Лекция 13
Основы Node.js

Содержание курса. Узнайте, как Node.js позволяет использовать JavaScript на серверной стороне и служит основой современной веб-разработки. Изучите архитектуру Node.js, цикл событий (Event Loop) и асинхронное выполнение кода. Затем научитесь запускать реальные программы и управлять модулями с помощью CommonJS и импорта ES Modules.

Получите прочную базу для создания масштабируемых и высокопроизводительных приложений на Node.js

1. Понимание Node.js и его основных концепций

Изучите, как Node.js позволяет запускать JavaScript вне браузера. Разберитесь в различиях между Node и клиентским JavaScript, изучите движок V8 и узнайте, как цикл событий обеспечивает неблокирующую модель выполнения Node.

1.1 Что такое Node.js?

Node.js — это **среда выполнения JavaScript,** построенная на основе движка V8 JavaScript от Chrome. В отличие от традиционных сред JavaScript, работающих в веб-браузерах, Node.js позволяет выполнять код JavaScript непосредственно на вашем компьютере или сервере, вне браузера. Это означает, что вы можете использовать JavaScript для создания приложений, работающих на стороне сервера, таких как веб-серверы, инструменты командной строки, API и даже настольные приложения. **Node.js** особенно популярен для создания масштабируемых сетевых приложений, поскольку он использует **неблокирующую, событийно-ориентированную архитектуру** , которая может обрабатывать множество соединений одновременно без замедления работы.

Node.js обычно используется для:

- Разработка веб-серверов и серверных приложений;
- Создание RESTful API для веб- и мобильных приложений;
- Разработка приложений реального времени, таких как чат-приложения и онлайн-игры;
- Разработка утилит командной строки и скриптов автоматизации;
- Прототипирование и быстрая разработка приложений.

Определение

«**Среда выполнения» ("runtime environment")** — это платформа, предоставляющая всё необходимое для выполнения кода, написанного на определённом языке. **Node.js** считается средой выполнения, поскольку он предоставляет движок (V8), библиотеки и системные интерфейсы, необходимые для запуска программ JavaScript вне браузера.

1.2 Node.js против JavaScript в браузере

При написании кода на JavaScript среда, в которой он выполняется, существенно влияет на то, что вы можете делать и как ваш код взаимодействует с внешним миром. Две наиболее распространенные среды для JavaScript — это Node.js и веб-браузеры. Понимание различий между этими средами поможет вам выбрать правильные инструменты и подходы для ваших проектов.

Ключевые различия между Node.js и JavaScript в браузере

Среда:

- Node.js выполняет JavaScript на сервере, вне контекста браузера;
- Браузеры выполняют JavaScript на стороне клиента, непосредственно внутри веб-страницы пользователя.

API и встроенные модули:

- Node.js включает в себя API для работы с файловой системой, сетью и запуском процессов;
- Браузеры предоставляют API для управления веб-страницей (DOM), обработки пользовательских событий и взаимодействия с веб-пространством.

Типичные области применения:

- Node.js используется для создания серверных приложений, инструментов командной строки и скриптов;
- JavaScript в браузерах используется для создания интерактивных веб-страниц и пользовательских интерфейсов.

Безопасность и разрешения:

- Скрипты Node.js по умолчанию могут получать доступ к локальной файловой системе и сети;
- JavaScript в браузере работает в изолированной среде в целях безопасности, ограничивая доступ к устройству пользователя.

Глобальные объекты:

- Node.js предоставляет глобальные объекты, такие как global, process, и Buffer;
- Браузеры предоставляют такие объекты, как window, document, и navigator.

Эти различия означают, что код, написанный для Node.js, часто не может работать в браузере без изменений, и наоборот.

Некоторые API уникальны для **Node.js**, например, fs модуль для операций с файловой системой, http для создания серверов и process для взаимодействия с запущенным процессом. В отличие от них, браузеры предлагают API, такие как **объектная модель документа (DOM),** для манипулирования HTML, window управления окнами браузера и

для сетевых запросов. Изучение документации по API **Node.js** и браузеров поможет вам понять, какие функции доступны в каждой среде.

1.3 Движок V8.

Чтобы понять, как Node.js так эффективно выполняет код JavaScript, необходимо изучить **движок JavaScript V8** . **V8** — это движок JavaScript с открытым исходным кодом, разработанный Google, первоначально созданный для веб-браузера Google Chrome. Со временем он стал основным компонентом, позволяющим Node.js запускать JavaScript вне браузерной среды.

