WebPack базовая сборка

Подготовка

Webpack - это инструмент сборки и веб разработки, который сует все файлы (html, css, скрипты, картинки) в один оптимизированный и готовый к развертыванию пакет.

Предоставляет функции по минификации и оптимизации кода, обработке изображений, шрифтов, поддержке препроцессоров CSS и т.д.

Основная идея в том, чтобы процесс разработки стал эффективнее и позволить разработчикам не тратить время на ручное управление файлами и зависимостями

1 - создаем проект, package.json

Для начала создадим проект. Открываем gitBush или командную строку (можно в самой IDE) и пишем

`npm init -y` - это команда в Node Package Manager (NPM), которая создает файл `package.json` для проекта с настройками по умолчанию без запросов на ввод данных от пользователя. Он использует настройки по умолчанию для имени проекта, версии, описания, точки входа и других параметров. Это полезно, когда вы начинаете новый проект и хотите быстро создать `package.json` без необходимости вводить каждое значение вручную.

Если вы используете `npm init` без флага `-y` **(автоматом yes)**, то команда будет интерактивной, и npm задаст вам ряд вопросов для настройки `package.json`. Это интерактивный режим, в котором вы должны будете вручную ввести данные о вашем проекте, такие как имя, версия, описание, точка входа, команда для тестирования и т.д.

ПРОЩЕ NPM, потому что у YARN меньше параметров будет создано.

yarn init -y

```
{
  "name": "web-pack",
  "version": "1.0.0",
```

npm init -y

```
{
  "name": "web-pack",
  "version": "1.0.0",
```

```
  "main": "index.js",
  "repository": "https:...",
  "author": "Anonim",
  "license": "MIT"
}
```

```
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo ..."
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Ну можно и yarn, потому что и в npm придется что-то дописывать.

Открываем в среде разработки папку. Смотрим в package.json. МОЖЕМ ВООБЩЕ НИЧЕГО ТУТ НЕ ТРОГАТЬ, а можем и удалить

Что тут написано?

- main: 'index.js' — точка входа,
- description — краткое описание проекта
- version — версия моего проекта
- script: test: "..."— то что запускаем. Ключ можем задавать произвольно, а значение — это описание что конкретно хочу сделать
- keywords: [] — это массив, помогающий найти модуль в репозитории npm
- author — собственно автор этого будущего шедевра.
- license — права на интеллектуальную собственность, это для webstorm

Напишем заранее в scripts команду для запуска сборки

Обязательно в двойных кавычках, в одинарных и без кавычек не работает - ругается. Это требования JSON формата, чтобы JSON-парсер мог правильно интерпретировать файл.

`scripts: {" build ": "webpack"}`

Остальное будет добавляться по ходу, либо будем редактировать то, что есть

2 - устанавливаем webpack

Теперь установим сам webpack, с флагом —save-dev, потому что это у нас dev зависимость.

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

Или yarn

```
yarn add webpack-cli webpack -D
```

Что тут написано?

- Флаг - `save` — информация об установленном пакете сохраняется в файле `package.json`, и пакет будет автоматически установлен при выполнении команды `npm install` без указания конкретного пакета. То есть я могу товарищу скинуть проект без nodemodules, но с package.json, а он у себя запустит yarn/npm install и все необходимые для работы этого проекта зависимости установятся.
- Флаг `-dev` указывает NPM на то, что пакет, который вы устанавливаете, должен быть добавлен в раздел `devDependencies` вашего `package.json`. Это означает, что пакеты будут доступны для использования в процессе разработки, но не будут включены в конечный продукт. Содержит пакеты только для сборки вашего приложения, не влияющие на работу приложения, и используется только для разработки. В финальный бандл для прода не будет включен.
- `webpack-cli` - command line interface это инструмент командной строки (CLI) для работы с Webpack. Он предоставляет простой способ управления и запуска сборки проектов.

После этой команды будут созданы файлы package-lock.json, node-modules, а в package.json установятся зависимости

```
{
  "name": "web-pack-rs-school",
  "scripts": {
        "build": "webpack"
    },
  "devDependencies": {
    "webpack": "^5.90.3",
    "webpack-cli": "^5.1.4"
```

