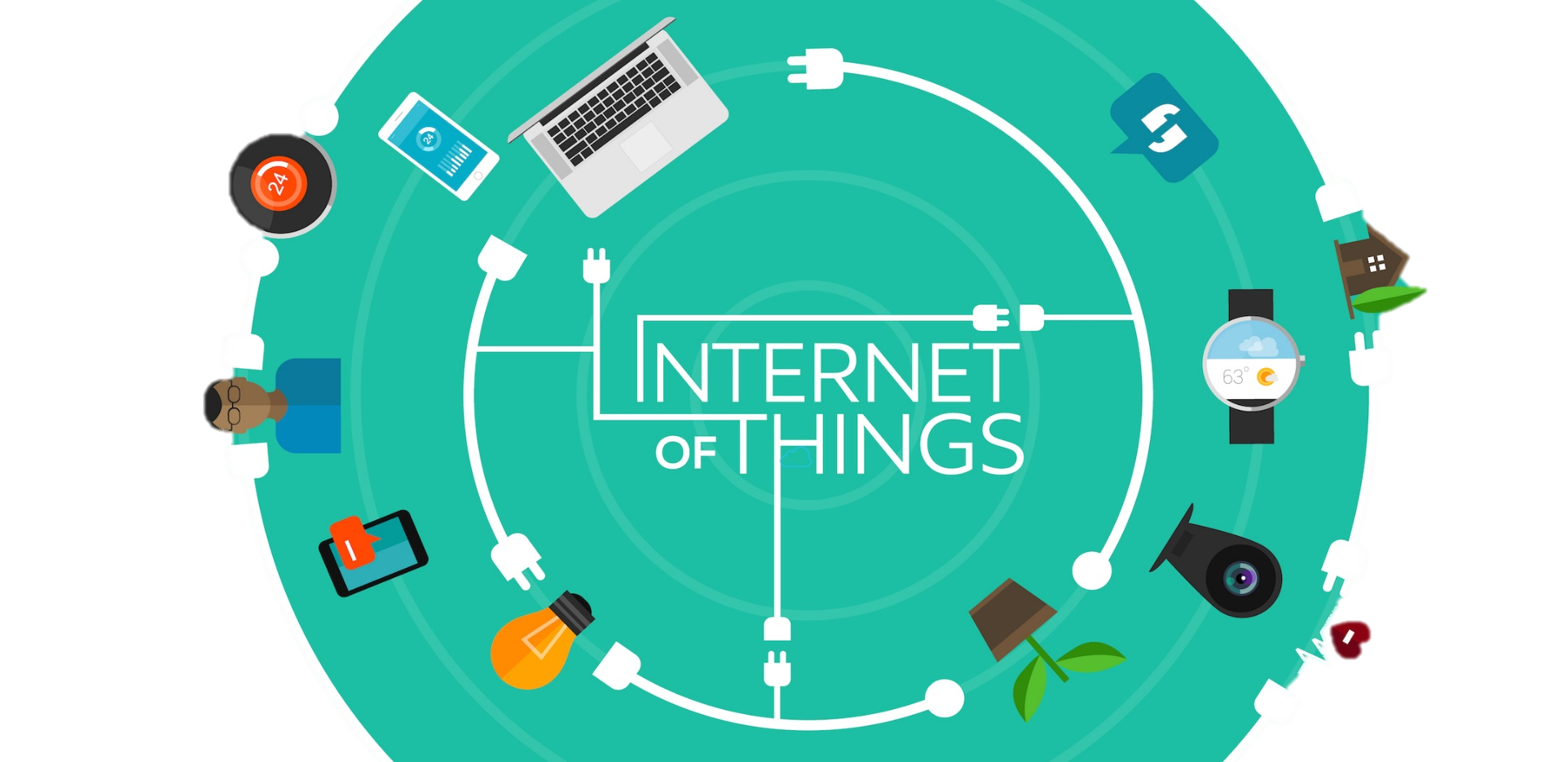
****

**Универзитет Св. Климент   
 Охридски -Битола**

**Факултет за информатички и  
комуникациски  
технологии**

ДИПЛОМСКА РАБОТА









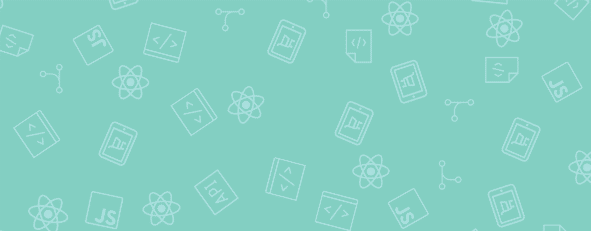


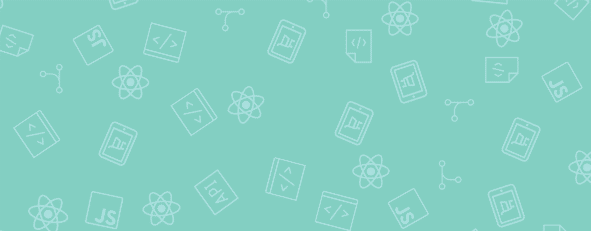
**Кандидат:** Мартина Талевска ИНКИ 53



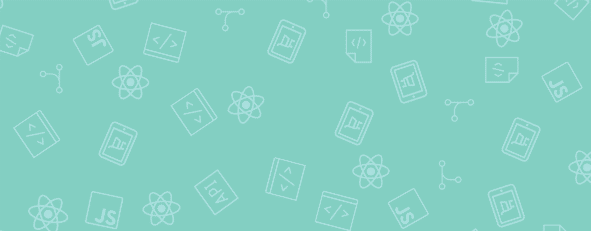
**Ментор:** проф. Д-р Никола Рендевски

2019, Битола

**СОДРЖИНА**

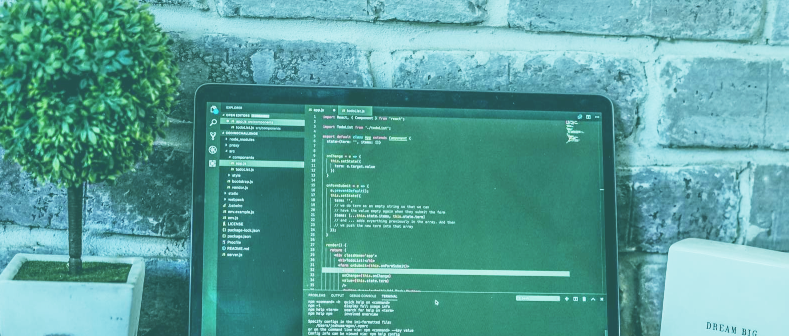






# Вовед

Идејата за изработување на дипломската работа е со цел да прикажаме како би изгледал ..........кое се надеваме ќе овозможи разбирање и потребни информации и знаење за да напредуваме во свет во кој повеќето физички предмети имаат свои дигитални животи благодарение на веб-протоколите. Секојдневно се користат солидни протоколи за да се изградат уште поголеми, попаметни но поедноставни работи - за да се изгради веб на нештата! Во текот на последните неколку години Интернетот на нештата (IoT) стана една од најпопуларните теми во технологијата и деловниот свет.



**1**

## Интернет на нештата и Веб на нештата

Во последните години, Internet of Things го бележи својот развој во технологијата и бизнисот. Визијата на светот каде мини компјутерите со сензори и комуникациски интерфејс се вградени во инфраструктурата на нашите градови, автомобили, канцеларии и домови ја револуционираат секоја област во нашите животи- како играме, како работиме и како живееме. До пред некое време, ИоТ проектите беа фокусирани на градење проекти кои се мали во размер, затворени и изолирани, каде што уредите не беа дизајнирани за лесен пристап или пак за репрограмирање. Спојувањето на уредите со апликациите во даден случај значеше дека секоја промена на веќе постоечкиот проект ќе биде комплексна и скапа. Ова водеше кон поставување на лимит во одржувањето и еволуцијата на ИоТ, бидејќи ресурсите (време, финансиски средства и технички способности) се задолжителни секој пат кога е додадена нова функција.

Во контраст, веб на нештата станува широко успешен во последните две декади поради што е поедноставен да се научи и користи, исто така и лесно го нагласува спојување помеѓу серверите, пребарувачите и апликациите. Web of Things е специјалност на Internet of Things и она што го направи веб толку успешен е што се применува на вградени уреди со цел да го направи последниот развој на Internet of Things колку што е можно подостапен за повеќето девелопери. Со Web of the Things – исто како и со секој веб-текст едитор и со основно ниво на познавање на веб стандардите ( HTML и HTTP) може лесно да поврземе уреди и објекти на веб. Исто така овозможува напредок на следно ниво и води кон ефективно градење на интерактивна и иновативна real-world апликација што ги вклопува физичкиот и дигиталниот свет.

За да ја доловиме есенцијалната дефиниција на Интернет на нештата една реченица не е доволна. Концептот постои повеќе декади и нема дефинирани граници околу тоа што е Internet of things и тоа што не е. Сепак, основната дефиниција за визијата на Интернет на нештата е свет каде што интернетот е многу повеќе од колекција на мултимедиска содржина: туку се проширува во физичкиот, real-time свет користејќи огромен број на мали или микро компјутери. Или накратко дефинирано, Internet of things е систем на физички објекти што можат да бидат истражувани, мониторирани, контролирани или можат да стапат во интеракција со електронски уреди што комуницираат со различни мрежни интерфејси и евентуално можат да бидат конектирани на поголема интернет мрежа.

Пред две декади, каде секојдневните објекти можат да го почуствуваат светот преку сензори и после тоа да анализираат, зачувуваат или разменуваат информации постоеше само во научно-фантастичните новели или the Jetsons. Денес, тие сценарија стануваат реалност благодарение на колосалниот процес во вградените уреди што воведија во светот нова класа на објекти: паметни објекти. A smart thing (паметен објект) ќе го дефинираме секој физички објект што е дигитално надграден или зголемен со едно од наведените:

* Сензори (за температура, светло, движење)
* Активатори (дисплеј, звук, мотор)
* Компјутери (програми и логика)
* Комуникациски интерфејси (жични или безжични)

Паметните објекти го проширија светот во кој живееме со нов размер на апликации  
(сл.1 и сл.2). Со имплементирање на микро компјутери со ниски цени-но сепак моќни насекаде околу нас, станува полесно за интеракција со физичкиот свет со многу поприлагодливи просторни и временски резолуции од порано.

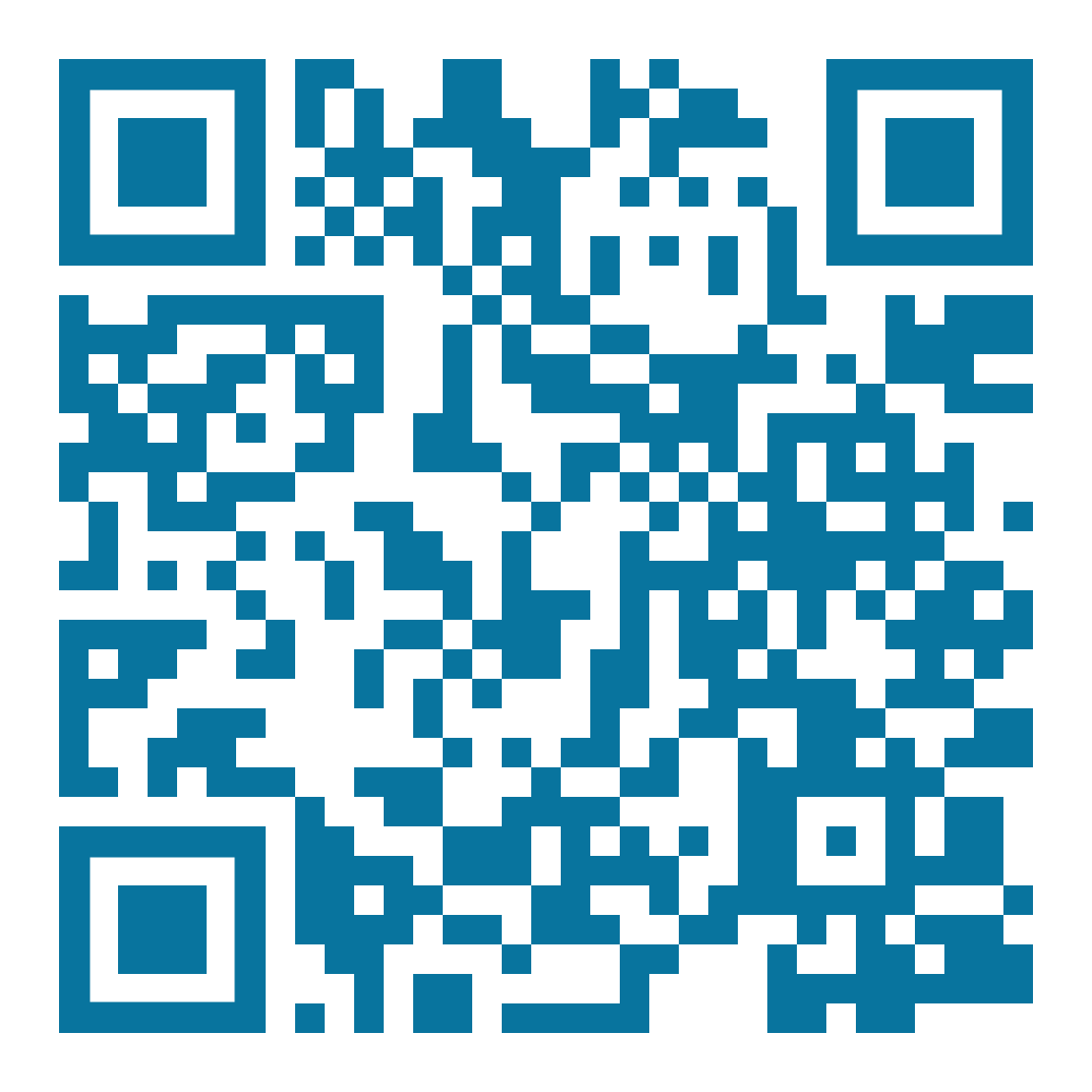
Околини

Машини

Уреди



Ознаки



Сл.1 Internet of things landscape

Компјутерска моќ и комплексност

Smart building

Smart car

Smart city

Arduino

Philips Hue

Raspberry Pi

iBeacon/BLE

QR ознака

NFC/RFID ознака

Со оглед на тоа што секојдневните објекти ќе бидат дигитално надградени, следниот логичен чекор е користење на World Wide Web екосистемот и инфраструктурата за да изградиме апликации за Internet of Things, воедно и ефективно ја прекршуваме шемата: ‘‘ one device, one protocol, one app‘‘ или ‘‘ еден уред, еден протокол, една апликација‘‘. Идејата за максимизирање на постојните и новите алатки и техники кои се користат на веб и применувајќи ги истите во развојот на Internet of things е ултималната цел на Web of things.