V8 берет код JavaScript и преобразует его в машинный код, который процессор вашего компьютера может выполнить напрямую. Именно этот процесс обеспечивает Node.js впечатляющую производительность при выполнении серверного JavaScript. В отличие от более старых движков, которые интерпретировали JavaScript построчно, V8 использует современные методы для оптимизации и ускорения выполнения кода.

Node.js работает на основе знаменитого движка V8 от Google. Однако не стоит путать это с термином «движок» — речь идёт не об автомобилях.

Этот исключительный движок компилирует код JavaScript в высокоэффективный машинный код, обеспечивая скорость и отзывчивость, характерные для приложений Node.js.

Способность V8 динамически компилировать и оптимизировать JavaScript является краеугольным камнем скорости и эффективности Node.js. Поскольку V8 был разработан для высокой производительности в браузерах, Node.js получает выгоду от тех же функций быстрого выполнения и управления памятью. Именно поэтому Node.js может обрабатывать крупномасштабные приложения с высокой пропускной способностью, использующие JavaScript на сервере.

Примечание. **Машинный код** — это язык, который компьютеры понимают напрямую. Он состоит из двоичных инструкций, которые указывают центральному процессору компьютера, что нужно делать. В контексте Node.js движок V8 компилирует код JavaScript в машинный код, что делает его более быстрым и эффективным для выполнения компьютером.

Определение. Компиляция «на лету» (JIT) — это метод, при котором код компилируется в машинный код во время выполнения, непосредственно перед его запуском. В контексте **V8** JIT-компиляция означает, что JavaScript не только интерпретируется, но и компилируется на лету, что позволяет оптимизировать часто используемые участки кода для повышения скорости. Это делает **V8** , а следовательно, и **Node.js** , намного быстрее традиционных интерпретаторов.

1.4 Цикл событий (The Event Loop)

Node.js разработан для эффективного одновременного выполнения множества задач, таких как обработка веб-запросов, чтение файлов или взаимодействие с базами данных. Основной механизм, обеспечивающий эту эффективность, называется **циклом событий (event loop)**.

Цикл событий - это фундаментальная часть Node.js, позволяющая выполнять **неблокирующие операции (non-blocking operations)**, то есть Node.js может начать задачу, перейти к следующей и вернуться к завершению первой задачи, когда она будет готова, не останавливая основной поток.

При выполнении кода в Node.js цикл событий постоянно работает в фоновом режиме. Он постоянно проверяет наличие новых задач, завершенных операций и событий, требующих обработки. Например, если вы отправляете запрос на чтение файла, Node.js передаст эту задачу системе и продолжит выполнение остального кода. Когда файл будет готов, цикл событий это заметит и сообщит вашему коду, чтобы вы могли обработать его содержимое.

Эта архитектура отличается от традиционного блокирующего кода, где каждая операция должна завершиться до начала следующей. Вместо этого цикл событий позволяет Node.js управлять тысячами операций одновременно, что делает его идеальным для серверов и приложений, которым необходимо обрабатывать множество пользователей или задач одновременно.

Примечание. Разница между синхронным и асинхронным выполнением кода. Синхронный код выполняет задачи одну за другой, ожидая завершения каждой из них, прежде чем перейти к следующей. Асинхронный код позволяет задачам начинаться и заканчиваться независимо, давая возможность выполняться другому коду, пока ожидается результат. Понимание этой разницы является ключом к освоению Node.js.

2. Запуск кода и работа с окружением Node.js

Запускайте программы JavaScript непосредственно на своем компьютере, используя инструменты командной строки Node. Выполняйте файлы, экспериментируйте с REPL и получайте доступ к глобальным переменным, таким как process, __dirname и __filename, чтобы составить четкое представление об среде выполнения Node.

При выполнении JavaScript-файла с помощью Node.js вы указываете Node.js прочитать файл, интерпретировать его содержимое и выполнить любой написанный в нем код. Процесс начинается, когда вы вводите команду node, а затем имя файла в терминале. Node.js загружает указанный файл, анализирует JavaScript-код и выполняет его построчно в среде Node.js. В приведенном выше примере Node.js находит оператор console.log и выводит сообщение в ваш терминал. Как только весь код в файле завершит выполнение, Node.js завершает процесс.

Как Node.js обрабатывает ошибки при выполнении скриптов?

Node.js обрабатывает ошибки во время выполнения скрипта, выводя сообщения об ошибках в консоль. Если в вашем коде возникает ошибка, Node.js описывает тип ошибки, сообщение и место её возникновения в файле. Подробнее об обработке ошибок в Node.js можно узнать в официальной документации.

