Node.js

# 1. Node.js

Още от създаването си **JavaScript** (1995) “живее” и работи в браузърите. Това е просто контекст, който дефинира какво може да се прави с езика, но не казва какво може да направи сам по себе си езика. JavaScript е “завършвен” (completed) език: може да се използва в много контексти и да се постигне всичко с него, както може да бъде постигнато с всички други “завършени ” езици.

**Node.js** е просто един друг контекст, който позволява да се пусне JavaScript код в back-end частта на един или повече сайтове, извън браузъра.

За да се пусне JavaScript код в back-end, трябва да се интерпретира и изпълни. В Node.js това става, чрез виртуалната машина за JavaScript на Google – **V8**, която се използва и в браузъра Google Chrome.

**Node.js** е софтуерна система от страна на сървъра, проектирана за писане на мащабни интернет приложения, предимно web server-и. Програмите се пишат от страна на сървъра с JavaScript, използващи **event-driver,** **asynchronous I/O** за да се минимизира overhead-а и да се максимизира мащабността.

Node.js създава web server oт само себе си, правейки ненужно използването на web server software, като Apache или Lighttpd. Node.js позволява пълен контрол върху това, как web server-а работи. Node.js позволява web разработчиците да създават цялостни web приложения използвайки само един език (JavaScript), и за front-end-а и за back-end-а на приложението.

Node.js е пакетирана компилация от **V8 JavaScript engine**-а на Google, **libUV аbstraction layer** платформа. Създаден е от **Ryan Dahl.** Проекта е започнат от 2009 година и разработката и поддръжката е спонсорирана от фирма **Joyent**.

Първоначалната цел на Dahl е да създава сайтове със push поддръжка (**push capabilities**, **push technology**), която може да видим в Gmail например. Пробвайки различни решения в няколко различни програмни езика, той избира JavaScript, поради липсата на I/O API. Това му позволява да дефинира конвенция за **non-blocking**, **event-driven I/O.**

Пример за **HTTP Server** със Node.js във файл simple\_http\_server.js:

***var* http = require('http');**

**http.createServer(*function* (req, res) {**

**res.writeHead(200, {'Content-Type': 'text/plain'});**

**res.end('Hello World\n');**

**}).listen(1337, '127.0.0.1');**

**console.log('Server running at** [**http://127.0.0.1:1337/**](http://127.0.0.1:1337/)**');**

След което, пишем в конзолата:

**node example.js**

и сървъра е стартиран на [http://127.0.0.1:1337/.](http://127.0.0.1:1337/.Като)

Като отворим браузъра на горе дадения url, ще ни се изпише на екрана “Hello world”, както сме му задали във res.end функцията. Чрез res.writeHead функцията му задаваме какъв да бъде header-а на изпратения отговор. В този случай изпращме обикновен текст (“text/plain”);

# Пример за **TCP Server**, който слуша на порт 1337 и пише в конзолата, всичко което му се изпрати:

***var* net = require('net');**

***var* server = net.createServer(*function* (socket) {**

**socket.write('Echo server\r\n');**

**socket.pipe(socket);**

**});**

**server.listen(1337, '127.0.0.1');**

# 2. Event-driven nature

**Event-driven architecture (EDA)**

Event-driven architecture е шаблон за софтуерна архитектура, която е базирана за работа със събития. Като event може да се определи, значителна промяна на състоянието на дадено нещо. Например, когато клиент си купува кола, състоянието на колата преминава от “за продан” към “продадена”, тоест си сменя текущия status. Всяка промяна на дадено нещо може да се нарече event (събитие). Софтуерната система на продавача на коли може да третира тази промяна на състоянието като събитие (event), при което системата да направи промяна на данните в друга част от софтуерната архитектура. От формална гледна точка, когато се произведе, продаде или купи дадено нещо се изпълнява съобщение наречено event notification. Това което известява е, че самото събитие се е случило.

Този архитектурен модел може да се прилага при проектирането на софтуерни системи, при които се предават събития между слабо свързани софтуерни компоненти или услуги. Event-driven системата обикновено се състои от event emmitters и event consumers или event sinks.