Сл.2 Internet of things landscape

Идеологијата на Интернет на нештата (IoT) може да се погледне како високо динамичен и радикално дистрибуиран мрежен систем составен од многу голем број на паметни предмети што можат да се идентификуваат. Овие објекти се во можност да комуницираат меѓу себе, со крајните корисници или други субјекти во мрежата. Влегувајќи во ерата на Интернет на нештата, употребата на мал, ефтин и флексибилен компјутерски хардвер што дозволува програмирање на крајниот корисник станува сегашност. Една од нив, разгледана во овој труд, е Raspberry Пи, целосно прилагодлива и програмабилна мала компјутерска табла. Компаративната анализа на нејзините клучни елементи и изведби со некои тековни постојни платформи на прототип IoT покажаа дека и покрај малкуте недостатоци, Raspberry Pi останува ефтин компјутер со многу успешно користење во разновиден спектар на апликации за истражување во IoT визијата.



**2**

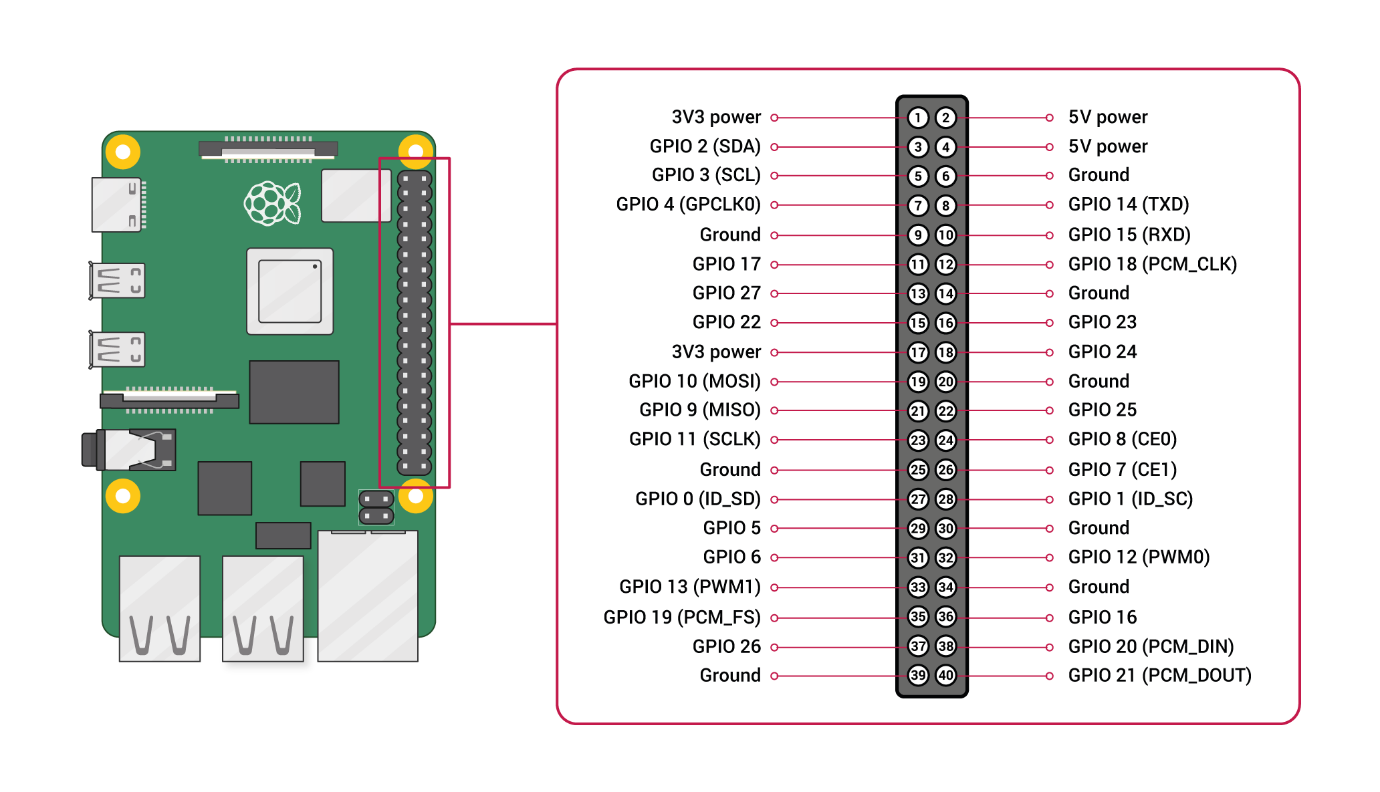
# Хардверски компоненти

Интернетот на нештата, или IoT, се однесува на милијарди физички уреди ширум светот кои се поврзани на интернет мрежа, и меѓусебно собираат и споделуваат податоци. Благодарејќи на евтините процесори и безжичниот интернет, сега е возможно од авто-пилот на авион до self-driving автомобил како дел од IoT. Со помош на Raspberry Pi and Node Js сега можеме да креираме ИоТ процес. Во продолжение ги опишуваме деловите кои ни се потребни и нивно синхронизирање во процесот на изработката на овај IoT процес.

## **Raspberry Pi**

**Raspberry Pi** е ARM architecture процесор базиран на боард дизајниран за електро-инженери, и исто така е една од најдоверливите платформи за развивање на проекти. Со поголема брзина на процесорот и од само 1 GB RAM, the PI може да се користи за проекти од поголеми размери.

Raspberry Pi е името на серијалот на single-board компјутери направени од Фондацијата Raspberry Pi, добротворна организација во Велика Британија, чија цел е да ги едуцира луѓето за компјутерите и да создаде полесен пристап за учење. Во големина е колку една кредитна картичка и се приклучува на компјутерскиот монитор или ТВ. Користат 700 MHz процесор ARM, а моделите се наоѓаат опремени со 256 MB или 512 MB RAM меморија. Разликуваме два модела на уредот: Model A и Model B. Единствените вистински разлики се додавањето на Ethernet и дополнителната USB порта на поскапиот Model B. Тие можат да поддржат најмногу пет USB 2.0 порти. Raspberry Pi може да користи MicroSD картичка или флеш меморија (до 4 GB) за нестабилно складирање и користи HDMI за дигитален аудио и видео излез. Нема мрежен интерфејс, но тие поддржуваат мрежно поврзување преку Wi-Fi и Ethernet преку USB. Raspberry Pis обично го работат Linux како нивен оперативен систем. Arch Piux, Fedora, Puppy Linux, Slackware, FreeBSD, NetBSD, OpenSUSE и RISC OS се достапни за Pi. Исто така е достапен и Raspbmc, оперативен систем специјално создаден за користење на Pi како дигитален центар за –медиуми или HTPC.



Сл.3 Raspberry Pi GPIO (general-purpose input/output) pins

## Други Компоненти

Други компоненти кои ни се потребни за комплетирање на проектот се:

* ****Пински каблиња (jumper wires)
* ****3x Отпорници од 3.3Ω
* ****ЛЕД диоди
* ****Breadboard

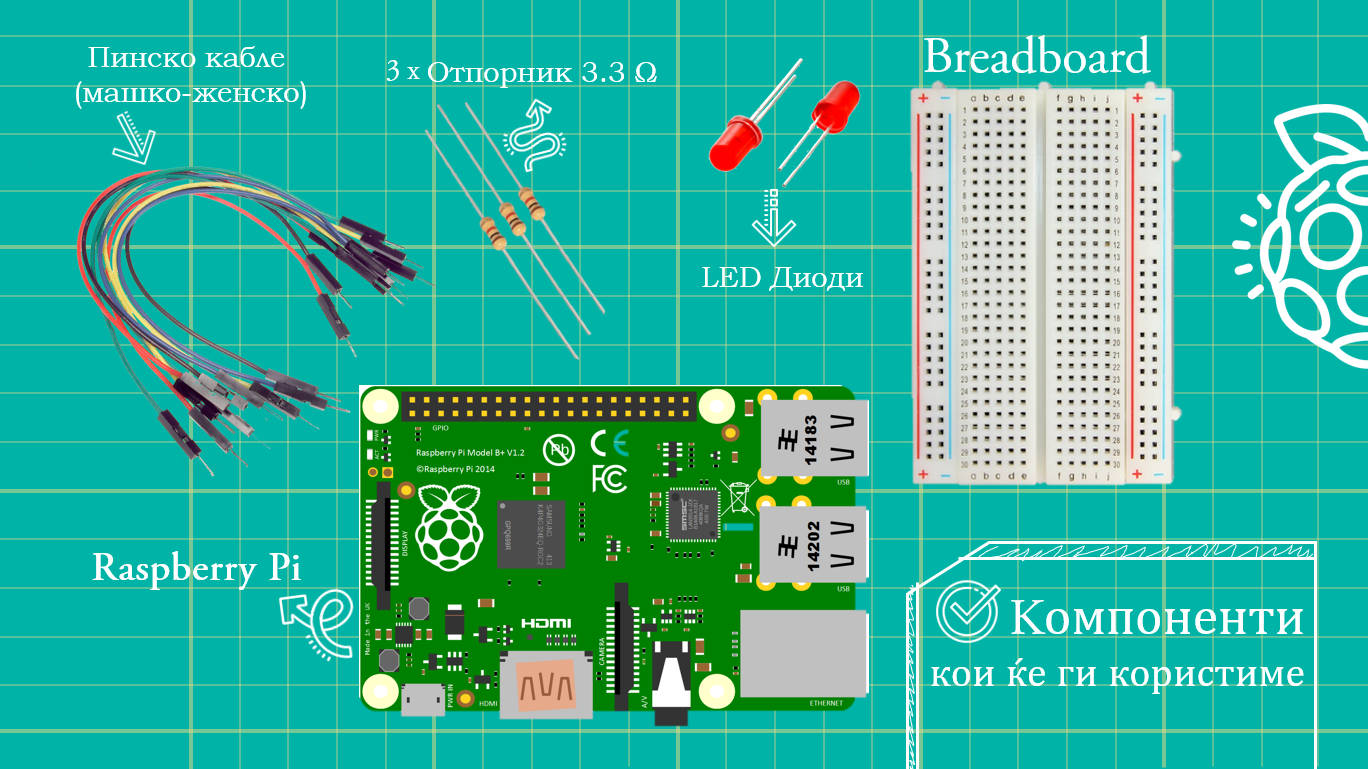
### 

### Пински каблиња (Jumper wires)

Поимот „jumper wire“ едноставно се однесува на спроводна жица што воспоставува електрична врска помеѓу две точки во коло. Пинските каблиња или jumper wires можете да ги користите за да модифицирате коло или да дијагностицирате проблеми во колото.

Пинското кабле е со конектор или игла на секој крај (или понекогаш без нив - едноставно „затегната“), што вообичаено се користи да ги поврзете компонентите на breadboard-от или друг прототип или тест коло, внатрешно или со друга опрема или компоненти, без лемење. Индивидуалните пински каблиња се вградени со вметнување на нивните „крајни конектори“ во слотите што се предвидени на breadboard-от, во header на конекторот на поклопецот или на парче опрема за тестирање.

Пинските каблиња обично доаѓаат во три верзии: машко-машко, машко- женско и женско- женско кабле. Разликата помеѓу секоја е во крајната точка на жицата. Машките краеви имаат испакната игла и можат да се вклучат во нештата, додека женските краеви немаат испакнати игли и се користат за приклучување во работите кои ги користиме со нив.



Сл.4 Компоненти кои ќе ги користиме

****

### Отпорник (Resistor)

Отпорникот е пасивна дво-терминална електрична компонента која спроведува електричен отпор како елемент на колото. Во електронските кола, отпорниците се користат за намалување на протокот на струја, прилагодување на нивото на сигналот, за поделување на напони, активни елементи на пристрасност, меѓу другите намени и прекинување на преносни линии. Главната функција на отпорниците во колото е да го контролираат протокот на струја во други компоненти. Доколку земеме една LED диода на пример: ако премногу струја тече низ ЛЕД диодата, таа е уништена. Значи, ги користиме отпорниците за да го ограничуваат протокот на електроните, намалувајќи ја струјата.Напонот се појавува со потенцијалната разлика во енергијата во резисторот. Математичкиот одговор е дека отпорник е двостепен електричен уред за спроведување, законот на Ом: V = IR. Отпорниците влијаат и на струјата и на напонот. Тие го прават тоа во линеарен начин... Напонот низ секој отпор ќе се разликува директно во однос на струјата што тече низ неа. Значи, во првиот отпорник од 10 оми, напонот низ него е 10 х 10.

За да го намалиме напонот на половина, едноставно формираме делител на напон помеѓу 2 отпорници со еднаква вредност (на пример, 2 10KΩ) отпорници. За да го поделиме напонот на половина, сè што треба да сториме е да поставиме какви било 2 отпорници со еднаква вредност во серија и потоа ставаме пинско кабле помеѓу резисторите.Иако отпорниците доаѓаат во различни форми, можеме да ги поделиме на само два основни типов:

„Фиксирани“ отпорници

Варијабилни отпорници (или „потенциометри“)

****

### LED диоди

Светлечка диода, односно ЛЕД (*Light-emitting Diode*) е електронски полуспроводничка компонента која испушта светлина кога низ неа ќе помине електрична струја. Светлината се произведува кога честичките што ја носат струјата (познати како електрони) се комбинираат заедно во полупроводничкиот материјал. Бидејќи светлината се создава во рамките на цврстиот полупроводнички материјал, LED диоди се опишуваат како уреди во цврста состојба. Терминот solid-state lighting, кој исто така опфаќа органски LED диоди (OLED), ја разликува оваа технологија на осветлување од други извори кои користат загреани нишки (лампи за блескаво и волфрам) или гас празнење (флуоресцентни ламби). ЛЕД се користат како индикатори во технички апликации, но сè повеќе и како извор на светло. Користени се како практичен електронски елемент уште од 1962 година, кога првите LED светеа со црвено слабо светло, додека новите модерни верзии се достапни со видливи, ултравиолетови и инфрацрвени бранови должини како и силна светлина.