Понимание __dirname и __filename

При работе с файлами и каталогами в Node.js часто возникает необходимость знать путь к файлу, который в данный момент выполняется, или к каталогу, в котором он находится. Node.js предоставляет для этого две специальные глобальные переменные: __dirnameи __filename.

__dirname - Предоставляет абсолютный путь к каталогу, содержащему текущий исполняемый файл. Это полезно, когда вы хотите создавать пути к другим файлам или каталогам относительно вашего скрипта, например, для чтения файлов конфигурации или обслуживания статических ресурсов.

__filename - Предоставляет абсолютный путь к самому файлу, включая имя файла. Вы будете использовать это, если вам нужно напрямую ссылаться на файл скрипта или регистрировать его местоположение. Оба параметра __dirname и __filename всегда разрешаются в абсолютные пути, поэтому вам не нужно беспокоиться о текущем рабочем каталоге вашего процесса.

Используйте этот параметр, __dirname когда вам нужно работать с другими файлами в том же каталоге или в известном относительном местоположении. Используйте его, __filename если вам нужен полный путь к текущему файлу скрипта, например, для ведения журнала, отладки или передачи местоположения скрипта другому процессу.

index.js

```
// Print the directory name and file name in a Node.js script
console.log("__dirname:", __dirname);
console.log("__filename:", __filename);
```

Примечание. В JavaScript, используемом в браузерах, нет переменных типа __dirname или __filename. Браузеры не предоставляют доступ к локальной файловой системе по соображениям безопасности, поэтому вы не можете напрямую получить путь к файлу скрипта или к содержащему его каталогу, как это можно сделать в Node.js.

3. Архитектура Node.js и модульная система

Погрузитесь в архитектуру, которая делает Node.js асинхронным и модульным. Изучите, как libuv управляет операциями ввода-вывода и как модульная система использует синтаксис require и ES import для эффективной организации и повторного использования кода.

3.1 libuv: Asynchronous I/O.

При работе с Node.js вы получаете преимущества от его способности обрабатывать множество задач одновременно, не дожидаясь завершения каждой из них перед началом следующей. Это возможно благодаря библиотеке **libuv** . Libuv — это ключевая часть Node.js, разработанная для обеспечения эффективного способа обработки асинхронных операций ввода-вывода (I/O). **Асинхронный ввод-вывод** означает, что Node.js может начать задачу, например, чтение файла или выполнение сетевого запроса, а затем перейти к другой работе, ожидая завершения задачи. Как только задача завершена, Node.js получает уведомление и может обработать результаты. Такой подход помогает Node.js эффективно обрабатывать множество соединений и запросов, что делает его идеальным для создания масштабируемых сетевых приложений.

Libuv написан на языке C и выступает в качестве моста между написанным вами JavaScript-кодом и базовой операционной системой. Он управляет низкоуровневыми операциями, такими как доступ к файловой системе, сетевое взаимодействие и таймеры, не блокируя основной поток. Это означает, что ваши приложения остаются отзывчивыми даже при выполнении ресурсоемких операций. Благодаря использованию libuv, Node.js может поддерживать тысячи одновременных подключений, что делает его популярным выбором для веб-серверов, API и приложений реального времени.

Libuv управляет многими асинхронными операциями в Node.js, включая:

- Чтение и запись файлов;
- Выполнение сетевых запросов (например, HTTP или TCP-соединений);
- Обработка DNS-запросов;
- Управление таймерами и интервалами.

3.2 Асинхронная модель и цикл событий

Node.js меняет правила игры благодаря своей уникальной архитектуре. В отличие от традиционных серверных сред, которые страдают от проблем с производительностью при обработке множества запросов, Node.js использует событийно-ориентированную, неблокирующую модель для одновременного управления многочисленными соединениями.

Результат? Высокомасштабируемые приложения, которые бесперебойно обрабатывают многочисленные взаимодействия и запросы пользователей.

3.3 Расширение возможностей разработчиков с помощью NPM (менеджера пакетов Node).

NPM, поставляемый в комплекте с Node.js, улучшает процесс разработки, являясь репозиторием многочисленных библиотек и инструментов с открытым исходным кодом. Эти ресурсы призваны

расширить возможности приложений Node.js, предлагая ряд готовых решений для улучшения функциональности.

NPM также упрощает сложную задачу управления версиями библиотек, освобождая разработчиков от рутинной работы и позволяя им сосредоточиться на создании исключительной логики приложения.