Sink-овете имат отговорност за прилагане на реакция, веднага след като event-а настъпи. Emmitter-ите са тези при който настъпва събитието. Например, вие като цъкнете на даден бутон в даден уеб сайт, този бутон се явява Emmitter (от него произтична събитието), а sink може да всичко, което съдържа логика за обработване на това събитие, като javascript функция или php скрипт със заявка до базата данни и т.н.

**Как става цялото това нещо в Node.js**

Традиционния начин на работа на един web server винаги е бил **thread-based-model.** Като се пусне Apache или някой друг web server, той започва да приема връзки (connections). Когато приеме някоя връзка, web server-а държи тази връзка отворена докато не се върне исканата страница, отговор от базата данни или отговор от някой друг сървар или каквато и друга информация се изпраща до клиента. Ако тази операция отнеме няколко милисекунди за да се намери страницата на диска или да се изведе резултат от базата данни, web server-а блокира тази операция вход/изход, докато не се извърши цялата работа. Това е познато като **Blocking I/O**. За да се мащабира този вид web server, трябва да се пуснат различни копия от сървъра, като всяко копие изисква различна нишка (thread) от операционната система, за това се казва и thread-based-model.

В контраст, **Node.js** използва **event-driven model**, където web сървъра приема заявка, започва да я обработва, като междувременно получава и други заявки. Когато първата заявка е завършена, **Node.js** се връща на нея и изпраща резултатите. Работи на принципа с **callbacks.**

Сега ще дадем пример с принципа на работа на **callbacks**:

**setTimeout(function() {**

**alert(“Hello World! Node.js is awesome!”);**

**}, 3000);**

Това е вградена функция в JavaScript, която изплълнява дадена функция, подадена като първи параметър след някакво време подадено в милисекунди като втори параметър. В нашият случай, като се извикаме дадената функция **setTimeout,** javascript изчаква 3 секунди и изпълнява анонимната функция, която е подадена като първи параметър. Тази функция, която е подадена се нарича callback.

Същия е и принципа при заявка до node.js. Имаме една заявка до сървара, която трябва да извелче много данни от базата данни, тази заявка си има callback и когато отговора на заявката е готов се изпълнява callback, като през останалото време server-а може да изпълнява и други заявки.

Така се постига не зависими една от друга заявки, спрямо сървъра. Самите заявки се подреждат в опашка и за всяка има callback, който се изпълнява когато заявката се изпълни, без да пречи на останалите. Този модел е високо ефективен и скалируем, защото web server-a, на практика винаги приема заявки, защото не чака за никакви операции за четене или писане. Това се нарича non-blocking I/O или event-driven I/O.

За **пример** нека видим следния процес:

1. Използваме нашия Web browser за да направим заявка за страницата about.html на Node.js web server.

2. Node.js сървъра приема заявката и вика функция, която взима този файл от диска.

3. Докато Node сървъра чака за файла, той обслужва следващия web request.

4. Когато файлът е получен от диска се вика callback функция която се намира в oпашката от такива функции на сървъра.

5. Node сървъра изпълнява тази функция, която в този случай обработва “about.html” страницата и я изпраща към web browser-а

Сега сме сигурни, че в този случай, ще отнеме само микросекунди за сървъра за да върне файла, но микросекундите имат значение, когато говорим за големи web сървъри. Така Node.js се различава от останалите сървъри и езици за сървърно програмиране.

# **3. Asynchronous I/O**

Асинхронния вход/изход в компютърните науки е форма на I/O обработка, която позволява при обработване на една операция, да започне обработка на втора, преди първата да е свършила. Входно/изходните операции в компютъра могат да бъдат много бавни. I/O устройството може да включва механични части, които трябва да се местят, като seeking операция на хард диска за четене и запис. Примерно поведение на I/O операция може да бъде: започване на операция и изчакване, докато операцията не се завърши.. Такова поведение се нарича synchronous I/O или blocking I/O, защото блокира процеса, докато програмата обработва процеса. Когато има много I/O операции, това означава, че процесора прекарва по много време в idle, чакайки I/O операциите да завършат.

