NODE.JS

LoftSchool
от мыслителя к создателю

Содержание

1

Универсальный Javascript

Javascript давно перестал быть только web-языком. В настоящее время он широко применяется в разных областях, отличных от создания сайтов.

Это стало возможным благодаря извлечению из браузера движка, который обрабатывает и выполняет JS. Одним из таких движков является **V8** - разработанный компанией Google.

Таким образом, стало возможным выполнять любой JS-код вне браузера. Но вот другой момент - а много ли смысла в том, чтобы просто выполнять JS-код вне браузера?

В 99% случаев программы на любом языке программирования пишутся с целью взаимодействия с чем-либо, с какой-либо подсистемой: файловая система, сеть и прочее.

2

NodeJS

NodeJS является той самой прослойкой между js-кодом и системой, на которой этот код выполняется.

Другими словами: **NodeJS** позволяет получаться доступ к ресурсам система(сеть, файлы, консоль) из JS-кода.

Скачать NodeJS можно здесь **https://nodejs.org**

Состав

NodeJS состоит из JS-движка V8 (не так давно, компания Microsoft подала заявку на внедрение своего движка - Chakra), самой среды исполнения JS-кода и менеджера пакетов NPM.

Как было сказано ранее - NodeJS - это прослойка между js-движком и системой.

NPM - это менеджер пакетов/зависимостей вашего приложения (об NPM было рассказано на одном из предыдущих занятий).

Но ключевую роль здесь играет среда исполнения.

Среда исполнения

Среда исполнения - это набор правил, по которым работает ваш код.
Так же, среда исполнения предоставляет так называемый **binding mechanism**, который и является мостиком между кодом кодом и системой.

На самом деле, код не напрямую работает с системой. Работа происходит со специальной библиотекой **libuv**, которая и осуществляет взаимодействие с системой.

Но не будем в это углубляться.

Запускать код в NodeJS - очень просто. Достаточно выполнить в консоли

```
node /path/to/file.js
```

То есть выполнить команду **node** и передать ей в качестве параметра путь к файлу с исходным кодом.

3

Встроенный модуль - fs

В **NodeJS** встроен ряд модулей, упрощающих разработку приложений. Например модуль для работы с файловой системой или модуль для работы с сетью.

NodeJS использует модульную системы **CommonJS**. Таким образов, чтобы подключить один из системных модулей, необходимо написать следующий код:

```
var fs = require('fs');
```

Здесь мы подключили модуль для работы с файловой системой.

Модуль представляет из себя объект с различными методами. Поэтому, для работы с ним, мы записали его в переменную.

Методы для работы с файловой систему достаточно высокоуровневые для того, чтобы можно было написать такой код:

```
var fs = require('fs');
var content = fs.readFileSync('./someFile.txt');

console.log(content);
```

После выполнения данного кода, на экран будет выведено содержимое файла someFile.txt который находится в папке с текущим js-файлом.

Но метод **readFileSync** выполняется синхронно(это видно из названия), то есть выполнение кода не может двигаться дальше, пока **readFileSync** не выполнится(не прочтет файл).

В арсенале **NodeJS** есть метод и для асинхронного чтения файлов:

```
var fs = require('fs');

var content = fs.readFile('./someFile.txt',
   function(error, buffer) {
       var content = buffer.toString('utf8');

    console.log(content);
});
console.log('я выведусь до окончания чтения файла!');
```

Как видите, здесь подход к решению задачи немного меняется.
Вызывается метод **readFIle** которому передается имя файла и анонимная **callback-функция.**

4

Callback

Callback-функция будет выполнена только после того, как файл будет прочитан, а код, идущий после вызова readFile, будет выполняться дальше, не ожидая завершения чтения файла. Таким образом фраза *“я выведусь до окончания чтения файла”* и правда будет выведена до окончания чтения файла. Ведь как и было сказано выше: метод **readFile** выполняется асинхронно, то есть не тормозит выполнение кода, а переданная ему **callback-функция** будет вызвана только после того, как файл будет прочитан.

Эта ситуация очень похожа на поведение **setTimeout**. Эта функция тоже не тормозит выполнение кода, а лишь ставит в очередь таймер, который сработает, когда подойдет его время.

В **NodeJS** всё немного сложнее. Как говорилось ранее - среда использует библиотеку **libuv** для связи с ресурсами системы. Так вот, **readFile** лишь просит **libuv** прочитать файл. И когда от **libuv** придет ответ об окончании чтения файла - среда выполнит **callback** переданный в **readFile**.

5

Цикл событий

Как было сказано на одном из предыдущих занятий: в JS нет встроенного способа задержать выполнение на определенный промежуток времени. Но, есть способ отложить выполнение какого-либо кода при помощи **setTimeout** и **setInterval**. Но даже в этом случае ничего не происходит параллельно. Указанные функции лишь ставят код в очередь на выполнение.

В **NodeJS** похожая ситуация, но, в отличии от браузерной реализации среды исполнения, в **NodeJS** присутствует гораздо больше этапов выполнения кода.

Совокупность этих этапов называется циклом событий.

Поверхностно можно описать цикл событий так: ***Среда выполняет основной поток кода.*** Если во время выполнения были поставлены какие-либо задачи, то среда находится в ожидании срабатывания таймеров или **IO-событий**.

Если задачи закончились и не были поставлены новые, то выполнение программы завершается.

IO-события - это события связанные с вводом/выводом информации. Например: окончание чтения файла.

Цикл событий NodeJS можно разделить как минимум на 3 фазы:
- выполнение таймеров
- выполнение io-callbacks
- выполнение остальных callback

Изначально, **NodeJS** выполняет основной код.

Если во время работы основного кода были добавлены таймеры или io-задачи, то после выполнения кода, **NodeJS** перейдет в режим ожидания и будет ждать пока что-то из вышеперечисленного не выполнится.

Если сработает таймер или произойдет io-событие (например завершилось чтение файла) то **nodejs** выполнит цикл.

В отличии от браузерной реализации, где задачи можно запланировать только при помощи **setTimeout/Interval**, в **NodeJS** есть дополнительные средства:

- process.nextTick(fn)
- setImmediate()

nextTick

process.nextTick(fn) запланирует выполнение указанной функции таким образом, что указанная функция будет выполнена после окончания текущей фазы(текущего исполняемого кода), но перед началом следующей(например перед началом выполнения io-callback).

```
console.log('первый');

process.nextTick(function() {
   console.log('второй');
});

console.log('третий');
```

На экран будет выведено:

первый
третий
второй

Именно в такой последовательности. Всё потому, что **nextTick** планирует указанную функцию таким образом, чтобы она была выполнена в конце текущей фазы. То есть сначала отработает код текущей фазы, а уже потом функции, которые были запланированы через **nextTick**.

setImmediate

setImmediate(fn) запланирует выполнение указанной функции сразу после фазы "**выполнение остальных callback**".

Посмотрите на пример, где вместе собраны все сделанные ранее утверждения:

```
var fs = require('fs');

console.log('первый');

process.nextTick(function() {
   console.log('второй')
});

var content = fs.readFile('./someFile.txt',
   function(error, buffer) {
       console.log('третий');

       setImmediate(function() {
           console.log('четвертый')
       });

       process.nextTick(function() {
       console.log('пятый')
       });
});

console.log('шестой');
```

Выведена будет следующая последовательность:

первый
шестой
второй
третий
пятый
четвертый

Более подробно пример будет разобран на вебинаре.