Главните полупроводнички материјали што се користат за производство на LED диоди се:  
  
     Индиум галиум нитрид (InGaN): сини, зелени и ултравиолетови LED диоди со висока осветленост

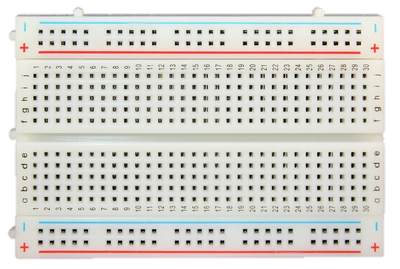
Фосфид на алуминиум галиум индиум (AlGaInP): жолти, портокалови и црвени LED диоди со висока осветленост

 Алуминиум галиум арсенид (AlGaAs): црвени и инфрацрвени LED диоди

Галиум фосфид (GaP): жолти и зелени LED диоди

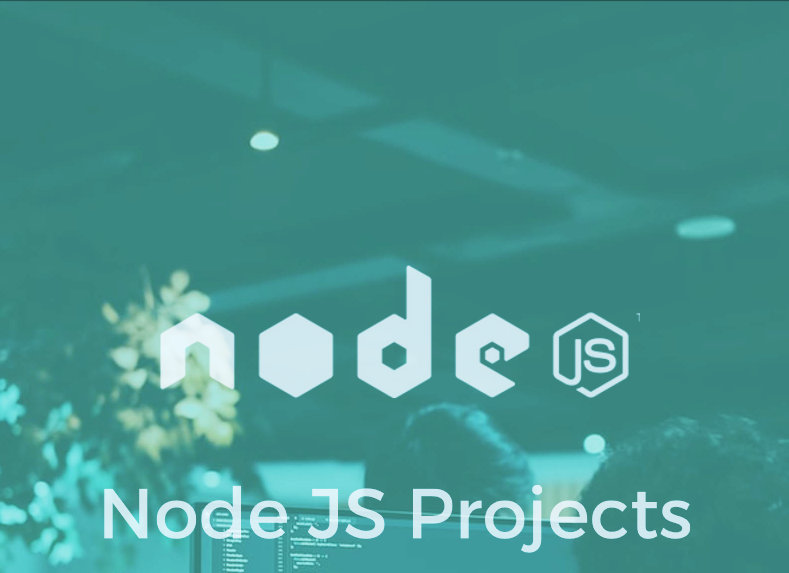
****

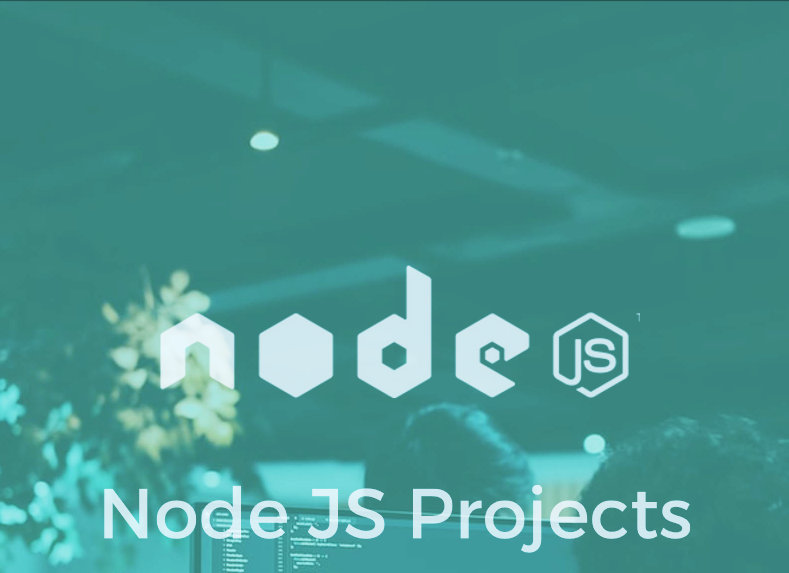
### Breadboard

Breadboard е градежна основа за прототипирање на електроника. Првично, зборот се однесувал буквално на табла за леб, полирано парче дрво кое се користи за сечење леб. Бидејќи solderless breadboard не бара лемење, таа може повторно да се употреби. Ja користимe за да изградеме и тестираме кола брзо пред да финализираме кој било дизајн на колото. Breadboard-от има многу дупки во кои можат да се вметнат компоненти на колото како IC и отпорниците.

Повеќето електронски компоненти во електронските кола можат да бидат меѓусебно поврзани со вметнување на нивните leads или терминали во дупките и потоа да поврзат преку жици каде што е соодветно. Breadboard-от има ленти од метал под таблата и ги поврзува дупките на горниот дел од таблата. Металните ленти се поставени како што е прикажано подолу. Забележете дека горниот и долниот ред на дупките се поврзани хоризонтално и се делат на средина додека преостанатите дупки се поврзани вертикално.

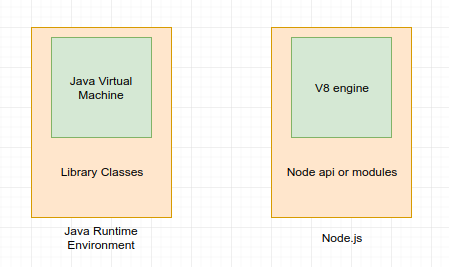
Сл.5 Breadboard





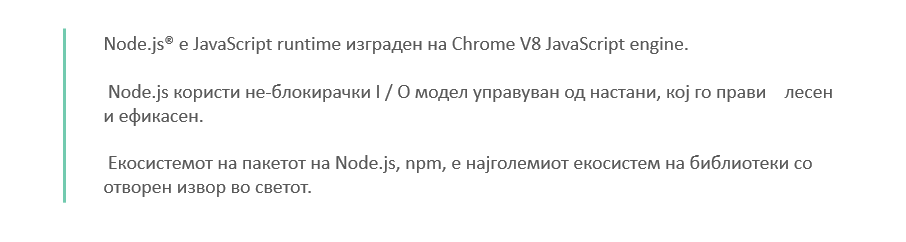
## Node.js

Доколку знаеме дека JavaScript е широко познат јазик креиран да се погрижи за интеракцијата на корисникот во веб пребарувачот, сега благодарение на node.js можеме да користиме JavaScript надвор од пребарувачот, и исто така во интеракција со хардвер уреди како што е Raspberry Pi. Node.js run-time околината вклучува сè што ни треба за да извршиме програма напишана во JavaScript. Node.js настана кога развивачите на JavaScript го проширија од нешто што можевме да го извршуваме само во прелистувачот во нешто што може да го извршиме на нашата машина како самостојна апликација. JavaScript сега има можност да прави работи што можат да ги прават другите јазици за скрипти како Python. И прелистувачот JavaScript и Node.js работат на V8 JavaScript runtime engine, со тоа што го зема нашиот JavaScript код и го претвора во побрз машински код. Машинскиот код е код на ниско ниво што компјутерот може да го извршува без претходно да го интерпретира.



Сл.6 Аналогија на Node JS со Java Runtime Environment

**Зошто Node.js?**

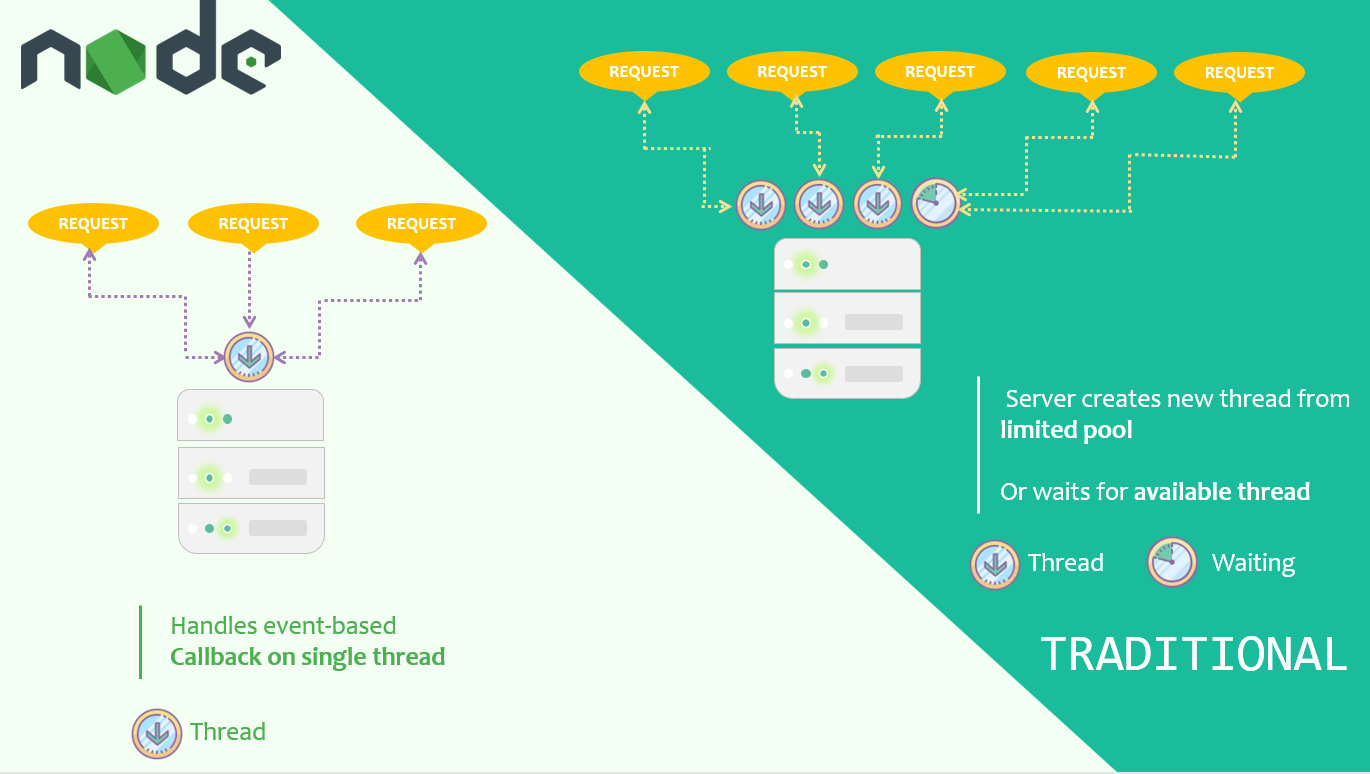
Во продолжение ја наведуваме дефиницијата како што е дадена на официјалната веб-страница на Node.js:

Input/Output или I/O се однесува на влез / излез, во спектар почнувајќи од читање / пишување локални датотеки до испраќање HTTP барање до API. I/O одзема време и може да блокира други функции. Како пример ќе земеме сценарио каде бараме backend база на податоци за деталите на user1 и user2, а потоа ги отпечатиме на екранот/ конзолата. Одговорот на ова барање одзема време, но и двете барања за кориснички податоци може да се извршат самостојно и истовремено.

**Како работи?**

Идејата позади Node.js е да користи не-блокирачки, настан-базиран (event-driven) I/O со цел да биди лесен и ефикасен пред real-time апликациите кои работат на дистрибуирани уреди. Покрај тоа, Node.js не е нова платформа што ќе доминира во светот за развој на веб-страници, наместо, тоа е платформа што исполнува одредена потреба. Доколку го разбираме ова, нема да сакаме да го користиме Node.js за CPU- интензивни операции (кои го преоптоваруваат процесорот), всушност користејќи го за тешки пресметки ќе ги поништиме сите негови предности. Креирањето брзи, скалабилни мрежни апликации е местото каде Node.js го зазема своето место, бидејќи е способен за справување со голем број на истовремени врски со голема пропустливост (throughput) што е еднакво на висока приспособливост.

Споредено со традиционалните web-serving техники каде што секоја конекција (request) поткрева нова нишка (thread), зафаќајќи повеќе RAM, а некогаш и целосната меморија достапна, Node.js оперира со една нишка (single-thread) користејќи non-blocking I/O повици, со што дозволува и поддржува десетици илјади истовремени врски што се случуваат во циклусот на настани.

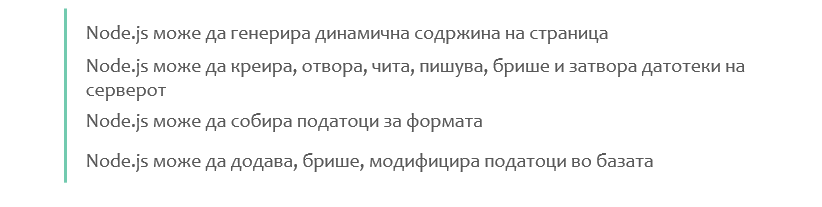


Сл.7 Node.js VS Traditional Web server

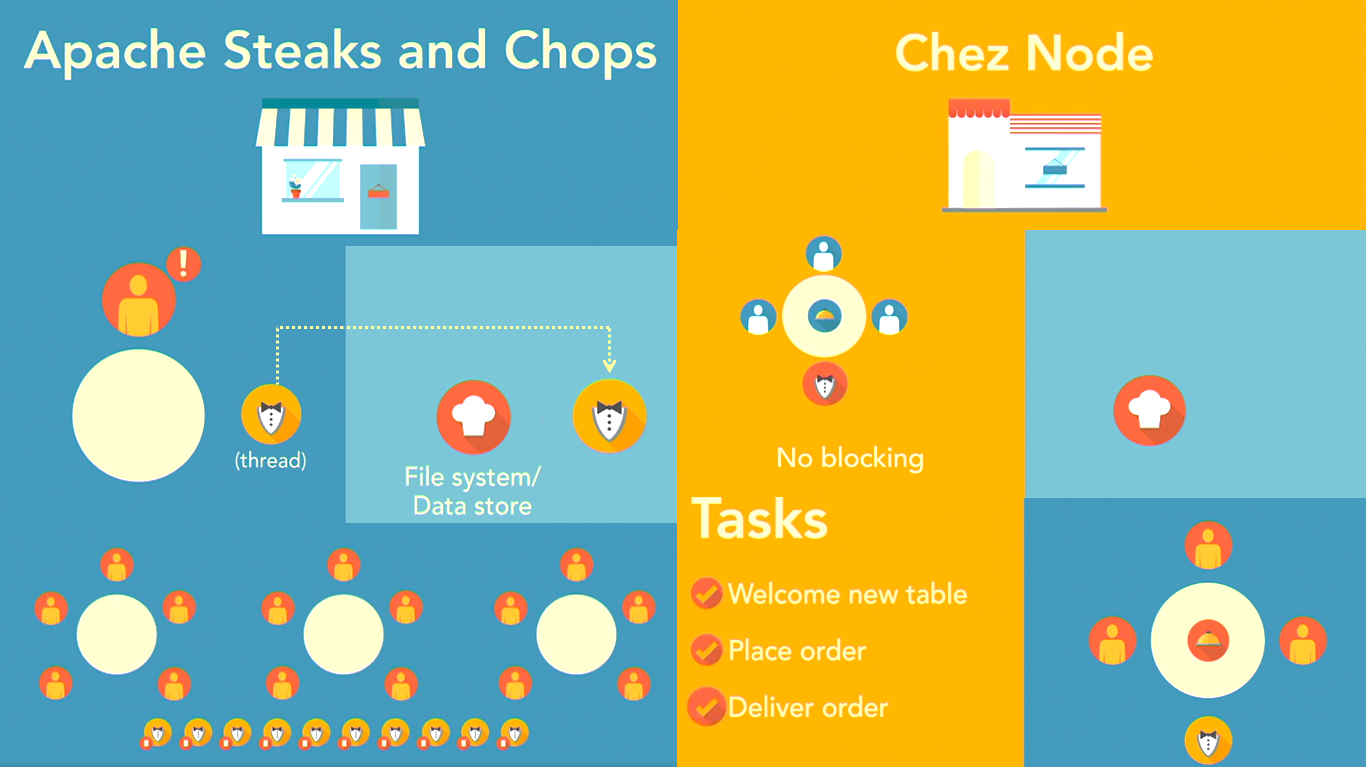
Пример: ако претпоставиме дека секоја нишка потенцијално има придружна меморија од 2МВ со неа, а работи на систем со 8GB RAM тогаш имаме теоретски максимум од 4.000 истовремени врски (пресметките се земени од статијата на Мајкл Абернети „What is Node.js? ”, објавено во IBM developerWorks во 2011 година), плус трошоците за context-switching помеѓу нишките. Ова е типично сценарио со кое би се справувале доколку користиме традиционалната техника на веб-сервирање. Со цел да го избегнеме ова, со Node.js постигнуваме нивоа на приспособливост од преку 1М истовремени врски и над 600к истовремени web sockets конекции.