```
  }
}
```

Установились зависимости

3 - webpack.config.js

Теперь нам нужно создать файл webpack конфига. Сами создаем webpack.config

Стартовый конфиг для простой сборки можно найти на главной странице сайта webpack (копируем и вставляем)

Только заменим формат js точки входа на ts

```
const path = require('path'); // -- это стандартный модуль Node.
js

module.exports = {
  entry: './src/index.ts',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
  },
};
```

В корне создаем файл конфига `webpack.config.js` и пихаем туда этот код.

Что тут написано?

- `const path = require('path');` — Стандартный импорт nodeJs, загружает модуль Node.js под названием 'path'. Этот модуль предоставляет утилиты для работы с путями к файлам и директориям в Node.js. У него свои методы, можете сами поковыряться где-нибудь, посмотреть.

  В output.path будем импользовать метод resolve

  Path.resolve(__dirname, ‘dist’) — означает определение абсолютного пути до папки dist — это название папки, куда будет собиратсья сборка. Она будет создаваться автоматически при запуске скрипта npm run build (этот скрипт напишем в package.json)

- module.exports — Это **часть системы модулей Node. js, которая позволяет разделять код на отдельные файлы и затем импортировать и использовать этот код в других файлах**

  Получается, вложенные module.exports объекты являются объектами и функциями, которые берет webpack при сборке. Как инструкции. И если нет указаний, то делает по умолчанию

- entry — точка входа в наше приложение. тут написано index.js, сейчас создадим
- output — это куда будем билдить наш готовый проект.
- bundle.js - это процесс выявления импортированных файлов и объединения их в один «собранный» файл (часто называемый «bundle» или «бандл»). Этот бандл после подключения на веб-страницу загружает всё приложение за один раз

Для начала **создадим папку src и файл index.ts**

Внутри напишем например console.log('Hello world')

4 - mode

Запустим первую сборку `npm run build`

Вывалится warning "The 'mode' option has not been set ... "

Напишем в webpack.config.js mode production

```
const path = require("path");

module.exports = {
    mode: "production",
  entry: "./src/index.ts",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "bundle.js",
  },
};
```

Если исправим так, то без warning

ИЛИ можно в файле package json для скрипта build сделать флаг

```
{
  "name": "web-pack-rs-school",
  "scripts": {
    "build": "webpack --mode development"
  },
  "devDependencies": {
    "webpack": "^5.90.3",
    "webpack-cli": "^5.1.4"
  }
}
```

После этой команды создается папка dist с budle.js. Внутри увидим тот же console.log('Hello world')

▼ Что такое mode?

`mode` в конфигурации Webpack определяет в каком режиме будет работать сборка. Этот параметр позволяет указать Webpack на оптимизации и установить определенные значения по умолчанию в зависимости от выбранного режима. Вот доступные значения для `mode`:

1. **development** (разработка): Этот режим оптимизирован для разработки. Включает дополнительные инструменты (source maps) и дружественные для разработчика предупреждения. Остаются некоторые оптимизации, которые облегчают отладку. В этой сборке мы не будем подключать source maps.
2. **production** (производство): Этот режим оптимизирован для производства, то есть уже готов для выкладывания на сервере. Webpack будет применять оптимизации, такие как минификация и объединение файлов, чтобы уменьшить размер и улучшить производительность вашего приложения. Исходные карты могут быть отключены, чтобы уменьшить объем файлов.
3. **none** (нет): В этом режиме Webpack не применяет никаких дополнительных оптимизаций или установок по умолчанию. Это позволяет полностью контролировать конфигурацию и оптимизации вручную.

Удалим папку dist

5 - gitignore (можем не делать, если не будем делать git репозиторий)

Закинем в проект еще файл gitignore

.gitignore — чтобы наши nodemodules не попадали в репу, ну и .idea

