https://github.com/learning-zone/nodejs-interview-questions

Mit jelent a futási környezet a Node.js -ben?

runtime: nodejs10

A runtime environment szó szerint csak az a környezet, amelyben az alkalmazás fut. Ez leírhatja az alkalmazást futtató hardvert és szoftvert. Mennyi RAM-ra, milyen node-verzióra, milyen operációs rendszerre, mennyi CPU-magra lehet hivatkozni, amikor futási környezetről beszélünk.

Mi a Node.js?

A Node.js egy nyílt forráskódú szerveroldali futási környezet, amely a Chrome V8 JavaScript-motorjára épül. Eseményvezérelt, nem blokkoló (aszinkron) I / O cross-platform runtime environment, amely biztosít skálázható szerveroldali alkalmazások futását JavaScript használatával.

Mi a Node.js process modell?

A Node.js egyetlen process-ben fut, az alkalmazáskód pedig egyetlen szálban fut, és így kevesebb erőforrásra van szüksége, mint más platformokon. A webalkalmazás összes felhasználói kérését egyetlen szál fogja kezelni, és az összes I / O vagy hosszú kérést aszinkron módon hajtja végre egy adott kéréshez. Tehát ennek az egyetlen szálnak nem kell megvárnia a kérelem teljesítését, és szabadon kezelheti a következő kérést. Amikor az aszinkron I / O munka befejeződik, tovább feldolgozza a kérést és elküldi a választ.

Milyen adattípusok vannak a Node.js-ben?

Primitive Types

- String
- Number
- Boolean
- Undefined
- Null
- RegExp
- `Buffer`: A Node.js tartalmaz további, Buffer nevű adattípust (nem érhető el a böngésző JavaScript-jében). A buffert elsősorban bináris adatok tárolására használják, miközben fájlból olvasnak, vagy csomagokat fogadnak a hálózaton keresztül.

Hogyan lehet létrehozni egy egyszerű szervert a Node.js-ben, amely visszaadja a Hello World-t?

01. lépés: Hozzon létre egy projektkönyvtárat

```
mkdir myapp
cd myapp
```

02. lépés: Inicializálja a projektet, és kapcsolja össze az npm-mel

```
npm init
```

Ez létrehoz egy package.json fájlt a myapp mappában. A fájl hivatkozásokat tartalmaz a projektjéhez letöltött összes npm csomagra. A parancs számos dolog megadására szólítja fel. Mindegyiken átjuthat, kivéve ezt:

```
entry point: (index.js)
```

Nevezze át erre:

```
app.js
```

03. lépés: Telepítse az Express-t a myapp könyvtárba:

```
npm install express --save
```

04 lépés: app.js:

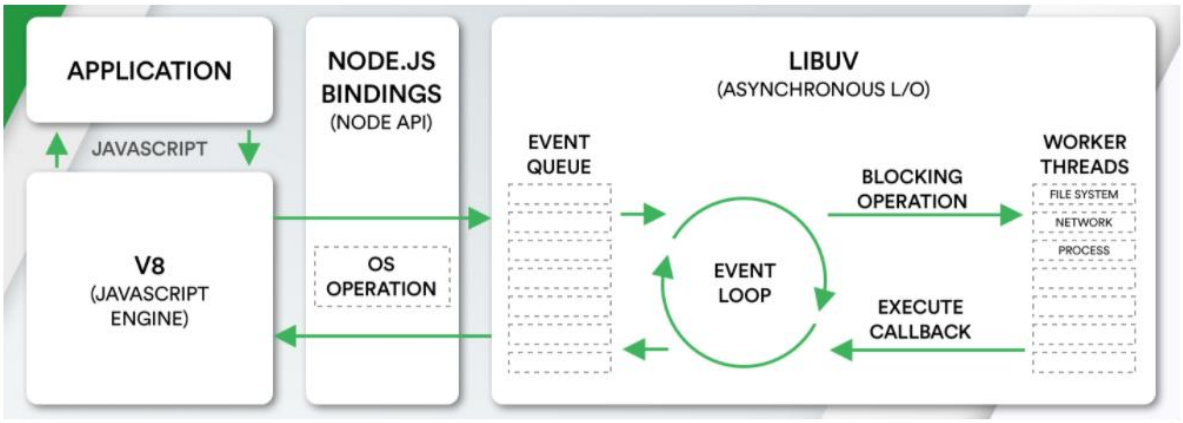
```
var express = require('express');
var app = express();
app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

05. lépés: Futtassa az alkalmazást

```
node app.js
```

Hogyan működik a Node.js?



A Node teljesen eseményvezérelt. Alapvetően a szerver egy szálból áll, amely az egyik eseményt a másik után dolgozza fel.

Az érkező új kérés egyfajta esemény. A szerver elkezdi feldolgozni, és blokkoló IO művelet esetén nem várja meg, amíg befejeződik, és regisztrál egy callback függvényt. Ezután a szerver azonnal elkezd feldolgozni egy másik eseményt (esetleg egy másik kérést). Amikor az IO művelet befejeződött, ez egy másik fajta esemény, és a szerver feldolgozza (azaz tovább dolgozik a kérésen) a callback végrehajtásával, amint ideje van rá.

Tehát a szervernek soha nem kell további szálakat létrehoznia vagy váltania a szálak között, ami azt jelenti, hogy nagyon kevés az általános költsége. Ha több hardvermagot szeretne teljes mértékben kihasználni, akkor egyszerűen indítsa el a node.js több példányát

A Node JS Platform nem követi a többszálas kérés / válasz model-t (Request/Response Multi-Threaded Stateless Model). A Single Threaded with Event Loop Model-t követi. Node JS feldolgozási modell főleg Javascript eseményalapú modellen, Javascript callback mechanizmuson alapul.

Egyszálú eseményhurok-modell feldolgozási lépései:

- Klinesek kérést küldenek a webszerverre.
- A Node JS Web Server belső korlátozott szálkészletet tart fenn, hogy szolgáltatásokat nyújtson az a klinenskérelmeknek.
- A Node JS Web Server fogadja ezeket a kéréseket, és sorba helyezi őket. „Eseménysor (Event Loop)” néven ismert.
- A Node JS webkiszolgálón belül van egy „Event Loop” néven ismert összetevő.
- Az Event Loop csak egyetlen szálat használ. Ez a Node JS Platform Processing Model fő szíve.
- Még a Loop is ellenőrzi, hogy az kliens-kérelmek az Eseménysorba kerülnek-e. Ha nem, akkor határozatlan ideig várja meg a beérkező kéréseket.
- Ha igen, akkor vegyen fel egy kliens kéréstaz eseménysorból
    - Elindítja a kliens kérés folyamatát
    - Ha a kliens kérés nem igényel blokkoló IO-műveleteket, akkor dolgozzon fel mindent, készítsen elő választ, és küldje vissza a kliensnek.

- Ha az kliens kérése bizonyos blokkoló IO műveleteket igényel, például az adatbázissal, a fájlrendszerrel, a külső szolgáltatásokkal való interakciót, akkor más megközelítést fog követni
    - A szálak rendelkezésre állását ellenőrzi a belső szálkészletből
    - Vegyen fel egy szálat, és rendelje hozzá ezt a kliens kérést ahhoz a szálhoz.
    - Ez a szál felelős a kérés fogadásáért, feldolgozásáért, az IO blokkolási műveleteinek végrehajtásáért, a válasz előkészítéséért és visszaküldéséért az Event loop-ba
    - Az Event Loop viszont elküldi a választ az adott kliensnek.

Mi az az error-first callback?

A Node.js összes aszinkron metódusában használt mintát Error-first Callback-nek hívják. Íme egy példa:

```
fs.readFile( "file.json", function ( err, data ) {
  if ( err ) {
    console.error( err );
  }
  console.log( data );
});
```

Bármely aszinkron módszer arra számít, hogy az egyik argumentum callback lesz. A teljes callback argumentum lista a hívó módjától függ, de az első argumentum mindig hibaobjektum vagy null. Amikor az aszinkron metódusra térünk, a függvény végrehajtása során dobott kivétel nem észlelhető a try / catch utasításban. Az esemény azután következik be, hogy a JavaScript motor elhagyta a próbálkozási blokkot.

Az előző példában, ha valamilyen kivétel merül fel a fájl olvasása során, az első és kötelező paraméterként a callback függvényre esik.

Mi az a callback-hell a Node.js-ben?

A callback-hell egy olyan jelenség, amely egy JavaScript-fejlesztőt sújt, amikor megpróbál egymás után több aszinkron műveletet végrehajtani.

Az aszinkron funkció az, ahol valamilyen külső tevékenységnek be kell fejeződnie, mielőtt az eredmény feldolgozható lenne; „aszinkron” abban az értelemben, hogy kiszámíthatatlan idő telik el, mire egy eredmény elérhetővé válik. Az ilyen funkciókhoz callback funkcióra van szükség a hibák kezeléséhez és az eredmény feldolgozásához.

```
getData(function(a){
    getMoreData(a, function(b){
        getMoreData(b, function(c){
            getMoreData(c, function(d){
                      getMoreData(d, function(e){
                                ...
                            });
                  });
```

```
        });
    });
});
```

Techniques for avoiding callback hell

1. Using Async.js
2. Using Promises
3. Using Async-Await

1.) Callback kezelése az Async.js használatával

Az Async egy igazán hatékony npm modul a JavaScript aszinkron jellegének kezelésére. A Node.js-sel együtt a böngészőkhöz írt JavaScript esetében is működik.

Az Async rengeteg hatékony segédprogramot kínál aszinkron folyamatokkal való együttműködéshez különböző esetekben.

```
npm install --save async
```

- **ASYNC WATERFALL**

```
var async = require('async');
async.waterfall([
    function(callback) {
        //doSomething
        callback(null, paramx); //paramx will be availaible as the first parameter to the next
function
        /**
            The 1st parameter passed in callback.
            @null or @undefined or @false control moves to the next function
            in the array
            if @true or @string the control is immedeatly moved
            to the final callback fucntion
            rest of the functions in the array
            would not be executed
        */
    },
    function(arg1, callback) {
        //doSomething else
      // arg1 now equals paramx
        callback(null, result);
    },
    function(arg1, callback) {
        //do More
        // arg1 now equals 'result'
        callback(null, 'done');
    },
    function(arg1, callback) {
        //even more
        // arg1 now equals 'done'
        callback(null, 'done');
    }
], function (err, result) {
    //final callback function
    //finally do something when all function are done.
    // result now equals 'done'
});
```

- **ASYNC SERIES**

```
var async = require('async');
async.series([
    function(callback){
        // do some stuff ...
        callback(null, 'one');
        /**
            The 1st parameter passed in callback.
            @null or @undefined or @false control moves to the next function
            in the array
            if @true or @string the control is immedeatly moved
            to the final callback fucntion with the value of err same as
            passed over here and
            rest of the functions in the array
            would not be executed
        */
    },
    function(callback){
        // do some more stuff ...
        callback(null, 'two');
    }
],
// optional callback
function(err, results){
    // results is now equal to ['one', 'two']
});
```

2.) A callback kezelése az promise segítségével

A promise-ok a callback-ek helyett az aszinkron kóddal foglalkoznak. A promise-ok visszaadják az eredmény értékét vagy egy hibakivételt. A promise-ok lényege a .then () függvény, amely az ígéret tárgyának visszaküldésére vár. A .then () függvény két opcionális függvényt vesz fel argumentumként, és a promise állapotától függően csak az egyiket hívják meg valah. Az első függvény akkor hívódik meg, ha az promise teljesül (sikeres eredmény). A második függvényt akkor hívják meg, amikor a promise-t elutasítják.

```
var outputPromise = getInputPromise().then(function (input) {
    //handle success
}, function (error) {
    //handle error
});
```

3.) Az Async Await használata

Az Async várakozás az aszinkron kódot úgy kezeli, mintha szinkron lenne. Ez csak a promise-ok újbóli bevezetése miatt volt lehetséges. Az Async-Await csak olyan függvényekkel működik, amelyek promise-t adnak vissza.

```
const getrandomnumber = function(){
    return new Promise((resolve, reject)=>{
        setTimeout(() => {
            resolve(Math.floor(Math.random() * 20));
        }, 1000);
    });
}

const addRandomNumber = async function(){
    const sum = await getrandomnumber() + await getrandomnumber();
    console.log(sum);
}
```

```
addRandomNumber();
```

Mik a promise-ok a Node.js-ben?

Lehetővé teszi a kezelők társítását egy aszinkron művelet esetleges sikerértékéhez vagy sikertelenségéhez. Ez lehetővé teszi, hogy az aszinkron módszerek olyan értékeket adjanak vissza, mint a szinkron módszerek.

Az promise-ok a node.js-ben megígérték, hogy elvégeznek némi munkát, majd külön callback hívásokat kaptak, amelyeket a siker és a kudarc, valamint az időkorlátok kezelése érdekében végrehajtanak. A node.js-ben az promise-ok gondolkodásának másik módja az volt, hogy olyan kibocsátók voltak, amelyek csak két eseményt tudtak kibocsátani: sikert és hibát. A promise-ok tulajdonsága, hogy függőségi láncokba vonhatja őket (a C ígéretet csak akkor teheti, amikor az A és a B ígéretet teljes).

Az promise alapgondolata az, hogy az promise egy aszinkron működés eredményét képviseli. Az promise három különböző állapot egyikében van:

- függőben (pending)- Az promise kezdeti állapota.
- teljesítve (fulfilled)- A sikeres műveletet képviselő promise állapota.
- elutasítva (rejected) - A sikertelen műveletet képviselő promise állapota.

Promise létrehozása:

```
var myPromise = new Promise(function(resolve, reject){
   ....
})
```

Milyen eszközökkel lehet biztosítani a következetes stílust?

- ESLint
- Standard

Mikor érdemes az npm-et és mikor a yarn-t használni?

1.) npm:

Ez az alapértelmezett módszer a csomagok kezelésére a Node.js futásidejű környezetben. Parancssori kliensre és az npm nyilvántartás néven ismert, nyilvános és prémium csomagokból álló adatbázisra támaszkodik. A felhasználók a kliensen keresztül férhetnek hozzá a nyilvántartáshoz, és böngészhetnek az npm webhelyen elérhető számos csomagban. Az npm-et és annak nyilvántartását is az npm, Inc. kezeli.

```
node -v
npm -v
```

2.) Yarn:

A Yarn-t a Facebook fejlesztette ki, hogy megpróbálja megoldani az npm néhány hiányosságát. A yarn technikailag nem helyettesíti az npm-et, mivel az npm rendszerleíró adatbázis moduljaira támaszkodik. Gondoljon a yarn-ra, mint új telepítőre, amely továbbra is ugyanazon az npm struktúrán alapszik. Maga a nyilvántartás nem változott, de a telepítési módszer más. Mivel a yarn hozzáférést biztosít ugyanazokhoz a csomagokhoz, mint az npm, az npm-ről a yarn-ra való áttéréshez nem szükséges változtatni a munkafolyamaton.