Секако, тука е и прашањето за споделување на една нишка меѓу сите барања на клиентите, и тоа е потенцијалната стапица во пишувањето на апликациите во Node.js. Како пример, тешките пресметки можат да ја задушат единствената нишка и да предизвикаат проблеми за сите клиенти со што последователните барања (requests) ќе бидат блокирани додека наведената пресметка не се заврши.

**Што може да направи Node.js?**



## Node JS VS. Apache

Со едноставен и сликовит приказ на следната страна ќе демонстрираме како Node JS функционира и зошто е толку брз начин на работење со споредба на два ресторани. Првиот е Аpache. Во овај ресторан, секој нов гостин претставува нов корисник. И правењето нарачка претставува ново барање или request. Со самата нарачка менаџерот ни доделува келнер кој ќе не служи само нас и ќе ги прима само нашите нарачки- келнерот претставува нашата нишка(thread) до кујната која во нашата аналогија ја толкуваме како File System/Data Store. Вака работи Apache, секое барање е single- threaded. Откако ќе ја земе нарачката келнерот ја доставува во кујната и чека додека да биде готова. Во меѓувреме неможе да прими друга нарачка, што значи шефот го блокира корисникот за да направи друга нарачка за чаша вода. Во Apache, нишката -the single thread чека додека File System да заврши со читање на фајловите за да продолжи да прави било што друго. Ова може да се протолкува како блокирање. Доколку има повеќе гости или во нашиот случај повеќе корисници, секој корисник има свој келнер или нишка, што е добра услуга. Но повеќе гости, значи повеќе келнери. Од друга страна пак, во Chez Node има само еден келнер, бидејќи Node JS e single threaded. Тоа значи келнерот ја зема нарачката од еден гостин и ја носи во кујната, и во меѓувреме оди кај другиот гостин за нова нарачка. Врз основа на ова можеме да кажеме дека тука келнерот се однесува асинхроно, односно секоја активност што треба да ја направи келнерот претставува нов настан или event. Настаните ќе бидат преземени во редослед како што се повикани. Келнерот тука не чека, нема блокирање или така наречено- non-blocking event driven IO. Сите корисници ја користат истата нишка (single thread). Node JS е асинхрон, што значи може да прави повеќе од една работа. Оваа способност за мултитаск е она што го прави Node JS толку брз.

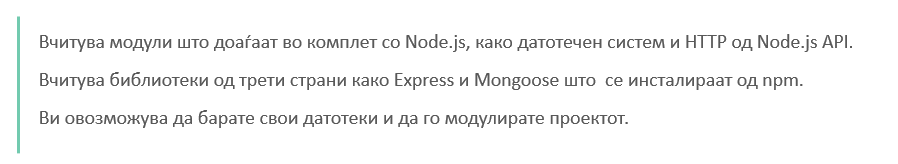
Сл.8 Node.js VS Apache

## Node.js NPM

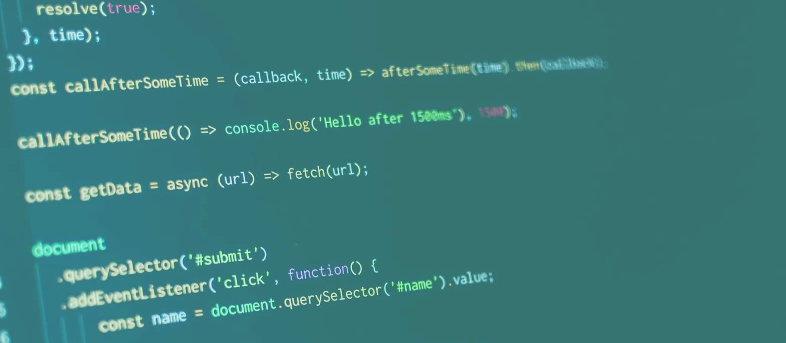
Стандардниот менаџер на пакети за Node.js или NPM (Node package manager) ни нуди пакети што можеме да ги користиме во апликации со цел нивниот развој да биде побрз и поефикасен. npm се инсталира во системот кога е инсталиран Node.js. Може да управува со пакети кои се локални зависности на одреден проект, како и со глобално инсталирани JavaScript алатки. Сепак, бидејќи секој може да креира библиотеки и нема процес на проверка, треба да бидеме претпазливи за оние со низок квалитет, несигурни или малициозни. npm се потпира на извештаите на корисниците за тоа кои библиотеки ги прекршуваат правила и е потребно да се симнат, но за полесна одлука која библиотека да ја користиме, вклучуваат статистички податоци како број на преземања и број на зависни пакети.

**Require**

Require е функција и прифаќа параметар „path“ и го враќа module.exports.

Require прави три работи:

**3**



# Програмски дел

# Зошто WebSocket сервер?

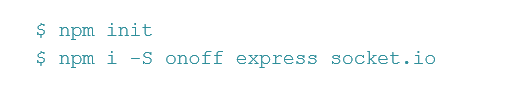
Доколку ја разбираме HTTP комуникацијата знаеме дека кога прелистувачот испраќа HTTP барање, прелистувачот одговара на ова барање со некои податоци. Откако податоците ќе бидат примени од прелистувачот, врската се прекинува.  
  
Оттука, за секое барање, ние создаваме нова HTTP врска за која е потребно извесно време да се воспостави и исто така содржи некои мета-податоци како заглавија(headers) и носивост(payload). Испраќањето на HTTP барање знае да биде тежокк процес и да одземи многу време.

Серверот не може да испрати податоци на клиентот, доколку клиентот не го побара тоа. Оттука, согледуваме дека HTTP е stateless I unidirectional- еднонасочен протокол за комуникација и не треба да се користи за двонасочна комуникација во реално време.

Спротивно на тоа, WebSocket конекцијата е двонасочен протокол за комуникација. Откако ќе се оствари web-socket конекцијата, таа останува отворена сè додека клиентот или серверот намерно не ја затвори. Откако ќе се воспостави врската, клиентот и серверот можат да си испраќаат настани еден на друг.

WebSocket конекцијата е надградба на HTTP конекција. Значи, клиентот прво испраќа HTTP барање до HTTP серверот со connection: Upgrade и Upgrade: WebSocket request headers.  
Тогаш е одговорност на HTTP серверот да воспостави WebSocket врска на истата TCP врска. Откако ќе се воспостави врската, серверот не може да испраќа настани до клиентот или да слуша настани од клиентот. Не сите прелистувачи имаат поддршка за протокол за WebSocket комуникација. Сепак, socket.io и socket.io-клиентот заедно обезбедува безбеден механизам за да се постигне целосно дуплекс механизам користејќи долго анкетирање кога протоколот WebSocket не е поддржан од прелистувачот.  
  
Освен испраќање едноставни настани, socket.io имаат богат API за управување со имиња на простори и простории за врски. Треба да ја разгледате нивната документација за да дознаете повеќе.

Кога ќе ја отвориме веб апликацијата во веб прелистувачот, таа се поврзува со WebSocket серверот, хостиран на Raspberry Pi. Кога корисникот ќе кликне на копчето (на веб-апликацијата), оваа веб-апликација испраќа порака (настан) до WebSocket серверот и серверот може да манипулира со состојбата на GPIO пиновите врз основа на информациите содржани во овој настан.



Од горенаведените команди, ја иницијализиравме датотеката package.json за да ги следиме project dependencies. Пакетот onoff обезбедува високо ниво на API за комуникација со GPIO на Raspberry Pi.

Пакетот express обезбедува АПИ на високо ниво за да се создаде HTTP-сервер во Node.js за да им служи на нашата веб-апликација која вклучува ги датотеките .html, .css и .js. Исто така, ќе се користи за поставување на WebSocket серверот.

Пакетот socket.io обезбедува API за создавање на WebSocket-сервер во Node.js. Овој пакет се користи од страната на серверот (Node.js). Сепак, socket.io-клиент обезбедува JavaScript датотека за веб-апликацијата (во прелистувачот) за комуникација со WebSocket серверот.

*****led-control.js* датотеката содржи GPIO логика за контрола на LED светлата. Оваа датотека извезува API функции за серверот.

**** *server.js* датотеката содржи express -сервер за да биде домаќин на нашата веб-апликација и WebSocket сервер кој се поврзува со оваа веб-апликација за да слушате прилагодени настани испратени од веб-апликацијата, како што се притискање на копчињата.

****веб-апликација / директорка содржи средства за нашата веб-апликација. Кога корисникот отвора специфична URL-адреса во прелистувачот, HTML. датотеката HTML ќе биде опремена по дифолт. Увезува лист за стилови.css во копчиња за стилови.

**** *main.js* содржи логика да се поврзете со серверот WebSocket и да ги слушате настаните за кликнување на копчето DOM. Се увезува според индексот.html.

led-control.js

// import `onoff` package

const { Gpio } = require( 'onoff' );

// configure LED pins

const pin\_red = new Gpio( 26, 'out' );

const pin\_green = new Gpio( 19, 'out' );

const pin\_blue = new Gpio( 13, 'out' );

// toggle LED states

exports.toggle = ( r, g, b ) => {

pin\_red.writeSync( r ? 1 : 0 );

pin\_green.writeSync( g ? 1 : 0 );

pin\_blue.writeSync( b ? 1 :0 );

};

Како што спомнавме и претходно, пакетот onoff го користи BCM системот за нумерирање на пиновите. Според дијаграмот на колото, ние доделивме соодветен број на пин на секоја боја на LED диодите. Воведуваме toggle функција за вклучување што зема булова (boolean) вредност за секоја светилка. Врз основа на овие вредности, 0 или 1 се запишува на соодветниот GPIO што прави излезот да биде 3.3V – светилката да светне или 0V.

## Server Setup

Следното што ни треба е HTTP и WebSocket сервер.

[server.js](https://gist.github.com/thatisuday/2a32f2dd9a94e1a27049875e1cef9039#file-server-js)

const path = require( 'path' );

const express = require( 'express' );

const socketIO = require( 'socket.io' );

// import LED control API

const { toggle } = require( './led-control' );

// create an express app

const app = express();

// send `index.html` from the current directory

// when `http://<ip>:9000/` route is accessed using `GET` method

app.get( '/', ( request, response ) => {

response.sendFile( path.resolve( \_\_dirname, 'web-app/index.html' ), {

headers: {

'Content-Type': 'text/html',

}

} );

} );

// send asset files

app.use( '/assets/', express.static( path.resolve( \_\_dirname, 'web-app' ) ) );

app.use( '/assets/', express.static( path.resolve( \_\_dirname, 'node\_modules/socket.io-client/dist' ) ) );

// server listens on `9000` port

const server = app.listen( 9000, () => console.log( 'Express server started!' ) );

// create a WebSocket server

const io = socketIO( server );

// listen for connection

io.on( 'connection', ( client ) => {

console.log( 'SOCKET: ', 'A client connected', client.id );

// listen to `led-toggle` event

client.on( 'led-toggle', ( data ) => {

console.log( 'Received led-toggle event.' );

toggle( data.r, data.g, data.b ); // toggle LEDs

} );

} );

Во горната датотека server.js, ги додаваме express и socket.io библиотеките. Бидејќи сакаме да ги контролираме GPIO пиновите од оваа датотека исто така треба да ја воведиме и функција toggle од датотеката led-control.js.

Прво, креираме HTTP сервер што не е ништо друго туку express апликација. Функција Еxpress() враќа таква апликација. Потоа треба да поставиме неколку рути за да ги опслужиме .html и други assets датотеки. Користиме app.get ('/', handler) на линија бр. 13,што значи секој што пристапува до нашата веб-апликација со / URL патека, ќе ја добие датотеката index.html сместена во web-app директориумот. Исто така, треба да се погрижиме да испратиме точни заглавија или headers. Следно нешто е то setup assets delivery. Бидејќи на датотекaта index.html и се потребни и main.js и style.css, ќе искористиме стандарден express.js setting to deliver assets.

Датотеката socket.io.js е исто така потребна во веб-апликацијата за комуникација со самиот SocketServer што го градиме. Оваа датотека се наоѓа во директориумот node\_modules / socket.io-client / dist. На линијата бр. 23, ние едноставно го упатуваме express.js да ги бара the assets датотеките во рамките на овој директориум, ако не се најдени во web-app директориумот.

Откако нашата express апликација е правилно конфигурирана, креираме HTTP сервер инстанца од оваа апликација. Со app.listen () повикот на линија бр. 26, HTTP-серверот се стартува на порт 9000. Аpp.listen повикот ја враќа референцата на HTTP-серверот што започна. Ова е она што му треба на нашиот WebSocket сервер.

Креираме WebSocket сервер користејќи ја инстанцата на HTTP серверот со socketIO (сервер) повик на линија бр. 29. За разлика од HTTP серверот, врската со WebSocket серверот е постојана. Оттука, комуникацијата помеѓу клиентот и серверот се случува преку настани(events).

Променливата io го претставува WebSocket серверот. Кога клиент (веб-апликација) е поврзана со овој сервер, (connection event) настанот за конекција е примен од серверот. Овој настан содржи податоци за конекцијата на клиентот, како што е ID на клиентот исто така како и API за комуникација само со поврзаниот клиент. Откако ќе се воспостави врската помеѓу серверот и клиентот, серверот може да емитува и слуша различни /custom настани од и до клиентот. Секој настан може да содржи дополнителни податоци или payload.

Сепак, во горенаведениот пример, ние само го слушаме led-toggle настанот предводен од клиентот. Клиентот го испраќа овој настан кога ќе притисне на копчето во веб-апликацијата. Овој настан содржи информации за состојбата (вклучена или исклучена) на различните копчиња. Врз основа на led-toggle настанот, можеме да ја смениме состојбата на GPIO-пиновите поврзани со RGB LED светлата. Бидејќи веќе имаме API за да се справиме со таа датотека led-control.js, ние едноставно ја повикуваме функцијата toggle кога ќе го прими настанот што се води со вклучување односно led-toggle event.