```
/node_modules
.idea
```

Это минимальный набор действий чтобы запустить webpack.

Но мы же работать будем с typescript, поэтому переходим в 6 пункт

6 - ts.config

Нужен файл ts.config — настраивает транспилятор ts в js

Сделаем это командой `tsc --init` . Создает конфиг именно в этом проекте.

Если ts глобально не установлен, то можно написать `npx tsc --init`

Создается файл **tsconfig.json**

Там настройки транспилятора.

Включим вот эти. Пропишем src, путь до папки с нашими исходниками

```
"noImplicitAny": true,   /* Enable error 'any' type. */
"rootDir": "./src",      /* Root folder */
```

Tsconfig будет красным, потому что у нас нет файлов ts. Теперь сделаем все файлы (index, test) в src на формат ts

Подготовительную работу сделали, но сборщик все еще бестолковый.

Он не умеет работать с HTML, CSS и всеми остальными форматами файлов.

Для решения этих вопросов у WebPack есть плагины и loader-ы.

Для любого формата файлов нужны свои loader-ы. Вместе с некоторыми loader используются еще и плагины, которые имеют больше возможностей по настройке.

Loader-ы и плагины

Typescript loader

Чтобы сборщик работал с современным синтаксисом JS ES6 и выше, нужно использовать loader **Babel**, чтобы современный транспилировал на более старую версию JS.

Но мы будем делать на примере TS, а не Babel, потому что мы будем работать с TS, а не JS.

```
npm i -D typescript ts-loader
```

Что тут написано?

- typescript - собственно
- -D — это сокращение от --save-dev
- ts-loader — это для loader для webpack, который позволяет компилировать TS файлы в JS.

Теперь подключим typescript и ts-loader в webpack.config.json

Чтобы подключить ts добавим модуль.

Добавим поле rules - массив, куда добавим ts-loader

Rules (правила) для webpack указывают webpack-у, как обрабатывать разные типы файлов при их загрузке в проект. Это может включать компиляцию, транспиляцию, преобразование файлов и многое другое.

```
const path = require("path");
```

```
module.exports = {
  mode: "production",
  entry: "./src/index.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "bundle.js",
  },
  module: {
    rules: [
      {
        test: /\.[tj]s$/,
        use: 'ts-loader',
        exclude: /node_modules/,
      }
    ]
  }
};
```

ЧТО ЗА ДИЧЬ ДОБАВИЛИ???

В массив rules добавляем объекты.

- test — регулярное выражение, которое говорит какие файлы нужно обрабатывать данным loader-ом.

  Вот эта строка `/\.[tj]s$/` это регулярное выражение

  - `/` — обозначает начало и конец регулярного выражения.
  - `\.` — экранированный точечный символ (`\.`), который сопоставляется с точкой в строке. Обратный слэш используется для экранирования точки, потому что в регулярных выражениях точка обычно означает "любой символ". Таким образом, `\.` сопоставляется только с самой точкой в строке, а не с любым символом.
  - `[tj]` — символьный класс `[...]`, который сопоставляется с одним из символов внутри квадратных скобок. В данном случае это `t` или `j`, то есть либо буква `t`, либо буква `j`.
  - `s` — просто буква `s`.
  - `$` — символ конца строки.

  Можно было бы написать `/\.ts | js$/`, а тут сократили.

- use: 'ts-loader', — использовать ts-loader для обработки файлов .ts и js
- exclude: /node_modules/, — исключить файлы из папки node_modules

Свойство resolve:{...} в webpack.config

Еще добавим свойство после module **resolve.** объект resolve в конфигурации webpack используется для настройки способа, которым webpack находит и разрешает модули. Например, вы можете использовать его для указания, где искать модули, какие расширения файлов использовать при разрешении, как обрабатывать импорты и т.д. Это может помочь избежать ошибок разрешения модулей и улучшить производительность сборки.

В resolve укажем **extensions**, массив, где указываем, какие расширения файлов нам не обязательно будет указывать при импортах и экспортах.