Node.js, благодаря своему движку V8, асинхронной архитектуре и обширной коллекции пакетов NPM, позволяет разработчикам создавать молниеносно быстрые и масштабируемые веб-приложения.

3.4 Использование «require» для импорта модулей.

3.4.1 Что такое модули?

Модули - это фундаментальная концепция в Node.js, позволяющая разбивать код на более мелкие, многократно используемые части. Модуль — это просто файл, содержащий JavaScript-код, изолированный от других файлов, если вы явно не используете общие части. Такой подход помогает организовать код, упрощая его сопровождение, отладку и повторное использование в разных проектах.

Использование модулей важно в Node.js, поскольку приложения часто становятся большими и сложными. Без модулей весь ваш код находился бы в одном файле, что затрудняло бы управление. Разделяя функциональность на отдельные модули, вы можете сосредоточиться на отдельных функциях или утилитах, избежать конфликтов имен и поощрять совместное использование кода. Модули также позволяют использовать сторонние библиотеки и делиться собственным кодом с другими.

greet.js

```
// This file exports a simple function
function greet(name) {
  return "Hello, " + name + "!";
}

module.exports = greet;
```

app.js

```
// This file imports the greet function and uses it
const greet = require('./greet');

console.log(greet("Node.js Developer"));
// Output: Hello, Node.js Developer!
```

Примечание. Модули Node.js бывают трех основных типов: основные модули, локальные модули и сторонние модули, устанавливаемые через npm.

3.4.2 Node.js использует **модульную систему CommonJS**, позволяющую организовывать код в отдельные файлы, или модули, и повторно использовать их во всем приложении.

Ядром этой системы является require функция, которая позволяет загружать как встроенные модули Node.js, так и ваши собственные пользовательские модули. При использовании require Node.js считывает указанный файл, выполняет его код и возвращает значение module.exports из этого файла. Это помогает поддерживать модульность, удобство сопровождения и упрощает тестирование кода.

Функция require работает синхронно, то есть она блокирует дальнейшее выполнение кода до тех пор, пока модуль не будет загружен. Вы можете использовать её для импорта модулей, указав либо имя встроенного модуля, либо относительный путь к пользовательскому модулю. Встроенные модули, например, fs для операций с файловой системой, поставляются в комплекте с Node.js и не требуют установки. Пользовательские модули — это файлы, которые вы создаёте в своём проекте, обычно заканчивающиеся на ".js.".

Пример 1 - index.js

```
// Requiring a built-in module
const fs = require('fs');

// Requiring a custom module (myModule.js in the same folder)
const myModule = require('./myModule');

// Using the built-in 'fs' module to read a file
const data = fs.readFileSync('example.txt', 'utf8');
console.log(data);

// Using a function from the custom module
myModule.sayHello();
```

Пример 2 - main.js

```
const greet = require('./greet');
greet('Alice');
```

greet.js

```
module.exports = function(name) {
  console.log('Hello, ' + name + '!');
};
```

В пользовательских модулях вы используете module.exports для указания того, какие функции, объекты или значения должны быть доступны при импорте модуля с помощью require. Только то, что назначено module.exports будет доступно другим файлам.

3.4.3 Как Node.js загружает модули

Для эффективной работы с модулями в Node.js необходимо понимать, как Node.js находит и загружает их. Этот процесс называется **разрешением модулей**. Когда вы используете функцию require для импорта модуля, Node.js следует определенному алгоритму для поиска запрашиваемого файла или пакета. Процесс разрешения учитывает расширения файлов, структуру папок и даже наличие специальных файлов, таких как package.json.

Node.js сначала проверяет, является ли подключаемый модуль основным модулем, например fs, или http. Если это не основной модуль, Node.js обрабатывает строку, которую вы передаете в require как относительный путь (начинающийся с ./ или ../), абсолютный путь или имя модуля. Для относительных и абсолютных путей Node.js пытается разрешить файл, проверяя следующие параметры в указанном порядке:

- Если файл с указанным именем существует;
- Если файл существует с .js добавленным расширением;
- Если файл существует с .json добавленным расширением;
- Если файл существует с .node добавленным расширением.

Если путь указывает на каталог, Node.js будет искать package.json файл внутри этого каталога. Если он найден, он использует main поле для определения того, какой файл загрузить.

Если поля нет package.json или они отсутствуют main, Node.js будет искать файл типа index.js, index.json или index.node внутри каталога.

