Лекция 15
Основы Node.js (продолжение)

1. Экспорт с параметрами и экспорт по умолчанию

Понимание того, как экспортировать значения из модуля, имеет важное значение при работе с **ES-модулями** в Node.js. ES-модули используют ключевые слова export и export default для того, чтобы сделать переменные, функции или классы доступными для других файлов.

Именованные экспорты позволяют экспортировать несколько значений из модуля по имени. Каждое экспортируемое значение должно быть явно указано, и при импорте необходимо использовать точные экспортируемые имена.

В отличие от этого, **экспорт по умолчанию** позволяет экспортировать одно значение в качестве основного экспортируемого объекта из модуля. При импорте экспорта по умолчанию вы можете использовать любое имя, которое выберете для импортируемого значения.

Пример 1 - mathUtils.js

```
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// Default export
const multiply = (a, b) => a * b;
export default multiply;
```

ES Modules (ESM) - это современная система модулей JavaScript. Она позволяет разделять большую программу на несколько файлов и передавать между ними функции, переменные и классы.

Один файл что-то **экспортирует**, а другой файл это **импортирует** и использует.

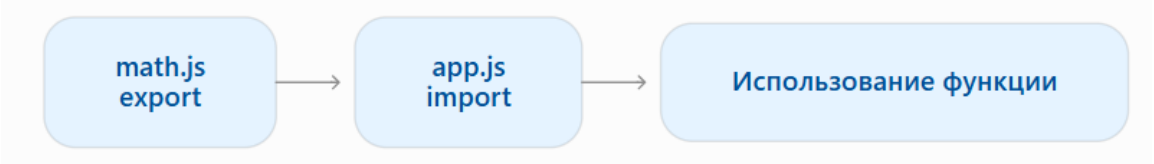
Пример 2. Создайте папку и два файла:

```
Обычный текст

my-project/
├── math.js
└── app.js
```

Файл math.js экспортирует функцию:	Файл app.js импортирует и вызывает эту функцию:	Запустите в терминале:
`export function add(a, b) { return a + b; }`	`import { add } from "./math.js"; const result = add(5, 3); console.log(result);`	`node app.js`

Общая схема:



Для нового проекта рекомендуется изучать ES Modules, но require() тоже нужно понимать, поскольку CommonJS часто встречается в существующих Node.js-проектах.

2. Импорт с помощью функции import

При использовании модулей ES в Node.js вы используете importоператор `input` для импорта функций, объектов или значений из других модулей. Этот importоператор поддерживает несколько вариантов в зависимости от того, как модуль экспортирует свое содержимое.

Пример 1.

mathUtils.js

```
export function add(a, b) {
  return a + b;
}

export function subtract(a, b) {
  return a - b;
}

const PI = 3.14159;
export default PI;
```

main.js

```
import PI, { add, subtract } from './mathUtils.js';

console.log(add(2, 3));         // 5
console.log(subtract(5, 2));    // 3
console.log(PI);                // 3.14159
```

- Если модуль использует export default, вы можете импортировать экспорт по умолчанию, присвоив ему любое выбранное вами имя, как показано в примере import PI from './mathUtils.js';
- Для именованных экспортов используются фигурные скобки, и указываются точные имена, например import { add, subtract } from './mathUtils.js' ;
- Вы также можете объединить обе формы в одном операторе: import PI, { add, subtract } from './mathUtils.js'. Это называется **деструктуризацией импорта** и позволяет выбрать из модуля только необходимые экспортируемые элементы.

Этот import оператор всегда должен находиться на верхнем уровне файла и не может использоваться условно или внутри функций. Такой синтаксис помогает писать модульный, поддерживаемый код, в котором зависимости четко перечислены в начале каждого модуля.

3. Совмещение CommonJS и ES Modules

При работе над проектами на Node.js вы можете столкнуться как с CommonJS, так и с ES Modules в одном и том же коде. Изначально Node.js использовал CommonJS (require/ module.exports), но теперь также поддерживает ES Modules (import/ export). Совмещение этих двух систем может быть сложным из-за различий в синтаксисе, расширениях файлов и поведении загрузки. Проблемы совместимости возникают, поскольку не все функции доступны при импорте одного типа модуля в другой. Вам следует понимать стратегии совместного использования обеих систем, а также ограничения, с которыми вы можете столкнуться.

Одна из распространенных стратегий — выбрать основную модульную систему для вашего проекта и использовать функции совместимости только при необходимости. Например, вы можете хранить большую часть кода на CommonJS для совместимости со старыми инструментами, но использовать ES Modules для более нового кода или когда вам нужны такие функции, как import.meta.url.

