

# МОДУЛИ И WEBRACK



МИХАИЛ КУЗНЕЦОВ / ING BANK



# МИХАИЛ КУЗНЕЦОВ

Разработчик в ING Bank



[@mkuznetcov](https://www.instagram.com/mkuznetcov)



# ПЛАН ЗАНЯТИЯ

1. [Что такое модули?](#)
2. [ES Modules](#)
3. [CommonJS](#)
4. [Webpack и его конфигурационный файл](#)



# ЧТО ТАКОЕ МОДУЛИ?

# СЛОЖНОСТЬ ПРИЛОЖЕНИЙ

Сложность JS-приложений очень сильно выросла: появилась сложная бизнес-логика на клиенте и необходимость управлять большим количеством объектов, и, как следствие, появилась необходимость "разделять" код (а не писать всё в одном файле).

Раньше было достаточно вручную при помощи тега `script` подключить несколько скриптов на HTML-страницу и легко следить за их зависимостями, то сейчас это невозможно поддерживать и расширять.



## ПРОБЛЕМЫ ПРИ УСТАРЕВШЕМ ПОДХОДЕ

- конфликты имён в глобальной области видимости (браузер просто "схлопывает" все подключенные JS-файлы в один)
- необходимо вручную следить за последовательностью загрузки и инициализации (для каждой HTML-страницы)

# ЧТО ТАКОЕ МОДУЛИ?

Модули - это возможность организовать разбиение кода на изолированные области видимости.

Каждый модуль может экспортировать какие-то имена (для использования в другом модуле) или импортировать их.

Необходимо явно указывать, что экспортируется (и только это можно импортировать) и что импортируется.

Таким образом, мы получаем контролируемую работу с именами.



# ПОДХОДЫ К ОРГАНИЗАЦИИ МОДУЛЕЙ

На сегодняшний день существует два ключевых подхода:

- CommonJS — система модулей, используемая Node.js
- ESM (ECMAScript Modules) — система модулей из стандарта ECMA\*



---

## Q & A

Q: почему их два?

A: причин несколько:

1. Ключевых платформ\*, на которых сейчас работает JS, две - браузер и Node.js, поэтому и подходы различаются
2. В Node.js система модулей появилась раньше, чем в стандарте

Node.js перенимает поддержку системы модулей из стандарта, но она до сих пор поддерживается в экспериментальном режиме.

Мы с вами рассмотрим оба подхода, поскольку вам так или иначе придётся работать с ними.

Примечание\*: про платформы мы будем говорить на следующих лекциях.

# ЗАДАЧА

В качестве задачи рассмотрим следующую: нам поручено написать приложение, которое интегрируется со сторонними сервисами.

Для простоты представим, что один из сервисов будет называться Logs (туда мы будем отправлять логи), а второй - корпоративная система статистики Stats, в которой мы будем фиксировать действия пользователя.

Мы, как разработчики приложения - не пишем ни Logs, ни Stats, но хотим подключить их к своему приложению и использовать.

Разработчики предоставили соответствующие библиотеки в виде файлов logs.js и stats.js (см. в репозитории с кодом)

Поэтому давайте разбираться, как это можно сделать в разных системах.



# ES MODULES

# EXPORT & IMPORT

В части модулей у нас есть две ключевые конструкции:

1. `export` - возможность из одного модуля экспортировать имена для использования в другом модуле
2. `import` - возможность из одного модуля импортировать из другого модуля имена для использования в текущем модуле

Сразу скажем, что импортировать можно только то, что экспортируется (в этом весь смысл инкапсуляции в модулях).



# EXPORT & IMPORT

Нам предоставляется огромный спектр возможностей в части определения того, что мы можем экспортировать и импортировать.

Большинство из них вы будете использовать достаточно редко (если вообще будете), поэтому мы с вами рассмотрим наиболее ключевые.

