



МОДУЛИ



ИЛЬЯ МЕДЖИДОВ / RAGEMARKET



ИЛЬЯ МЕДЖИДОВ

CEO в RageMarket




medzhidov@ragemarket.ru



ПЛАН ЗАНЯТИЯ

1. [Зачем нужны ES-модули?](#)
2. [Экспорт: export, export default](#)
3. [Импорт: import, смешанный импорт, import as, import * as](#)
4. [CommonJS](#)
5. [Webpack и его конфигурационный файл](#)



ЗАЧЕМ НУЖНЫ ES- МОДУЛИ?

ЗАЧЕМ НУЖНЫ ES-МОДУЛИ?

- сложность JavaScript-приложений очень сильно выросла с 2006 г., когда появился jQuery: появилась сложная бизнес-логика на клиенте и необходимость управлять большим количеством данных, и как следствие — файлами различных типов
- если раньше было достаточно вручную при помощи тега script подключить несколько скриптов на HTML-страницу и легко следить за их зависимостями, то современное приложение при таком подходе невозможно поддерживать и расширять



УСТАРЕВШИЙ ПОДХОД К МОДУЛЯМ

- паттерн проектирования «модуль» (может быть немедленно вызываемой функцией или функцией-конструктором)
- jQuery-плагины
- JavaScript в стиле ООП



ПРОБЛЕМЫ ПРИ УСТАРЕВШЕМ ПОДХОДЕ

- конфликты имен модулей в глобальной области видимости
- необходимо вручную следить за последовательностью загрузки и инициализации модулей с учетом зависимостей между ними (для каждой HTML-страницы)

ПОДХОДЫ К ОРГАНИЗАЦИИ МОДУЛЕЙ

- AMD (Asynchronous Module Definition) – одна из первых систем организации модулей
- CommonJS – система модулей, используемая Node.js

```
1  const students = require('./students.js');
2
3  console.log(students.getGroups());
4
5  // в students.js
6  exports.getGroups = () => {
7    // ...
8  };
```

- UMD (Universal Module Definition) – система модулей, совместимая с системой AMD и CommonJS.



КАК ПРАВИЛЬНО РАЗБИВАТЬ КОД НА МОДУЛИ?

Модуль — это файл с кодом, из которого экспортируется хотя бы одна переменная (иначе его нельзя переиспользовать).

Один класс, компонент или библиотека — один модуль.



**ЭКСПОРТ: EXPORT,
EXPORT DEFAULT**

ИМЕНОВАННЫЙ ЭКСПОРТ

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

```
export const getStudentsByGroup = (students, group) =>
  students.filter(student => student.group === group);
```

В случае экспорта заранее объявленной переменной/функции/класса необходимо взять их название в фигурные скобки:

```
1  const getStudentsByGroup = (students, group) =>
2    students.filter(student => student.group === group);
3
4  export { getStudentsByGroup }; // необходимо взять в фигурные скобки
```

Можно экспортировать сразу несколько переменных:

```
export { getStudentsByGroup, getGroupsByCourse };
```

ЭКСПОРТ ПО УМОЛЧАНИЮ

С использованием ключевого слова **default**, в одном файле может быть только один дефолтный экспорт:

```
const courseUtils = { ... };  
export default courseUtils;
```

Можно экспортировать по умолчанию и новый объект:

```
1 export default {  
2   getStudentsByGroup, // короткая запись ES6 для getStudentsByGroup: getStudentsByGroup  
3   getGroupsByCourse,  
4 };  
◀
```

ПЕРЕИМЕНОВАНИЕ ПРИ ЭКСПОРТЕ (EXPORT AS)

При помощи ключевого слова **as** можно экспортировать переменную/функцию/класс под другим именем:

```
export { getStudentsByGroup as getStudents, getGroupsByCourse as getGroups };
```

**ИМПОРТ: IMPORT,
СМЕШАННЫЙ ИМПОРТ,
IMPORT AS, IMPORT * AS**

ИМЕНОВАННЫЙ ИМПОРТ

```
1 // модуль courseUtils .js
2 const courseUtils = {...};
3
4 export const getStudentsByGroup = (students, group) =>
5   students.filter(student => student.group === group);
6
7 export const getGroupsByCourse = (groups, course) =>
8   students.filter(group => group.course === course);
9
10 export default courseUtils;
```

Импорт одной или нескольких переменных, функций или классов по имени:

```
import { getStudentsByGroup, getGroupsByCourse } from './courseUtils.js';
```

Имя при экспорте и при импорте в данном случае должно совпадать.

ESLINT

ESLint с настройками Airbnb будет "ругаться" на расширение модуля при импорте. Поэтому (если вы используете бандлер - Webpack), расширение указывать не нужно.