Based on the payload of led-toggle event, we can toggle the state of GPIO pins connected to the RGB LED lights. Since we have already written a clean API to handle that inside led-control.js file, we are simply calling the toggle function when the led-toggle event is received.

Веб-апликација е обично само веб-страница која обезбедува корисни статични информации или многу богат кориснички интерфејс за комуникација со серверот.  
Во нашиот случај, дури и нашата веб-апликација ќе содржи само некои копчиња на кои треба да кликнеме, ние треба да комуницираме со серверот за да ги емитуваме овие настани со притискање на копчињата. Бидејќи треба да ги испратиме овие настани во реално време, ние користиме WebSocket врска преку традиционална HTTP врска.  
Прво, треба да дизајнираме релативно добар кориснички интерфејс.

[index.html](https://gist.github.com/thatisuday/d3c974ea9f9b4d89f711ea7f18f5e1f1#file-1-index-html)

<!DOCTYPE html>

<html>

<head>

<title>LED IoT Example</title>

<!-- stylesheets -->

<link rel="stylesheet" href="/assets/style.css"/>

<link href="https://fonts.googleapis.com/css?family=Roboto+Condensed:400,700&display=swap" rel="stylesheet">

</head>

<body>

<p class="title">RGB LED IoT Appliation</p>

<div class="buttons-container">

<div class="buttons-container\_\_button buttons-container\_\_button--red" id="button-red">Red</div>

<div class="buttons-container\_\_button buttons-container\_\_button--green" id="button-green">Green</div>

<div class="buttons-container\_\_button buttons-container\_\_button--blue" id="button-blue">Blue</div>

</div>

<!-- scripts -->

<script src="/assets/socket.io.js"></script>

<script src="/assets/main.js"></script>

</body>

</html>

[**style.css**](https://gist.github.com/thatisuday/d3c974ea9f9b4d89f711ea7f18f5e1f1#file-2-style-css)

html, body {

font-size: 14px;

font-family: 'Roboto Condensed', sans-serif;

color: #333333;

}

body {

padding: 30px;

margin: 0px;

}

\* {

box-sizing: border-box;

user-select: none;

}

/\* Application title \*/

.title {

font-size: 24px;

font-weight: 700;

text-transform: uppercase;

letter-spacing: 1px;

margin: 0;

line-height: 1;

}

/\* Buttons container \*/

.buttons-container {

margin-top: 30px;

display: inline-flex;

justify-content: center;

align-items: center;

padding: 30px 10px;

background-color: #dce4e8;

}

.buttons-container\_\_button {

display: flex;

width: 80px;

height: 60px;

border-radius: 3px;

font-size: 16px;

font-weight: 700;

text-transform: uppercase;

color: #ffffff;

margin: 0 20px;

cursor: pointer;

flex: 0 0 auto;

justify-content: center;

align-items: center;

}

.buttons-container\_\_button:hover{

opacity: 0.75;

}

.buttons-container\_\_button--red { background-color: red; }

.buttons-container\_\_button--green { background-color: green; }

.buttons-container\_\_button--blue { background-color: blue; }

Во продолжение е прикажан кодот од index.html датотеката која обезбедува едноставен кориснички интерфејс со три копчиња. (Јас го користев кондензираниот фонт Roboto од серверот фонтови на Google за подобар UI.)

Исто така, треба да го обрнеме внимание на the asset files import. Откако го конфигуриравме express серверот да ги доставува asset files од /assets path , тоа е патеката која ние треба да ја користиме во .html фајлот за да ги импортираме ассетс.

Последното што ни преостанува е да ја имплементираме вистинската логика на клиент-сервер комуникацијата во следната датотеката main.js.

[main.js](https://gist.github.com/thatisuday/f4aad692fca0c26dd9d53baeefc827e6#file-main-js)

// get button elements

var button\_red = document.getElementById( 'button-red' );

var button\_green = document.getElementById( 'button-green' );

var button\_blue = document.getElementById( 'button-blue' );

// initial button states

var button\_red\_state = false;

var button\_green\_state = false;

var button\_blue\_state = false;

// check for active connection

var isConnectionActive = false;

// connect to the Web Socket server

var connection = io( 'http://192.168.1.5:9000' );

// when connection is established

connection.on( 'connect', () => {

isConnectionActive = true;

} );

connection.on( 'disconnect', () => {

isConnectionActive = false;

} );

// WebSocket event emitter function

var emitEvent = function( event ) {

if( ! isConnectionActive ) {

return alert( 'Server connection is closed!' );

}

// change button state

if( event.target.id === 'button-red') { button\_red\_state = ! button\_red\_state; }

if( event.target.id === 'button-green') { button\_green\_state = ! button\_green\_state; }

if( event.target.id === 'button-blue') { button\_blue\_state = ! button\_blue\_state; }

// emit `led-toggle` socket event

connection.emit( 'led-toggle', {

r: button\_red\_state,

g: button\_green\_state,

b: button\_blue\_state,

} );

};

// add event listeners on button

button\_red.addEventListener( 'click', emitEvent );

button\_blue.addEventListener( 'click', emitEvent );

button\_green.addEventListener( 'click', emitEvent );

Во горенаведената JavaScript main.js датотека, создадовме референци до секое индивидуално копче во ДОМ и ја одржуваме состојбата на секое копче (вклучено или исклучено) преку линиите бр. 1 до бр. 9. Со импортирање на датотеката socket.io.js во претходниот index.html, таа ни обезбедува API за комуникација на високо ниво со WebSocket серверот што го изградивме со употреба на socket.io. Тоа го обезбедува интерфејсот io кој се користи за поврзување со далечински/remote SocketServer креиран со употреба на socket.io. Во горенаведениот пример, ја користевме IP адресата на Raspberry Pi. Бидејќи се обидуваме да се поврземе со сервер, треба да го обезбедиме и бројот на портот на тој сервер за да направиме успешна socket конекција.  
  
Откако успешно ќе се креира socket врската, таа содржи API за комуникација со socket серверот. Веднаш штом врската e oстварена, таа го прима настанот за поврзување (connection event) од серверот. Ова е безбедна област за иницијализирање на какви било зависни од конекција (connection dependent) променливи или за испраќање настани до серверот. Во случајот ние ги слушаме настаните со кликнување на копчето и откако ќе биде притиснато копче, ние го емитираме led-toggle настанот до WebSocket серверот на објектот за врска(connection object).

Повикот connection.emit() прима два аргументи. Прво е името на настанот што треба да се емитува и втор е payload-oт или товарот што треба да се испрати заедно со настанот. Структурата на носивоста(payload) на настанот треба да одговара на структурата на носивоста што ја очекува SocketServer (видете ја датотеката server.js).

## Starting the Web Application Server and Socket Server

Now that everything is in place, it’s time to fire up the engine. The only thing we need to do is to run the server.js file with Node.js. We do that using the below command.

$ node server.js

Once you execute this command, it will start an HTTP server on port 9000. It will also start a WebSocket server but this server also runs on the same port. We can open http://<rpi-host>:9000 URL in the browser to access the web application we just built.

Once you open this web application, it will get connected to the WebSocket server. You can add additional logs to main.js (*inside connect and disconnect events listeners*) to check if the socket connection to the server is made successfully or not.

However, if you see the logs printed by the server.js, you should see a log message displaying the id of the connected socket client.

Express server started!  
SOCKET: A client connected XaOGSG9KlEDul37hAAAA

If you see this message in the terminal where node server.js command was executed, then the client-server communication is established. Now you can press on the Red, Green or Blue button in the web application and your Raspberry Pi should toggle the LED lights just fine.

**WebSocket** is a computer [communications protocol](https://en.wikipedia.org/wiki/Communications_protocol), providing [full-duplex](https://en.wikipedia.org/wiki/Full-duplex) communication channels over a single [TCP](https://en.wikipedia.org/wiki/Transmission_Control_Protocol) connection. The WebSocket protocol was standardized by the [IETF](https://en.wikipedia.org/wiki/Internet_Engineering_Task_Force) as [RFC](https://en.wikipedia.org/wiki/Request_for_Comments) 6455 in 2011, and the WebSocket [API](https://en.wikipedia.org/wiki/Application_programming_interface) in [Web IDL](https://en.wikipedia.org/wiki/Web_IDL) is being standardized by the [W3C](https://en.wikipedia.org/wiki/World_Wide_Web_Consortium).

WebSocket is distinct from [HTTP](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol). Both protocols are located at layer 7 in the [OSI model](https://en.wikipedia.org/wiki/OSI_model) and depend on TCP at layer 4. Although they are different, [RFC 6455](https://tools.ietf.org/html/rfc6455) states that WebSocket "is designed to work over HTTP ports 80 and 443 as well as to support HTTP proxies and intermediaries," thus making it compatible with the HTTP protocol. To achieve compatibility, the WebSocket [handshake](https://en.wikipedia.org/wiki/Handshaking) uses the [HTTP Upgrade header](https://en.wikipedia.org/wiki/HTTP/1.1_Upgrade_header)[[1]](https://en.wikipedia.org/wiki/WebSocket#cite_note-1) to change from the HTTP protocol to the WebSocket protocol.

The WebSocket protocol enables interaction between a [web browser](https://en.wikipedia.org/wiki/Web_browser) (or other client application) and a [web server](https://en.wikipedia.org/wiki/Web_server) with lower overhead than half-duplex alternatives such as HTTP polling, facilitating real-time data transfer from and to the server. This is made possible by providing a standardized way for the server to send content to the client without being first requested by the client, and allowing messages to be passed back and forth while keeping the connection open. In this way, a two-way ongoing conversation can take place between the client and the server. The communications are usually done over TCP [port](https://en.wikipedia.org/wiki/Port_(computer_networking)) number 80 (or 443 in the case of [TLS](https://en.wikipedia.org/wiki/Transport_Layer_Security)-encrypted connections), which is of benefit for those environments which block non-web Internet connections using a [firewall](https://en.wikipedia.org/wiki/Firewall_(computing)). Similar two-way browser-server communications have been achieved in non-standardized ways using stopgap technologies such as [Comet](https://en.wikipedia.org/wiki/Comet_(programming)).

WebSocket е протокол за компјутерски комуникации, кој обезбедува целосен дуплекс комуникациски канал преку единствена TCP врска. Протоколот WebSocket е стандардизиран од IETF како RFC 6455 во 2011 година, а WebSocket API во Web IDL се стандардизира од W3C.  
WebSocket се разликува од HTTP. Двата протоколи се наоѓаат на слојот 7 во моделот OSI и зависат од TCP на слојот 4. Иако се различни, RFC 6455 наведува дека WebSocket "е дизајниран да работи преку HTTP порти 80 и 443 како и да поддржува HTTP прокси и посредници, "со што се прави компатибилен со протоколот HTTP. За да се постигне компатибилност, ракувањето на WebSocket го користи заглавието за надградба на HTTP [1] за промена од протоколот HTTP во протоколот WebSocket.  
Протоколот WebSocket овозможува интеракција помеѓу веб-прелистувач (или друга апликација на клиент) и веб-сервер со пониски опции од полу-дуплекс, како што се анкетирање HTTP, олеснување на преносот на податоци во и во серверот во реално време. Ова е овозможено со обезбедување на стандардизиран начин серверот да испраќа содржина до клиентот без претходно да биде побаран од клиентот и дозволувајќи им на пораките да се пренесуваат напред и назад, додека ја одржувате врската отворена. На овој начин, може да се одржи двонасочен тековен разговор помеѓу клиентот и серверот. Комуникациите обично се вршат преку порта TCP број 80 (или 443 во случај на шифрирана врска со TLS), што е од корист за оние средини што блокираат не-веб Интернет врски со користење на заштитен allид. Слични двонасочни комуникации на прелистувач-сервер се постигнати на нестандардизирани начини со користење на застанувачки технологии како што е Комета.

––

### Чекор по чекор

## Setup Nodemon to auto restart Node.js application server

There are several tools you can use to auto restart your Node server after every change so you don’t have to deal with that. My preferred tool is [Nodemon](https://nodemon.io/) which has worked really well for me in my projects.

If you look at the package.json file, you will see that nodemon is listed under the devDependencies, so you can start using it right away.

## Rendering HTML in the Browser

Instead of just sending text to the browser when someone hits a route, we can send some HTML as most websites do. We can author the HTML files by hand and specify what file to send to the browser once a GET request hits a route, but it’s almost always better to use a template engine to generate HTML files on the fly.

### There are several template engines you can use with Express. [Pug](https://pugjs.org/), [Mustache](https://github.com/janl/mustache.js/), and [EJS](http://ejs.co/) are some of the most popular ones. I’ll be using Pug here because I’m comfortable with the syntax but you can do the tutorial in another templating engine if you wish.

### Pug basics

The first thing to know is that Pug relies on indentation to describe the structure of the template and there are no closing tags.

Here’s the basic syntax for Pug that you need to understand to complete this tutorial along with the HTML equivalent in a comment below the Pug code.

You have your element, a space and the contents just like we’ve done above:

### Passing variables in Pug

You can pass information from your route to your template by passing an object when you render the template like this:

### Working with static content

We need to tell express where static files (such as stylesheets, fonts or images) for our website are placed so that it knows how to serve them correctly.

Change your server.js file to look like this:

### Working with JSON data

In the root folder, there is a people.json file which we are going to use to construct the website’s pages. If you inspect it, you will see a profiles key which is an array that contains a few objects each representing a person’s profile.

In a real world application, you will likely fetch this data from a database somewhere, but this method should serve to illustrate the concept well.

Let’s construct the website homepage. We need to pass the json data to our index template and render a card for each person defined within.

Change your server.js file to look like this:

The each keyword in pug allows us to iterate over arrays and objects. Each object in the people array can be accessed under the person key for each iteration and we use that to construct a card for each person on the homepage.

### Making sure everything’s installed correctly

Now that you have Node.js installed, let’s use terminal to make sure everything is installed correctly. Open terminal (or GitBash) and type the following command at the prompt: node-v.

The output of this command should show you the version of Node.js you’ve just installed. Similarly, you can check the version of npm that you’ve installed by running the command npm -v at the command prompt.

NOTE: If your terminal responds with an error or with nothing at all, it’s possible that your installation of Node.js was not successful. In the case of an error, try copying and pasting that error into a search engine to look for common solutions or simply try repeating the installation process.

Now that you have Node.js installed and your terminal running, you need somewhere to write your code. Although text editors come in many different forms and can be used to make non-code files as well, text editors designed specifically for developers often come prepackaged with helpful tools and plugins. I recommend installing the Atom text editor, which you can download at [atom.io](https://atom.io/).