# БРАУЗЕР

Если мы собираемся использовать модули в браузере, то сам скрипт подключается особым образом:

```
<script src="./js/app.js" type="module"></script>
```

Указание `type="module"` переводит файл в режим модулей (т.е. имена из него не экспортируются в глобальную область видимости).

Кроме того, сам скрипт автоматически интерпретируется в `strict mode`.

# ПОДКЛЮЧЕНИЕ МОДУЛЕЙ

Раньше, чтобы подключить JS-файл, мы его были обязаны прописать в теге скрипт.

В случае с модулями, мы можем использовать ключевое слово `import` для подключения модуля (при этом никакой `script` не нужен):

```
// файл app.js
import './logs.js';
import './stats.js';
```

При таком импорте браузер делает HTTP-запрос на URL (URL разрешается относительно текущего файла), подгружает его и выполняет его содержимое:

Name	Status	Type	Initiator
 app.js	200	script	(index)
 logs.js	200	script	app.js:1
 stats.js	200	script	app.js:2

# ПУТИ

При импорте модуля обязательно должен быть указан путь (без пути нельзя):

```
// файл app.js
import './logs.js';
import 'stats.js'; // <- неправильно
```





При таком импорте браузер сгенерирует ошибку:

```
✖ Uncaught TypeError: Failed to resolve module specifier "stats.js".  
  Relative references must start with either "/", "./", or "../".  
127.0.0.1/:1
```



# ПОВТОРНЫЙ IMPORT

Браузер достаточно умный, и если нескольких модулях запрашивается какой-то ещё один модуль (в нашем случае `lib.js`), то подгружен он будет всего один лишь раз:

Name	Status	Type	Initiator
 app.js	200	script	<u>(index)</u>
 logs.js	200	script	<u>app.js:1</u>
 stats.js	200	script	<u>app.js:2</u>
 lib.js	200	script	<u>logs.js:1</u>

# ЭКСПОРТ

Как мы уже видели, `import` позволяет подгрузить и выполнить модуль. Но в большинстве случаев, модули не содержат какого-то исполняемого кода\*, чаще они содержат объявление имён, функций и типов.

Ключевое слово `export` ставится перед объявлением переменных, функций и классов:

```
// файл stats.js
// можем ставить export перед функциями и классами
export class Client { }
// можем перед объявлением переменной
export const timeout = 5;

const internal = 42;

const defaultClient = new Client();
// export default можно писать только рядом с объявлением класса или функции
// но для переменной нужно писать отдельно
// важно: default экспорт может быть только один!
export default defaultClient;

console.log('stats executed');
```

# DEFAULT EXPORT

`default export` может быть только один на весь модуль и чаще всего в него "кладут" самое часто используемое имя (в нашем случае - `client`):

```
const defaultClient = new Client();  
export default defaultClient;
```

**Важно:** если вы с первого взгляда не можете сказать, что у вас "самое часто используемое", тогда не используйте `export default` - он не обязателен.

# ИМПОРТ

Вариант с раздельным импортом:

```
1 // файл app.js
2 // импорт по имени
3 // (имя при экспорте и при импорте должно совпадать):
4 import { Client, timeout } from './stats.js';
5 // импорт default:
6 import client from './stats.js'; // можем именовать как хотим
7
8 // дальше можем использовать client, Client и timeout
```

Совмещённый вариант (самый часто используемый вариант):

```
1 // файл app.js
2 import client, { Client, timeout } from './stats.js';
3 // т.е. имя для default export не берётся в фигурные скобки
4
5 // дальше можем использовать client, Client и timeout
```

# ИМПОРТ

При попытке импорта неэкспортированного значения получим ошибку:

```
1 | // файл app.js
2 | import { internal } from './stats.js';
```

```
✖ Uncaught SyntaxError: The requested module './stats.js' does not
  provide an export named 'unexpored'                                127.0.0.1/:1
```

При попытке импорта несуществующего экспорта тоже:

```
1 | // файл app.js
2 | import { external } from './stats.js';
```

```
✖ Uncaught SyntaxError: The requested module './stats.js' does not
  provide an export named 'external'                                127.0.0.1/:1
```

# ESLINT

ESLint с настройками Airbnb будет "ругаться" на расширение модуля при импорте. Но браузер синтаксис без расширения (.js или .mjs) принимать не будет.