```
npm install yarn --global
```

A yarn vs npm összehasonlítása:

- Gyors: A yarn minden letöltött csomagot tárol, így soha többé nem kell. Ez párhuzamosítja a műveleteket az erőforrás-kihasználás maximalizálása érdekében, így a telepítés ideje gyorsabb, mint valaha.
- Megbízható: A Yarn részletes, de tömör, lockfile formátumot és determinisztikus algoritmust alkalmazva garantálja, hogy az egy rendszeren működő telepítés ugyanúgy fog működni bármely más rendszeren.
- Biztonságos: A yarn ellenőrző összegekkel ellenőrzi minden telepített csomag integritását a kód futtatása előtt.
- Offline mód: Ha korábban telepített egy csomagot, akkor internetkapcsolat nélkül újra telepítheti.
- Determinisztikus: Ugyanazokat a függőségeket ugyanolyan pontosan telepítik minden gépre, a telepítés sorrendjétől függetlenül.
- Hálózati teljesítmény: A yarn hatékonyan sorba állítja a kéréseket, és elkerüli a kérések vízeséseit a hálózat kihasználásának maximalizálása érdekében.
- Több regiszter: Telepítsen bármelyik csomagot akár az npm-től, akár a Bower-től, és a csomag munkafolyama változatlan marad.
- Hálózati rugalmasság: Egyetlen kérelem meghiúsulása nem okozza a telepítés meghiúsulását. A kéréseket sikertelenség esetén újrapróbáljuk.
- Flat mód: Oldja meg a függőségek nem egyező verzióit egyetlen verzióval, hogy elkerülje a másolatok létrehozását.

Mi a stub?

A node.js tesztek nyomozása és ellenőrzése. Lehetővé teszi az egymásba ágyazott kóddarabok, például a metódusok, a required () és az npm modulok vagy akár az osztályok példányainak viselkedésének ellenőrzését és felülbírálását. Ezt a könyvtárat a node-gently, MockJS and mock-require ihlette.

A Stub jellemzői:

- könnyűsúlyú objektumokat állít elő
- Kompatibilis a Nodejs-sel

- Könnyen bővíthető közvetlenül vagy az ExtensionManager segítségével
- Előre definiált, használható kiterjesztésekkel érkezik

A Stub-ok olyan funkciók / programok, amelyek a komponensek / modulok viselkedését szimulálják. A stub.ok válaszokat adnak a tesztesetek során indított függvény hívásokra. Azt is megerősítheti, hogy hívták ezeket a stub-okat.

Felhasználási eset lehet olvasott fájl, amikor nem akar aktuális fájlt olvasni:

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

var readFileStub = sinon.stub(fs, 'readFile', function (path, cb) {
  return cb(null, 'filecontent');
});

expect(readFileStub).to.be.called;
readFileStub.restore();
```

Hogyan védheti meg a HTTP cookie-kat az XSS támadásokkal szemben?

1. Amikor a webszerver sütiket állít be, további attribútumokat adhat meg annak biztosítására, hogy rosszindulatú JavaScript használatával a sütik ne legyenek elérhetők. Az egyik ilyen tulajdonság a HttpOnly.

```
Set-Cookie: [name]=[value]; HttpOnly
```

A HttpOnly biztosítja, hogy a cookie-kat csak arra a domainre küldjék be, ahonnan származnak.

2. A " Secure" attribútum megbizonyosodhat arról, hogy a cookie-kat csak a biztonságos csatornán továbbítják.

```
Set-Cookie: [name]=[value]; Secure
```

3. A webszerver használhatja az X-XSS-Protection válaszfejlécet, hogy megbizonyosodjon arról, hogy az oldalak nem töltődnek be, ha észlelik reflected cross-site scripting (XSS) támadásait.

```
X-XSS-Protection: 1; mode=block
```

4. A webszerver a HTTP Content-Security-Policy válaszfejléc segítségével szabályozhatja, hogy a felhasználói kliensek milyen erőforrásokat tölthessenek be egy adott oldalra. Segíthet megelőzni a különféle típusú támadásokat, például a Cross Site Scripting (XSS) és az adatinjekciós támadásokat.

```
Content-Security-Policy: default-src 'self' *.http://sometrustedwebsite.com
```

Hogyan lehet megbizonyosodni arról, hogy függőségei biztonságosak-e?

Az egyetlen lehetőség a függőségek frissítésének / biztonsági ellenőrzésének automatizálása. Ehhez vannak ingyenes és fizetett lehetőségek:

1. npm outdated
2. Trace by RisingStack
3. NSP
4. GreenKeeper
5. Snyk
6. npm audit
7. npm audit fix

Mi az Event loop a Node.js-ben? És hogyan működik?

Az eseményhurok az, amely lehetővé teszi a Node.js számára, hogy nem blokkoló I / O műveleteket hajtson végre - annak ellenére, hogy a JavaScript egyszálú - a műveleteknek a rendszermagba történő terelésével, amikor csak lehetséges.

A Node.js egyszálú alkalmazás, de támogatja az egyidejűséget az esemény és a callback fogalmán keresztül. A Node.js minden API-je aszinkron, és egyszálú, aszinkron függvényhívásokat használnak az egyidejűség fenntartására. A Node observer mintát használ. A Node szál megtartja az eseményhurkot (event loop), és amikor egy feladat befejeződik, elindítja a megfelelő eseményt, amely jelzi az esemény-figyelő (event-listener) funkció végrehajtását.

1.) Eseményvezérelt programozás (Event-Driven Programming)

Eseményvezérelt alkalmazásban általában van egy fő ciklus, amely figyeli az eseményeket, majd callback függvényt indít el, amikor az egyik eseményt észleli.

Bár az események meglehetősen hasonlítanak a callback-ekre, a különbség abban rejlik, hogy a callback függvényeket akkor hívják meg, amikor egy aszinkron függvény visszaadja eredményét, míg az eseménykezelés az observer mintán működik. Az eseményeket figyelő függvények Observer-ként működnek. Amikor egy esemény elindul, a listener függvény elindul. A Node.js több beépített eseményt kínál, amelyek az eseménymodulon és az EventEmitter osztályon keresztül érhetők el, és amelyek az események és az eseményfigyelők (event-listeners) összekapcsolására szolgálnak az alábbiak szerint

```
// Import events module
var events = require('events');

// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();
```

Example:

```
// Import events module
var events = require('events');

// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();

// Create an event handler as follows
var connectHandler = function connected() {
   console.log('connection succesful.');
```

```
   // Fire the data_received event
   eventEmitter.emit('data_received');
}

// Bind the connection event with the handler
eventEmitter.on('connection', connectHandler);

// Bind the data_received event with the anonymous function
eventEmitter.on('data_received', function() {
   console.log('data received succesfully.');
});

// Fire the connection event
eventEmitter.emit('connection');

console.log("Program Ended.");
```

Mi az a REPL? Milyen célra használják?

A REPL (READ, EVAL, PRINT, LOOP) a Shell (Unix / Linux) és a parancssorhoz hasonló számítógépes környezet. A Node telepítve van a REPL környezettel. A rendszer a felhasznált parancsok / kifejezések kimenetein keresztül kölcsönhatásba lép a felhasználóval. Hasznos a kódok írásakor és hibakeresésében. A REPL munkája teljes formájából megérthető:

- Read: Beolvassa a felhasználók inputjait, és elemzi a JavaScript adatstruktúrájába. Ezután a memóriába kerül.
- Eval: Az elemzett JavaScript adatszerkezetet kiértékelik az eredmények alapján.
- Print: Az eredmény kiértékelődik.
- Loop: Ismétli a bemeneti parancsot. A NODE REPL kijutásához nyomja meg kétszer a ctrl + c billentyűkombinációt

```
$ node
> 10 + 20
30
> 10 + ( 20 * 30 ) - 40
570
>
```

Mi a különbség az aszinkron (Asynchronous) és a nem blokkoló (Non-blocking) között?

1. Aszinkron (Asynchronous)

Az aszinkron felépítése megmagyarázza, hogy az elküldött üzenet nem azonnal adja meg a választ, mint ahogy mi is küldjük a levelet, de a választ nem kapjuk meg azonnal. Nincs függősége vagy rendje. Ezért javítja a rendszer hatékonyságát és teljesítményét. A szerver tárolja az információt, és a művelet végrehajtása után értesítést kap.

2. Nem blokkoló (Non-blocking)

Nem blokkoló (Non-blocking) azonnal válaszol bármilyen rendelkezésre álló adattal. Ezenkívül nem blokkol semmilyen végrehajtást, és a kéréseknek megfelelően tovább fut. Ha a választ nem sikerült lekérni, akkor ezekben az esetekben az API azonnal hibával tér vissza. A nem blokkolást többnyire I /

O-val (bemenet / kimenet) használják. A Node.js maga a nem blokkoló I / O modellen alapszik. Kevés olyan kommunikációs mód van, amelyet egy nem blokkoló I / O befejezett. A callback függvényt a művelet befejeztével kell meghívni. A nem blokkoló hívás a javascript segítségét használja, amely callback függvényt biztosít.

3.) Aszinkron VS nem blokkoló

Az Asynchronous nem reagál azonnal, míg a Nonblocking azonnal reagál, ha rendelkezésre állnak az adatok, és ha nem, akkor egyszerűen hibát ad vissza.

Az Asynchronous javítja a hatékonyságot azáltal, hogy gyorsan elvégzi a feladatot, mivel a válasz később érkezhet, miközben más feladatokat is elvégezhet. A non-blocking nem blokkol semmilyen végrehajtást, és ha rendelkezésre állnak az adatok, gyorsan lekéri az információkat.

Az aszinkron a szinkron ellentéte, míg a nem blokkoló I / O a blokkolás ellentéte. Mindkettő meglehetősen hasonló, de különböznek egymástól is, mivel az aszinkron tágabb működési tartományban használatos, míg a nem blokkolást többnyire I / O-val.

Hogyan lehet debug-olni egy alkalmazást a Node.js-ben?

- **node-inspector**

```
npm install -g node-inspector
```

Run

```
node-debug app.js
```

- **Debugging**
    - Debugger
    - Node Inspector
    - Visual Studio Code
    - Cloud9
    - Brackets
- **Profiling**

```
1. node --prof ./app.js
2. node --prof-process ./the-generated-log-file
```

- **Heapdumps**
    - node-heapdump with Chrome Developer Tools
- **Tracing**
    - Interactive Stack Traces with TraceGL
- **Logging**
  Libraries that output debugging information
    - Caterpillar

- Tracer
- scribbles

Libraries that enhance stack trace information

- Longjohn

Melyek a Node.js legnépszerűbb moduljai?

- Async: Az Async egy segédprogram modul, amely egyszerű, hatékony funkciókat biztosít az aszinkron JavaScript használatához.
- Browserify: A Browserify rekurzívan elemzi az összes require() hívást az alkalmazásában annak érdekében, hogy olyan csomagot hozzon létre, amelyet egyetlen <script> címkében szolgálhat a böngésző számára.
- Bower: A Bower az internet csomagkezelője. Úgy működik, hogy az egész területről behozza és telepíti a csomagokat, gondoskodik a vadászatról, a keresett dolgok megkereséséről, letöltéséről és mentéséről.
- Backbone: A Backbone.js struktúrát ad a webalkalmazásoknak azáltal, hogy kulcsérték-összerendeléssel és egyedi eseményekkel, számtalan függvényt tartalmazó gazdag API-val rendelkező gyűjteményeket, deklaratív eseménykezeléssel rendelkező nézeteket kínál, és mindezt RESTful JSON felületen keresztül összekapcsolja a meglévő API-val.
- A Csv: csv modul négy almodullal rendelkezik, amelyek CSV generálást, elemzést, átalakítást és sorosítást biztosítanak a Node.js számára.
- Debug: A Debug egy aprócska node.js hibakereső segédprogram, amely a node core hibakeresési technikája alapján készült.
- Express: Az Express egy gyors, minimalista web keretrendszer. Kicsi, robusztus eszközöket biztosít a HTTP szerverek számára, ez nagyszerű megoldást jelent egyoldalas alkalmazásokhoz, weboldalakhoz, vagy nyilvános HTTP API-khoz.
- Forever: Egyszerű CLI eszköz annak biztosítására, hogy egy adott node szkript folyamatosan (azaz örökké) futjon.
- Grunt: egy JavaScript Task Runner, amely megkönnyíti az új projektek létrehozását, és olyan ismétlődő, de szükséges feladatok elvégzését, mint például a linting, unit testing, concatenating and minifying files.
- Gulp: egy streaming build rendszer, amely segít a nehézkes vagy időigényes feladatok automatizálásában a fejlesztési munkafolyamatban.
- Hapi: egy streaming build rendszer, amely segít a nehézkes vagy időigényes feladatok automatizálásában a fejlesztési munkafolyamatban.
- Http-server: egy egyszerű, nulla konfigurációjú parancssori http-kiszolgáló. Elég hatékony a production használatához, de elég egyszerű. Helyi fejlesztésre és tanulásra lehessen használni.
- Inquirer: Gyakori interaktív parancssori felhasználói felületek gyűjteménye.
- Jquery: A jQuery egy gyors, kicsi és funkciókban gazdag JavaScript könyvtár.
- Jshint: Statikus elemző eszköz a JavaScript-kód hibáinak és lehetséges problémáinak felderítésére, valamint a csapat kódolási konvencióinak kikényszerítésére.

- Koa: A Koa egy webalkalmazás-keret. A node.js kifejező HTTP köztes szoftvere a webalkalmazások és az API-k élvezetesebbé tételéhez.
- Lodash: A lodash könyvtár node modulként exportálva. A Lodash egy modern JavaScript segédkönyvtár, amely modularitást, teljesítményt és extrákat nyújt.
- Less: Less library node modul-ként exportált.
- Moment: JavaScript dátumkönyvtár a dátumok elemzéséhez, érvényesítéséhez, kezeléséhez és formázásához.
- Mongoose: Ez egy MongoDB objektum-modellező eszköz, amelyet aszinkron környezetben történő működésre terveztek.
- MongoDB: A Node.js hivatalos MongoDB illesztőprogramja. Magas szintű API-t biztosít a mongodb-core tetején, amelyet a végfelhasználóknak szánnak.
- Npm: a javascript csomagkezelője.
- Nodemon: Ez egy egyszerű monitor szkript, amely a node.js alkalmazás fejlesztése során használható. Meg fogja nézni a fájlokat abban a könyvtárban, amelyben a nodemon elindult, és ha bármely fájl megváltozik, a nodemon automatikusan újraindítja a node alkalmazást.
- Nodemailer: Ez a modul lehetővé teszi az e-mail küldését a Node.js alkalmazásokból.
- Optimist: egy node.js könyvtár az argv hash-val történő elemzéshez.
- Phantomjs: A PhantomJS NPM telepítője, JS API-val. Gyors és natív támogatást nyújt a különféle webes szabványokhoz: DOM kezelés, CSS választó, JSON, Canvas és SVG.
- Passport: Egyszerű, nem feltűnő hitelesítési köztes szoftver a Node.js fájlhoz. A Passport a stratégiákat használja a kérelmek hitelesítésére. A stratégiák a felhasználónév és jelszó hitelesítő adatok ellenőrzésétől vagy az OAuth vagy OpenID használatával történő hitelesítésig terjedhetnek.
- Q: A Q a promise-ok tárháza. A promise olyan objektum, amely a visszatérési értéket vagy a dobott kivételt képviseli, amelyet a függvény végül adhat.
- Request: A Request egyszerűsített HTTP kérés kliens lehetővé teszi a http hívások kezdeményezését. Támogatja a HTTPS-t, és alapértelmezés szerint követi az átirányításokat.
- Socket.io: Ez egy node.js valós idejű keretrendszer-kiszolgáló.
- Sails: API-alapú keretrendszer valós idejű alkalmazások készítéséhez, MVC-egyezmények felhasználásával (az Express és a Socket.io alapján)
- Through: Lehetővé teszi a stream elkészítését. Könnyű olyan stream létrehozása, amely egyszerre olvasható és írható.
- Underscore: Az Underderscore.js egy segéd-könyvtár a JavaScript-hez, amely támogatást nyújt a szokásos funkciókra (each, map, reduce, filter...) anélkül, hogy kiterjesztenék az alapvető JavaScript objektumokat.
- Validator: Nodejs modul a string validátorok és sanitizer-ekhez.
- Winston: multi-transport async logging library a Node.js-hez
- Ws: Egyszerűen használható, gyors és alaposan tesztelt websocket kliens, szerver és konzol a node.js fájlhoz
- Xml2js: Egyszerű XML-JavaScript objektum-átalakító.
- Yo: CLI eszköz a Yeoman generátorok futtatásához
- Zmq: A node.js és az io.js kötései a ZeroMQ-hoz. Ez egy nagy teljesítményű aszinkron üzenetküldő könyvtár, amelyet elosztott vagy egyidejű alkalmazásokban kívánnak használni.

