http://blog.vigowebs.com/post/nodejs-interview-questions-part-2

Mi a middleware? Mi a middleware eszközök fogalma az Express.js-ben?

A middleware-ek, amelyek hozzáférést biztosítanak a kérésobjektumhoz (req), a válaszobjektumhoz (res) és az alkalmazás kérés-válasz ciklusának next middleware funkciójához. A next middleware függvényét általában egy next nevű változóval jelöljük.

A middleware funkciók a következő feladatokat hajthatják végre:

- Végezzen el bármilyen kódot.
- Végezze el a kérés (request) és a válaszobjektumok (response) módosítását.
- Fejezze be a kérés-válasz ciklust.
- Hívja meg a verem next middleware funkcióját.

alkalmazásobjektum egy példányához az app.use () és az app.METHOD () függvények használatával, ahol a METHOD a köztes program függvény által kezelt kérés (például GET, PUT vagy POST) kisbetűvel.

```
var app = express()

app.use(function (req, res, next) {
  console.log('Time:', Date.now())
  next()
})
```

Mi a Child Process és a Cluster? Mi a különbség közöttük?

A Node.js-ben található gyermekfolyamat (Child Process) egy gyermekfolyamat futtatására szolgál a Node.js alatt. Ennek két módja van: exec és spawn. Az exec példa:

```
var exec = require('child_process').exec;
exec('node -v', function(error, stdout, stderr) {
  console.log('stdout: ' + stdout);
  console.log('stderr: ' + stderr);
  if (error !== null) {
    console.log('exec error: ' + error);
}
});
```

Az exec első argumentumaként adjuk át a parancsunkat, és három válaszra számítunk a callback-ben. Az stdout az a kimenet, amelyet elvárhatunk a végrehajtástól.

A Cluster egyetlen folyamat több folyamatra vagy worker-re bontására használják a Node.js terminológiában. Ezt egy Cluster modul segítségével lehet elérni. A Cluster modul lehetővé teszi gyermekfolyamatok (workers) létrehozását, amelyek megosztják az összes szerver portot a fő Node folyamattal (master).

A cluster hasonló worker-ek csoportja, amely egy szülő Node-folyamat alatt fut.

Mik a globális objektumok a Node.js-ben?

A Node.js tartalmaz néhány globális objektumot, amelyek minden modulban elérhetők. Ezen objektumok némelyike valójában nem globális, hanem a modul hatókörű.

- __dirname: Az aktuális modul könyvtár neve. Ez megegyezik a __filename path.dirname () nevével.
- __filename: Az aktuális modul fájlneve. Ez az aktuális modulfájl abszolút elérési útja.
- exports: A module.exports hivatkozás, amely rövidebb leírni. az export valójában nem globális, hanem lokális az egyes modulok számára.
- require: Modulok igényléséhez valójában nem globális, hanem lokális az egyes modulok számára.

Miért használja forever-t a Node.js során?

a forever egy node.js csomag, amelyet a szerver életben tartására használnak akkor is, ha a szerver összeomlik / leáll. Amikor a Node szerver valamilyen hiba, kivétel stb. miatt leáll, forever automatikusan újraindítja.

```
forever start app.js
```

Végül a forever célja, hogy a gyermekfolyamatot (például a node.js webszerverünket) folyamatosan futtassa, és automatikusan újraindítsa, ha váratlanul kilép. A forever néhány beállítása

- forever nyomon követi a * .fvr fájlokban futó folyamatokat, amelyek a / tmp / forever / pids fájlba kerülnek
- Minden forever folyamat egyedi naplófájlt generál az /tmp/forever/*.log fájlba
- Hacsak másként nincs meghatározva, a gyermek folyamat 'stdout és stderr kimenete a fenti naplófájlba kerül.

Melyek a Node.js használatának fő előnyei és hátrányai?

A Node.js néhány előnye:

- Könnyen megtanulható: Mivel a JavaScript az egyik legnépszerűbb programozási nyelv a front-end fejlesztéshez, szinte minden front-end fejlesztő ismeri ezt az univerzális nyelvet. Ezért sokkal könnyebb áttérni a Node.js használatára.
- Több szabadság: Ellentétben néhány olyan háttérprogrammal, mint a Ruby on Rails, amely egy olyan keret, amely szabályokat és irányelveket ír elő a szoftverek speciális fejlesztésére, a Node.js sokkal több teret és szabadságot ad a saját módunk szerint.