Настройка ESLint:

```
"rules": {  
  "import/extensions": [  
    "error",  
    "ignorePackages"  
  ]  
}
```

Чуть позже мы познакомимся с бандлером (Webpack), который позволит не указывать расширения.

Детали [по ссылке](#).

# .MJS

Расширение `mjs` было предложено разработчиками Node.js для явного указания того, что файл использует систему ESM.

Браузеру наличие расширения `mjs` не принципиально.

Ключевое - если вы используете модули без бандлера (нативно в браузере), веб-сервер должен отдавать для файлов `mjs` заголовок `Content-Type: application/javascript`

✗ Failed to load module script: The server responded with a non-JavaScript MIME raw:1 type of "application/octet-stream". Strict MIME type checking is enforced for module scripts per HTML spec.

# КОНФЛИКТ ИМЁН

А теперь представим ситуацию, когда модуль `logs` писал тот же человек, что и `stats`, и он решил сделать одинаковый "интерфейс" (в данном случае имеется в виду набор экспортированных имён):

```
1 // файл logs.js
2     import './lib.js';
3
4 // можем ставить export перед функциями и классами
5 export class Client { }
6 // можем перед объявлением переменной
7 export const timeout = 5;
8
9 const defaultClient = new Client();
10 // export default можно писать только рядом с объявлением класса или функции
11 // но для переменной нужно писать отдельно
12 export default defaultClient; //
13
14 console.log('logs executed');
```



# КОНФЛИКТ ИМЁН

На резонный вопрос "Зачем?", он ответил, что модули обеспечивают инкапсуляцию и имена не будут конфликтовать с другими модулями.

Инкапсуляция действительно обеспечивается (поскольку имена в модулях теперь не в одной области видимости), но импортировать оба в `app.js` одинаково не получится:

```
1 // файл app.js
2 import client, { Client, timeout } from './logs.js';
3 import client, { Client, timeout } from './stats.js';
```

✖ Uncaught SyntaxError: Identifier 'client' has already been declared app.js:17

# КОНФЛИКТ ИМЁН

Мы можем переименовать один из `client` 'ов (например в `logsClient`), либо переименовать оба:

```
1 // файл app.js
2 import logsClient, { Client, timeout } from './logs.js';
3 import statsClient, { Client, timeout } from './stats.js';
```

Но конфликт с другими именами (в частности `Client`) останется:

```
✖ Uncaught SyntaxError: Identifier 'Client' has already been declared    app.js:21
```

# ПЕРЕИМЕНОВАНИЕ ПРИ ИМПОРТЕ

Для именованных экспортов, мы можем использовать переименование при импорте, что снимет проблему:

```
1 // файл app.js
2 import logsClient, { Client as LogsClient, timeout as logsTimeout } from './logs.js';
3 import statsClient, { Client as StatsClient, timeout as statsTimeout } from './stats.js';
```

## \* IMPORT

Иногда удобнее не перечислять все имена, а импортировать их разом, "сложив" в объект:

```
1 // файл app.js
2 import * as logs from './logs.js';
3 import * as stats from './stats.js';
4
5 console.log(stats);
```

▼ Module {Symbol(Symbol.toStringTag): "Module"} ⓘ

app.js:30

- ▶ Client: class Client
- ▶ default: Client
  - timeout: 5
  - Symbol(Symbol.toStringTag): "Module"
- ▶ get Client: f ()
- ▶ set Client: f ()
- ▶ get default: f ()
- ▶ set default: f ()
- ▶ get timeout: f ()
- ▶ set timeout: f ()

# КОМБИНИРОВАННЫЙ ИМПОРТ

При этом никто не запрещает комбинировать все три вида, но при этом можно совместно использовать только:

- default и именованный
- default и \*

Т.е. именованный и \* вместе нельзя.

```
1 // файл app.js
2 import * as logs from './logs.js';
3 import statsClient, * as stats from './stats.js';
4
5 console.log(statsClient);
```

# DYNAMIC IMPORT

Важно: в таком формате `import 'path';` `import` может использоваться только на самом верхнем уровне скрипта (т.е. его нельзя "положить" в условие, завернуть в try-catch или функцию).