Mi az EventEmitter a Node.js-ben?

Minden eseményt kibocsátó objektum az EventEmitter osztály tagja. Ezek az objektumok egy eventEmitter.on () függvényt tárnak fel, amely lehetővé teszi egy vagy több függvény csatolását az objektum által kibocsátott elnevezett eseményekhez.

Amikor az EventEmitter objektum eseményt bocsát ki, az adott eseményhez csatolt összes függvényt szinkron módon hívják meg. A hívott listener-ek által visszaküldött összes értéket figyelmen kívül hagyják, és elvetik.

Example:

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

// listener #1
var listner1 = function listner1() {
   console.log('listner1 executed.');
}

// listener #2
var listner2 = function listner2() {
   console.log('listner2 executed.');
}

// Bind the connection event with the listner1 function
eventEmitter.addListener('connection', listner1);

// Bind the connection event with the listner2 function
eventEmitter.on('connection', listner2);

var eventListeners = require('events').EventEmitter.listenerCount
   (eventEmitter,'connection');
console.log(eventListeners + " Listner(s) listening to connection event");

// Fire the connection event
eventEmitter.emit('connection');

// Remove the binding of listner1 function
eventEmitter.removeListener('connection', listner1);
console.log("Listner1 will not listen now.");

// Fire the connection event
eventEmitter.emit('connection');

eventListeners = require('events').EventEmitter.listenerCount(eventEmitter,'connection');
console.log(eventListeners + " Listner(s) listening to connection event");

console.log("Program Ended.");
```

Now run the main.js

```
$ node main.js
```

Output

```
2 Listner(s) listening to connection event
listner1 executed.
listner2 executed.
Listner1 will not listen now.
listner2 executed.
1 Listner(s) listening to connection event
Program Ended.
```

Hányféle adatfolyam található a node.js-ben?

A stream-ek olyan objektumok, amelyek segítségével folyamatosan olvashat adatokat egy forrásból, vagy adatokat írhat a célállomásra. Négyféle stream létezik

- Readable - Stream, amelyet olvasási műveletekhez használnak.
- Writable - Íráshoz használt adatfolyam.
- Duplex - Stream, amely egyaránt használható olvasási és írási műveletekhez.
- Transform - A duplex adatfolyam típusa, ahol a kimenetet a bemenet alapján számítják ki.

A Stream minden típusa egy EventEmitter-példány, és több eseményt különböző időpontokban dob.

data - Ez az esemény akkor indul el, ha rendelkezésre állnak adatok olvashatóak.

end - Ez az esemény akkor indul el, ha nincs több olvasnivaló adat.

error - Ez az esemény akkor indul el, ha hiba történt az adatok fogadásakor vagy írásakor.

finish - Ez az esemény akkor indul el, amikor az összes adat flushed.

Reading from a Stream

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

// Create a readable stream
var readerStream = fs.createReadStream('input.txt');

// Set the encoding to be utf8.
readerStream.setEncoding('UTF8');

// Handle stream events --> data, end, and error
readerStream.on('data', function(chunk) {
   data += chunk;
});

readerStream.on('end',function() {
   console.log(data);
});

readerStream.on('error', function(err) {
   console.log(err.stack);
});

console.log("Program Ended");
```

Writing to a Stream

```
var fs = require("fs");
var data = 'Simply Easy Learning';

// Create a writable stream
var writerStream = fs.createWriteStream('output.txt');

// Write the data to stream with encoding to be utf8
writerStream.write(data,'UTF8');

// Mark the end of file
writerStream.end();

// Handle stream events --> finish, and error
writerStream.on('finish', function() {
   console.log("Write completed.");
});
```

```
writerStream.on('error', function(err) {
   console.log(err.stack);
});

console.log("Program Ended");
```

Piping the Streams

A piping egy olyan mechanizmus, ahol az egyik adatfolyam kimenetét bemenetként biztosítjuk egy másik adatfolyamhoz. Általában arra szolgál, hogy adatokat gyűjtsön egy adatfolyamból, és az adatfolyam kimenetét továbbítsa egy másik adatfolyamhoz. A csővezeték-üzemeltetésnek nincs korlátja.

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

// Create a readable stream
var readerStream = fs.createReadStream('input.txt');

// Create a writable stream
var writerStream = fs.createWriteStream('output.txt');

// Pipe the read and write operations
// read input.txt and write data to output.txt
readerStream.pipe(writerStream);

console.log("Program Ended");
```

Chaining the Streams

A láncolás (chaining) olyan mechanizmus, amely összeköti az egyik adatfolyam kimenetét egy másik folyammal, és létrehozza a több adatfolyam műveletét. Általában csővezeték műveleteknél használják.

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

// Compress the file input.txt to input.txt.gz
fs.createReadStream('input.txt')
   .pipe(zlib.createGzip())
   .pipe(fs.createWriteStream('input.txt.gz'));

console.log("File Compressed.");
```

Mi a titkosítás a Node.js-ben? Hogyan titkosítja a Node.js biztonságos adatait?

A Node.js Crypto modul támogatja a titkosítást. Kriptográfiai funkciókat kínál, amelyek wrapper-eket tartalmaznak a nyitott SSL HMAC hash-jához, titkosításához, megfejtéséhez, aláírásához és ellenőrzéséhez.

Hash: A hash egy rögzített hosszúságú bitekből álló sztring, determinisztikusan, a forrásadatok tetszőleges blokkjából keletkezik.

HMAC: A HMAC rövidítés alapú üzenet-hitelesítési kódot jelent. Ez egy eljárás hash algoritmus alkalmazására mind az adatokra, mind pedig egy titkos kulcsra, amely egyetlen végső kivonatot eredményez.

Titkosítási példa Hash és HMAC használatával

```
const crypto = require('crypto');
const secret = 'abcdefg';
const hash = crypto.createHmac('sha256', secret)
                   .update('Welcome to JavaTpoint')
                   .digest('hex');
console.log(hash);
```

Titkosítási példa a Cipher segítségével

```
const crypto = require('crypto');
const cipher = crypto.createCipher('aes192', 'a password');
var encrypted = cipher.update('Hello JavaTpoint', 'utf8', 'hex');
encrypted += cipher.final('hex');
console.log(encrypted);
```

Dekódolási példa a Decipher segítségével

```
const crypto = require('crypto');
const decipher = crypto.createDecipher('aes192', 'a password');
var encrypted = '4ce3b761d58398aed30d5af898a0656a3174d9c7d7502e781e83cf6b9fb836d5';
var decrypted = decipher.update(encrypted, 'hex', 'utf8');
decrypted += decipher.final('utf8');
console.log(decrypted);
```

Mi a DNS modul használata a Node.js-ben?

A DNS egy olyan node modul, amelyet névfeloldási lehetőség elvégzésére használnak, amelyet az operációs rendszer biztosít, valamint egy tényleges DNS-kereséshez. Nincs szükség IP-címek memorizálására - a DNS-kiszolgálók remek megoldást kínálnak a tartomány- vagy aldomainnevek IP-címekké történő átalakítására. Ez a modul aszinkron hálózati wrapper-t biztosít, és a következő szintaxissal importálható.

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

Example: `dns.lookup()` function

```
const dns = require('dns');
dns.lookup('www.google.com', (err, addresses, family) => {
  console.log('addresses:', addresses);
  console.log('family:',family);
});
```

Example: `resolve4()` and `reverse()` functions

```
const dns = require('dns');
dns.resolve4('www.google.com', (err, addresses) => {
  if (err) throw err;
  console.log(`addresses: ${JSON.stringify(addresses)}`);
  addresses.forEach((a) => {
    dns.reverse(a, (err, hostnames) => {
      if (err) {
        throw err;
      }
      console.log(`reverse for ${a}: ${JSON.stringify(hostnames)}`);
    });
  });
});
```

Example: print the localhost name using `lookupService()` function

```
const dns = require('dns');
dns.lookupService('127.0.0.1', 22, (err, hostname, service) => {
  console.log(hostname, service);
    // Prints: localhost
});
```

Milyen biztonsági mechanizmusok érhetők el a Node.js fájlban?

A Helmet modul használata

A Helmet számos HTTP header beállításával segíti az Express alkalmazások biztonságát, például:

- X-Frame-Options to mitigates clickjacking attacks,
- Strict-Transport-Security to keep your users on HTTPS,
- X-XSS-Protection to prevent reflected XSS attacks,
- X-DNS-Prefetch-Control to disable browsers DNS prefetching

```
const express = require('express')
const helmet = require('helmet')
const app = express()

app.use(helmet())
```

Felhasználói bevitel ellenőrzése

A felhasználói adatok ellenőrzése az egyik legfontosabb tennivaló az alkalmazás biztonsága szempontjából. Ha nem teszi meg helyesen, az alkalmazások és a felhasználók számos támadást nyithatnak meg, beleértve a command injection, SQL injection or stored cross-site scripting

A felhasználói bevitel ellenőrzéséhez az egyik legjobb könyvtár, amelyet kiválaszthat, a joi. A Joi egy objektum sémaleírás nyelve és a JavaScript objektumok ellenőrzője.

```
const Joi = require('joi');

const schema = Joi.object().keys({
    username: Joi.string().alphanum().min(3).max(30).required(),
    password: Joi.string().regex(/^[a-zA-Z0-9]{3,30}$/),
    access_token: [Joi.string(), Joi.number()],
    birthyear: Joi.number().integer().min(1900).max(2013),
    email: Joi.string().email()
}).with('username', 'birthyear').without('password', 'access_token')

// Return result
const result = Joi.validate({
    username: 'abc',
    birthyear: 1994
}, schema)
// result.error === null -> valid
```

A reguláris kifejezések biztonsága

A reguláris kifejezések nagyszerű módja a szövegek manipulálásának és a szükséges részek megszerzésének. Van azonban egy olyan támadásvektor, amelyet a Regular Expression Denial of

Service támadásnak neveznek, ami feltárja azt a tényt, hogy a legtöbb reguláris kifejezés megvalósítás extrém helyzetekbe kerülhet a speciálisan kialakított bemeneteknél, amelyek rendkívül lassan működnek.

Azok a reguláris kifejezések, amelyek ilyesmire képesek, általában Evil Regexes néven szerepelnek.

Példák a Evil Regexes kifejezések mintáira:

```
(a+)+
([a-zA-Z]+)*
(a|aa)+
```

Security.txt

A Security.txt meghatároz egy szabványt, amely segít a szervezeteknek meghatározni a biztonsági kutatók számára a biztonsági rések biztonságos feltárásának folyamatát.

```
const express = require('express')
const securityTxt = require('express-security.txt')

const app = express()

app.get('/security.txt', securityTxt({
  // your security address
  contact: 'email@example.com',
  // your pgp key
  encryption: 'encryption',
  // if you have a hall of fame for securty resourcers, include the link here
  acknowledgements: 'http://acknowledgements.example.com'
}))
```

Nevezze meg az API függvények típusait a Node.js-ben?

Kétféle API-függvény létezik a Node.js-ben:

- Aszinkron, nem blokkoló függvények
- Szinkron, blokkoló függvények

1. Blokkoló függvények

Blokkolási műveletben az összes többi kód végrehajtását blokkolják, amíg be nem következik egy várakozó I / O esemény. A blokkoló függvények szinkron módon futnak.

```
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
console.log(data);
// moreWork(); will run after console.log
```

A második kódsor addig blokkolja a további JavaScript futtatását, amíg a teljes fájlt be nem olvassa. A moreWork () csak a Console.log után lesz meghívva

2. Nem blokkoló funkciók

Nem blokkoló műveletben több I / O hívás hajtható végre a program végrehajtásának leállítása nélkül. A nem blokkoló funkciók aszinkron módon hajtanak végre.

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
  console.log(data);
});
// moreWork(); will run before console.log
```

Mivel az fs.readFile () nem blokkoló, a moreWork () szolgáltatásnak nem kell megvárnia az olvasott fájl befejezését, mielőtt meghívná. Ez nagyobb teljesítményt tesz lehetővé.

Hogyan kezeli a Node.js a gyermekszálakat?

A Node.js egyetlen szálú nyelv, amely a háttérben több szálat használ az aszinkron kód végrehajtásához. A Node.js nem blokkoló, ami azt jelenti, hogy az összes függvényt (callbacks) delegálják az eseményhurokhoz, és ezeket különböző szálak hajtják végre (vagy hajthatják). Ezt a Node.js run-time kezeli.

- A Nodejs Primary alkalmazás eseményhurokban fut, amely egyetlen szálban van.
- A háttér I / O olyan szálkészletben fut, amely csak a C / C ++ vagy más fordított / natív modulok számára érhető el, és többnyire átlátszó a JS számára.
- A v11 / 12 csomópont most kísérleti worker_threads-el rendelkezik, ami egy másik lehetőség.
- A Node.js támogatja a több folyamat elágazását (amelyeket különböző magokon hajtanak végre).
- Fontos tudni, hogy az állapot nincs megosztva a master és a forked folyamat között.
- Átadhatjuk az üzeneteket a forked folyamatnak (ami más szkript), és a folyamatot elágazó folyamatból kezelhetjük függvényküldéssel.

Mi a preferált módszer a Node.js-ben a nem kezelt kivételek feloldására?

A Node.js-ben szereplő kezeletlen kivételeket a folyamat szintjén lehet elkapni, ha csatol egy kezelőt az uncaughtException eseményhez.