- Egyidejű kérelmek kezelése: A Node.js biztosítja a nem blokkoló IO rendszert, amely lehetővé teszi számunkra, hogy egyszerre számos kérést feldolgozzunk. A rendszer sokkal jobbá teszi az egyidejű kéréskezelést, mint más nyelveken, mint a Ruby vagy a Python.
- Gazdag közösség: A Node.js fejlesztői közösség egy nagyon aktív és élénk fejlesztői csoport, akik hozzájárulnak a Node.js folyamatos fejlesztéséhez.

Néhány hátrány:

- Instabil API és következetlenségek: A Node.js egyik legnagyobb hátránya, hogy hiányzik a következetesség. A Node.js API-ja gyakran változik, és a változások gyakran nem kompatibilisek egymással.
- Nem alkalmas nagy számításos alkalmazásokhoz: A Node.js még nem támogatja a többszálas programozást. Sokkal bonyolultabb alkalmazásokat képes kiszolgálni, mint a Ruby, de nem alkalmas nagy számítások végrehajtására.
- Eseményvezérelt modell: Az eseményvezérelt modell sok programozót összezavar, akik újak a JavaScript-ben. A callback chain nagyon hosszú lehet, ami megnehezíti a karbantartást.

Miért használjuk a __filename –t a Node.js-ben?

A __filename a Node.js egyik globális objektuma, és a végrehajtandó kód fájlnevét jelenti. Ez a kódfájl feloldott abszolút útvonala.

A program helye alapján a következőképpen adja vissza a fájl nevét:

```
//main.js
console.log( __filename );
```

```
/web/com/1427091028_21099/main.js
```

Felépíthetünk egy REST szolgáltatást a Node.js-sel?

Igen. A kiszolgálón az útválasztás annak meghatározására vonatkozik, hogy az alkalmazás hogyan reagál egy adott végpont kliens kérésére, amely egy URI (vagy elérési út) és egy adott HTTP kérési módszer (GET, POST és így tovább). Az alábbiakban két alapvető útvonalat határoztunk meg (/ task, és / task / taskId), különböző módszerekkel.

A '/ task' metódusoknak kell lennie (GET és POST), míg a / task / taskId GET, PUT és DELETE.

```
'use strict';
module.exports = function(app) {

  // todoList Routes
  app.route('/tasks')
    .get(/* get method response */)
    .post(/* delete method response */);


  app.route('/tasks/:taskId')
    .get(/* get method response */)
    .put(/* put method response */)
```

```
    .delete(/* delete method response */);
};
```

Melyek a Node.js fontos parancssori beállításai?

A leggyakrabban használt Node.js CLI-beállítások a következők:

- --version vagy -v - A --version, vagy röviden, node -v csomópont használatával kiírhatjuk a Node.js általunk használt verziót.
- --eval vagy -e - A --eval opcióval futtathatjuk a JavaScript kódot közvetlenül a terminálról. A REPL-ben előre definiált modulok igény nélkül is használhatók, például a http vagy az fs modul.
- --inspect [= host: port] - A node használata --inspect aktiválja az inspector-t a megadott gazdagépen és porton. Ha nincsenek megadva, akkor az alapértelmezett érték 127.0.0.1:9229. A Node.js példányokhoz csatolt hibakereső eszközök tcp porton keresztül kommunikálnak a Chrome hibakeresési protokoll segítségével.
- --zero-fill-buffers - A Node.js a --zero-fill-buffers parancssori opcióval indítható, hogy az újonnan kiosztott Buffer-példányokat automatikusan nullára töltsék a létrehozáskor. Ennek oka az, hogy az újonnan kiosztott puffer példányok érzékeny adatokat tartalmazhatnak.
- --check vagy -c - A --check opció utasítja a Node.js-t, hogy ellenőrizze a megadott fájl szintaxisát anélkül, hogy azt ténylegesen végrehajtaná.
- --prof-process - A --prof-process használatával a Node.js folyamat a v8 profiler kimenetét adja ki.

Hogyan kezelhetjük a Node.js programban a kezeletlen kivételeket?

A production rendszerben nagyon fontos az el nem kapott kivételek rögzítése is. Például. hiányzik egy egyszerű típusú konverzió, vagy egy fájl írásvédett, akkor alkalmazásunk nem működik, és nem tudjuk, miért. Vagy az alkalmazásunk automatikusan újraindul, és nem látunk hibát.

A Node.js fájlban egy uncaughtException esemény indul. Az eseményt egyszerű módon rögzíthetjük:

```
process.on("uncaughtException", function(err) { ... });
```

Mi a QueryString használata a Node.js-ben?