Если вам требуется модуль по имени (а не по пути), Node.js будет искать его в node_modules папках, начиная с текущего каталога и продвигаясь вверх по дереву каталогов, пока не достигнет корневого каталога.

```
// Requiring a local module without extension
const myModule = require('./myModule');

// Requiring a local module with .js extension
const myModuleJs = require('./myModule.js');

// Requiring a directory (Node.js will look for index.js inside 'utils' folder)
const utils = require('./utils');

// Requiring a core module
const path = require('path');
```

Примечание. Node.js кэширует модули после их первой загрузки. Это означает, что если вы запрашиваете один и тот же модуль несколько раз, он выполняется только один раз, а последующие вызовы возвращают тот же кэшированный объект. Это повышает производительность и обеспечивает согласованность работы всего приложения.

3.4.4 Экспорт с помощью модуля “module.exports”

Когда вы хотите использовать одни и те же функции, объекты или значения из одного файла, чтобы другой файл мог их использовать, вы используете module.exports в Node.js. Это основа взаимодействия модулей. Вы определяете свои функции или объекты в одном файле, а затем присваиваете их module.exports. Любой файл, использующий require для загрузки этого модуля, получит то, что вы присвоили module.exports.

mathUtils.js

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

const calculator = {
  subtract(a, b) {
    return a - b;
  }
};

module.exports = {
  add,
  calculator
};
```

Начните с создания функции или объекта, который вы хотите экспортировать. В приведенном выше примере add функция и calculator объект определены в файле mathUtils.js. Присвоив объект, содержащий и функцию add, и объект, calculator файлу module.exports, вы сделаете их доступными для других файлов.

Когда другой файл, например app.js, использует require('./mathUtils'), Node.js загружает модуль и возвращает значение module.exports. Это означает, mathUtils что app.js — это объект, который вы экспортировали, поэтому вы можете вызвать mathUtils.add(5, 3) или mathUtils.calculator.subtract(10, 4).

Вы можете экспортировать одно значение, например, функцию, или объект, содержащий несколько значений. Если вы присвоите новое значение переменной module.exports, именно это значение будет возвращено require.

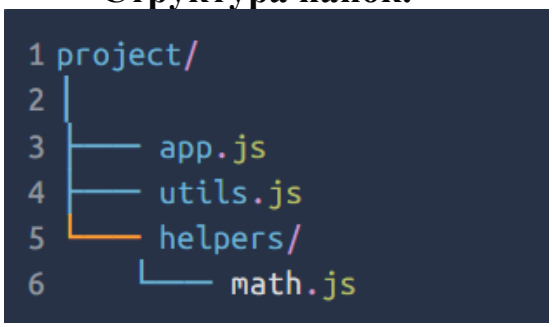
Всегда помните: Только то, что вы добавите к module.exports, будет доступно за пределами модуля.

Примечание. В Node.js `exports` — это просто сокращенная ссылка на module.exports. Вы можете использовать exports.greet = ... для добавления свойств, но если вы переназначаете module.exports напрямую, сокращенная ссылка больше не будет применяться.

3.4.5 require и относительные пути

Использование require относительных путей в Node.js позволяет точно указать Node.js, где найти ваши пользовательские модули. Всегда начинайте относительный путь с “ ./ ” для текущего каталога или “ ../ ”для перехода на один каталог выше. Это делает ваш код понятнее и позволяет избежать случайных конфликтов со встроенными модулями или установленными пакетами.

Структура папок.



utils.js

```
function logMessage(msg) {
  console.log("Log:", msg);
}

module.exports = logMessage;
```

helpers/math.js

```
const logMessage = require('../utils');

function add(a, b) {
  const result = a + b;
  logMessage(`Result: ${result}`);
  return result;
}

module.exports = add;
```

app.js

```
const add = require('./helpers/math');

add(2, 3); // Log: Result: 5
```

Следует избегать пропуска ./ или ../ при подключении собственных файлов. Если вы напишете require('utils') вместо require('./utils'), Node.js будет искать основной модуль или пакет в node_modules, а не в вашем файле. Это может привести к ошибкам, которые трудно отладить.

Поддерживайте организованную структуру проекта и используйте четкие, явные относительные пути. По мере роста проекта рассмотрите возможность использования абсолютных путей с помощью таких инструментов, как path.join или настройки псевдонимов модулей, но всегда

будьте последовательны и дважды проверяйте пути при перемещении файлов. Помните, что относительные пути определяются из файла, где require вызывается функция, а не из места запуска скрипта.

Источник:
https://codefinity.com/courses/v2/e7f47954-ae36-44cc-a69e-83034dc8fe0a/2729519a-2f68-48d5-852c-667da11e6503/1b0c5e9d-0ae8-4cc8-9836-d910ec31778c