```
process.on('uncaughtException', function(err) {
    console.log('Caught exception: ' + err);
});
```

A folyamat egy globális objektum, amely információkat nyújt a jelenlegi Node.js folyamatról. A folyamat olyan figyelő funkció, amely mindig figyeli az eseményeket.

Néhány esemény:

1. Exit

2. disconnect
3. unhandledException
4. rejectionHandled

Hogyan támogatja a Node.js a többprocesszoros platformokat, és teljes mértékben kihasználja-e az összes processzor erőforrását?

Mivel a Node.js alapértelmezés szerint egyetlen szálú alkalmazás, egyetlen processzormagon fog futni, és nem fogja teljes mértékben kihasználni a többmagos erőforrásokat. A Node.js azonban támogatást nyújt a többmagos rendszerek telepítéséhez, a hardver nagyobb előnyeinek kihasználása érdekében. A Cluster modul az egyik alapvető Node.js modul, és lehetővé teszi több Node.js worker process futtatását, amelyek ugyanazt a portot használják.

A Cluster modul segít új folyamatok létrehozásában az operációs rendszerben. Minden folyamat önállóan működik, ezért nem használhatja a megosztott állapotot a gyermekfolyamatok között. Minden folyamat IPC-vel kommunikál a fő folyamattal

A Cluster kétféle terheléselosztást támogat:

- A fő folyamat hallgatja a portot, elfogadja az új kapcsolatot és hozzárendeli egy gyermek folyamathoz round robin elv alapján.
- A fő folyamat a portot hozzárendeli egy gyermek folyamathoz, és a gyermek folyamat maga figyeli a portot.

Mi általában az első argumentum, amelyet a Node.js callback handler-nek adnak át?

Bármely callback handler első argumentuma opcionális hibaobjektum

```
function callback(err, results) {
    // usually we'll check for the error before handling results
    if(err) {
        // handle error somehow and return
    }
    // no error, perform standard callback handling
}
```

Hogyan olvassa be a Node.js egy fájl tartalmát?

A Node.js "normális" módja valószínűleg egy fájl tartalmának beolvasása nem blokkoló, aszinkron módon. Vagyis szóljon a Node-nak, hogy olvassa be a fájlt, majd adjon callback-et, amikor a fájlolvasás befejeződött. Ez lehetővé tenné számunkra, hogy több kérést párhuzamosan nyújtsunk be.

A fájlrendszer modul általános használata:

- Read files
- Create files

- Update files
- Delete files
- Rename files

Read Files

index.html

```
<html>
<body>
  <h1>My Header</h1>
  <p>My paragraph.</p>
</body>
</html>
```

```
// read_file.js

var http = require('http');
var fs = require('fs');
http.createServer(function (req, res) {
  fs.readFile('index.html', function(err, data) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(data);
    res.end();
  });
}).listen(8080);
```

Initiate read_file.js:

```
node read_file.js
```

Mi a JIT és hogyan kapcsolódik a Node.js-hez?

A Node.js a V8 JavaScript-motortól függ, hogy biztosítsa a kód végrehajtását a nyelven. A V8 egy JavaScript motor, amelyet a google fejlesztőközpontban fejlesztettek Németországban. Nyílt forráskódú és C ++ nyelven íródott. Kliens oldali (Google Chrome) és szerver oldali (node.js) JavaScript alkalmazásokhoz egyaránt használható. A V8 motor központi része, amely lehetővé teszi a JavaScript nagy sebességű futtatását, a JIT (Just In Time) fordító. Ez egy dinamikus fordító, amely optimalizálni tudja a kódot futás közben. A V8 első kiadása a JIT Compiler FullCodegen névre hallgatott. Ezután a V8 csapata implementálta a Crankshaft-ot, amely számos teljesítményoptimalizálást tartalmazott, amelyeket a FullCodegen nem hajtott végre.

A V8-at először úgy fejlesztették ki, hogy növelje a JavaScript futtatásának teljesítményét a böngészőkben. A sebesség elérése érdekében a V8 fordító használata helyett hatékonyabb gépi kódra fordítja a JavaScript kódot. A JavaScript kódot futás közben állítja össze egy JIT (Just-In-Time) fordító megvalósításával, mint sok modern JavaScript motor, például a SpiderMonkey vagy a Rhino (Mozilla). A fő különbség a V8-mal szemben az, hogy nem hoz létre bájtkódot vagy bármilyen köztes kódot.

Mi a különbség a put és a patch között?

A PUT és a PATCH HTTP metódusok, mindkettő egy erőforrás frissítésére vonatkozik. Egy PUT kérésben az entitás a szerveren tárolt erőforrás módosított verziójának tekinthető, és a kliens a tárolt verzió cseréjét kéri.

A PATCH használatával azonban a entitás utasításokat tartalmaz, amelyek leírják, hogy a szerveren jelenleg tartózkodó erőforrást hogyan kell módosítani egy új verzió előállításához.

Egy másik különbség az, hogy amikor egy erőforrást PUT kéréssel kíván frissíteni, akkor a teljes adatot kérésként, míg a PATCH segítségével csak azokat a paramétereket küldi el, amelyeket frissíteni szeretne.

A POST, GET, PUT, DELETE HTTP leggyakrabban használt metódusok hasonlóak az adatbázis CRUD (Create, Read, Update and Delete) műveleteihez. Tehát az alábbiakban összehasonlítjuk őket.

- `POST` - create
- `GET` - read
- `PUT` - update
- `DELETE` - delete

PATCH: Részleges módosítást nyújt be egy erőforráshoz. Ha csak egy mezőt kell frissítenie az erőforráshoz, akkor érdemes a PATCH módszert használni.

Felsorolja a Node.js által támogatott Http-kérések típusait?

A HTTP core modul a Node.js networking kulcsmodulja.

```
const http = require('http')
```

http.METHODS

```
require('http').METHODS
[ 'ACL',
  'BIND',
  'CHECKOUT',
  'CONNECT',
  'COPY',
  'DELETE',
  'GET',
  'HEAD',
  'LINK',
  'LOCK',
  'M-SEARCH',
  'MERGE',
  'MKACTIVITY',
  'MKCALENDAR',
  'MKCOL',
  'MOVE',
  'NOTIFY',
  'OPTIONS',
  'PATCH',
  'POST',
  'PROPFIND',
  'PROPPATCH',
  'PURGE',
  'PUT',
  'REBIND',
  'REPORT',
  'SEARCH',
  'SUBSCRIBE',
```

```
  'TRACE',
  'UNBIND',
  'UNLINK',
  'UNLOCK',
  'UNSUBSCRIBE' ]
```

http.STATUS_CODES

```
require('http').STATUS_CODES
{ '100': 'Continue',
'101': 'Switching Protocols',
'102': 'Processing',
'200': 'OK',
'201': 'Created',
'202': 'Accepted',
'203': 'Non-Authoritative Information',
'204': 'No Content',
'205': 'Reset Content',
'206': 'Partial Content',
'207': 'Multi-Status',
'208': 'Already Reported',
'226': 'IM Used',
'300': 'Multiple Choices',
'301': 'Moved Permanently',
'302': 'Found',
'303': 'See Other',
'304': 'Not Modified',
'305': 'Use Proxy',
'307': 'Temporary Redirect',
'308': 'Permanent Redirect',
'400': 'Bad Request',
'401': 'Unauthorized',
'402': 'Payment Required',
'403': 'Forbidden',
'404': 'Not Found',
'405': 'Method Not Allowed',
'406': 'Not Acceptable',
'407': 'Proxy Authentication Required',
'408': 'Request Timeout',
'409': 'Conflict',
'410': 'Gone',
'411': 'Length Required',
'412': 'Precondition Failed',
'413': 'Payload Too Large',
'414': 'URI Too Long',
'415': 'Unsupported Media Type',
'416': 'Range Not Satisfiable',
'417': 'Expectation Failed',
'418': 'I\'m a teapot',
'421': 'Misdirected Request',
'422': 'Unprocessable Entity',
'423': 'Locked',
'424': 'Failed Dependency',
'425': 'Unordered Collection',
'426': 'Upgrade Required',
'428': 'Precondition Required',
'429': 'Too Many Requests',
'431': 'Request Header Fields Too Large',
'451': 'Unavailable For Legal Reasons',
'500': 'Internal Server Error',
'501': 'Not Implemented',
'502': 'Bad Gateway',
'503': 'Service Unavailable',
'504': 'Gateway Timeout',
'505': 'HTTP Version Not Supported',
'506': 'Variant Also Negotiates',
'507': 'Insufficient Storage',
'508': 'Loop Detected',
'509': 'Bandwidth Limit Exceeded',
'510': 'Not Extended',
'511': 'Network Authentication Required' }
```

Making HTTP Requests

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

request('https://nodejs.org/', function(err, res, body) {
    console.log(body);
});
```

Az első argumentum lehet URL-karakterlánc vagy opció objektum. Íme néhány a leggyakoribb lehetőségek közül, amelyekkel találkozhat:

- url: A HTTP-kérés URL-je
- method: Használandó HTTP metódus (GET, POST, DELETE stb.)
- headers: A kérésben beállítandó HTTP fejlécek (kulcs-érték) objektuma
- form: Kulcs- értéket tartalmazó form objektum

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

const options = {
    url: 'https://nodejs.org/file.json',
    method: 'GET',
    headers: {
        'Accept': 'application/json',
        'Accept-Charset': 'utf-8',
        'User-Agent': 'my-reddit-client'
    }
};

request(options, function(err, res, body) {
    let json = JSON.parse(body);
    console.log(json);
});
```

Az Options objektum használatával ez a kérés a GET metódussal használja a JSON-adatok közvetlen lekérését a Reddit-ből, amelyet karakterláncként ad vissza a törzsmezőbe. Innen használhatja a JSON.parse metódust és az adatokat normál JavaScript objektumként használhatja.

Miért érdemes használni az Express.js-t?

Az ExpressJS egy előre felépített NodeJS keretrendszer, amely segítséget nyújt a szerver-oldali webalkalmazások gyorsabb és intelligensebb létrehozásában. Az egyszerűség, a minimalizmus, a rugalmasság, a skálázhatóság néhány jellemzője, és mivel magában a NodeJS-ben készült.

Az Express 3.x egy könnyűsúlyú (light-weight) webalkalmazás-keretrendszer, amely segíti a webalkalmazás szerveroldalon történő MVC architektúrába rendezését. Ezután használhat olyan adatbázist, mint a MongoDB és a Mongoose, hogy háttérszolgáltatást nyújtson a Node.js alkalmazáshoz. Az Express.js alapvetően segít minden kezelésében, az útvonalaktól (routes) kezdve a kérelmek és nézetek kezeléséig.

A node.js szabványos szerver keretrendszerré vált. Az Express a MEAN stack része. A MEAN egy ingyenes és nyílt forráskódú JavaScript szoftver dinamikus weboldalak és webalkalmazások készítéséhez, amely a következő összetevőket tartalmazza:

- MongoDB - A szokásos NoSQL adatbázis
- Express.js - Az alapértelmezett webalkalmazás-keretrendszer
- Angular.js - A webes alkalmazásokhoz használt JavaScript MVC keretrendszer
- Node.js - skálázható szerveroldali és hálózati alkalmazásokhoz használt keretrendszer.

Az Express.js keretrendszer nagyon megkönnyíti egy olyan alkalmazás kifejlesztését, amely felhasználható többféle kérés kezelésére, mint például a GET, PUT, POST és DELETE kérések.

using Express

```
var express=require('express');
var app=express();
app.get('/',function(req,res) {
  res.send('Hello World!');
});
var server=app.listen(3000,function() {});
```

Írja le az Express JS alkalmazás beállításának lépéseit?

1. Install Express Generator

```
C:\node>npm install -g express-generator
```

2. Create an Express Project

```
C:\node>express --view="ejs" nodetest1
```

3. Edit Dependencies

Rendben, most van néhány alapvető struktúránk, de még nem vagyunk készen. Megjegyezzük, hogy az express-generator rutin létrehozta a package.json nevű fájlt a nodetest1 könyvtárban. Nyissa meg ezt egy szövegszerkesztőben, és így fog kinézni:

```
// C:\node\nodetest1\package.json
{
  "name": "nodetest1",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "cookie-parser": "~1.4.3",
    "debug": "~2.6.9",
    "ejs": "~2.5.7",
    "express": "~4.16.0",
    "http-errors": "~1.6.2",
    "morgan": "~1.9.0"
  }
}
```

Ez egy alapvető JSON fájl, amely leírja az alkalmazásunkat és annak függőségeit. Hozzá kell tennünk néhány dolgot hozzá. Konkrétan a MongoDB-t és a Monk hívásokat.

```
C:\node\nodetest1>npm install --save monk@^6.0.6 mongodb@^3.1.13
```

4. Install Dependencies

```
C:\node\nodetest1>npm install
C:\node\nodetest1>npm start
```

Node Console

```
> nodetest1@0.0.0 start C:\node\nodetest1
> node ./bin/www
```

Mivel a Node egyetlen szálú folyamat, hogyan lehet felhasználni az összes CPU-t?

A Node.js egyetlen szálú nyelv, amely a háttérben több szálat használ az aszinkron kód végrehajtásához. A Node.js nem blokkoló, ami azt jelenti, hogy az összes függvényt (callback) delegálják az eseményhurokhoz, és ezeket különböző szálak hajtják végre (vagy hajthatják végre). Ezt a Node.js run-time kezeli.

- A Node.js támogatja több folyamat elágazását (amelyeket különböző magokon hajtanak végre).
- Fontos tudni, hogy az állapot nincs megosztva a master és a forked folyamat között.
- Átadhatjuk az üzeneteket a fork-olt folyamatnak (ami más szkript), és a folyamatot elágazó folyamatból kezelhetjük függvényküldéssel.

A Node.js egyetlen példánya egyetlen szálban fut. A többmagos rendszerek előnyeinek kihasználása érdekében a felhasználó néha a Node.js folyamatok klaszterét akarja elindítani a terhelés kezelésére. A cluster modul lehetővé teszi a gyermekprocesszek egyszerű létrehozását, amelyek mindegyike megosztja a kiszolgáló portjait.

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  console.log(`Master ${process.pid} is running`);

  // Fork workers.
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} died`);
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
  }).listen(8000);

  console.log(`Worker ${process.pid} started`);
}
```

A Node.js futtatása most megosztja a 8000-es portot a worker-ek között:

```
$ node server.js
```

```
Master 3596 is running
Worker 4324 started
Worker 4520 started
Worker 6056 started
Worker 5644 started
```

A worker folyamatokat a child_process.fork () metódus segítségével hozzák létre, hogy az IPC-n keresztül kommunikálhassanak a szülővel,

A cluster modul két módszert támogat a bejövő kapcsolatok elosztására.

Az első (és az alapértelmezett minden platformon, a Windows kivételével) a round-robin megközelítés, ahol a master process hallgatja a portot, új kéréseket fogad el és round.robin módon elosztja a worker-ek között, néhány beépített funkcióval, hogy elkerülje a worker folyamat túlterhelését.

A második megközelítés az, amikor a master folyamat létrehozza a listen socketet és elküldi a worker-eknek. A worker-ek ezután közvetlenül fogadják a bejövő kapcsolatokat.