Node.js позволяет указывать тип модуля в package.json файле с помощью "type" поля "commonjs" или "module". Файлы с .cjs всегда обрабатываются как CommonJS, а .mjs файлы всегда являются ES Modules, независимо от "type" поля. Это дает вам гибкость в смешивании типов модулей в рамках проекта, но вы должны помнить об ограничениях и различиях в способах загрузки и выполнения модулей.

Пример 1.

math.cjs

```
module.exports.add = (a, b) => a + b;
```

app.mjs

```
module.exports.add = (a, b) => a + b;
```

В приведенном примере кода показано, как можно использовать модуль CommonJS внутри модуля ES в Node.js. В math.cjs файле используется синтаксис CommonJS, экспортируется add функция с помощью module.exports.add = (a, b) => a + b; . Эта функция складывает два числа и возвращает результат.

Этот app.mjs файл является модулем ES. Модули ES не могут использовать его require напрямую, поэтому вы импортируете createRequire функцию из встроенного module пакета.

Вызвав функцию createRequire(import.meta.url), вы создаете require функцию, которая работает в контексте текущего файла модуля ES. Затем вы используете это require для импорта модуля CommonJS (math.cjs).

При вызове функции math.add(2, 3)код выводит результат 5 в консоль. Такой подход позволяет смешивать CommonJS и ES Modules в одном проекте, используя преимущества каждой системы там, где это необходимо.

4. Работа с модулем fs

Модуль fs (File System) в Node.js позволяет работать с файлами на вашем компьютере. Вы можете использовать его для чтения, записи, удаления или изменения файлов.

При работе с файлами часто приходится выбирать между **синхронными** и **асинхронными** методами.

Пример 1 – index.js

```
// Reading a file synchronously
const fs = require('fs');
const data = fs.readFileSync('example.txt', 'utf8');
console.log('Synchronous read:', data);

// Reading a file asynchronously
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error reading file:', err);
```

```
    return;
  }
  console.log('Asynchronous read:', data);
});
```

- Синхронные операции с файлами, такие как чтение или запись fs.readFileSync, блокируют выполнение вашей программы до завершения операции с файлом. Это означает, что никакой другой код не выполняется, пока Node.js читает или записывает файл, что может замедлить работу вашего приложения, если файл большой или если вы выполняете много таких операций;
- Асинхронные операции с файлами, такие как `<filename>` fs.readFile, не блокируют программу. Вместо этого вы предоставляете функцию обратного вызова, которая запускается после завершения операции. Это позволяет Node.js продолжать выполнение другого кода, пока он ожидает завершения работы файловой системы, что делает ваши приложения более эффективными и отзывчивыми, особенно при работе с большим количеством файлов или медленными дисками.

5. Работа с путями с помощью модуля path

При работе с файлами и каталогами в Node.js крайне важно правильно обрабатывать пути к файлам. Разные операционные системы используют разные форматы путей, поэтому вам необходим надежный способ построения, разрешения и извлечения информации из путей к файлам.

Модуль path — это встроенная утилита Node.js, которая помогает работать с путями к файлам на разных платформах. Используя методы, предоставляемые модулем path, вы избегаете скрытых ошибок и делаете свой код переносимым между Windows, macOS и Linux. Модуль path не взаимодействует с самой файловой системой, а помогает безопасно и согласованно форматировать и анализировать строковые пути.

Пример 1 – index.js

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

// Join several path segments into one normalized path
const projectDir = '/Users/alex/projects';
const fileName = 'app.js';
const fullPath = path.join(projectDir, 'src', fileName);
// fullPath: '/Users/alex/projects/src/app.js'

// Resolve an absolute path from segments
const resolvedPath = path.resolve('src', 'utils', 'helper.js');
// resolvedPath: '/absolute/path/to/current/dir/src/utils/helper.js'

// Get the base name (file name) from a path
const base = path.basename(fullPath);
// base: 'app.js'

console.log('Joined path:', fullPath);
```

```
console.log('Resolved path:', resolvedPath);
console.log('Base name:', base);
```

Приведённый пример кода демонстрирует, как использовать path модуль для обработки путей к файлам таким образом, чтобы это работало в любой операционной системе.
Сначала необходимо подключить path модуль, а затем использовать его path.join для объединения сегментов имен каталогов и файлов в единую нормализованную строку пути. Это гарантирует использование правильных разделителей пути независимо от того, используете ли вы системы Windows или Unix.

Далее, path.resolve эта функция используется для преобразования ряда относительных сегментов пути в абсолютный путь, основанный на текущем рабочем каталоге. Это полезно, когда вам нужно надежно ссылаться на файлы независимо от того, где выполняется ваш скрипт.

Наконец, path.basename функция извлекает только имя файла из полного пути, что полезно, когда вам нужно только имя файла, а не полный путь.
Использование этих методов делает ваш код более надежным и переносимым в различных средах.