```
const path = require("path");

module.exports = {
  mode: "production",
  entry: "./src/index.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "bundle.js",
  },
  module: {
    rules: [
      {
        test: /\.[tj]s$/,
        use: "ts-loader",
        exclude: /node_modules/,
      },
    ],
  },
  resolve: {
    extensions: [".ts", ".js"],
  },
};
```

> Это позволит нам при написании в коде import не писать для них в конце их расширение
>
> Например есть отдельный файл test.ts, где напишем функцию и экспортируем ее

```
export function test() {
  console.log("Здарова, атец");
}
```

> А в index.ts будем ее импортировать

```
import { test } from './test.ts'

console.log("Hello world");
test();
```

> Нам теперь при импортах не обязательно указывать расширения, если они ts или js

Запустим проверку сборки — `npm run build`

Запустился, бандл собрался.

Удалим бандл и продолжим

На данный момент наш webpack может собирать ts файлы в bundle.js

---

HTML плагин

Начнем с html плагина. Что это такое

> плагин для Webpack, который упрощает создание HTML-файлов для веб-приложений. Он автоматически генерирует HTML-файл и добавляет в него ссылки на сжатые JavaScript и CSS файлы, которые были сгенерированы в результате сборки проекта с помощью Webpack. Это позволяет избежать ручного создания и обновления HTML-файлов при изменениях в коде.

После установки плагина через npm или yarn, вы можете использовать `require` , чтобы подключить его в своем конфигурационном файле Webpack и настроить его для своего проекта.

`npm i -D html-webpack-plugin`

Устанавливаем, теперь подключаем модуль к webpack.config.

```
const HtmlWebpackPlugin = require('html-webpack-plugin')
```

Это конструктор.

функция `require` используется для импорта модулей.

Добавим поле **plugins**: [...]

Там будут те свойства, которые будут использоваться в проекте.

Нам нужно создать в этом массиве конструктором объект, в котором можем перечислять опции. Например title. Это будет title тег внутри генерируемого html.

Либо можем создать свой index.html, который наполним своими тегами.

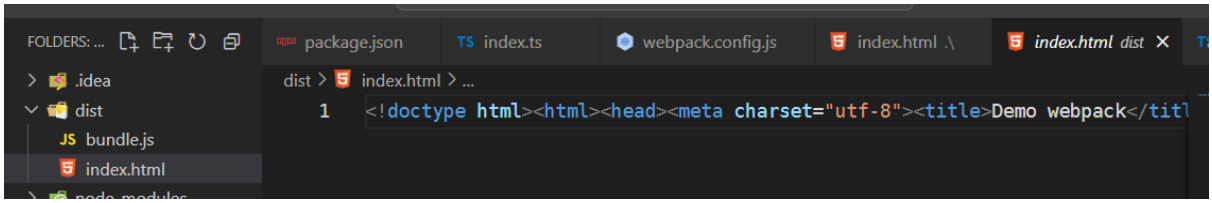
Тогда нужно будет тут, в опциях указать к нему путь.

Эта опция называется `template`, куда передаем путь до файла. Допустим мы его в src сделаем (index.html). Тогда webpack возьмет за основу существующий html файл. **ОБЯЗАТЕЛЬНАЯ НАСТРОЙКА, иначе html создаваться не будет** и при сборке далее могут возникнуть проблемы!

Можем указать ключ filename: указывает название html файла в бандле

```
  },
  plugins: [
    new HtmlWebpackPlugin({
      title: "Demo webpack",
      template: "./src/index.html",
    }),
  ],
};
```

Попробуем с title сначала, без template. Запускаем сборку



Создал с тем title, который указали

Удалим dist.

Теперь с template — сначала сделаем свой html.

Смотрим

Работает

Удалим dist.

Assets

Это наши иконки, картинки, медиафайлы, шрифты

Раньше для работы с ассетами требовалось устанавливать отдельный file loader. Теперь , с новым webpack нет. Теперь можно без них.

Для этого идем в module.exports.module.rules, подключаем тут новым объектом: test: регулярное выражение, на какие файлы импорты не нужен формат

и добавим type.

Также для шрифтов

```
  },
  module: {
    rules: [
      {
        test: /\.[tj]s$/,
        use: "ts-loader",
        exclude: /node_modules/,
      },
      {
        test: /\.(?:ico|gif|png|jpg|jpeg|svg)$/i,
                type: 'asset/resource',
      },
```