Mit csinál az emitter és mi a diszpécser?

A Node.js core API aszinkron eseményvezérelt architektúrán alapul, amelyben bizonyos típusú kibocsátóknak (emitter) nevezett objektumok rendszeresen olyan eseményeket bocsátanak ki, amelyek listener objektumok meghívását okozzák.

Minden eseményt kibocsátó objektum az EventEmitter osztály tagja. Ezek az objektumok egy eventEmitter.on () függvényt tárnak fel, amely lehetővé teszi egy vagy több függvény csatolását az objektum által kibocsátott elnevezett eseményekhez.

Amikor az EventEmitter objektum eseményt bocsát ki, az adott eseményhez csatolt összes függvényt szinkron módon hívják meg. A hívott listener-ek által visszaküldött összes értéket figyelmen kívül hagyják, és elvetik.

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();
myEmitter.on('event', function(a, b) {
  console.log(a, b, this);
  // Prints:
  //   Technoetics Club MyEmitter {
  //     domain: null,
  //     _events: { event: [Function] },
  //     _eventsCount: 1,
  //     _maxListeners: undefined }
});
myEmitter.emit('event','Technoetics', 'Club');
```

Itt létrehozunk egy myEmitter objektumot, és a végén eseményt bocsátunk ki, amely elindítja a callback függvényt, és képesek vagyunk megszerezni a kívánt kimenetet.

Alapértelmezés szerint az adott eseményobjektumhoz csatolt összes listener-t az EventListener objektum szinkron módon hívja meg abban a sorrendben, amelyben regisztrálták vagy csatolták őket az eseményobjektumhoz.

Diszpécser

A diszpécser olyan funkcióival rendelkezik, amelyek nem biztosítottak és nem várhatók az EventEmitter programban, a legjelentősebb a waitFor, amely lehetővé teszi az store számára, hogy egy művelet hatására egy másik store frissítése megtörténjen, mielőtt tovább haladna.

Mintázat szerint a diszpécser is singleton, míg az EventEmitter egy olyan API, amelyet objektumként rendelhet több store-hoz.

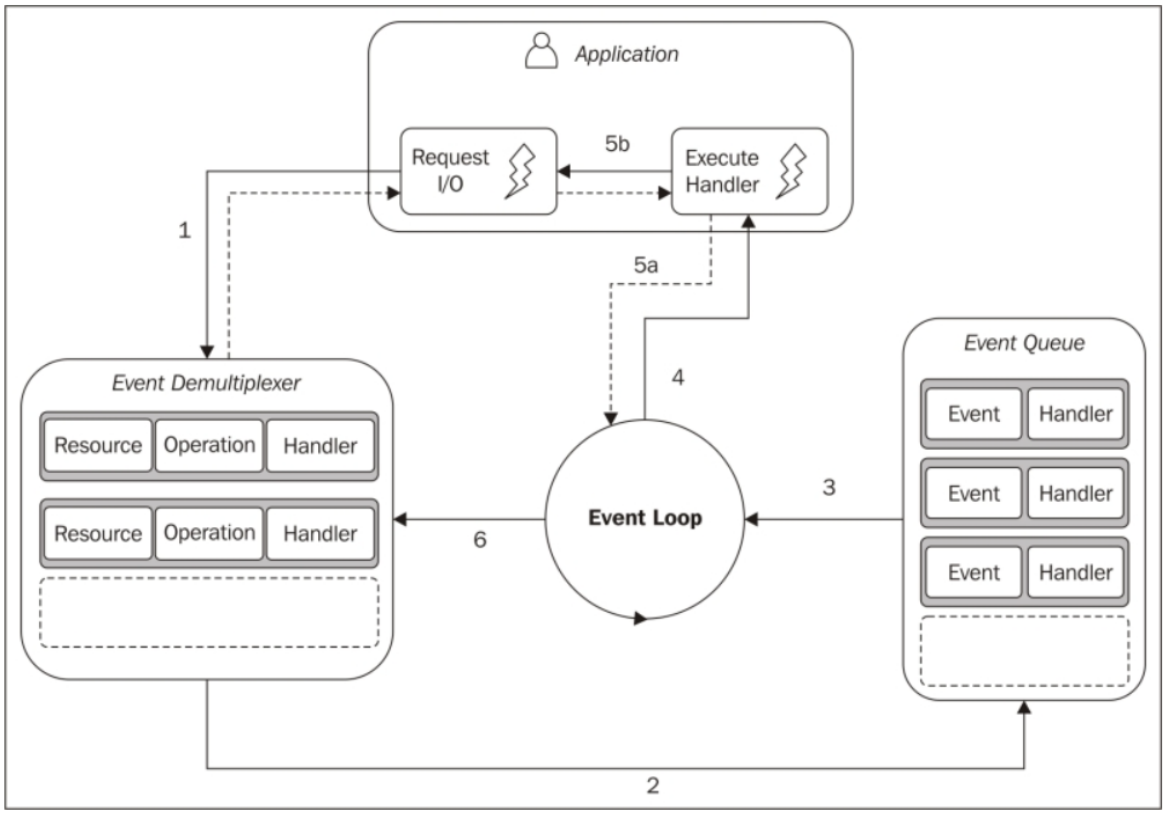
Mit ért a Reaktor mintán a Node.js-ben?

A Reaktorminta a Node.js nem blokkoló I / O műveleteinek ötlete. Ez a minta egy kezelőt biztosít (a Node.js esetében egy callback függvény), amely minden I / O művelethez társul. Amikor I / O kérést generálnak, azt egy demultiplexerhez továbbítják.

Ez a demultiplexer egy értesítési felület, amely a blokkolás nélküli I / O módban a párhuzamosság kezelésére szolgál, és minden kérést esemény formájában összegyűjt, és sorba állítja az egyes eseményeket. Így a demultiplexer biztosítja az eseménysort. Amikor a kérést a demultiplexer összegyűjti, akkor a vezérlőt visszaküldi a rendszernek, így nem blokkolja az I / O-t. Ugyanakkor létezik egy eseményhurok, amely az eseménysorban szereplő elemeket ismétli. Minden eseményhez tartozik callback függvény, és ez a callback függvény meghívásra kerül, amikor az Esemény ckilus (Event Loop) ismétlődik.

A callback függvénynek többnyire más callback függvényei is vannak, amelyek néhány aszinkron műveletet képviselnek. Ezeket a műveleteket a demultiplexer illeszti be az eseménysorba, és készen állnak a feldolgozásra, amint az Event Loop megismétli őket. Ezért a más műveletekre irányuló hívásoknak aszinkronnak kell lenniük.

Ha az Eseménysorban szereplő összes elem feldolgozásra kerül, és nem marad függőben lévő művelet, a Node.js automatikusan leállítja az alkalmazást.



1. Az alkalmazás új I / O műveletet generál azáltal, hogy kérelmet nyújt be az Event Demultiplexerhez. Az alkalmazás megad egy kezelőt is, amelyet a művelet befejezésekor hívnak meg. Új kérelem beküldése az Event Demultiplexer számára nem blokkoló hívás, és azonnal visszaküldi a vezérlőt az alkalmazásnak.
2. Amikor egy I / O művelet befejeződik, az Event Demultiplexer az új eseményeket az Esemény sorába tolja.
3. Ezen a ponton az Event Loop ismétli az eseménysor elemeit.
4. Minden eseménynél a hozzá tartozó kezelőt hívják meg.
5. A kezelő, amely az alkalmazáskód része, visszaadja a vezérlést az Event Loop-nak, amikor a végrehajtása befejeződik (5a). A kezelő (5b) végrehajtása során azonban új aszinkron műveletekre lehet szükség, ami új műveletek beillesztését okozhatja az Event Demultiplexer (1) -be, mielőtt a vezérlőt visszaadnák az Event Loop-nak.
6. Az eseménysor összes elemének feldolgozása után a ciklus ismét blokkolni fog az Event Demultiplexer alkalmazásban, amely ekkor újabb ciklus indít el.

Melyek a Node.js legfontosabb jellemzői?

- Az aszinkron eseményvezérelt IO segíti az konkurrens kéréskezelést - A Node.js összes API-ja aszinkron. Ez azt jelenti, hogy ha egy Node kérést kap valamilyen Input/Output műveletről, akkor azt a háttérben végrehajtja, és folytatja a többi kérés feldolgozását. Így nem várja meg a korábbi kérések válaszát.

- Gyors kódfuttatás - A Node.js a V8 JavaScript Runtime motort használja, amelyet a Google Chrome használ. A Node rendelkezik egy wrapper-rel a JavaScript motor felett, amely sokkal gyorsabbá teszi a futásidejű motort, és így a Node.js-en belüli kérelmek feldolgozása is gyorsabbá válik.
- Egyszálú, de nagymértékben méretezhető - A Node.js egyetlen szálas modellt használ az event loop-hoz. Lehetséges, hogy ezekből az eseményekből érkező válasz nem éri el azonnal a szervert. Ez azonban nem zárja le a többi műveletet. Így a Node.js nagyon skálázható. A hagyományos szerverek korlátozott szálakat hoznak létre a kérelmek kezelésére, míg a Node.js egyetlen szálat hoz létre, amely sokkal nagyobb számú ilyen kérést szolgáltat.
- A Node.js könyvtár JavaScriptet használ - Ez a Node.js másik fontos szempontja a fejlesztő szempontjából. A fejlesztők többsége már jól ismeri a JavaScript-et. Ezért a Node.js fejlesztése könnyebbé válik egy olyan fejlesztő számára, aki ismeri a JavaScript-et.
- Van egy aktív és élénk közösség a Node.js keretrendszerhez - Az aktív közösség mindig frissíti a keretrendszert a webfejlesztés legújabb trendjeivel.
- Nincs pufferelés - A Node.js alkalmazások soha nem pufferelnek adatokat. Egyszerűen csak darabokban adják ki az adatokat.

Mik a globálisok a Node.js-ben?

Három kulcsszó található a Node.js-ben, amelyek globálisként szerepelnek. Ezek a Global, a Process és a Buffer.

1. Global

A Global kulcsszó a globális névtér objektumot képviseli. Konténerként működik az összes többi globális objektum számára. Ha beírjuk a console.log (global) szót, akkor mindegyiket kiírja.

Fontos megjegyezni a globális objektumokkal kapcsolatban, hogy nem mindegyik tartozik globális hatókörbe, némelyik a modul hatókörébe tartozik. Tehát bölcs dolog a var kulcsszó használata nélkül deklarálni őket, vagy hozzáadni a Global objektumhoz.

A var kulcsszóval deklarált változók lokalizálódnak a modul számára, míg a nélkül deklaráltak feliratkoznak a globális objektumra.

2. Process

Ez egyike a globális objektumoknak, de további funkciókat tartalmaz a szinkron funkció aszinkron callback hívássá alakítására. Nincs korlátozás, bárhonnan elérheti a kódban. Ez az EventEmitter osztály példánya. És minden Node alkalmazás objektum a Process objektum példánya.

Elsősorban az alkalmazással vagy a környezettel kapcsolatos információkat adja vissza.

- <process.execPath> - a Node alkalmazás végrehajtási útvonalának lekérése.
- <process.Version> - az aktuálisan futó Node verzió lekérése.
- <process.platform> - a szerverplatform lekérése.

Néhány további hasznos Process a következő.

- <process.memoryUsage> - A Node alkalmazás által használt memória megismerése.
- <process.NextTick> - Callback függvény csatolása, amelyet a következő ciklus során hívnak meg. Késést okozhat egy funkció végrehajtásában.

3. Buffer

A Buffer a Node.js osztálya bináris adatok kezelésére. Hasonló az egész számok listájához, de nyers memóriaként tárolja a V8 heap-en kívül.

Átalakíthatjuk a JavaScript karakterlánc objektumokat bufferekké. Ehhez azonban kifejezetten meg kell említeni a kódolási típust.

- <ascii> - 7 bites ASCII adatokat határoz meg.
- <utf8> - Többbájtos kódolású Unicode karakterkészletet jelent.
- <utf16le> - 2 vagy 4 bájt, Unicode karaktereket jelöl.
- <base64> - Base64 karakterlánc kódoláshoz.
- <hex> - Az egyes bájtokat két hexadecimális karakterként kódolja.

A Buffer osztály használatának szintaxisa.

```
var buffer = new Buffer(string, [encoding]);
```

A fenti parancs új buffert fog kiosztani, amely alapértelmezett kódolásként az utf8 karakterláncot tartja. Ha azonban egy karakterláncot szeretne írni egy meglévő bufferobjektumhoz, akkor használja a következő kódsort.

```
buffer.write(string)
```

Ez az osztály más módszereket is kínál, mint például a readInt8 és writeUInt8, amelyek lehetővé teszik a különféle típusú adatok olvasását / írását a bufferbe.

Mi a láncolási folyamat a Node.js-ben?

Ez egy megközelítés, amely összeköti az egyik adatfolyam kimenetét egy másik adatfolyam bemenetével, így létrehozva a több adatfolyam működését.

Mi az a control flow függvény? milyen lépéseket hajt végre?

Ez egy általános kóddarab, amely több aszinkron függvényhívás között fut, vezérlési folyamat függvényként ismert.

A következő lépéseket hajtja végre.

- A végrehajtás sorrendjének ellenőrzése.

- Adatgyűjtés.
- Konkurrencia korlátozása.
- Meghívja a program következő lépését.

Mi az npm a Node.js-ben?

Az NPM a Node Package Manager rövidítése. A következő két fő funkciót biztosítja.

- Online tárolóként működik a <nodejs.org> webhelyen található node.js csomagokhoz / modulokhoz.
- Parancssori segédprogramként működik a csomagok telepítéséhez, a Node.js csomagok verziókezeléséhez és függőségkezeléséhez. Az NPM a Node.js telepíthetőségével együtt érkezik. Verzióját a következő paranccsal ellenőrizhetjük:

```
npm --version
```

Az NPM az alábbi paranccsal segíti a Node.js modulok telepítését.