A Node.js Query String való a query string kezelésére. Használható a query string konvertálására JSON objektummá és fordítva.

A query string használatához a requre('querystring') parancsot kell használnunk.

A querystring.parse () metódus elemzi az URL-lekérdezési karakterláncot (str) kulcs- és értékpárok gyűjteményévé. Például a 'foo = bar & abc = xyz & abc = 123' lekérdezési karakterlánc a következőre van elemezve:

```
{
```

```
  foo: 'bar',
  abc: ['xyz', '123']
}
```

A querystring.parse () metódussal visszaadott objektum prototípusosan nem öröklődik a JavaScript objektumból. Ez azt jelenti, hogy a tipikus Object módszerek, mint például az obj.toString (), az obj.hasOwnProperty () és mások, nincsenek meghatározva, és nem fognak működni.

Hogyan védhetjük a sütiket a Node.js fájlban?

Az alábbi lépéseket követhetjük a sütik biztosításához:

- HttpOnly - A HttpOnly egy olyan jelző, amely bekerülhet a Set-Cookie válasz fejlécébe. Ennek a jelzőnek a jelenléte arra szólítja fel a böngészőket, hogy ne engedélyezzék a klines oldali szkript hozzáférését a cookie-hoz (ha a böngésző támogatja). Ez azért fontos, mert segít megvédeni a cookie-adatainkat a rosszindulatú szkriptektől, és enyhíti a leggyakoribb XSS-támadásokat.
- Secure - Ugyanolyan fontos, mint a HttpOnly flag a Secure flag. Ez is szerepel a Set-Cookie válasz fejlécében. A secure flag jelenléte arra utasítja a böngészőket, hogy ezt a cookie-t csak HTTPS-végpontokhoz érkező kérésekben küldjék el. Ez nagyon fontos, mivel a sütire vonatkozó információkat nem titkosítatlan csatornán küldjük el. Ez segít enyhíteni néhány olyan kihasználást, amikor a böngészője egy webhely HTTP-végpontjára irányul át, nem pedig a HTTPS-végpontra, és ezáltal potenciálisan kiteszi a cookie-kat a forgalom közepén lévő valakinek.

Mit jelent a tilde és a caret a package.json fájlban?

A legegyszerűbben kifejezve, a tilde egyezik a legfrissebb kisebb verzióval (a középső számmal). Az ~ 1.2.3 minden 1.2.x verziónak megfelel, de az 1.3.0-nak nem.

Az caret viszont lazább. Frissítésre kerül a legfrissebb nagyobb verzióra (az első számra). ^ Az 1.2.3 bármelyik 1.x.x kiadással megegyezik, beleértve az 1.3.0 verziót is, de a 2.0.0 verzió már nem.

Hogyan ellenőrizhetjük a Node.js csomagjainak verziószámát?

Ha az npm 5 előtti verzióját használjuk, akkor használhatjuk az npm shrinkwrap alkalmazást. Ez lezárja a jelenleg használt npm modulok verzióit..

Az npm 5 kiadásával automatikusan létrehozza a shrinkwrap-hoz hasonló package-lock.json fájlt, amikor az npm telepítést alapértelmezés szerint futtatjuk. Kommitolni kell a lock vagy shinkwrap fájlokat is.

Megnézhetjük a yarn-t is. A egy yarn.lock fájllal kerül ki, amely ugyanúgy működik, de további teljesítménysebességgel és offline képességekkel rendelkezik.

Különbség a dependencies és a dev-dependencies között?

Dependecy-k szükségesek és kulcsfontosságúak az alkalmazásunk futtatásához, a devDependency-k csak a fejlesztéshez szükségesek, pl .: unit tests, ES6 to Javascript transpilation, minification.

A fejlesztés során ez nem számít, mert a rendszeres npm install úgyis mindkét függőségi készletet telepíti, de ha a NODE_ENV production értékre van állítva, akkor az npm kihagyja a devDependencies-eket. A --only opcióval kifejezetten kiválaszthatjuk, hogy mit akarunk telepíteni:

npm install --only = prod [uction] csak függőségeket telepít

npm install --only = dev [elopment] csak a devDependencies-eket telepíti

Tegyük fel, hogy continous deployment szolgáltatást használunk. Ha a NODE_ENV-t production-ra állítjuk, és a webpack a devDependencies alatt van felsorolva (mint sok általam látott kódbázisban), akkor a production sikertelen lesz, mert a webpack nem lesz telepítve. Mindig tesztelhetjük, hogy megfelelően helyeztük-e el a függőségeket, ha telepítjük őket a --only = prod opcióval.