В современном стандарте появилась возможность "динамически" подключать модули с помощью **функции** `import`:

```
1 // файл app.js
2 import('./dynamic.js').then(mod => console.log(mod));
```

```
1 // файл dynamic.js
2 export default {
3   id: 'default',
4 };
5
6 export const named = {
7   id: 'named',
8 };
9
10 console.log('dynamic executed');
```

# DYNAMIC IMPORT

Функция `import` возвращает `Promise`, при этом при успешном подключении код модуля автоматически выполнится, а результаты будут положены в специальный объект:

```
▼ Module {Symbol(Symbol.toStringTag): "Module"} ⓘ  
  ▼ default: Object  
    id: "default"  
    ► __proto__: Object  
  ▼ named: Object  
    id: "named"  
    ► __proto__: Object  
    Symbol(Symbol.toStringTag): "Module"  
    ► get default: f ()  
    ► set default: f ()  
    ► get named: f ()  
    ► set named: f ()
```

[app.js:35](#)

Обратите внимание, что `default export` был положен в свойство `default`.



# COMMONJS



# MODULE.EXPORTS

В CommonJS, если мы хотим сделать имя (функцию, переменную либо объект) доступным из нашего модуля, то:

```
1  class Client { }
2  const timeout = 5;
3
4  const defaultClient = new Client();
5
6  module.exports = {
7    Client,
8    timeout,
9    defaultClient,
10 };
11
12 console.log('stats executed');
```

`module.exports` - это специальный объект, определяющий, что будет экспортировано из модуля.

# EXPORTS VS MODULE.EXPORTS

Помимо `module.exports` есть ещё имя `exports`, которое указывает на тот же объект, что и `module.exports`.

Самое важное, что нужно запомнить: всегда экспортируется то, на что указывает `module.exports`.

```
1  class Client { }
2  const timeout = 5;
3
4  const defaultClient = new Client();
5
6  exports.Client = Client;
7  exports.timeout = timeout;
8  exports.defaultClient = defaultClient;
9
10 // теперь exports !== module.exports
11 module.exports = {
12   Client,
13   timeout,
14   defaultClient,
15 };
```

# REQUIRE

Если мы хотим использовать имя, экспортированное из другого модуля, в своём модуле, то:

```
1  const fs = require('fs');
2  const stats = require('./stats');
3      // можно использовать и destructuring
4  const { defaultClient as statsClient } = require('./stats');
5  // а можно просто "достать" свойство
6  const client = require('./stats').defaultClient;
7  // то же самое с logs
```

При этом код модуля так же будет выполнен (запускать надо через `node app.js`).

# REQUIRE

## Важно:

1. `require` - это функция и можно её использовать внутри `if` и других конструкций
2. Расширение модуля писать не принято
3. Если пакет установлен через `npm install` или является частью стандартной библиотеки, то путь пишется не относительно файла, а относительно стандартной библиотеки или `node_modules`:

```
1 | const fs = require('fs');  
2 | const moment = require('moment');  
3 | // нужно не забыть установить: npm install moment
```

# MAIN

На прошлой лекции мы проходили npm и в файле `package.json` было свойство `main`.