```
      {
        test: /\.(?:woff(2)|eot|ttf|otf)$/i,
        type: 'asset/inline'
      }
    ],
  },
  res
```

Опция `type: "asset/resource"` означает, что эти файлы будут обрабатываться как ресурсы и копироваться в выходную директорию, но имя будет каракульное. Копироваться в формат resource будут файлы, по дефолтной настройке webpack, более 8 кб.

А `asset/inline` — означает, что файлы будут встроены непосредственно в сгенерированный JavaScript

Если покопаться глубже в документации, можно настроить размер вставляемого файла. По дефолту, если файл больше 8 кб, то скопируется как resource, а не инлайн. Если меньше, то инлайн.

Я бы хотел, чтобы картинки или шрифты, или медиафайлы были не в корне выходной папки сборки dist, а в какой-то отдельной.

Нужна доп настройка module.exports.output. `assetModuleFilename: 'assets/[name][ext]'`

```
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
    assetModuleFilename: 'assets/[name][ext]',
  },
```

Так мы webpack-у говорим, "братанский, сохраняй все наши ассеты в папку (создай ее, с таким именем) assets/{с таким именем файла][таким расширением]", где в квадратных скобках это шаблонное название.

▼ Что такое assetModuleFilename?

`assetModuleFilename` - это опция конфигурации в webpack, которая позволяет настраивать формат и путь для сохранения ресурсов, обрабатываемых через Asset Modules.

В вашем примере `'assets/[name][ext]'`:

- `'assets/'` указывает на каталог, в который будут сохраняться ресурсы.
- `[name]` заменяется на имя файла ресурса.
- `[ext]` заменяется на расширение файла ресурса.

Таким образом, с помощью этой конфигурации, ресурсы будут сохраняться в каталоге `assets` с их оригинальными именами и расширениями.

Например, если у вас есть файл изображения с именем `logo.png`, то он будет сохранен по следующему пути: `assets/logo.png`.

Это может быть полезно для организации файлов в структуру проекта или для задания специфического формата и пути для сохранения ресурсов в зависимости от требований вашего проекта.

Теперь проверим как это работает.

Создадим папку в src, например images, положим туда картинку.jpeg.

Создадим в index.html тег img и положим туда эту картинку.

Сделаем сборку и будем думать, что у нас в dist появится эта картинка.

Запускаем и видим, что никакой картинки нет. И в bundle.js тоже

Причина в том, что webpack пока не умеет работать с тем, что находится внутри index.html. Позже научим, с плагином HTML-loader

Удалим img из index.html

Попробуем тогда через index.ts

Сделаем импорт картинки в index.ts.

НО ВОЗНИКНЕТ ПРОБЛЕМА — typescript Ругается. Тип не знает какой.

Можно покопаться в документации и найти способ вставить через импорт, но я чет устал искать, поэтому воспользуемся другим способом

Сделаем вставку с помощью require

```
const img = require('./assets/tom.jpeg')
```

Запустим сборку.

Вуаля, у нас в собранном пакете появилась папка assets с тем же именем картинки, которая была в оригинале.

Также она появилась в bundle.js.

---

2 варианта импорта img в структуру html в сборке

Если создать в index.ts картинку img, через require, и не вставлять его в index.html, то при сборке он будет в выходном пакете отдельным файлом, но в структуре html этой картинки не будет. Она просто будет существовать без привязки к структуре html. А нам его нужно ее туда прикрепить, чтобы, если мы настроим стили для картнки, эти стили применялись к html в сборке.

Пока webpack так не умеет. Но это пока. Мы можем его научить.

Чтобы это реализовать есть 2 варианта

1 вариант — через index.ts

Удалим test.ts, он нам больше не понадобится.

Создадим картинку в index.ts, а из html удалим ее

html