TIP: If you ever forget where you installed Node.js or npm, you can open a command window and type either *which node* or *which npm* at the prompt to see the corresponding location. From a Windows command-line prompt, use *where* in place of *which*.

## Planning Your App

Imagine that you want to build an application for your city’s community-supported agriculture (CSA) club. Through this application, users could subscribe to receive food from local farms and distributors. The application ensures that your community gets healthy food and stays connected. You plan to use Node.js to build this web application and you want to start by verifying users’ zip codes to see if they live close enough for delivery. The question is: will you need to build your own tool to make this possible?

Luckily for us, the answer is no, npm can be used to install Node.js packages, libraries of code others have written that you can use to add specific features to your application. In fact, there’s a package for verifying locations based on zip codes. We’ll take a closer look at that package and how to install it in a little bit.

## Creating a Node.js Module

A Node.js application is ultimately made up of many JavaScript files. For your application to stay organized and efficient, these files need to have access to each other’s contents when necessary. Each file, whose code is collectively related, is called a module. Let’s look at our app again and add some positive messages to it. You can create a file called messages.js with the following code:

let messages = ["You are great!", "You can accomplish anything!", "Success is in your future!"];

Keeping these messages separate from the code you’ll write to display them will make your code more organized. To manage these messages in another file, you need to change the let variable definition to use the exports object, like so:

exports.messages =["You are great!", "You can accomplish anything!", "Success is in your future!"];

Just like other JavaScript objects, you are adding a messages property on the Node.js exports object, which can be shared between modules.

NOTE: The *exports* object is actually a property of the *moduleobject*. *module* is both the name of the code files in Node.js and one of its global objects. Using *exports* is essentially a shorthand for *module.exports*.

[Report Advertisement](https://www.sitepoint.com/build-a-simple-web-server-with-node-js/#report-ad)

The module is ready to be required (imported) by another JavaScript file. You can test this by creating another file called printMessages.js, whose purpose is to loop through the messages and log them to your console with the code in listing 1. First, require the local module by using the require object and the module’s filename (with or without a .js extension). Then, refer to the module’s array by the variable set up in printMessages.js.

**Listing 1. log messages to console in printMessages.js**

const messageModule = require(’./messages’); 1

messageModule.messages.forEach( (m) =&gt; { 2

console.log(m);

});

1. Require the local messages.js module.
2. Refer to the module’s array through messageModule.messages.

require is another Node.js global object used to locally introduce methods and objects from other modules. Node.js interprets require('./messages'); to look for a module called messages.js within your project directory and allow code within printMessages.js to use any properties added to the exports object.

Next, we’ll use npm, another tool for adding modules to your project.

## Running npm Commands

With your installation of Node.js, you also got Node Package Manager (npm). As the name suggests, npm is responsible for managing the external packages (modules others have built and made available online) in your application. Throughout application development, npm will be used to install, remove, and modify these packages. Entering npm -l in your terminal brings up a list of npm commands with brief explanations.

Listing 2 contains a few npm commands that you’ll want to know about.

**Listing 2. Npm commands to know**

* npm init. Initializes a Node.js application and creates a package.json file
* npm install <package>. Installs a Node.js package.
* npm publish. Saves and uploads a package you built to the npm package community.
* npm start. Runs your Node.js application (provided the package.json file is set up to use this command). npm stop will quit the running application.

When using the npm install <package>, appending --save to your command installs the package as a dependency for your application. Appending --global installs the package globally on your computer to be used anywhere within terminal. These command extensions, called flags, have the shorthand forms of -S and -g, respectively. npmuninstall <package> reverses the install action. Should a project call for it, the npm install express -S can be used to install the Express.js framework, and npm install express-generator -g to install the Express.js generator for use as a command-line tool.

### Modules and packages

Throughout your development with Node.js, you’ll hear the terms module and package thrown around a lot. Here’s what you need to know:

* **Modules** are individual JavaScript files containing code that pertains to a single concept, functionality, or library.
* **Packages** may contain multiple modules or a single module. Packages are used to group together files offering relevant tools.

NOTE: “Dependencies” are Node.js modules used by an application or another module. If a package is considered an application dependency, it must be installed (at the version specified by the application) before the application can be expected to run successfully.

If there’s some functionality you’d like to incorporate into your application, there’s likely a package you can find online at npmjs.com to perform that task. Let’s go ahead and add the ability to find where your users are located based on their zip code.

To do this, we’ll need to install the [cities package](https://www.npmjs.com/package/cities), built to convert text addresses into location coordinates. However, we’re still missing one thing from this project before the package can be installed successfully. We’ll need to properly initialize a Node.js project and create a package.json file used by npm for installing cities, which we will do in the next section.

**Quick Check Exercise**: Which flag do you use if you want to install a package globally on your computer?

[Report Advertisement](https://www.sitepoint.com/build-a-simple-web-server-with-node-js/#report-ad)

## Initializing a Node.js Application

Every Node.js application or module will contain a package.json file to define the properties of that particular project. This file lives at the root level of your project. Typically, this file is where you’ll specify the version of your current release, the name of your application, and the main application file. This file is important for npm to save any packages to the node community online.

To get started, create a folder called zip\_connection, navigate to your project directory in terminal, and use the npm init command to initialize your application. You’ll be prompted to fill out the name of your project, the application’s version, a short description, the name of the file from which you’ll start the app (entry point), test files, git repositories, your name (author), and a license code. For now, just be sure to enter your name and press enter to accept all the default options. Once you confirm all these changes, you should see a new package.json file in your project directory resembling the contents of Listing 3.

**Listing 3. Result of your package.json file in your zip\_connection project**

{

"name": "zip\_connection",

"version": "1.0.0",

"description": "An app to promote locally sourced food.",

"main": "printMessages.js",

"scripts": {

"test": "echo \"Error: no test specified\" &amp;&amp; exit 1"

},

"author": "Jon Wexler",

"license": "ISC"

}

This example package.json contains a name, version, description, starting file, custom scripts, an author, and license.

Now your application has a starting point for saving and managing application configurations and packages. You should be able to install cities by navigating to your project folder and running the following in terminal:

npm install cities --save

Also, with this installation, your project folder will have just gained a new folder called node\_modules. The code contents of the cities package you just installed lives in this folder.

TIP: The *--save* flag will save the *cities* package as a dependency for this project. Check your *package.json* file now to see how the package is listed under dependencies. Because your *node\_modules* folder will grow in size, it is not recommended to include it when you share the project code online. However, anyone who downloads the project can enter *npm install* to automatically install all the project dependencies listed in this file.

Test out this new package by adding the lines in Listing 4 to main.js. We’ll start by requiring the locally installed cities package and make it available in this file. Then we’ll use the zip\_lookup method from the cities package to find a city by its zip code. The result is stored in a variable called myCity.

**Listing 4. Implementing the cities package in main.js**

const cities = require(’cities’);

var myCity = cities.zip\_lookup(10016);

console.log(myCity);

The code above does the following:

1. Require the cities package.
2. Store the resulting city from using the zip\_lookup method.
3. Log the results to your console.

The resulting data from that zip code is printed to console as shown in Listing 5.

**Listing 5. Sample result from running the main.js in terminal**

{ zipcode: ’10016’,

state\_abbr: ’NY’,

latitude: ’40.746180’,

longitude: ’-73.97759’,

city: ’New York’,

state: ’New York’ }

The cities zip\_lookup method returns a JavaScript object with coordinates.

[Report Advertisement](https://www.sitepoint.com/build-a-simple-web-server-with-node-js/#report-ad)

**Quick Check Exercise**: Which terminal command initializes a Node.js application with a package.json file?

## Building a Simple Web Server in Node.js

The tech community raves about Node.js and its use of JavaScript as a server-side language, and that’s exactly what we’re going to do now: build it!

This section covers some basic functions of the http module, a Node.js library of code used for handling requests over the internet. Through a few short steps, you’ll convert a few lines of JavaScript into an application with which you can communicate on your web browser. Web servers are the foundation of most Node.js web applications. They allow you to load images and HTML web pages to users of your app.

Let’s consider that application that we’re working on to connect users from the same zip code. Before you deliver the complete application, the community might like to see a simple site with the flexibility to improve and add features in the future. How long do you think it will take you to build a prototype?

With Node.js, you can use the http module to get a web server with sufficient functionality built within hours.

### Understanding web servers

Before getting started, let’s discuss some important web server concepts. After all, the final product will look and feel a lot better if you have clear expectations for the result ahead of time.

#### Web servers and HTTP

A **web server** is software designed to respond to requests over the internet by loading or processing data. Think of a web server like a bank teller, whose job is to process your request to deposit, withdraw, or simply view money in your account. Just as the bank teller follows a protocol to ensure that they process your request correctly, web servers follow the Hypertext Transfer Protocol (HTTP), a standardized system globally observed for the viewing of web pages and sending of data over the internet.

One way a client (your computer) and a server communicate is through the use of HTTP verbs. These verbs indicate what type of request is being made. For example, is the user trying to load a new web page or are they updating information in their profile page? The context of a user’s interaction with an application is an important part of the request–response cycle.

Here are the two most widely used HTTP methods you’ll encounter:

* GET. This method is used to request information from a server. Typically, a server responds with content you can view back on your browser (e.g. clicking a link to see the home page of a site).
* POST. This method is used to send information to the server. A server may respond with an HTML page or redirect you to another page in the application after processing your data (e.g. filling out and submitting a sign-up form).

Consider what happens when you visit google.com. Behind the scenes, you’re actually making a request to Google’s servers, which in turn sends a response back to you, rendering their famous “Google Search” landing page. This request–response relationship allows for a channel of communication between the user and the application. Notice in Figure 3 how a bundle of data is sent to the application’s server in the form of a request and, once the server processes the request, it issues a bundle of data back in the form of a response. This is how most of your interactions on the internet are facilitated.

Figure 3. A web server sends your browser web pages, images and other resources on request

When you enter the URL you want to see in your browser, an HTTP request is sent to a physical computer elsewhere. This request contains some information indicating whether you want to load a web page or send information to that computer.

[Report Advertisement](https://www.sitepoint.com/build-a-simple-web-server-with-node-js/#report-ad)

You might build a fancy application with many bells and whistles, but at the core there is web server, handing its communication on the internet. These concepts will make more sense the more practice you get. For now, let’s start building our web server.

You might build a fancy application with many bells and whistles, but at the core there is web server, handing its communication on the internet. These concepts will make more sense the more practice you get. For now, let’s start building our web server.

**Quick Check Exercise**: What does a web server receive from the client and what does it send back?

### Initialize the application with npm

To get started with a Node.js web application, you first need to initialize the project in your project folder on terminal. Open a terminal window and create a new directory called simple\_server with mkdir. You can initialize the project with npm init.

NOTE: npm stands for Node package manager. Your node projects rely on this tool to install and build Node.js applications.

Running the npm init command initiates a prompt to create a package.json file (Figure 4). As the prompt explains, you’ll walk through configuring the most basic settings of your Node.js application in this file.

### Callbacks in Node.js

Part of what makes Node.js so fast and efficient is its use of callbacks. Callbacks are not new to JavaScript, but they are overwhelmingly used throughout Node.js and are thus worth mentioning.

A **callback** is essentially an anonymous function (a function without a name) that is set up to be invoked as soon as another function completes. The benefit of callbacks is you don’t have to wait for the original function to complete processing before other code is run.

Consider virtually depositing a check to your bank account by uploading a picture to your bank’s mobile app. A callback is equivalent to receiving a notification a couple days later to let you know the check has been verified and deposited. In between that time you were able to go about your normal routine.

In the http web server example, incoming requests from the client are received on a rolling basis, and thereupon pass the request and response as JavaScript objects to a callback function.

Warning: Method names are case sensitive. For example, using createserver() will throw an error.

The createServer() method generates a new instance of http.Server, a built-in Node.js class with tools to evaluate HTTP communication. With this newly created server instance, your app is prepared to receive HTTP requests and send HTTP responses. The argument in createServer() is a callback function that is invoked whenever a certain event occurs within the server. For example, once the server is running and your application’s root URL (home page) is accessed, an HTTP request event will trigger this callback and allow you to run some custom code. In this case, the server will return a simple HTML response.

NOTE: 200 is the HTTP status code for “OK”, used to indicate that there was no issue in returning content in an HTTP response header. To get a list of other HTTP status codes, enter *http.STATUS\_CODES* in the Node.js REPL shell.

Right below that line, you’re writing one line of HTML in the response with write() and closing the response with end(). You must end your response with end() to tell the server you are no longer writing content. Not doing so will leave the connection open to send more in the response and the server will ultimately never consider the response complete.

With this code in place, you’re ready to start your Node.js application from the terminal.

**Quick Check Exercise**: Why should you use const instead of var to store the HTTP server in your application?

## Run the Application

The last step is an easy one: navigate to your project’s directory using terminal and run node main.js in your terminal window. Next, open any browser to the address localhost:3000. You won’t see any indication in the terminal window that the server has started, but you’ll soon see in the browser that everything is working properly. Your terminal window should resemble Figure 5.

## Starting the Web Application Server and Socket Server

Now that everything is in place, it’s time to fire up the engine. The only thing we need to do is to run the server.js file with Node.js. We do that using the below command.

$ node server.js

Once you execute this command, it will start an HTTP server on port 9000. It will also start a WebSocket server but this server also runs on the same port. We can open http://<rpi-host>:9000 URL in the browser to access the web application we just built.

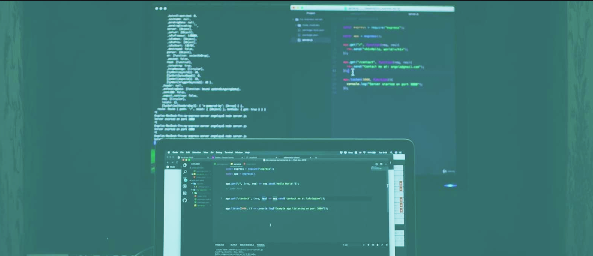
Once you open this web application, it will get connected to the WebSocket server. You can add additional logs to main.js (*inside connect and disconnect events listeners*) to check if the socket connection to the server is made successfully or not.

However, if you see the logs printed by the server.js, you should see a log message displaying the id of the connected socket client.

Express server started!  
SOCKET: A client connected XaOGSG9KlEDul37hAAAA

If you see this message in the terminal where node server.js command was executed, then the client-server communication is established. Now you can press on the Red, Green or Blue button in the web application and your Raspberry Pi should toggle the LED lights just fine.

**4**



# Заклучок

Internet of Things (IoT) is emerging as the next big revolution in the world of technology. It facilitates an environment where web-enabled devices are connected with embedded sensors, communication hardware and processors to disseminate insightful information over the defined network. It involves a range of applications, standards, protocols, architecture and data analysis techniques whereby devices and items are connected to the internet with some business, industrial or human purpose in mind.