Но браузер (без бандлера) синтаксис без расширения (.js или .mjs) принимать не будет.

Настройка ESLint:

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

Детали [по ссылке](#).

.MJS

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

Браузеру и Webpack'у 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`

ИМПОРТ ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ

```
1 // модуль courseUtils.js
2 const courseUtils = {...};
3
4 export const getStudentsByGroup = (students, group) =>
5     students.filter(student => student.group === group);
6
7 export const getGroupsByCourse = (groups, course) =>
8     students.filter(group => group.course === course);
9
10 export default courseUtils;
```

```
import courseUtils from './courseUtils.js';
```

имя может быть и другим, не обязательно courseUtils

СМЕШАННЫЙ ИМПОРТ

Импорт по умолчанию и по имени в одной строке

```
1 import courseUtils, { getStudentsByGroup, getGroupsByCourse } from './courseUtils.js';  
2 // вместо:  
3 import courseUtils from './courseUtils.js';  
4 import { getStudentsByGroup, getGroupsByCourse } from './courseUtils.js';
```

ИМПОРТ ВСЕГО СОДЕРЖИМОГО МОДУЛЯ

`import * as`, при этом необходимо дать модулю имя

```
import * as utils from './courseUtils.js';
```

Обращаться к конкретным переменным через `utils`:

```
const { getStudentsByGroup, getGroupsByCourse, courseUtils } = utils;
```

ИМПОРТ С ПЕРЕИМЕНОВАНИЕМ (IMPORT AS)

Если после импорта переменной хочется обращаться к ней по имени, отличающимся от того, с которым ее экспортировали:

```
import { getStudentsByGroup as getStudents, getGroupsByCourse as getGroups } from './courseUtils.js';
```

КАК В ДРУГОМ МОДУЛЕ ИМПОРТИРОВАТЬ ОБЪЕКТ COURSEUTILS?

```
1  // модуль courseUtils.js
2  const courseUtils = {...};
3
4  export const sortStudentsByMark = students => (
5    students.sort((a, b) => {
6      if (a.mark > b.mark) {
7        return 1;
8      }
9      if (a.mark < b.mark) {
10       return -1;
11     }
12     return 0;
13   })
14 );
15
16 export const getBestStudent = students => sortStudentsByMark(students)[0];
17
18 export default courseUtils;
```

ВАРИАНТЫ:

1. `import { courseUtils } from './courseUtils.js';`
2. `import courseUtils from './courseUtils.js';`
3. `import * from './courseUtils.js';`
4. `import { default as courseUtils } from './courseUtils.js';`

КАК ИМПОРТИРОВАТЬ ФУНКЦИЮ SORTSTUDENTSBYMARK ПОД ИМЕНЕМ SORTSTUDENTS?

```
1  // модуль courseUtils.js
2  const courseUtils = {...};
3
4  export const sortStudentsByMark = students => (
5    students.sort((a, b) => {
6      if (a.mark > b.mark) {
7        return 1;
8      }
9      if (a.mark < b.mark) {
10       return -1;
11     }
12     return 0;
13   })
14 );
15
16 export const getBestStudent = students => sortStudentsByMark(students)[0];
17
18 export default courseUtils;
```


ВАРИАНТЫ:

1. `import * as sortStudents from './courseUtils.js';`
2. `import sortStudentsByMark as sortStudents from './courseUtils.js';`
3. `import { sortStudentsByMark as sortStudents } from './courseUtils.js';`

КАК ОБРАТИТЬСЯ К ФУНКЦИИ GETBESTSTUDENT?

```
1  // модуль courseUtils.js
2  const courseUtils = {...};
3
4  export const sortStudentsByMark = students => (
5    students.sort((a, b) => {
6      if (a.mark > b.mark) {
7        return 1;
8      }
9      if (a.mark < b.mark) {
10       return -1;
11      }
12     return 0;
13   })
14 );
15
16 export const getBestStudent = students => sortStudentsByMark(students)[0];
17
18 export default courseUtils;
```

```
import * as utils from './courseUtils.js';
```

ВАРИАНТЫ:

1. `utils.getBestStudent()`
2. `getBestStudent()`
3. нельзя к ней обратиться

КАК ИСПОЛЬЗОВАТЬ МОДУЛИ СЕЙЧАС?

Поддержка модулей присутствует в свежих версиях популярных браузеров:

```
1 <script type="module">
2   import { groups } from './jsCourse.js';
3   console.log(groups);
4 </script>
```

Или вот так можем сказать браузеру, что в подгружаемом скрипте используются модули:

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

Для более полной поддержки браузерами рекомендуется использовать транспайлеры, такие как Babel, а также сборщики, например, Webpack.