```
<body>
  <div>
    <h1>Text</
  </div>
</body>
```

index.ts

```
// import img from './images/IMG.jpg'
const imgPath = require('./images/IMG.jpg')

const img = document.createElement('img')

img.src = imgPath
img.alt = 'samurai'

const body = document.querySelector('body')
body?.append(img)
```

Запустим тест

Отлично, картинка в структуре html и в bundle.js

Но согласись, так неудобно.

Мы хотим чтобы было проще. Хотим чтобы из html все картинки сами подтягивались.

2 вариант — через html. Тогда нужно установить Html-loader

Соответственно, удаляем импорт картинки из index.ts

Устанавливаем html-loader

`npm install --save-dev html-loader`

или

`yarn add -D html-loader`

Вставляем картинку в index.html

В webpack.config.js устанавливаем rules

```
...
    {
    test: /\.html$/i,
    loader: "html-loader",
  },
...
```

И делаем импорт index.html в index.ts

```
const html = require('./index.html')
```

Проверяем.

Отлично. Работает.

Особенность assets

assets находятся в кэше. И пользователь может не увидеть новые картинки, если мы сделаем изменения в билде. Это нужно фиксить

`assetModuleFilename: 'assets/[name][ext]'` Вот тут нужно будет тогда вместо name указать hash. Это уникальный id ассета. И если обновим проект, то он тоже

обновится, следовательно другой, следовательно из кэша удалится старый, появится новый и тогда все ок

Если мы в конфиге добавили assetModuleFilename: assets/[hash][ext], то при билде имя файла будет отличаться. Не так как в исходнике назвали.

Теперь сделаем build. Из-за того, что писали [hash], у нашего файла картинки будет каракульное имя dsf2e583lgkfdngl.jpeg

Это если мы работаем с файлами, которые могут изменяться.

Вернем [name][ext]

Плагин clean-webpack-plugin

`npm i -D clean-webpack-plugin`

- каждый раз после команды npm run build он сам удаляет старый билд

подключаем плагин к webpack.config

ПИШЕМ С ФИГУРНЫМИ СКОБКАМИ, т.к. его экспорт не по дефолту.

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const CopyPlugin = require("copy-webpack-plugin");
const { CleanWebpackPlugin } = require("clean-webpack-plugin")
```

Тоже есть свои настройки. Например можно сказать, чтобы он не удалял то, что не изменилось

```
...
    new CleanWebpackPlugin({
    cleanStaleWebpackAssets: false
  })
],
```

Плагин copy-webpack-plugin

Копирует отдельные файлы или целые каталоги, в папку билда.

Умеет копировать файлы из одной папки в другую.

`npm i -D copy-webpack-plugin`

Часто в проектах бывает, что нам не нужен какой-то новый hash, а нужно оставлять оригинальное имя файла.

Создадим папку dopAssets, в которой лежит еще одна картинка. И мы в билде хотим видеть эту картинку в определенном каталоге, либо в корне, с тем же названием, как и в нашем проекте.

подключаем плагин к webpack.config.

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const CopyPlugin = require("copy-webpack-plugin")

module.exports = {
...
  },
  plugins: [
    new HtmlWebpackPlugin({
      // title: "Demo webpack",
      template: "./src/index.html",
    }),
    new CopyPlugin({
      patterns: [
        {from: './src/dopAssets/', to: 'bundleAssets' }
      ]
    })
  ],
};
```

patterns - свойство объекта copyPlugin конструктора, в котором массив с объектом, который состоит из from = путь откуда копировать, to — куда вставлять

Папка создастся автоматом. Если не указывать to, то скопирует прямо в корень dist

Запускаем, проверяем. Работает.

Предостережение — если файлов не будет в папке dopAssets, то билд не соберется.

Если бы не использовали HTML Loader, то картинку можно было бы вставить в html с помощью CopyLoader — тогда путь до картинки нужно писать, используя файл, который скопировали в папку dist, т.е. не от оригинала, а уже от копии. Т.е. в данном случае `<img src = ‘./bundleAssets/useMemo.png’ alt=’ttt’ />`

Также и в случае, если буду делать это в index.ts. Т.е. require не нужен, а просто пишем путь до файла из папки bundleAssets

---

Стили, лоадеры для стилей

`npm i -D css-loader sass-loader sass mini-css-extract-plugin`

css-loader для работы со стилями

Препроцессор sass-loader, sass

`mini-css-extract-plugin` - извлекает css из JS, а то сам не умеет

ПОРЯДОК RULES ИМЕЕТ ЗНАЧЕНИЕ!!!