IoT is an umbrella term used for a broad range of underlying technologies and services, including:

* Artificial intelligence
* Advanced analytics
* Augmented and virtual reality
* Next-gen cyber security
* Big data
* Cloud computing
* Blockchain, and a lot more.

A fast growing IoT technology market is driving down the hardware costs. Products equipped with IoT sensors can alert the customers when a service is needed. This creates opportunities for the businesses for ongoing customer relationships and creating revenue streams for what previously was a onetime sale.

IoT device deployment provides the data and insights needed to automate processes, streamline workflows, visualize usage patterns and compete more effectively in the changing business landscape. Captured, aggregated and analyzed data are leveraged for several use cases, including real-time monitoring, semi-autonomous and autonomous decisions, new business models and many more.

**IoT network and connectivity shifts**

In the recent years, focus of IoT has shifted from the pure aspect of connecting devices and gathering data to bridging the gap between physical, digital and human spheres through networks and connected processes to turn the insight into knowledgeable actions. The usage of IoT also happens at different speeds. It is far higher in manufacturing industry than the consumer internet of things (CIoT) space.

Internet of Things is an umbrella term, and often distinction is made between the Consumer Internet of Things (CIoT) and Industrial Internet of Things (IIoT). However, both CIoT and IIoT cover many use cases, and there are overlaps between them.

IoT is an essential driver for customer-facing innovation, digital transformation, connectivity of things, bridging the divide between physical and digital, data driven optimization and revenue streams across various business sectors.

Weather

Throughout the human history, people were keen to know about the weather, its parameters and its impacts on their daily lives. By the virtue the technological advancement, in this era, the information about the weather lies in our hands (through mobile phones or websites). We can now make ourselves aware of not only our location’s temperature, humidity etc. but also any part of the world. In this article, we will learn how to make a weather application using pure JavaScript. We will also be familiarizing ourselves with the JSON extraction during this process. So without further ado, let’s get started!

## Prerequisites

In this section, we will discuss the basic components that we need in order to make the weather application.

### Tools

I am going to use the following tools:

* Sublime Text (you can use whatever editor you want)
* XAMPP (Optional; for running the webpage locally; you can run the application by opening the index.html page in the web-browser) If you want to use Sublime Text, go to the following link in order to download it:

https://www.sublimetext.com/3

Make sure to download the latest version!

### Programming Language(s)

For setting up, we would require the following:

* JavaScript (for programming the core of weather app)
* HTML (front-end; for the general client end page)
* CSS (for styles) We will discuss each one of them later in the article, but first, we have to look at the API that is required for getting the weather updates.

### OpenWeatherMap API

For making the weather application, we are going to use the API provided by the following service:

OpenWeatherMap

In order to get the weather updates, we need to sign-up to OpenWeatherMap’s service. For that, go to the following link and sign-up:

https://home.openweathermap.org/users/sign\_up

After signing-up, you should copy the API key into the Text document by going to the following link:

https://home.openweathermap.org/api\_keys

If you do not have any API key there, make sure to generate one. The option of generating the API key is available on the same link given above.

Now that we have done the preliminary steps, it is time to dive into the coding part of this application.

## Designing Weather App

First make the following three files in a folder:

* index.html (file for writing the HTML code)
* index.js (file for writing the JavaScript code)
* style.css (file for prettifying the webpage using CSS)

Once you create the aforementioned files, you now need to add the following

The above code has the following important features:

* src="js/index.js": We added the index.js file right at the end of the body section. In this project, I have made a different JavaScript folder with the name js; however, it is not mandatory.
* The code is divided into two chunks: header, and the main. The parent of both these chunks is the div tag with the class name container.
* Right at the end of the head tag, the css (named style.css) file is linked in order to prettify the HTML.

The code of style.css file is given as follow:

There is only one point that needs our attention regarding the above code:

* In order to centrally align the whole container, we first need to set the value of position as absolute, then left and top are set to 50%, and after that the container is translated towards left by half of its width and towards north by half of its height.

After saving the aforementioned files, the app would look like this:

The above app is divided into three sections:

* The left (lemon coloured) section is meant to display the city name and the icon for the weather condition.
* The upper right (purplish coloured) section is meant to display the temperature of the city.
* The lower right (reddish) section is meant to display the humidity in the city.
* The input **Enter City Name** works in the following two ways: you can type the city name and hit **Enter** using your keyboard, or the other way is that you can type the city name and click the **search** button to get the updates.

## Weather App JavaScript

Now the last file **index.js**, which we have added into our HTML script at the start, needs to be added. The code for the JavaScript file is given as follows:

Following are the important properties and methods of the aforementioned code:

* appKey: App Key is required to get the Weather updates from OpenWeatherMap.
* The format of the URL for the HTTP request is as follows: "<https://api.openweathermap.org/data/2.5/weather?q=>" + searchInput.value + "&appid="+appKey
* In order to parse the JSON response getting from OpenWeatherMap API, JSON.parse method is used. The rest of the code is self-explanatory! Now it’s time to see the DEMO of the Weather App!

As shown in above figure, there are 40output pins for the PI. But when you look at the second figure,

you can see not all 40 pin out can be programmed to our use. These are only 26 GPIO pins which can

be programmed. These pins go from **GPIO2 to GPIO27**.

These **26 GPIO pins can be programmed** as per need. Some of these pins also perform some special

functions, we will discuss about that later. With special GPIO put aside, we have 17 GPIO remaining

(Light green Cirl).

Each of these 17 GPIO pins can deliver a maximum of **15mA current**. And the sum of currents

from all GPIO cannot exceed 50mA. So we can draw a maximum of 3mA in average from each

of these GPIO pins. So one should not tamper with these things unless you know what you are doing.

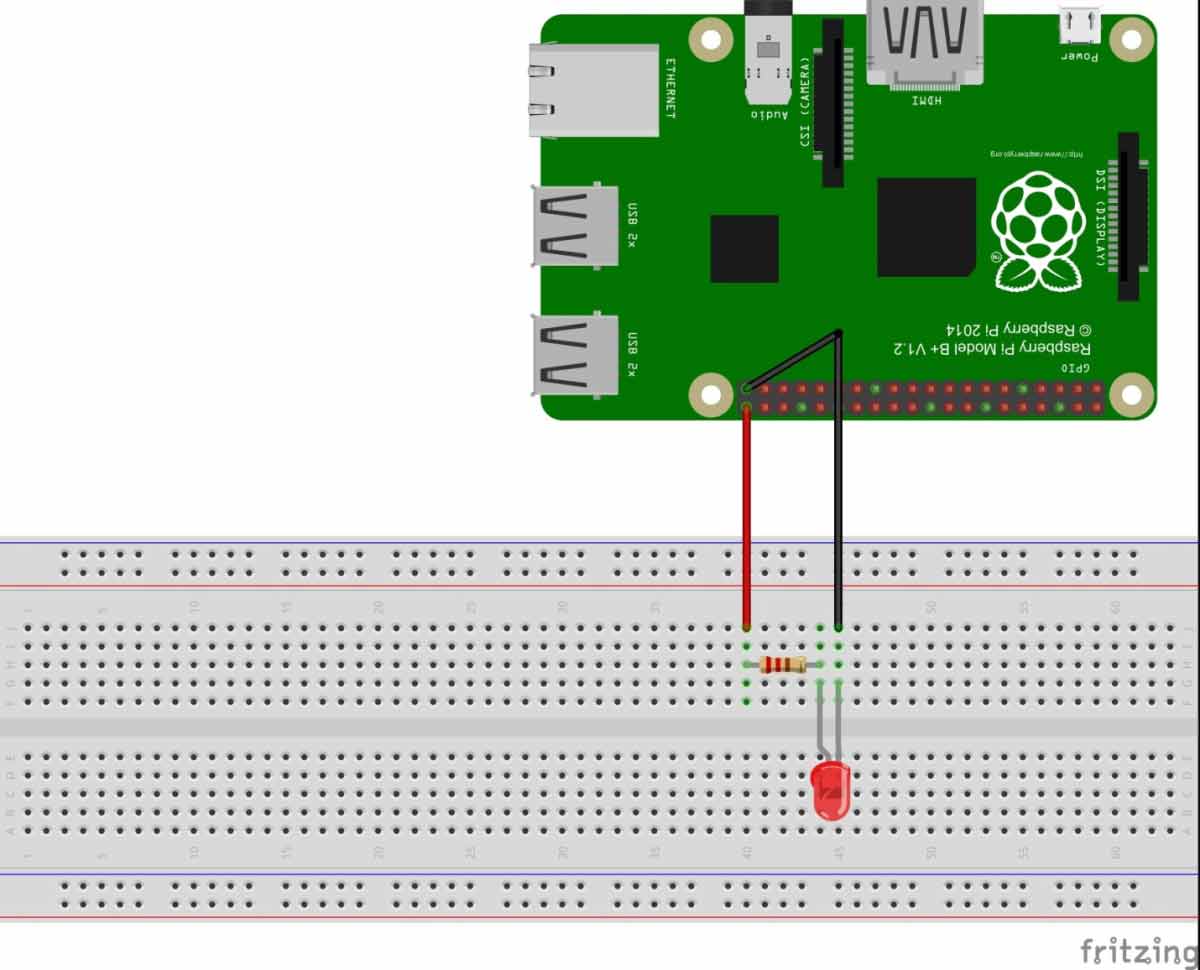
### Components Required:

Here we are using **Raspberry Pi 2 Model B with NOOBS OS**. All the basic Hardware and

Software requirements are previously discussed, you can look it up in the [Raspberry Pi Introduction](http://circuitdigest.com/microcontroller-projects/getting-started-with-raspberry-pi),

other than that we need:

* Connecting pins
* 330Ω or 820KΩresistor
* LED
* Bread Board



As shown in the circuit diagram we are going to connect an LED between PIN40 (GPIO21)

and PIN39 (GROUND).  As said earlier, we cannot draw more than 15mA from any one

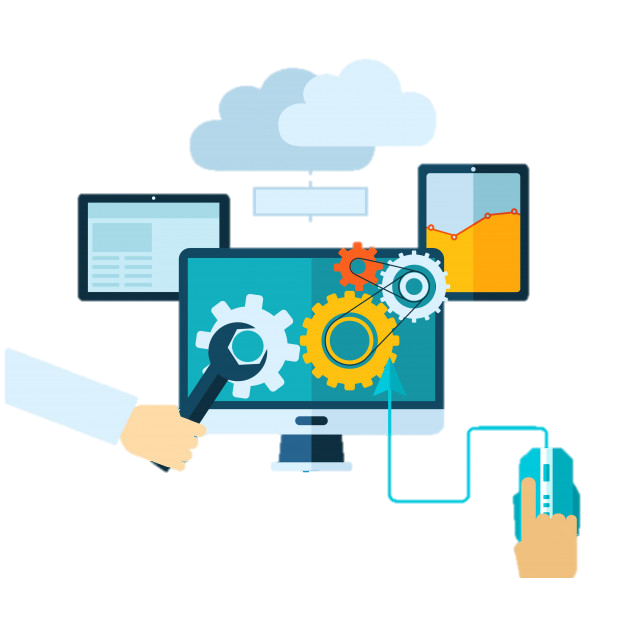
of these pins, so to limit the current we are connecting a 220Ω or 1KΩ resistor in series

with the LED.

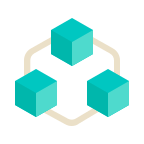


Развој на апликација за инфо диспеј со Raspberry Pi

[*https://www.youtube.com/watch?v=OEilz2Cq\_xY&fbclid=IwAR183OUoPwVzasQ-iiu6\_ps4\_1Ay6c85v9SEVkALk5CeO94qQas5\_NANBDE*](https://www.youtube.com/watch?v=OEilz2Cq_xY&fbclid=IwAR183OUoPwVzasQ-iiu6_ps4_1Ay6c85v9SEVkALk5CeO94qQas5_NANBDE)



[*https://circuitdigest.com/microcontroller-projects/raspberry-pi-led-blinking?fbclid=IwAR2wPOgA5eGDSkS-zEnZw1uPc0Qqhux32CQlumlg-s4qqQpE1RXkiMMjyGk*](https://circuitdigest.com/microcontroller-projects/raspberry-pi-led-blinking?fbclid=IwAR2wPOgA5eGDSkS-zEnZw1uPc0Qqhux32CQlumlg-s4qqQpE1RXkiMMjyGk)



**Introduction**

JavaScript’s rising popularity has brought with it a lot of changes, and the face of web development today is dramatically different. The things that we can do on the web nowadays with JavaScript running on the server, as well as in the browser, were hard to imagine just several years ago, or were encapsulated within sandboxed environments like Flash or Java Applets.

Before digging into [Node.js solutions](https://www.toptal.com/services/nodejs-development), you might want to read up on the benefits of using [JavaScript across the stack](https://www.toptal.com/javascript/guide-to-full-stack-javascript-initjs) which unifies the language and data format (JSON), allowing you to optimally reuse developer resources. As this is more a benefit of JavaScript than Node.js specifically, we won’t discuss it much here. But it’s a key advantage to incorporating Node in your stack.

As Wikipedia states: “Node.js is a packaged compilation of Google’s V8 JavaScript engine, the libuv platform abstraction layer, and a core library, which is itself primarily written in JavaScript.” Beyond that, it’s worth noting that Ryan Dahl, the creator of Node.js, was aiming to create **real-time websites with push capability**, “inspired by applications like Gmail”. In Node.js, he gave developers a tool for working in the non-blocking, event-driven I/O paradigm.

After over 20 years of stateless-web based on the stateless request-response paradigm, we finally have web applications with real-time, two-way connections.

In one sentence: Node.js shines in real-time web applications employing push technology over websockets. What is so revolutionary about that? Well, after over 20 years of stateless-web based on the stateless request-response paradigm, we finally have web applications with real-time, two-way connections, where both the client and server can initiate communication, allowing them to exchange data freely. This is in stark contrast to the typical web response paradigm, where the client always initiates communication. Additionally, it’s all based on the open web stack (HTML, CSS and JS) running over the standard port 80.

One might argue that we’ve had this for years in the form of Flash and Java Applets—but in reality, those were just sandboxed environments using the web as a transport protocol to be delivered to the client. Plus, they were run in isolation and often operated over non-standard ports, which may have required extra permissions and such.

With all of its advantages, Node.js now plays a critical role in the technology stack of many high-profile companies who depend on its unique benefits. The Node.js Foundation has consolidated all the best thinking around why enterprises should consider Node.js in a short presentation that can be found on the [Node.js Foundation’s Case Studies page](https://nodejs.org/en/foundation/case-studies/).

In this Node.js guide, I’ll discuss not only how these advantages are accomplished, but also why you might want to use Node.js—and *why not*—using some of the classic web application models as examples.