Это свойство определяет, какой файл будет импортироваться, если указать не путь, а имя npm-пакета `main` в `package.json`:

```
1 // файл нашего приложения
2 const moment = require('moment');
3 // файл package.json пакета moment
4 {
5     ...
6     "main": "./moment.js",
7     "name": "moment",
8     ...
9 }
10 // файл moment.js пакета moment
11 ;(function (global, factory) {
12     typeof exports === 'object' && typeof module !== 'undefined' ? module.exports
13     : typeof define === 'function' && define.amd ? define(factory) :
14     global.moment = factory()
15 })(this, (function () { ... }));
```

# MOMENT.JS

[Moment.js](#) - это удобная библиотека для работы со временем. Работает она как в Node.js, так и в браузере.

Происходит это благодаря анализу тек самых имён, которые мы проходили:

```
1  ;(function (global, factory) {  
2      // если node.js  
3      typeof exports === 'object' && typeof module !== 'undefined' ? module.exports  
4      // если amd (ещё одна система модулей)  
5      typeof define === 'function' && define.amd ? define(factory) :  
6      // если браузер  
7      global.moment = factory()  
8  }(this, (function () { ... })));
```

`this` в браузере будет указывать на `window`.

И в то же время, внутри самой библиотеки есть каталог `dist`, в котором расположена версия для ESM.



# ESM VS COMMONJS

# ЧТО ИСПОЛЬЗОВАТЬ?

Придётся использовать обе, т.к. платформа Node.js поддерживает ES Modules в экспериментальном режиме.

Детали на странице <https://nodejs.org/api/esm.html>

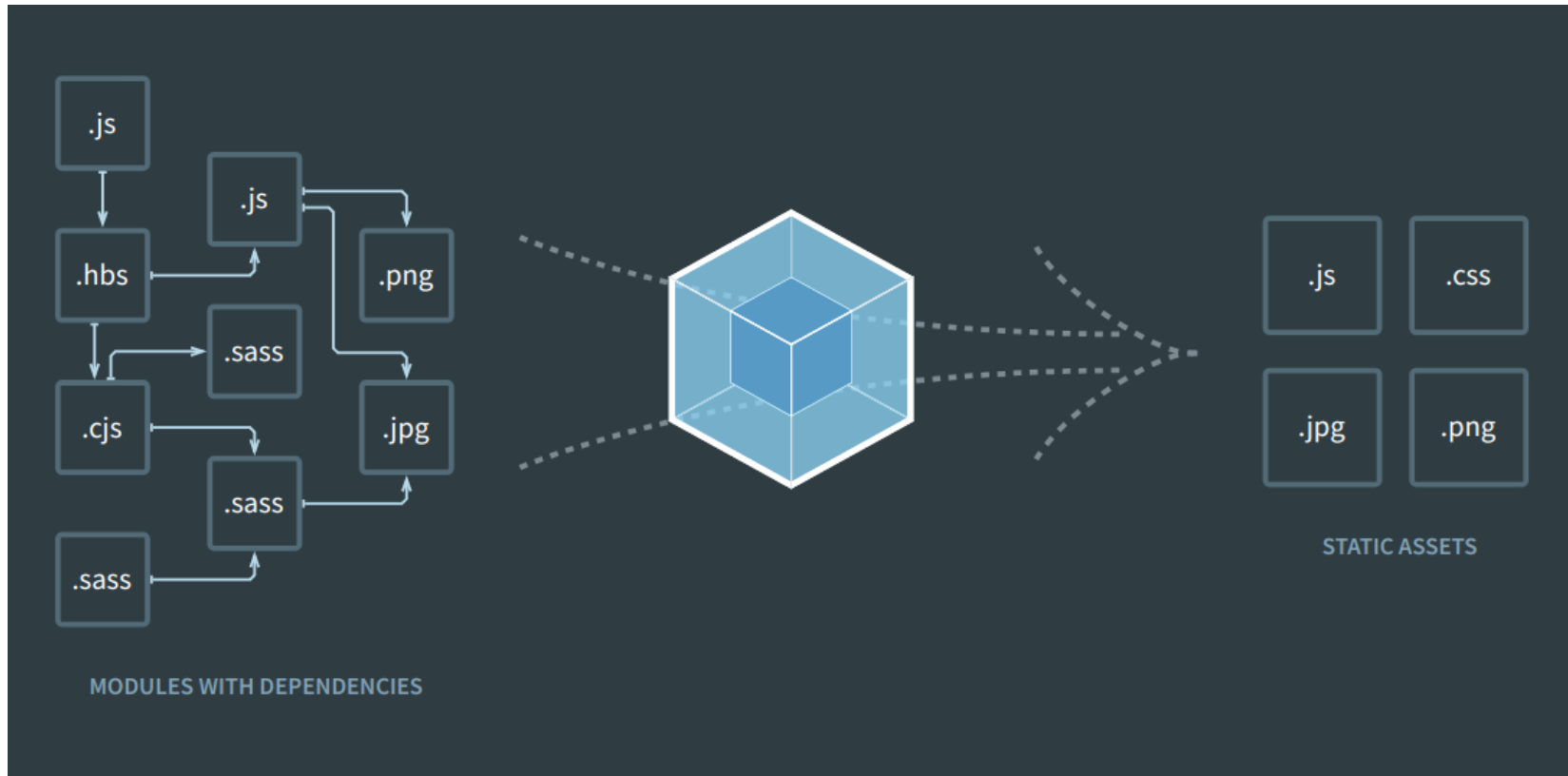
Кроме того, большинство библиотек рассчитаны на использование как в Node.js, так и в ESM.