```
npm install <Module Name>
```

Például a következő parancs egy híres Node.js web-keretrendszer, az úgynevezett express-

```
npm install express
```

Mikor érdemes használni a Node.js-t, és mikor nem?

Mikor érdemes használni a Node.js-t?

Ideális a Node.js használatával olyan streaming vagy eseményalapú valós idejű alkalmazások fejlesztésére, amelyek kevesebb CPU-használatot igényelnek, mint pl.

- Chat alkalmazások.
- Játék szerverek - A Node.js alkalmas gyors és nagy teljesítményű szerverekre, amelyek több ezer felhasználói kérés egyidejű kezelésének szükségességével szembesülnek.
- Jó együttműködő környezetben - Olyan környezetekhez alkalmas, ahol több ember dolgozik együtt. Például feladják dokumentumaikat, módosítják őket a dokumentumok ki- és bejelentkezésével. A Node.js támogatja az ilyen helyzeteket azzal, hogy eseménydokumentumot hoz létre a dokumentum minden módosításához. A Node.js „Event Loop” szolgáltatása lehetővé teszi több esemény egyidejű kezelését anélkül, hogy blokkolódna.

- Hirdetési szerverek (Advertisement Servers) - Itt is vannak olyan szervereink, amelyek több ezer kérést kezelnek a hirdetések letöltésére egy központi állomásról. A Node.js pedig ideális megoldás az ilyen feladatok kezelésére.
- Streaming szerverek - A Node.js másik ideális forgatókönyve a multimédiás streaming szerverek számára, ahol az ügyfelek a szerver felé indítják a kéréseket, hogy különféle multimédiás tartalmakat töltsenek le róla.

Összefoglalva: jó használni a Node.js-t, amikor magas szintű párhuzamosságra van szükség, de kevesebb dedikált CPU-időre van szükség.

Végül, de nem utolsósorban, mivel a Node.js belsőleg használja a JavaScript-et, így a legjobban illeszkedik a JavaScript-et is használó kliensoldali alkalmazások felépítéséhez.

Mikor ne használja a Node.js-t:

Ugyanakkor a Node.js-t számos alkalmazáshoz használhatjuk. De ez egy szálú keretrendszer, ezért ne használjuk olyan esetekre, amikor az alkalmazás hosszú feldolgozási időt igényel. Ha a szerver valamilyen számítást végez, akkor nem fog tudni más kéréseket feldolgozni. Ezért a Node.js a legjobb, ha a feldolgozáshoz kevesebb dedikált CPU-idő szükséges.

Magyarázza el, hogyan működik a Node.js?

A Node.js alkalmazás egyetlen szálat hoz létre. Amikor a Node.js kérést kap, először befejezi annak feldolgozását, mielőtt továbblépne a következő kérésre.

A Node.js aszinkron módon működik az event loop és a callback függvények használatával, több párhuzamosan érkező kérelem kezelésére. Az Event loop egy olyan függvény, amely kezeli és feldolgozza az összes külső eseményt, és csak átalakítja őket callback függvénnyé. Megfelelő időben meghívja az összes eseménykezelőt. Így rengeteg munka zajlik a háttérben, egyetlen kérés feldolgozása közben, így az új beérkező kérelemnek nem kell várnia, ha a feldolgozás nem teljes.

A kérelem feldolgozása közben a Node.js callback függvényt csatol hozzá, és áthelyezi a háttérbe. Most, amikor a válasz készen áll, egy eseményt hívnak meg, amely elindítja a kapcsolódó callback függvényt a válasz elküldéséhez.

A Node.js teljes egészében egyszálon alapszik?

Igen, igaz, hogy a Node.js minden kérést egyetlen szálon dolgozza fel. De ez csak a Node.js tervezés mögött álló elmélet része. Valójában az egyszálas mechanizmuson kívül eseményeket és callback függvényeket használ fel nagyszámú kérés aszinkron kezelésére.

Ezenkívül a Node.js optimalizált kialakítású, amely mind a JavaScriptet, mind a C ++ -ot használja a maximális teljesítmény garantálásához. A JavaScript a szerver oldalon fut a Google Chrome v8 motor által. És a C ++ lib UV könyvtár gondoskodik a nem szekvenciális I / O-ról a hatter worker-ön keresztül.

Tegyük fel, hogy a Node.js sorban száz kérelem sorakozik fel. A terveknek megfelelően a Node.js event loop fő szála megkapja mindet, és továbbítja a background worker-eknek végrehajtásra. Miután a worker-ek befejezték a kérelmek feldolgozását, a regisztrált callback-ek értesítést kapnak az event loop száláról, hogy továbbítsák az eredményt a felhasználónak.

Hogyan lehet post kérést Node.js-ben készíteni?

```
var request = require('request');
    request.post('http://www.example.com/action', { form: { key: 'value' } },
    function (error, response, body) {
        if (!error && response.statusCode == 200) {
            console.log(body)
        }
    });
```

Http szerver létrehozása a Node.js-ben?

Ehhez használhatjuk a <http-server> parancsot.

```
var http = require('http');
var requestListener = function (request, response) {
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.end('Welcome Viewers\n');
}
var server = http.createServer(requestListener);
server.listen(4200); // The port where you want to start with.
```

Hogyan lehet betölteni a HTML-t a Node.js fájlba?

A HTML Node.js fájlba történő betöltéséhez meg kell változtatnunk a HTML-kód „Content-type” szövegét text / plain helyett text / html-re.

```
fs.readFile(filename, "binary", function(err, file) {
    if(err) {
        response.writeHead(500, {"Content-Type": "text/plain"});
        response.write(err + "\n");
        response.end();
        return;
    }

response.writeHead(200);
response.write(file, "binary");
response.end();
});
```

Most módosítjuk ezt a kódot HTML-re plain text helyett.

```
fs.readFile(filename, "binary", function(err, file) {
    if(err) {
        response.writeHead(500, {"Content-Type": "text/html"});
        response.write(err + "\n");
        response.end();
        return;
    }

response.writeHead(200, {"Content-Type": "text/html"});
response.write(file);
response.end();
});
```

Hogyan hallgathatja a 80-as porton a Node?

Ahelyett, hogy a 80-as porton futtatnánk, a 80-as portot átirányíthatjuk az alkalmazásod portjára (> 1024)

```
iptables -t nat -I PREROUTING -p tcp --dport 80 -j REDIRECT --to-port 3000
```

Mi a különbség a működési és a programozói hibák között?

A működési hibák nem hibák, hanem a rendszer problémái, például a kérés időkorlátja vagy a hardver hiba. Másrészt a programozói hibák tényleges hibák.

Miért hasznos az npm shrinkwrap?

Az NPM shrinkwrap segítségével lezárhatja (lock) a telepített csomagok verzióit és azok leszármazott csomagjait. Segít ugyanazon csomagverziók használatában minden környezetben (fejlesztés, futtatás, production mód), valamint javítja a letöltési és telepítési sebességet. Ha minden környezetben ugyanazok a verziók találhatók, segíthet a rendszerek tesztelésében és a magabiztos telepítésben. Ha az összes teszt átmegy egy gépen, biztos lehet benne, hogy az átmegy az összes többi gépen is, mert tudja, hogy ugyanazt a kódot használja!

```
npm shrinkwrap
```

Új npm-shrinkwrap.json fájlt kell létrehoznia, amely tartalmazza az összes használt csomagot.

Mi a kedvenc HTTP keretrendszered és miért?

Express.js

Az Express egy vékony réteget biztosít a Node.js tetején olyan webalkalmazás-funkciókkal, mint az alapvető útválasztás, middleware, template engine és statikus fájlok kiszolgálása, így a Node.js drasztikus I / O teljesítménye nem sérül.

Az Express egy minimális keret. nem alkalmazza az elterjedt tervezési mintákat, mint például az MVC, az MVP, az MVVM vagy bármi más. Az egyszerűség kedvelői számára ez egy nagy plusz az összes többi keretrendszer között, mert saját preferenciájával és felesleges tanulási görbe nélkül készítheti el az alkalmazását. Ez különösen akkor előnyös, ha új személyes projektet hoznak létre, amelynek nincs történelmi terhe, de a projekt vagy a fejlesztő csapat növekedésével a szabványosítás hiánya többletmunkához vezethet a projekt- / kódmenedzsment számára, a legrosszabb esetben pedig képtelenség a fenntarthatóság.

Generator

A keretrendszer rendelkezik a generátorral, amely létrehozza a projekt mappájának struktúráját. Az express-generator npm csomag telepítése és generátor paranccsal történő létrehozása után világos hierarchiával rendelkező alkalmazásmappa jön létre, amely segít a képek, a stíluslapfájlok (stylesheet files) és a HTML-sablonfájlok rendezésében.

```
npm install express-generator -g
express helloapp
```

Middleware

A Middleware alapvetően csak olyan függvények, amelyek teljes hozzáféréssel rendelkeznek mind a kérés, mind a válaszobjektumokhoz.

```
var app = express();

app.use(cookieParser());
app.use(bodyParser());
app.use(logger());
app.use(authentication());

app.get('/', function (req, res) {
  // ...
});

app.listen(3000);
```

Az Express alkalmazás lényegében a Node.js, sok middleware függvénnyel, függetlenül attól, hogy testre akarja-e szabni a saját köztes szoftverét, vagy kihasználja a keretrendszer beépített köztes eszközeit, az Express természetes és intuitívvá tette a folyamatot.

Sablonmotor (Template Engine)

A sablonmotorok (Template Engine) lehetővé teszik a fejlesztő számára, hogy háttér-változókat beágyazhasson HTML-fájlokba, és kérésre a sablonfájl egyszerű HTML-formátumba kerül, a változók interpolálva a tényleges értékeikkel. Alapértelmezés szerint az expressz-generátor a Pug (eredetileg Jade néven ismert) sablonmotort használja, de más opciók, például a Mustache és az EJS is zökkenőmentesen működnek az Express-szel.

Adatbázis-integráció

Minimális keretrendszerként az Express nem tekinti az adatbázis-integrációt kötelező szempontnak a csomagjában, ezért semmiféle speciális adatbázis-használathoz sem hajlik. Míg egy adott adattárolási technológiát alkalmaz, legyen az MySQL, MongoDB, PostgreSQL, Redis, ElasticSearch vagy valami más, csak az adott npm csomag adatbázis-illesztőprogramként történő telepítéséről van szó. Ezek a harmadik féltől származó adatbázis-illesztőprogramok nem felelnek meg az egységes szintaxisnak a CRUD utasítások végrehajtása során, ami nagy gondot és hibát okoz az adatbázis-váltáson.

Milyen kihívások vannak a Node.js használatával?

Kihívások a Node.js alkalmazáskarbantartással

Az alkalmazás nem megfelelő karbantartása a stabilitással vagy rugalmassággal kapcsolatos kérdéseket eredményezhet, ami gyakran az alkalmazás meghibásodásához vezethet. Ha a kód nincs jól megírva, vagy ha a fejlesztők elavult eszközöket használnak, a teljesítmény károsodhat, és a felhasználók több hibát és alkalmazás összeomlást tapasztalhatnak. Ráadásul a rossz minőségű kód akadályozhatja az alkalmazás méretezési képességét és az alkalmazás továbbfejlesztését. A legrosszabb esetben lehetetlenné válhat új funkciók bevezetése anélkül, hogy a kódbázist átírnánk a semmiből.

- A skálázhatóság kihívásai
- Gyenge dokumentáció

A karbantartási problémák kezelése

- Code review
- Használjon microservices-t
- Javítsa a kód minőségét
- Tesztelje az új szolgáltatás bevezetése előtt
- Javítsa a dokumentációt

Hogyan oldja meg a Node.js az I / O műveletek blokkolásának problémáját?

A Node.js úgy oldja meg ezt a problémát, hogy az eseményalapú modellt állítja középpontjába, szálak helyett event loop használatával.

A Node.js ehhez event loop használ. Az event loop „olyan entitás, amely kezeli és feldolgozza a külső eseményeket, és callback hívásokká alakítja azokat”. Ha adatokra van szükség, a nodejs regisztrál egy callback-et és elküldi a műveletet erre az event loop-ra. Amikor az adatok rendelkezésre állnak, a callback-et hívják.

Említse meg azokat a lépéseket, amelyekkel aszinkronizálhatja a Node.js fájlt?

Az ES 2017 aszinkron funkciókat vezetett be. Az aszinkron funkciók lényegében tisztább módszer a JavaScript-ben lévő aszinkron kód használatára.

1.) Async/Await

Aszinkron kód legújabb módja.

Nem blokkoló (csakúgy, mint az promise-ok és callback-ek).

Az Async / Await azért jött létre, hogy leegyszerűsítse a láncolt promise-okkal való munka és írás folyamatát.

Az Async függvények promise-t adnak vissza. Ha a függvény hibát dob, az promise rejected-re kerül. Ha a függvény visszaad egy értéket, akkor az promise resolved.

```
// Normal Function
function add(x,y){
  return x + y;
}
// Async Function
async function add(x,y){
  return x + y;
}
```

2.) Await

Az aszinkron függvények felhasználhatják az await kifejezést. Ez szünetelteti az aszinkron függvények, és a továbblépés előtt megvárja a promise resolved legyen.

```
function doubleAfter2Seconds(x) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(x * 2);
    }, 2000);
  });
}

async function addAsync(x) {
  const a = await doubleAfter2Seconds(10);
  const b = await doubleAfter2Seconds(20);
  const c = await doubleAfter2Seconds(30);
  return x + a + b + c;
}


addAsync(10).then((sum) => {
  console.log(sum);
});
```

Melyek a Node.js időzítési jellemzői?

A Performance Timing API biztosítja a W3C Performance Timeline specifikáció megvalósítását. Az API célja a nagy felbontású teljesítménymutatók támogatása. Ez ugyanaz a Performance API, mint a modern webböngészőkben.

```
const { PerformanceObserver, performance } = require('perf_hooks');

const obs = new PerformanceObserver((items) => {
  console.log(items.getEntries()[0].duration);
  performance.clearMarks();
});
obs.observe({ entryTypes: ['measure'] });

performance.mark('A');
doSomeLongRunningProcess(() => {
  performance.mark('B');
  performance.measure('A to B', 'A', 'B');
});
```

request modul

A népszerű request modul beépített módszerrel rendelkezik a HTTP időzítés mérésére. Engedélyezheti a time property-vel.