Алтернативата е да се започне комуникация и да се извърши processing, който не изисква I/O операцията да е завършена. Това поведение се нарича asynchronous input/output. Всяка задача, която всъщност зависи от I/O oперациите, които не са завършили, не може да бъде изпълнена, но пък, тези който не зависят от конкретна конкретна не изпълнена операция, може да продължи по работата си.

Така можем с node.js да обработваме файлове много по-бързо, докато в сръвнение с php, това не е възможно, защото работи с blocking I/0

# **4. V8 JavaScript Engine**

**JavaScript Engine** с отворен код, разработен от Google. Използва се в браузъра Google Chrome. Първата версия излиза, когато излиза и първата версия на Google Chrome, 2ри Септември 2008.

V8 компилира JavaScript до native машинен код преди изпълнението му. Както други традиционни техники като Java Virtual Machine изпълнява до bytecode или го интерпретира. Компилирания код е допълнително оптимизиран динамично в runtime въз основа на евристични методи за изпълнение на код. Използват се техники за оптимизиране като inlining, elision, inline кеширане и много други.

Garbage collector-а на V8 е базиран на Generational Incremental Collector. V8 е предназначен да се използва както в браузъра така и като самостоятелен високо производителен JavaScript Enginе, който може да се интегрира в различни проекти, както е случая за Node.js.

# 5. NPM

**Какво можем да наричаме npm?**

NPM **конзолно приложение** и също така е **пакетен мениджър (Package Manger)** за Node.js, който се изпълнява в терминала и управлява зависимостите (dependancies) на приложението. Node.js използва и други пакетни мениджъри, но предимно npm. Според неговия автор, npm, не е акроним от Node.js Package Manager. От версия 0.6.3, npm идва вградено заедно със средата на Node.js.

Поддържа инсталация и деинсталация на пакети:

Команди:

**npm install express** – инсталира локално пакетите за текущия проект по който работим и се препоръчва от обществото на node.js.

**npm install -g grunt-cli –** инсталираглобално за всички проекти. По някога е наложително.

**npm uninstall express –** деинсталира локално за текущия проект по който работим.

**npm uninstall -g grunt-cli –** деинсталираглобално за всички проекти.

Съществува файл **package.json,** в който се описват зависимостите. Зависимостите биват, твърди зависимости (**--production**), меки зависимости(**--dev**). Файла съдържа също така мета данни за публикуване.

# **6. Тестване на node.js приложения**

В тази част ще споменем, най-често използваните инструменти за **тестване** на node.js приложения:

**Mocha**

Mocha е богат на функционалност JavaScript test framework. Може да се пусне и в node и в браузъра. Прави асинхронното тестване опростено и забавно. Mocha тестовете се изпълняват последователно и позволяват гъвкаво и детайлно отчитане на резултатите (reporting).

**Should.js**

Обширен **test framework** за node. Разширява **обектно-прототипния** модел, като спомага на програмиста да опише, какво точно трябва да е поведението на даден обект.  **Should** буквално наследява концепцията assert за unit tests, като ще дадем следния пример:

**var user = {**

**name: 'tj'**

**, pets: ['tobi', 'loki', 'jane', 'bandit']**

**};**

**user.should.have.property('name', 'tj');**

**user.should.have.property('pets').with.lengthOf(4);**

Вижда се, че проверката за нещо става буквално като чист английски език.

# 7. Предимства, които ни дава node.js пред други web servers

- Бърз е. Концепцията осъществена от V8 virtual machine, прави изпълнението на JavaScript, супер бързо

- Бърз by design. Повечето приложения, които пишем днес са силно обвързани с I/O процесите. Поради event-driven природата на JavaScript (и Node.js), каквото и да напишем, то е супер бързо

- Един език за всичко. Всяко сериозно web приложение днес използва JavaScript, за подобряване на UI-a и UX-a. Например Facebook и Twitter използват много JavaScript за интеракциите между страниците. Имайки възможността да пишем на този език и за back-end частта от нашето приложение ни прави атеративни.

- Много подходящ за уеб в реално време, тъй като Node.js е уникално пригодено за това