# WEBPACK

# WEBPACK





# WEBPACK

[Webpack](#) - Module Bundler для JS-приложений. Позволяет объединять все ресурсы нашего приложения в Bundle (преобразованные, минимизированные, оптимизированные) и готовые для использования в продакшн-среде.

На сегодняшний день - самый популярный инструмент сборки в мире JS. Содержит интеграции с большинством других популярных инструментов.

# УСТАНОВКА

Для установки Webpack и поддержки Babel выполним следующую команду:

```
$ npm install --save-dev webpack webpack-cli babel-loader
```

В скриптах заменим `build`:

```
"scripts": {  
  ...  
  "build": "webpack --mode production"  
},
```



# BABEL-LOADER

babel-loader понимает конфиг Babel, поэтому настройка осуществляется аналогично тому, как мы это делали в предыдущей лекции.

# СБОРКА

Выполним сборку:

```
$ npm run build
```

Удостоверимся, что появился каталог `dist`, в котором находится минимизированный файл `main.js`.

# ОСНОВНЫЕ КОНЦЕПЦИИ

- **Бандл** (Bundle)

Бандлы состоят из некоторого количества модулей и содержат итоговые версии исходных файлов, которые уже прошли компиляцию.

- **граф зависимостей** (Dependency graph)

Когда Webpack обрабатывает JavaScript-приложение, он строит внутренний граф зависимостей, который сопоставляет каждый модуль приложения и генерирует один или несколько бандлов.

- **точка входа в приложение** (Entry point)

Точка входа указывает, какой модуль webpack должен использовать, чтобы начать строить свой граф зависимостей. webpack выясняет, от каких модулей и библиотек зависит эта точка входа (напрямую и через зависимости его зависимостей). По умолчанию точкой входа считается файл `./src/index.js`, но можно указать другой (или несколько других) в конфигурационном файле webpack.

`webpack.config.js`:

```
1 | module.exports = {  
2 |   entry: './path/to/my/entry/file.js',  
3 | }
```

# МОДУЛЬ В WEBPACK

Под модулем в Webpack понимаются не только файлы с расширением .js, .mjs, .cjs или .json, а всё, что импортируется с использованием синтаксиса `import` / `require`.

В том числе модулями Webpack будут являться изображения, css-файлы и другие типы файлов, если произведены соответствующие настройки Webpack.





# ENTRY POINT

В рамках курса мы не рекомендуем в `index.js` размещать какую-то логику.

Мы используем этот файл только в качестве входной точки Webpack (для подключения зависимостей).