```
{
  test: /\.css$/i,
  use: ["style-loader", "css-loader"],
},
  {
  test: /\.s[ac]ss$/i,
  use: [
    // Creates `style` nodes from JS strings
    "style-loader",
    // Translates CSS into CommonJS
    "css-loader",
    // Compiles Sass to CSS
    "sass-loader",
  ],
},
```

Мы не устанавливали style-loader. Так что он нам не нужен. Нужно его заменить. Будем использовать плагин mini-css-extract

```
...
const MiniCssExtractPlugin = require("mini-css-extract-plugin")

module.exports = {
...
    new MiniCssExtractPlugin({
      filename: 'styles.css'
    })
  ],
};
```

А теперь для rules пропишем, что сначала должен отработать MiniCssExtractPlugin

ТОЛЬКО В ТАКОМ ПОРЯДКЕ!

```
{
  test: /\.css$/,
  use: [MiniCssExtractPlugin.loader, 'css-loader'],
},
{
  test: /\.s[ac]ss$/,
  use: [
        MiniCssExtractPlugin.loader,
        'css-loader',
        'sass-loader' ],
},
```

Теперь стили будут экспортироваться в файл style.css

Но можно использовать и такой шаблон

```
  new MiniCssExtractPlugin({
    filename: '[name].[hash].css'
  })
```

Чтобы наши стили тоже имели хэш и имя файла будет меняться

Для проверки сделаем стили style.scss

```scss
body {
  background-color: lemonchiffon;
  & button {
    background-color: chocolate;
  }
}
```

ЧТОБЫ эти стили были импортированы в проект, их нужно импортировать в index.ts. Без этого файл стилей мы не увидим

```ts
import './style.scss'

const imgPath = require('./images/самурай без меча.jpg')

const img = document.createElement('img')

img.src = imgPath
img.alt = 'samurai'

const body = document.querySelector('body')
body?.append(img)
```

Разные конфиги (две сборки, и потом еще одна на всякий случай)

Часто бывает так, что у нас может быть несколько config. 1 — для develop сборки с source map (в этой сборке без source map), без минификации кода с настроенным dev сервером , 2 - для production сборки, где уже добавлены плагины для минификации, оптимизации и чего-то еще.

Покажем как это можно сделать. В package.json сделаем скрипты для обеих сборок.

```
"scripts": {
      "build": "webpack --config ./webpack-prod.config.json",
    "dev-build": "webpack --config ./webpack-dev.config.json"
}
```

The end. Можно впринципе больше ничего не делать, но мы пойдем дальше.

Но можно сделать по другому.

Можно не делать 2 файла конфига, оставить скрипт как был, но дополнить свойством с кастомным флагом и Mode develop как переменная

У самого webpack есть такая возможность.

Для этого нужно module.exports обернуть в колбек и использовать переменную

Чтобы использовать переменную в package.json нужно использовать флаг `--env` и внести название переменной, у нас тут будет `develop`

```
  "scripts": {
    "build": "webpack",
    "dev": "webpack --env develop"
  },
```

```
module.exports = ({develop}) => ({
  mode: develop ? "development" : "production",
  entry: "./src/index.ts",
  output: {
...
})
```

А принимать будет значение деструктуризации develop.

Создадим функцию

Есть такая настройка devServer — в самом низу webpack.config напишем ее

▼ Что такое devServer

- это опция конфигурации, которая позволяет настроить локальный сервер разработки для вашего веб-приложения при использовании Webpack.

Некоторые наиболее распространенные параметры конфигурации `devServer`:

1. `contentBase` (устаревший). Вместо него нужно использовать `static: {…}`: Указывает путь к статическим файлам, которые будут доступны через сервер разработки. Обычно это каталог, содержащий ваши HTML, CSS и другие ресурсы. Также при параметре static, если подключить автообновление сервера, он будет обновляться самостоятельно при изменении кода в исходном src
2. `port`: Порт, на котором будет запущен локальный сервер.
3. `open`: Автоматически открывает браузер при запуске локального сервера.

Пример конфигурации `devServer` в файле `webpack.config.js`:

```
module.exports = {
  // Остальные параметры конфигурации...

  devServer: {
        open: true,
        port: 9000,
    // contentBase: path.join(__dirname, 'src'),
        static: {
        directory: path.join(__dirname, "src"), // Каталог проек
      },
  },
};
```

У нее есть несколько параметров. Можно ознакомиться в оф доке.

Стоит отметить, что в свойстве directory в строке пишем название корневой папки нашего проекта (не сборки), а это src

contentBase больше не поддерживается в новых версиях webpack-dev-server. Вместо нее нужно использовать static

```
  ],
  devServer: {
    open: true,
    port: 9000,
```

```
    // contentBase: path.join(__dirname, "src"),
        static: {
        directory: path.join(__dirname, "src"),
      },
  }
});
```

Но, devServer нужен в production mode, а в develop он не нужен. Поэтому вместо этого пишем функцию, а на это место, где был devServer будем ее использовать

Вызываем функцию, которая вернет объект в зависимости от того, в каком моде запускаем webpack

▼ **ДОПОЛНЕНИЕ! - можно использовать 1 из 2х вариантов за отслеживанием изменений проекта**

contentBase и static (ниже) примерно одинаковые настройки, разница в том, что contentBase — сервер разработки будет следить за файлами в указанной папке. Static то же самое, но можно писать несколько папок. Однако contentBase уже depricated, поэтому используем static. И еще раз напомню, что при этой настройке будет автообновление сервера

```
static: {
  directory: path.join(__dirname, "src"),
},
```

Либо в package.json можно написать “`dev”: “… --watch“` Этот флаг — то же самое что “следи”, т.е. обновляй сам сервер, если что-то изменилось.

Получается можно указать один параметр из 2х `--watch` в package.json либо в webpack.config `static`

```
const devServer = (isDev) =>
  !isDev
    ? {}
    : {
        devServer: {
          open: true,
          port: 9000,
```

```
                static: {
                  directory: path.join(__dirname, "src"),
                },
        },
      };

module.exports = ({ develop }) => ({
...
  ],
  ...devServer(develop)
});
```

Теперь, чтобы он(сервер) у нас заработал при старте сборки, нам нужно обновить скрипты в package.json

И вот он всякий случай

Давайте еще один скрипт сделаем — serve, позже объясню зачем

```
  "scripts": {
    "build": "webpack",
    "dev": "webpack --env develop",
        "serve": "webpack serve --env develop",
  },
```

serve это команда, которая запускает dev-сервер webpack, предоставляя возможность разрабатывать приложение в режиме разработки. Когда вы запускаете `npm run dev`, webpack запускает dev-сервер, который автоматически пересобирает ваше приложение при изменении файлов и предоставляет его на локальном сервере по указанному в настройках порту (по умолчанию 8080). Это позволяет вам видеть изменения в реальном времени без необходимости каждый раз вручную пересобирать проект.

Возникнет ПРЕДУПРЕЖДЕНИЕ что для использования SERVE необходимо **установить пакет webpack-dev-server**, предложит установить, отвечаем YES!

Запускаем сборку build, пакет dist собирается в моде production

Запускаем сборку dev, пакет dist собирается в моде development

Запускаем сборку serve, пакет dist по идее должен собраться в моде development, НО МЫ ВИДИМ, ЧТО ПАПКА dist ПУСТАЯ.

Причина в том, что так работает webpack-dev-server — он скрывает бандл папку от разрабов, потому что считает что эта папка тебе больше не нужна, а нужна только ему, для запуска сервера. Поэтому она недоступна.

Webpack-dev-server запускается в памяти

По другому увидеть свои файлы дист можно так — http://localhost:9000/webpack-dev-server на своем локальном сервере.

Ну да, видны файлы, но все в формате JSON

THE END