COMMONJS

MODULE.EXPORTS

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

```
1 module.exports = {  
2   variable: { ... },  
3   method: function() { ... },  
4 };
```

REQUIRE

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

```
1  const mod = require('<path_to_module>');  
2  // mod.variable - доступ к конкретному имени  
3  // mod.method() - доступ к конкретному имени
```



ESM VS COMMONJS



ЧТО ИСПОЛЬЗОВАТЬ?

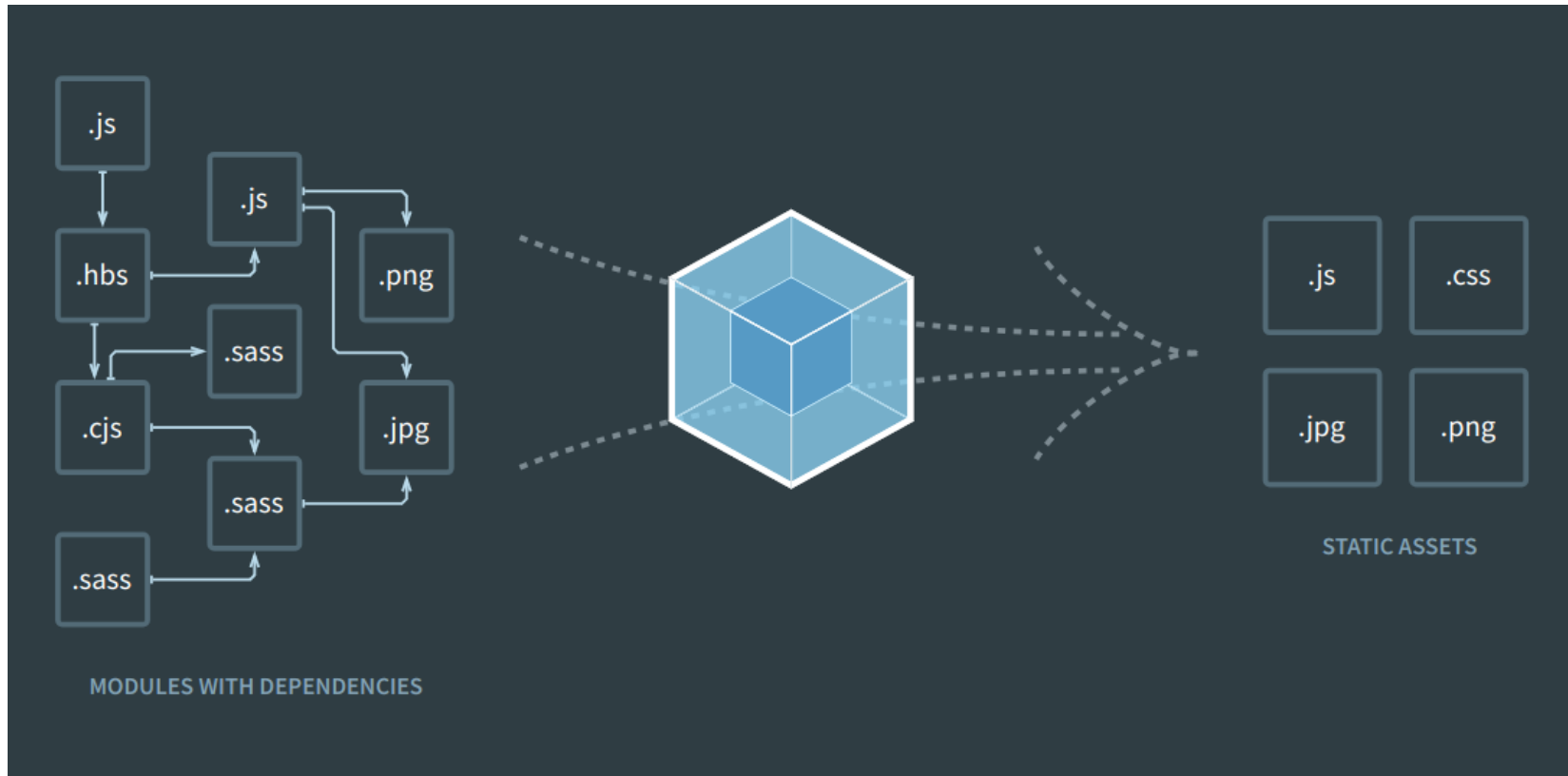
Придётся использовать обе, т.к. платформа Node.js поддерживает ES Modules в экспериментальном режиме.

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



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"  
},
```

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

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 дипломной 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 сделанных 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



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!
Время задавать вопросы 😊

ИЛЬЯ МЕДЖИДОВ

✉ medzhidov@ragemarket.ru