## NPM: The Node Package Manager

When discussing Node.js, one thing that definitely should not be omitted is built-in support for [package management using NPM](https://www.toptal.com/javascript/a-guide-to-npm-the-node-package-manager), a tool that comes by default with every Node.js installation. The idea of NPM modules is quite similar to that of Ruby Gems: a set of publicly available, reusable components, available through easy installation via an online repository, with version and dependency management.

A full list of packaged modules can be found on the [npm website](https://npmjs.com/), or accessed using the npm CLI tool that automatically gets installed with Node.js. The module ecosystem is open to all, and anyone can publish their own module that will be listed in the npm repository.

Some of the most useful npm modules today are:

* [**express**](http://expressjs.com) - Express.js—or simply Express—a Sinatra-inspired web development framework for Node.js, and the de-facto standard for the majority of Node.js applications out there today.
* **hapi** - a very modular and simple to use configuration-centric framework for building web and services applications
* [**connect**](http://www.senchalabs.org/connect/) - Connect is an extensible HTTP server framework for Node.js, providing a collection of high performance “plugins” known as middleware; serves as a base foundation for Express.
* [**socket.io**](http://socket.io) and [**sockjs**](https://github.com/sockjs) - Server-side component of the two most common websockets components out there today.
* **pug** (formerly **Jade**) - One of the popular templating engines, inspired by HAML, a default in Express.js.
* [**mongodb**](https://npmjs.org/package/mongodb) and [**mongojs**](https://github.com/gett/mongojs) - MongoDB wrappers to provide the API for MongoDB object databases in Node.js.
* [**redis**](https://github.com/mranney/node_redis) - Redis client library.
* **lodash (underscore, lazy.js)** - The JavaScript utility belt. Underscore initiated the game, but got overthrown by one of its two counterparts, mainly due to [better performance](http://philosopherdeveloper.com/posts/introducing-lazy-js.html) and modular implementation.
* [**forever**](https://npmjs.org/package/forever) - Probably the most common utility for ensuring that a given node script runs continuously. Keeps your Node.js process up in production in the face of any unexpected failures.
* **bluebird** - A full featured Promises/A+ implementation with exceptionally good performance
* **moment** - A JavaScript date library for parsing, validating, manipulating, and formatting dates.

The list goes on. There are tons of really useful packages out there, available to all (no offense to those that I’ve omitted here).

## Examples of Where Node.js Should Be Used

### CHAT

Chat is the most typical real-time, multi-user application. From IRC (back in the day), through many proprietary and open protocols running on non-standard ports, to the ability to implement everything today in Node.js with websockets running over the standard port 80.

The chat application is really the sweet-spot example for Node.js: it’s a lightweight, high traffic, data-intensive (but low processing/computation) application that runs across distributed devices. It’s also a great use-case for learning too, as it’s simple, yet it covers most of the paradigms you’ll ever use in a typical Node.js application.

Let’s try to depict how it works.

In the simplest example, we have a single chatroom on our website where people come and can exchange messages in one-to-many (actually all) fashion. For instance, say we have three people on the website all connected to our message board.

On the server-side, we have a simple Express.js application which implements two things:

1. A GET / request handler which serves the webpage containing both a message board and a ‘Send’ button to initialize new message input, and
2. A websockets server that listens for new messages emitted by websocket clients.

On the client-side, we have an HTML page with a couple of handlers set up, one for the ‘Send’ button click event, which picks up the input message and sends it down the websocket, and another that listens for new incoming messages on the websockets client (i.e., messages sent by other users, which the server now wants the client to display).

When one of the clients posts a message, here’s what happens:

1. Browser catches the ‘Send’ button click through a JavaScript handler, picks up the value from the input field (i.e., the message text), and emits a websocket message using the websocket client connected to our server (initialized on web page initialization).
2. Server-side component of the websocket connection receives the message and forwards it to all other connected clients using the broadcast method.
3. All clients receive the new message as a push message via a websockets client-side component running within the web page. They then pick up the message content and update the web page in-place by appending the new message to the board.

Node.js will still be operating under the same basic principles: reacting to events, handling many concurrent connections, and maintaining fluidity in the user experience.

### API ON TOP OF AN OBJECT DB

Although Node.js really shines with real-time applications, it’s quite a natural fit for exposing the data from object DBs (e.g. MongoDB). JSON stored data allow Node.js to function without the impedance mismatch and data conversion.

For instance, if you’re using Rails, you would convert from JSON to binary models, then expose them back as JSON over the HTTP when the data is consumed by Backbone.js, Angular.js, etc., or even plain jQuery AJAX calls. With Node.js, you can simply expose your JSON objects with a REST API for the client to consume. Additionally, you don’t need to worry about converting between JSON and whatever else when reading or writing from your database (if you’re using MongoDB). In sum, you can avoid the need for multiple conversions by using a uniform data serialization format across the client, server, and database.

### QUEUED INPUTS

If you’re receiving a high amount of concurrent data, your database can become a bottleneck. As depicted above, Node.js can easily handle the concurrent connections themselves. But because database access is a blocking operation (in this case), we run into trouble. The solution is to acknowledge the client’s behavior before the data is truly written to the database.

With that approach, the system maintains its responsiveness under a heavy load, which is particularly useful when the client doesn’t need firm confirmation of a the successful data write. Typical examples include: the logging or writing of user-tracking data, processed in batches and not used until a later time; as well as operations that don’t need to be reflected instantly (like updating a ‘Likes’ count on Facebook) where [eventual consistency](http://www.allthingsdistributed.com/2007/12/eventually_consistent.html) (so often used in NoSQL world) is acceptable.

Data gets queued through some kind of caching or message queuing infrastructure—like RabbitMQ or ZeroMQ—and digested by a separate database batch-write process, or computation intensive processing backend services, written in a better performing platform for such tasks. Similar behavior can be implemented with other languages/frameworks, but not on the same hardware, with the same high, maintained throughput.

In short: with Node, you can push the database writes off to the side and deal with them later, proceeding as if they succeeded.

**DATA STREAMING**

In more traditional web platforms, HTTP requests and responses are treated like isolated event; in fact, they’re actually streams. This observation can be utilized in Node.js to build some cool features. For example, it’s possible to process files while they’re still being uploaded, as the data comes in through a stream and we can process it in an online fashion. This could be done for [real-time audio or video encoding](https://transloadit.com/blog/2010/12/realtime-encoding-over-150x-faster), and proxying between different data sources (see next section).

**PROXY**

Node.js is easily employed as a server-side proxy where it can handle a large amount of simultaneous connections in a non-blocking manner. It’s especially useful for proxying different services with different response times, or collecting data from multiple source points.

An example: consider a server-side application communicating with third-party resources, pulling in data from different sources, or storing assets like images and videos to third-party cloud services.

Although dedicated proxy servers do exist, using Node instead might be helpful if your proxying infrastructure is non-existent or if you need a solution for local development. By this, I mean that you could build a client-side app with a Node.js development server for assets and proxying/stubbing API requests, while in production you’d handle such interactions with a dedicated proxy service (nginx, HAProxy, etc.).

**BROKERAGE - STOCK TRADER’S DASHBOARD**

Let’s get back to the application level. Another example where desktop software dominates, but could be easily replaced with a real-time web solution is brokers’ trading software, used to track stocks prices, perform calculations/technical analysis, and create graphs/charts.

Switching to a real-time web-based solution would allow brokers to easily switch workstations or working places. Soon, we might start seeing them on the beach in Florida.. or Ibiza.. or Bali.

**APPLICATION MONITORING DASHBOARD**

Another common use-case in which Node-with-web-sockets fits perfectly: tracking website visitors and visualizing their interactions in real-time.

You could be gathering real-time stats from your user, or even moving it to the next level by introducing targeted interactions with your visitors by opening a communication channel when they reach a specific point in your funnel. *(If you’re interested, this idea is already being productized by* [*CANDDi*](http://canddi.com/)*.)*

Imagine how you could improve your business if you knew what your visitors were doing in real-time—if you could visualize their interactions. With the real-time, two-way sockets of Node.js, now you can.

**SYSTEM MONITORING DASHBOARD**

Now, let’s visit the infrastructure side of things. Imagine, for example, an SaaS provider that wants to offer its users a service-monitoring page, like GitHub’s status page. With the Node.js event-loop, we can create a powerful web-based dashboard that checks the services’ statuses in an [asynchronous](https://www.toptal.com/javascript/asynchronous-javascript-async-await-tutorial) manner and pushes data to clients using websockets.

Both internal (intra-company) and public services’ statuses can be reported live and in real-time using this technology. Push that idea a little further and try to imagine a [Network Operations Center (NOC)](https://en.wikipedia.org/wiki/Network_operations_center) monitoring applications in a telecommunications operator, cloud/network/hosting provider, or some financial institution, all run on the open web stack backed by Node.js and websockets instead of Java and/or Java Applets.

**Where Node.js Can Be Used**

**SERVER-SIDE WEB APPLICATIONS**

Node.js with Express.js can also be used to create classic web applications on the server-side. However, while possible, this request-response paradigm in which Node.js would be carrying around rendered HTML is not the most typical use-case. There are arguments to be made for and against this approach. Here are some facts to consider:

Pros:

* If your application doesn’t have any CPU intensive computation, you can build it in Javascript top-to-bottom, even down to the database level if you use JSON storage Object DB like MongoDB. This eases development (including hiring) significantly.
* Crawlers receive a fully-rendered HTML response, which is far more SEO-friendly than, say, a Single Page Application or a websockets app run on top of Node.js.

Cons:

* Any CPU intensive computation will block Node.js responsiveness, so a threaded platform is a better approach. Alternatively, you could try scaling out the computation [\*].
* Using Node.js with a relational database is still quite a pain (see below for more detail). Do yourself a favour and pick up any other environment like Rails, Django, or ASP.Net MVC if you’re trying to perform relational operations.

*[\*] An alternative to these CPU intensive computations is to create a highly scalable MQ-backed environment with back-end processing to keep Node as a front-facing ‘clerk’ to handle client requests asynchronously.*

**Where Node.js Shouldn’t Be Used**

**~~SERVER-SIDE WEB APPLICATION W/ A RELATIONAL DB BEHIND~~**

Comparing Node.js with Express.js against Ruby on Rails, for example, there used to be a clean decision in favor of the latter when it came to accessing relational databases like PostgreSQL, MySQL, and Microsoft SQL Server.

Relational DB tools for Node.js were still in their early stages. On the other hand, Rails automatically provides data access setup right out of the box together with DB schema migrations support tools and other Gems (pun intended). Rails and its peer frameworks have mature and proven Active Record or Data Mapper data access layer implementations.[\*]

But things have changed. [Sequelize](http://sequelizejs.com), [TypeORM](https://github.com/typeorm/typeorm), and [Bookshelf](https://github.com/bookshelf/bookshelf) have gone a long way towards becoming mature ORM solutions. It might also be worth checking out [Join Monster](https://github.com/acarl005/join-monster) if you’re looking to generate SQL from GraphQL queries.

*[\*] It’s possible and not uncommon to use Node solely as a front-end, while keeping your Rails back-end and its easy-access to a relational DB.*

**Related:** [The Back End: Using Gatsby.js and Node.js for Static Site Updates](https://www.toptal.com/nodejs/gatsby-js-node-js-static-site-generator-pt-1)

**HEAVY SERVER-SIDE COMPUTATION/PROCESSING**

When it comes to heavy computation, Node.js is not the best platform around. No, you definitely don’t want to build a [Fibonacci computation server in Node.js](http://zef.me/4561/node-js-and-the-case-of-the-blocked-event-loop). In general, any CPU intensive operation annuls all the throughput benefits Node offers with its event-driven, non-blocking I/O model because any incoming requests will be blocked while the thread is occupied with your number-crunching—assuming you’re trying to run your computations in the same Node instance you’re responding to requests with.

As stated previously, Node.js is single-threaded and uses only a single CPU core. When it comes to adding concurrency on a multi-core server, there is some work being done by the Node core team in the form of a cluster module [ref: http://nodejs.org/api/cluster.html]. You can also run several Node.js server instances pretty easily behind a [reverse proxy via nginx](http://blog.argteam.com/coding/hardening-node-js-for-production-part-2-using-nginx-to-avoid-node-js-load/).

With clustering, you should still offload all heavy computation to background processes written in a more appropriate environment for that, and having them communicate via a message queue server like RabbitMQ.

Even though your background processing might be run on the same server initially, such an approach has the potential for very high scalability. Those background processing services could be easily distributed out to separate worker servers without the need to configure the loads of front-facing web servers.

Of course, you’d use the same approach on other platforms too, but with Node.js you get that high reqs/sec throughput we’ve talked about, as each request is a small task handled very quickly and efficiently.

**Conclusion**

We’ve discussed Node.js from theory to practice, beginning with its goals and ambitions, and ending with its sweet spots and pitfalls. When people run into problems with Node, it almost always boils down to the fact that **blocking operations are the root of all evil**—99% of Node misuses come as a direct consequence.

In Node, blocking operations are the root of all evil—99% of Node misuses come as a direct consequence.

Remember: Node.js was never created to solve the compute scaling problem. It was created to solve the I/O scaling problem, which it [does really well](http://www.srirangan.net/2012-05-node-js-critics-which-part-of-event-driven-non-blocking-io-model-you-dont-understand).

Why use Node.js? If your use case does not contain CPU intensive operations nor access any blocking resources, you can exploit the benefits of Node.js and enjoy fast and scalable network applications. Welcome to the real-time web.

### What is Node.js?

Node.js is a server-side JavaScript run-time environment. It's open-source, including Google's V8 engine, libuv for cross-platform compatibility, and a core library. Notably, Node.js does not expose a global "window" object, since it does not run within a browser.

### What is Node.js used for?

Node.js is primarily used for non-blocking, event-driven servers, due to its single-threaded nature. It's used for traditional web sites and back-end API services, but was designed with real-time, push-based architectures in mind.

### What is a web framework?

Web frameworks like Angular and React are libraries that help organize and generate the front-end code that runs in a web browser. They reduce development time, reusing code for common operations. Some web frameworks are "full stack," meaning they also generate the back-end code that runs on a web server.

### Is Node.js a framework?

No, it's an environment, and back-end frameworks run within it. Popular ones include Express.js (or simply Express) for HTTP servers and Socket.IO for WebSocket servers.

### Why is Node.js popular?

Aside from being effective at what it does, Node.js is popular because it has a huge, active, open-source, JavaScript-based ecosystem. Also, it doesn't tend to break compatibility between versions in major ways.

### What is the difference between Node.js and Angular/AngularJS?

Node.js executes JavaScript code in its environment on the server, whereas Angular is a JavaScript framework that gets executed on the client (i.e. within a web browser.)

### Why is Node.js bad?

Node.js, being single-threaded, may be a bad choice for web servers doubling as computational servers, since heavy computation will block the server's responsiveness. However, Node.js itself isn't bad: The technology is quite mature and widely used for many different types of servers.



.