---

# WEBPACK

Webpack собирает модули и их зависимости в выходной файл (для простоты пока будем считать, что это один файл в формате .js).

Webpack понимает, что css, картинки и другие файлы тоже являются зависимостями, поэтому также собирает их в бандл. Если настроить сборку соответствующим образом, то он сможет .css и изображения встраивать не в .js, а выносить в отдельные файлы, сжимать и т.д.

Таким образом, ключевая задача Webpack'a - собрать всё наше дерево зависимостей, чтобы получить бандл, готовый для развёртывания.

# ВЫХОДНОЙ ФАЙЛ (OUTPUT)

Свойство `output` указывает Webpack, куда выводить создаваемые бандлы и как их назвать. Название по умолчанию для главного выходного файла `./dist/main.js`.

`webpack.config.js`:

```
1  const path = require('path'); // Node.js модуль для разрешения путей файлов
2
3  module.exports = {
4    entry: './src/index.js',
5    output: {
6      path: path.resolve(__dirname, 'dist'),
7      filename: 'app.bundle.js',
8    },
9  };
```

Свойства `output.filename` и `output.path` указывают имя выходного файла бандла, и куда его сохранить.

## ЗАГРУЗЧИКИ (LOADERS)

По умолчанию Webpack понимает как работать только с JavaScript и JSON.

Loader'ы используются для добавления поддержки bundling'a других типов файлов, например: .css, .png, .txt.

Список наиболее популярных loader'ов: <https://webpack.js.org/loaders>

# ЗАГРУЗЧИКИ (LOADERS)

Два основных свойства для настройки загрузчика:

- **test** - какие типы файлов должны быть обработаны Webpack
- **use** - какой загрузчик(и) нужно использовать для загрузки файлов указанного типа.

`webpack.config.js`:

```
1 // используем CJS, т.к. webpack запускается Node.js
2 const path = require('path');
3
4 module.exports = {
5   module: {
6     rules: [
7       {
8         test: /\.txt$/, // маска для имени файла
9         use: 'raw-loader' // какой загрузчик использовать
10      },
11     ],
12   },
13 };
```

Когда в каком-то JavaScript-файле встретится `import (require)` файла `txt`, то будет использоваться загрузчик `raw-loader` для его обработки перед добавлением в бандл (`raw-loader` выдаст содержимое `.txt`-файла как строку).

---

## ЗАГРУЗЧИКИ (LOADERS)

Поскольку большинство загрузчиков поставляются в виде отдельных пакетов, для большинства из них необходимо использовать `npm install` (дetailedнее читайте в документации на конкретный loader).



## ПЛАГИНЫ (PLUGINS)

Loader'ы используются для загрузки модулей.

Для других операций (например, оптимизация, управление ресурсами, минимизация, mangling) Webpack предлагает концепцию плагинов.

# ПЛАГИНЫ (PLUGINS)

webpack.config.js:

```
1 | const HtmlWebpackPlugin = require('html-webpack-plugin'); // устанавливается через npm
2 | // const webpack = require('webpack'); // для получения доступа ко встроенным плагинам
```



# HTML PLUGIN

Для сборки HTML-файлов нужен отдельный плагин: HTML Webpack Plugin.

Он позволит по шаблону генерировать выходной файл и в него встраивать ссылки на .js (и другие файлы).

Для установки выполним следующую команду:

```
$ npm install --save-dev html-webpack-plugin
```

---

# HTML PLUGIN

```
1 | plugins: [  
2 |   new HtmlWebpackPlugin({  
3 |     template: "./src/index.html",  
4 |     filename: "./index.html"  
5 |   })  
6 | ]
```

# CSS PLUGIN

Для поддержки CSS нам нужны ладеры и плагин MiniCSSExtractPlugin:

```
$ npm install --save-dev mini-css-extract-plugin css-loader
```

Плагин MiniCSSExtractPlugin содержит в своём составе ещё и loader.