```
const request = require('request')

request({
  uri: 'https://nodejs.org',
  method: 'GET',
  time: true
}, (err, resp) => {
  console.log(err || resp.timings)
})
```

Mi a Node.js LTS kiadása, miért fontos?

A Node.js LTS (Long Term Support) verziója megkapja az összes kritikus hibajavítást, biztonsági frissítést és teljesítményt

A Node.js LTS verzióit legalább 18 hónapig támogatják, és páros verziószámokkal jelzik (pl. 4, 6, 8). A legjobbak a production-höz, mivel az LTS a stabilitásra és a biztonságra összpontosít, míg a jelenlegi kiadási sor rövidebb élettartammal és gyakoribb frissítéssel látja el a kódot. Az LTS verzióinak változtatásai a stabilitás hibajavításaira, a biztonsági frissítésekre, az esetleges npm frissítésekre, a dokumentáció frissítésére és bizonyos teljesítménybeli fejlesztésekre korlátozódnak, amelyekről bebizonyítható, hogy nem törik meg a meglévő alkalmazásokat.

Miért válassza el az Express „app”-ot és a „server”-t?

Az API deklarációnak a hálózattal kapcsolatos konfigurációtól (port, protokoll stb.) elkülönítve tartása lehetővé teszi az API tesztelését, hálózati hívások végrehajtása nélkül. Azt is lehetővé teszi, hogy ugyanazt az API-t rugalmasan és eltérő hálózati körülmények között telepítsék. Bónusz: az kód jobb elkülönítése és a tisztább kód.

Az API deklarációnak az app.js fájlban kell lennie:

```
var app = express();
app.use(bodyParser.json());
app.use("/api/events", events.API);
app.use("/api/forms", forms);
```

A szerver hálózati deklarációnak a / bin / www könyvtárban kell lennie:

```
var app = require('../app');
var http = require('http');

/**
 * Get port from environment and store in Express.
 */

var port = normalizePort(process.env.PORT || '3000');
app.set('port', port);

/**
 * Create HTTP server.
 */

var server = http.createServer(app);
```

Mi a különbség a process.nextTick () és a setImmediate () között?

A process.nextTick () és a setImmediate () közötti különbség az, hogy a process.nextTick () elhalasztja egy művelet végrehajtását az event loop körüli következő lépésig, vagy egyszerűen meghívja acallback függvényt, amint az event loop folyamatban lévő végrehajtása befejeződik. míg a setImmediate () callback-et hajt végre az event loop következő ciklusán, és visszaadja az event loop-ot az I / O műveletek végrehajtásához.

Mi a különbség a JavaScript és a Node.js között?

	JavaScript	**Node JS**
Type	A JavaScript programozási nyelv. Minden webböngészőben fut, megfelelő böngészőmotorral.	Ez a JavaScript értelmezője és környezete, néhány speciális könyvtárral, amelyeket a JavaScript programozása külön használhat.
Utility	kliens oldali tevékenységhez, például az attribútum esetleges érvényesítéséhez vagy az oldal frissítéséhez egy adott időközönként, vagy bizonyos dinamikus változásokat biztosít a weboldalakon az oldal frissítése nélkül.	Főleg bármely operációs rendszer nem blokkoló műveletének elérésére vagy végrehajtására használták, például shell parancsfájl létrehozására vagy végrehajtására, vagy bármilyen hardverspecifikus információ elérésére, vagy bármilyen háttérprogram feladat futtatására.
Running Engine	Bármilyen motort futtató JavaScript, például Spider monkey (FireFox), JavaScript Core (Safari), V8 (Google Chrome).	A Node JS csak egy V8-as motorban fut, amelyet főleg a Google Chrome használ. És a javascript program, amelyet e Node JS-ben írunk, mindig a V8 Engine-ben fog futni.

Mi a különbség az események és a callback-ek között?

A Node.js egyszálú alkalmazás, de támogatja a konkurrens feldolgozás az esemény és a callback fogalmán keresztül. A Node.js minden API-je aszinkron, és egyszálú, aszinkron függvényhívásokat használnak a konkurrencia fenntartására. A Node szál megtartja az event loop-ot, és amikor egy feladat befejeződik, elindítja a megfelelő eseményt, amely jelzi az esemény-figyelő (event-listener) függvénynek végrehajtást.

Callback függvényeket hívunk meg, amikor egy aszinkron függvény visszaadja eredményét, míg az eseménykezelés az observer mintán működik. Azok a függvényeket, amelyek eseményeket figyelnek Observer-nek nevezzük. Amikor egy esemény elindul, a figyelő függvény elindul. A Node.js több beépített eseményt kínál, amelyek az eseménymodulon és az EventEmitter osztályon keresztül érhetők el, amelyek események és eseményfigyelők összekapcsolására szolgálnak

1. Callback: A callback függvény egy másik függvénybe argumentumként átadott függvény, amelyet a külső függvény belsejében hívnak meg valamilyen rutin vagy művelet elvégzésére.

Példa: szinkron callback

```
function greeting(name) {
  alert('Hello ' + name);
}

function processUserInput(callback) {
  var name = prompt('Please enter your name.');
  callback(name);
}

processUserInput(greeting);
```

2. Események: A számítógép minden művelete esemény. A Node.js lehetővé teszi számunkra, hogy egyedi eseményeket hozzunk létre és kezeljünk egyszerűen az esemény modul segítségével. Az Event modul tartalmaz EventEmitter osztályt, amely felhasználható egyedi események felvetésére és kezelésére.

```
var event = require('events');
var eventEmitter = new event.EventEmitter();

//  Add listener function for Sum event
eventEmitter.on('Sum', function(num1, num2) {
    console.log('Total: ' + (Number(num1) + Number(num2)));
});

//  Call Event.
eventEmitter.emit('Sum', '10', '20');
```

Magyarázza a RESTful webszolgáltatásokat a Node.js-ben?

A REST jelentése REpresentational State Transfer. A REST web szabványokon alapuló architektúra és HTTP protokollt használ. Ez egy architektúrális minta, valamint kommunikációs célú megközelítés, amelyet gyakran használnak a különféle webszolgáltatások fejlesztésében. A REST kiszolgáló egyszerűen hozzáférést biztosít az erőforrásokhoz, a REST kliens pedig hozzáférést biztosít és módosítja az erőforrásokat a HTTP protokoll segítségével.

HTTP metódusok

- GET - Csak olvasható hozzáférést biztosít egy erőforráshoz.
- PUT - Új erőforrást hoz létre.
- DELETE - erőforrás eltávolítása.
- POST - Meglévő erőforrás frissítése vagy új erőforrás létrehozása.
- PATCH - Erőforrás frissítése / módosítása

A REST alapelvei

- Egységes felület
- Hontalan
- Stateless
- Client-Server

- Réteges rendszer
- Igény szerinti kód (nem kötelező)

users.json

```
{
   "user1" : {
      "id": 1,
      "name" : "Ehsan Philip",
      "age" : 24
   },

   "user2" : {
      "id": 2,
      "name" : "Karim Jimenez",
      "age" : 22
   },

   "user3" : {
      "id": 3,
      "name" : "Giacomo Weir",
      "age" : 18
   }
}
```

List Users (GET method)
Let's implement our first RESTful API listUsers using the following code in a server.js file −

```
var express = require('express');
var app = express();
var fs = require("fs");

app.get('/listUsers', function (req, res) {
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
      console.log( data );
      res.end( data );
   });
})

var server = app.listen(3000, function () {
   var host = server.address().address
   var port = server.address().port
   console.log("App listening at http://%s:%s", host, port)
});
```

Add User (POST method)
Following API will show you how to add new user in the list.

```
var express = require('express');
var app = express();
var fs = require("fs");

var user = {
   "user4" : {
      "id": 4,
      "name" : "Spencer Amos",
      "age" : 28
   }
}

app.post('/addUser', function (req, res) {
   // First read existing users.
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
      data = JSON.parse( data );
      data["user4"] = user["user4"];
```

```
      console.log( data );
      res.end( JSON.stringify(data));
   });
})

var server = app.listen(3000, function () {
   var host = server.address().address
   var port = server.address().port
   console.log("App listening at http://%s:%s", host, port)
})
```

Delete User

```
var express = require('express');
var app = express();
var fs = require("fs");

var id = 2;

app.delete('/deleteUser', function (req, res) {
   // First read existing users.
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
      data = JSON.parse( data );
      delete data["user" + 2];
      console.log( data );
      res.end( JSON.stringify(data));
   });
})

var server = app.listen(3000, function () {
   var host = server.address().address
   var port = server.address().port
   console.log("App listening at http://%s:%s", host, port)
})
```

Hogyan kezeljük a fájlok feltöltését a Node.js-ben?

express: A Node.js tetejére épített népszerű web- framework, amelyet a REST-API létrehozására használnak.

body-parser: A beérkező kérelem törzsek elemzése egy middleware-be

multer: Middleware a multipart/form-data- fájl feltöltések kezeléséhez

1. Installing the dependencies

```
npm install express body-parser multer –save
```

2. Package.json

```
{
  "name": "file_upload",
  "version": "0.0.1",
  "dependencies": {
    "express": "4.13.3",
    "multer": "1.1.0"
  },
  "devDependencies": {
    "should": "~7.1.0",
    "mocha": "~2.3.3",
    "supertest": "~1.1.0"
  }
}
```

3. Server.js

```
var express = require("express");
var bodyParser = require('body-parser');
var multer  = require('multer');
var app = express();

// for text/number data transfer between clientg and server
app.use(bodyParser());

var storage =   multer.diskStorage({
  destination: function (req, file, callback) {
    callback(null, './uploads');
  },
  filename: function (req, file, callback) {
    callback(null, file.fieldname + '-' + Date.now());
  }
});

var upload = multer({ storage : storage}).single('userPhoto');

app.get('/', function(req, res) {
      res.sendFile(__dirname + "/index.html");
});

// POST: upload for single file upload
app.post('/api/photo', function(req, res) {
    upload(req,res,function(err) {
        if(err) {
            return res.end("Error uploading file.");
        }
        res.end("File is uploaded");
    });
});

app.listen(3000, function(){
    console.log("Listening on port 3000");
});
```

4. index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Multer-File-Upload</title>
</head>
<body>
    <h1>MULTER File Upload | Single File Upload</h1>

    <form id = "uploadForm"
         enctype = "multipart/form-data"
         action = "/api/photo"
         method = "post"
    >
      <input type="file" name="userPhoto" />
      <input type="submit" value="Upload Image" name="submit">
    </form>
</body>
</html>
```

Magyarázza el a body-parser, a cookie-parser, a morgan, a nodemon, a pm2, a serve-favicon, a cors, dotenv, az fs-extra, a moment kifejezéseket az Express JS-ben?

a) body-parser

body-parser kivonja a bejövő kérésfolyam teljes törzsrészét, és kiteszi azt a req.body-ra. Ez a törzselemző modul elemzi a HTTP POST kéréssel elküldött JSON-, puffer-, karakterlánc- és URL-kódolt adatokat.

Example:

```
npm install express ejs body-parser


// server.js

var express = require('express')
var bodyParser = require('body-parser')

var app = express()

// create application/json parser
var jsonParser = bodyParser.json()

// create application/x-www-form-urlencoded parser
var urlencodedParser = bodyParser.urlencoded({ extended: false })

// POST /login gets urlencoded bodies
app.post('/login', urlencodedParser, function (req, res) {
  res.send('welcome, ' + req.body.username)
})

// POST /api/users gets JSON bodies
app.post('/api/users', jsonParser, function (req, res) {
  // create user in req.body
})
```

b) cookie -parser

A cookie egy olyan adat, amelyet egy kéréssel küldenek az kliens oldalra, és amelyet a felhasználó által használt webböngésző maga a kliens oldalon tárol.

A cookie –parser köztes szoftver cookieParser függvénye egy titkos karakterláncot vagy string-tömböt vesz fel első argumentumként, és egy opcióobjektumot második argumentumként.

Installation

```
npm install cookie-parser
```

Example:

```
var express = require('express')
var cookieParser = require('cookie-parser')

var app = express()
app.use(cookieParser())

app.get('/', function (req, res) {
  // Cookies that have not been signed
  console.log('Cookies: ', req.cookies)

  // Cookies that have been signed
  console.log('Signed Cookies: ', req.signedCookies)
})

app.listen(3000)
```

c) morgan

HTTP kérés naplózó middleware a node.js-hez.

Installation

```
npm install morgan
```

Example: write logs to a file

```
var express = require('express')
var fs = require('fs')
var morgan = require('morgan')
var path = require('path')

var app = express()

// create a write stream (in append mode)
var accessLogStream = fs.createWriteStream(path.join(__dirname, 'access.log'), { flags: 'a' })

// setup the logger
app.use(morgan('combined', { stream: accessLogStream }))

app.get('/', function (req, res) {
  res.send('hello, world!')
```

d) nodemon

A Nodemon egy olyan segédprogram, amely figyeli az esetleges forrásbeli változásokat, és automatikusan újraindítja a szervert.

Installation

```
npm install -g nodemon
```

Example:

```
{
  // ...
  "scripts": {
    "start": "nodemon server.js"
  },
  // ...
}
```

e) pm2

A P (rocess) M (anager) 2 (pm2) a Node.js alkalmazások production folyamatkezelője, beépített terheléselosztóval. Lehetővé teszi az alkalmazások örök életben maradását, állásidő nélküli újratöltését, és megkönnyíti a rendszer rendszergazdai feladatait.

Installation

```
npm install pm2 –g
```

Start an application

```
pm2 start app.js
```

f) serve-favicon

Node.js köztes szoftver egy favicon kiszolgálására. A serve-favicon modul lehetővé teszi, hogy kizárjuk a favicon iránti kérelmeket a naplózó köztes szoftverünkből. Gyorsítótárban tárolja az ikont a memóriában, hogy javítsa a teljesítményt a lemezelérés csökkentésével. Ezenkívül az ikon tartalma alapján nyújt ETag-ot, nem pedig a fájlrendszer tulajdonságait.

Installation

```
npm install serve-favicon
```

Example:

```
var express = require('express')
var favicon = require('serve-favicon')
var path = require('path')

var app = express()
app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')))

// Add your routes here, etc.

app.listen(3000)
```

g) cors

A Cross-Origin Resource Sharing (CORS) fejlécek lehetővé teszik a böngészőben futó alkalmazások számára, hogy különböző domain-ekben (más néven origin-okban) működő szervereknek kérelmeket nyújtsanak be. A CORS fejlécek a szerver oldalon vannak beállítva - a HTTP szerver felel azért, hogy jelezze, hogy egy adott HTTP kérés cross-origin-e. A CORS meghatározza a böngésző és a szerver kölcsönhatásának módját, és megállapíthatja, hogy biztonságos-e engedélyezni a cross-origin kérelmet.

Installation

```
npm install cors
```

Example: Enable All CORS Requests

```
var express = require('express')
var cors = require('cors')
var app = express()

app.use(cors())

app.get('/products/:id', function (req, res, next) {
  res.json({msg: 'This is CORS-enabled for all origins!'})
})

app.listen(8080, function () {
  console.log('CORS-enabled web server listening on port 80')
})
```

Example: Enable CORS for a Single Route

```
var express = require('express')
var cors = require('cors')
var app = express()

app.get('/products/:id', cors(), function (req, res, next) {
  res.json({msg: 'This is CORS-enabled for a Single Route'})
})
```

```
app.listen(8080, function () {
  console.log('CORS-enabled web server listening on port 80')
})
```

h) dotenv

Amikor egy NodeJs alkalmazás fut, egy process.env nevű globális változót injektál be, amely információkat tartalmaz az alkalmazás futásának állapotáról. A dotenv betölti az .env fájlban tárolt környezeti változókat a process.env fájlba.

Installation

```
npm install dotenv
```

Usage

```
// .env

DB_HOST=localhost
DB_USER=admin
DB_PASS=root
// config.js

const db = require('db')

db.connect({
  host: process.env.DB_HOST,
  username: process.env.DB_USER,
  password: process.env.DB_PASS
})
```

i) fs-extra

Az fs-extra olyan módszereket tartalmaz, amelyek nem szerepelnek a Node.js fs csomagban. Ilyen például a rekurzív mkdir, copy, and remove.

Installation

```
npm install fs-extra
```

Usage

```
const fs = require('fs-extra')


// Async with callbacks:
fs.copy('/tmp/myfile', '/tmp/mynewfile', err => {
  if (err) return console.error(err)
  console.log('success!')
})
```

j) moment

JavaScript dátumkönyvtár a dátumok elemzéséhez, érvényesítéséhez, kezeléséhez és formázásához.