# MINI CSS EXTRACT PLUGIN

```
1  module.exports = {  
2    module: {  
3      rules: [  
4        ...  
5      ],  
6      test: /\.css$/,  
7      use: [  
8        MiniCssExtractPlugin.loader, 'css-loader',  
9      ],  
10     },  
11   ],  
12 },  
13 plugins: [  
14   new MiniCssExtractPlugin({  
15     filename: '[name].css'  
16   }),  
17 ],
```

# ПОДКЛЮЧЕНИЕ CSS

Теперь мы можем подключать css в entry point (для этого мы и использовали плагины и loader'ы):

```
1 | import '../css/styles.css';
```

Подключение в html будет автоматически выполнено за нас.

# КАРТИНКИ

Для обработки ссылок в html-файлах (``) и css-файлах (`background: url(../img/bg.png)`) необходимо будет установить соответствующие ладеры.

Вы это проделаете самостоятельно в рамках домашней работы.

## WEBPACK. MODE (РЕЖИМ)

Можно включить встроенную оптимизацию Webpack для конкретного окружения: **development** (разработка), **production** (продакшн) или **none** (не установлен)

`webpack.config.js:`

```
1 | module.exports = {  
2 |   mode: 'production',  
3 | };
```

Можно также передавать в качестве флага в командной строке.



# WEBRACK. СОВМЕСТИМОСТЬ С БРАУЗЕРАМИ

Webpack поддерживает все браузеры, совместимые с ES5 (IE8 и ниже не поддерживаются).





# WEBPACK DEV SERVER

После всех проделанных манипуляций, Live Server (который мы настраивали на предыдущей лекции) нам уже не особо поможет в разработке.

Но решение есть - [Webpack Dev Server](#).

# УСТАНОВКА

Для установки выполним следующую команду:

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

И в скриптах заменим `start`:

```
"scripts": {  
  "start": "webpack-dev-server --mode development",  
  "lint": "eslint .",  
  "build": "webpack --mode production"  
},
```

# ЗАПУСК

Запустим Dev Server и удостоверимся, что Live Reload работает при изменении файлов:

```
$ npm start
```

```
i [wds]: Project is running at http://localhost:8080/
i [wds]: webpack output is served from /
i [wdm]: Hash: 582f4199a90ed6e26bf1
Version: webpack 4.28.3
Time: 941ms
Built at: 2019-01-02 14:54:00
   Asset      Size  Chunks             Chunk Names
./index.html  285 bytes                [emitted]
  main.js    420 KiB             main [emitted] main
```

# APP.JS

В итоге Webpack вполне нормально будет обрабатывать синтаксис ESM без указания путей и расширений:

```
import * as logs from './logs';  
import * as stats from './stats';  
import moment from 'moment';  
  
console.log(moment.now());
```

Именно такой синтаксис вы чаще всего будете встречать (в том числе в различных статьях).



# WEBPACK MERGE

На практике часто разделяют конфигурации для production mode и development mode.

При этом общую часть выносят в отдельный конфигурационный файл.

Для сбора итоговой конфигурации используют дополнительный инструмент - [Webpack Merge](#).



# ИТОГИ

# ЧЕМУ МЫ НАУЧИЛИСЬ

- использовать системы ESM и CommonJS
- использовать экспорт: `export`, `export default`
- использовать импорт: `import`, смешанный импорт, `import as`, `import * as`
- основным концепциям Webpack и его настройке через конфигурационный файл



## ВАЖНО

Начиная с сегодняшнего дня во всех домашних заданиях мы будем требовать от вас: использования Webpack для сборки ваших проектов.



# ССЫЛКИ НА ДОПОЛНИТЕЛЬНЫЕ МАТЕРИАЛЫ

1. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import>
2. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export>
3. [Node.js Modules](#)
4. <https://webpack.js.org/>
5. [Концепции Webpack](#)
6. [Документация Webpack Dev Server](#)



Спасибо за внимание!!! Жду ваших вопросов 😊

**МИХАИЛ КУЗНЕЦОВ**