Installation

```
npm install moment --save
```

Usage

- Format Dates

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

moment().format('MMMM Do YYYY, h:mm:ss a'); // October 24th 2020, 3:15:22 pm
moment().format('dddd');                    // Saturday
moment().format("MMM Do YY");               // Oct 24th 20
```

- Relative Time

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

moment("20111031", "YYYYMMDD").fromNow(); // 9 years ago
moment("20120620", "YYYYMMDD").fromNow(); // 8 years ago
moment().startOf('day').fromNow();        // 15 hours ago
```

- Calendar Time

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

moment().subtract(10, 'days').calendar(); // 10/14/2020
moment().subtract(6, 'days').calendar();  // Last Sunday at 3:18 PM
moment().subtract(3, 'days').calendar();  // Last Wednesday at 3:18 PM
```

Hogyan működik a routing a Node.js-ben?

Az routing határozza meg, hogy az kliens kéréseket hogyan kezelik az alkalmazás végpontjai. Meghatározzuk az útválasztást (routing) az Express alkalmazásobjektum HTTP-módszereknek megfelelő metódusai segítségével; például az app.get () a GET kérések kezelésére és az app.post a POST kérések kezelésére, az app.all () az összes HTTP módszer kezelésére, az app.use () pedig a middleware-ekhez történő megadására.

Ezek az útválasztási módszerek a megadott útvonal (ok) nak és módszer (ek) nek megfelelő kéréseket "hallgatják", és amikor egyezést észlel, meghívja a megadott callback.

Szintaxis:

```
app.METHOD(PATH, HANDLER)
```

- Az app az expressz példány.
- A METHOD egy HTTP kérési módszer.
- A PATH egy elérési út a szerveren.
- A HANDLER az útvonal egyezésekor végrehajtott függvény.

a) Route methods

```
// GET method route
app.get('/', function (req, res) {
  res.send('GET request')
})

// POST method route
app.post('/login', function (req, res) {
```

```
  res.send('POST request')
})

// ALL method route
app.all('/secret', function (req, res, next) {
  console.log('Accessing the secret section ...')
  next() // pass control to the next handler
})
```

b) Útvonalak

Az útvonal elérési útjai egy kérési módszerrel kombinálva meghatározzák azokat a végpontokat, amelyeken kéréseket lehet tenni. Az útvonal útvonalai lehetnek karakterláncok, karakterlánc minták vagy reguláris kifejezések.

A ?, +, *, és () karakterek a reguláris kifejezés megfelelőinek részhalmazai.

```
// This route path will match requests to /about.
app.get('/about', function (req, res) {
  res.send('about')
})


// This route path will match acd and abcd.
app.get('/ab?cd', function (req, res) {
  res.send('ab?cd')
})


// This route path will match butterfly and dragonfly
app.get(/.*fly$/, function (req, res) {
  res.send('/.*fly$/')
})
```

c) Útvonal paraméterei

Az útvonalparaméterek olyan elnevezett URL-szegmensek, amelyek az URL-ben a pozíciójukban megadott értékek rögzítésére szolgálnak. A rögzített értékeket a req.params objektum tölti ki, az útvonalban megadott útvonalparaméter nevével.

```
app.get('/users/:userId', function (req, res) {
  res.send(req.params)
})
```

Response methods

Method	Description
`res.download()`	Kérjen egy fájlt letöltésre.
`res.end()`	Fejezze be a válaszfolyamatot.
`res.json()`	Küldjön JSON-választ.

Method	Description
`res.jsonp()`	Küldjön JSON-választ JSONP-támogatással.
`res.redirect()`	Átirányít egy kérést.
`res.render()`	View template renderelése.
`res.send()`	Küldjön különféle típusú választ.
`res.sendFile()`	Fájl küldése.
`res.sendStatus()`	Állítsa be a válasz állapotkódját, és küldje el annak karakterlánc-reprezentációját válaszként.

d) Router method

```
var express = require('express')
var router = express.Router()

// middleware that is specific to this router
router.use(function timeLog (req, res, next) {
  console.log('Time: ', Date.now())
  next()
})

// define the home page route
router.get('/', function (req, res) {
  res.send('Birds home page')
})

// define the about route
router.get('/about', function (req, res) {
  res.send('About birds')
})

module.exports = router
```

Hogyan akadályozza meg a node.js a blokkoló kódot?

Blokkolás vs blokkolás nélküli

A blokkolás akkor következik be, amikor a Node.js folyamatban további JavaScript végrehajtásának meg kell várnia, amíg egy nem JavaScript művelet befejeződik. Ez azért történik, mert az event loop nem tudja folytatni a JavaScript futtatását, miközben blokkolási művelet történik.

A Node.js szabványos könyvtárában a libuv-ot használó szinkron módszerek a leggyakrabban használt blokkoló műveletek. A natív modulok blokkolási módszerekkel is rendelkezhetnek. A blokkolási módszerek szinkron, a nem blokkoló módszerek pedig aszinkron.

```
// Blocking
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
console.log(data);
moreWork(); // will run after console.log

// Non-blocking
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
  console.log(data);
});
moreWork(); // will run before console.log
```

Mi a különbség a promise és az async-await között a Node.js-ben?

1. Promise

A promise egy művelet aszinkron eredményének kezelésére szolgál. A JavaScript úgy lett kialakítva, hogy ne várjon egy aszinkron kódblokk teljes végrehajtását, mielőtt a kód más szinkron részei futtathatók. Promise-okkal elhalaszthatjuk a kódblokk végrehajtását mindaddig, amíg az aszinkron kérés be nem fejeződik. Így más műveletek megszakítás nélkül folytathatók.

A Promise állapota:

- Pending (Függőben): Kezdeti állapot, mielőtt az Promise resolved vagy rejected lenne.
- Resolved: Teljesített promise
- Rejected (Elutasítva): Sikertelen promise, hibát dob

```
function logFetch(url) {
  return fetch(url)
    .then(response => {
      console.log(response);
    })
    .catch(err => {
      console.error('fetch failed', err);
    });
}
```

2. Async-await

Az Await alapvetően syntactic sugar a promise-okhoz. Az aszinkron kód sokkal inkább hasonlít a synchronous/procedural kódra, amelyet az emberek könnyebben megértenek.

Ha az async kulcsszót egy függvény elé helyezi, a függvény visszaadja a promise-t. Ha a kód olyasmit ad vissza, amely nem promise, akkor a JavaScript automatikusan resolved-olja a promise-t az értékkel. Az await kulcsszó egyszerűen arra készteti a JavaScriptet, hogy várjon, amíg az Promise resolved, majd visszaadja eredményét.

```
async function logFetch(url) {
  try {
    const response = await fetch(url);
    console.log(response);
  }
  catch (err) {
    console.log('fetch failed', err);
  }
}
```

Hogyan használható a JSON Web Token (JWT) a Node.js-ben történő hitelesítéshez?

A JSON Web Token (JWT) egy nyílt szabvány, amely kompakt és önálló módszert határoz meg az információk biztonságos továbbítására a felek között JSON objektumként. Ez az információ ellenőrizhető és megbízható, mert digitálisan aláírt.

A JWT-nek néhány előnye van:

Tisztán állapotmentes. A munkamenet információk tárolásához nincs szükség további szerverre.

Könnyen megosztható a szolgáltatások között.

A JSON web tokenek három részből állnak, pontokkal (.) Elválasztva, amelyek a következők:

```
jwt.sign(payload, secretOrPrivateKey, [options, callback])
```

(Header) Fejléc - Két részből áll: a token típusából (azaz JWT) és az signing algoritmusból (azaz HS512)

Payload- Olyan állításokat tartalmaz, amelyek információkat nyújtanak egy hitelesített felhasználóról, valamint egyéb információkat, például a tokenek lejárati idejét.

Signature - A token utolsó része, amely beburkolja a kódolt fejlécet és a payload-ot, valamint az algoritmus és a secret

Installation

```
npm install jsonwebtoken bcryptjs --save
```

Example: AuthController.js

```
var express = require('express');
var router = express.Router();
var bodyParser = require('body-parser');
var User = require('../user/User');

var jwt = require('jsonwebtoken');
var bcrypt = require('bcryptjs');
var config = require('../config');


router.use(bodyParser.urlencoded({ extended: false }));
router.use(bodyParser.json());

router.post('/register', function(req, res) {

  var hashedPassword = bcrypt.hashSync(req.body.password, 8);
```

```
  User.create({
    name : req.body.name,
    email : req.body.email,
    password : hashedPassword
  },
  function (err, user) {
    if (err) return res.status(500).send("There was a problem registering the user.")
    // create a token
    var token = jwt.sign({ id: user._id }, config.secret, {
      expiresIn: 86400 // expires in 24 hours
    });
    res.status(200).send({ auth: true, token: token });
  });
});
```

config.js

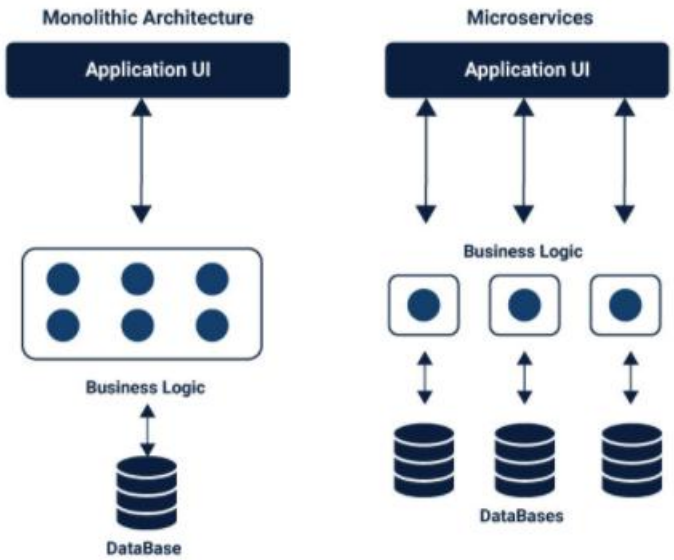
```
// config.js
module.exports = {
  'secret': 'supersecret'
};
```

A jwt.sign () metódus egy payload-ot és a config.js-ben definiált titkos kulcsot veszi paraméterként. Egyedi karakterláncot hoz létre, amely a payload-ot képviseli. Esetünkben a payload egy olyan objektum, amely csak a felhasználó azonosítóját tartalmazza.

Hogyan készítsünk microservices architektúrát a Node.js segítségével?

microservices

A microservices a szolgáltatásorientált architektúra (SOA) stílusa, ahol az alkalmazás összekapcsolt szolgáltatások összességére épül. A microservices-el az alkalmazás architektúrája lightweigth protokollokkal épül fel. A szolgáltatások alaposan be vannak építve az architektúrába. A mikroszolgáltatások kisebb szolgáltatásokra bontják az alkalmazást, és lehetővé teszik a jobb modularitást.



Kevés dolgot érdemes kiemelni a mikroszolgáltatások és általában az elosztott rendszerek és a monolit architektúrával szemben:

- Modularitás - az egyes műveletekért való felelősség az alkalmazás külön részeihez van rendelve
- Egységesség - a mikroszolgáltatások interfészei (API-végpontok) egy objektumot azonosító alap URI-ból és az objektum manipulálására használt standard HTTP-módszerekből (GET, POST, PUT, PATCH és DELETE)
- Robusztusság - az alkatrészek meghibásodása csak egy adott funkcionalitási egység hiányát vagy csökkenését okozza
- Karbantarthatóság - a rendszerelemek egymástól függetlenül módosíthatók és telepíthetők
- Méretezhetőség - egy szolgáltatás példányai hozzáadhatók vagy eltávolíthatók, hogy alkalmazkodjanak a kereslet változásaira.
- Elérhetőség - új funkciók adhatók a rendszerhez a 100% -os rendelkezésre állás fenntartása mellett.
- Tesztelhetőség - az új megoldásokat közvetlenül a production környezetben tesztelhetjük, ha korlátozott számú felhasználói szegmens számára alkalmazzuk őket, hogy lássuk, hogyan viselkednek a való életben.

Mikroszolgáltatások létrehozása a Node.js segítségével

01. lépés: Kiszolgáló létrehozása a kérések elfogadásához

Ez a fájl létrehozza szerverünket, és az összes kérés feldolgozásához útvonalakat rendel.

```
//  server.js

const express = require('express')
const app = express();
const port = process.env.PORT || 3000;

const routes = require('./api/routes');
routes(app);
app.listen(port, function() {
   console.log('Server started on port: ' + port);
});
```

02. lépés: Az útvonalak meghatározása

A következő lépés az, hogy meghatározza a mikroszolgáltatások útvonalait, majd mindegyiket hozzárendeli egy célhoz a vezérlőben. Két végpontunk van. Az egyik „about” nevű végpont, amely információt ad az alkalmazásról. És egy "távolság" végpont, amely két útparamétert tartalmaz, a Lego áruház mindkét irányítószámát. Ez a végpont adja vissza a két irányítószám közötti távolságot mérföldben.

```
const controller = require('./controller');

module.exports = function(app) {
   app.route('/about')
       .get(controller.about);
   app.route('/distance/:zipcode1/:zipcode2')
       .get(controller.getDistance);
```

```
};
```

03. lépés: Vezérlő (Controller) logika hozzáadása

A vezérlőfájlon belül két tulajdonságú vezérlőobjektumot fogunk létrehozni. Ezek a tulajdonságok azok a funkciók, amelyek az útvonali modulban definiált kéréseket kezelik.

```
var properties = require('../package.json')
var distance = require('../service/distance');

var controllers = {
   about: function(req, res) {
       var aboutInfo = {
           name: properties.name,
           version: properties.version
       }
       res.json(aboutInfo);
   },
   getDistance: function(req, res) {
           distance.find(req, res, function(err, dist) {
               if (err)
                   res.send(err);
               res.json(dist);
           });
       },
};

module.exports = controllers;
```

Hogyan lehet használni a Q promise-t a Node.js-ben?

A promise olyan objektum, amely a visszatérési értéket vagy a dobott kivételt képviseli, amelyet a függvény végül adhat. A promise egy távoli objektum proxyjaként is használható a késés leküzdésére.

A promise az aszinkron működés viszonylag egyszerű megvalósítása. A függvényből visszakapott promise objektum még nem befejezett műveletet jelent, de garantálja a művelet hívójának, hogy a művelet a jövőben befejeződik.

A promise a következő állapotokkal rendelkezik:

- Pending (Függőben) - az aszinkron művelet még nem fejeződött be.
- Fulfilled (Teljesítve) - az aszinkron művelet sikeresen befejeződött.
- Rejected (Elutasítva) - az aszinkron működés hibával szűnik meg.
- Settled- az aszinkron művelet teljesül vagy elutasításra kerül.
- Callback - a függvény akkor kerül végrehajtásra, ha a promise-t értékkel hajtják végre.
- Errback - függvény végrehajtásra kerül, ha a promise-t elutasítják.

Az ilyesmi kódokat kerülhetjük el vele (**Pyramid of Doom)**:

```
step1(function (value1) {
    step2(value1, function(value2) {
        step3(value2, function(value3) {
            step4(value3, function(value4) {
                // Do something with value4
            });
        });
    });
});
```

With a promise library, it can flatten the pyramid.

```
Q.fcall(promisedStep1)
.then(promisedStep2)
.then(promisedStep3)
.then(promisedStep4)
.then(function (value4) {
    // Do something with value4
})
.catch(function (error) {
    // Handle any error from all above steps
})
.done();
```