Node.js

v4.0.0

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Mi a Node.js?

- JavaScript futtatókörnyezet a böngészőn kívül
  - Chrome V8 motor
  - Első verzió: 2009 május 27
  - Aktuális verzió: 4.0.0, 2015 szeptember 8.
- Segítségével gyorsan, nagyon jól skálázható, hálózatba kapcsolt alkalmazásokat lehet készíteni.
- Könnyű és hatékony eseményvezérelt, nem blokkoló I/O architektúrát használ
- Nagyon jó választás adat intenzív, valós idejű elosztott alkalmazások készítéséhez.
- Böngészőn kívüli JavaScript futtatás ☺

Get Ready for Node v0.12

NODE IS MORE POPULAR THAN EVER

Five years after its debut, Node.js is the third most popular project on GitHub.

OVER 2 MILLION DOWNLOADS PER MONTH

OVER 20 MILLION DOWNLOADS OF v0.10x

OVER 475 WORLDWIDE MEETUPS

NODE IS DEPLOYED BY BIG BRANDS

Big brands are using Node to power their business

Manufacturing

- BMW
- GE
- General Motors
- Johnson Controls
- SIEMENS

Financial

- ADP
- citigroup
- Goldman Sachs
- PayPal
- WELLS FARGO

eCommerce

- amazon.com
- BEST BUY
- ebay
- TARGET
- Zappos.com

Media

- beatsMUSIC
- CONDÉ NAST
- DOWJONES
- The New York Times
- SONY

Technology

- salesforce.com
- box
- intel
- YAHOO!

Nem blokkoló I/O műveletek

- Minden ami nem közvetlenül a memórián hajtódik végre
  - Fájlműveletek
  - Adatbázis műveletek
  - Külső szolgáltatások használata
  - ...
- Probléma
  - Blokkoló műveleteknél meg kell várni amíg az adott metódus végez
- Megoldás
  - Ezeket a műveleteket külön szálon hajtsuk végre
  - Legtöbb nyelvben van erre támogatás
- Node.js-ben csak aszinkron I/O műveletek érhetők el! (ami véletlenül mégsem azt nem használjuk ☺)

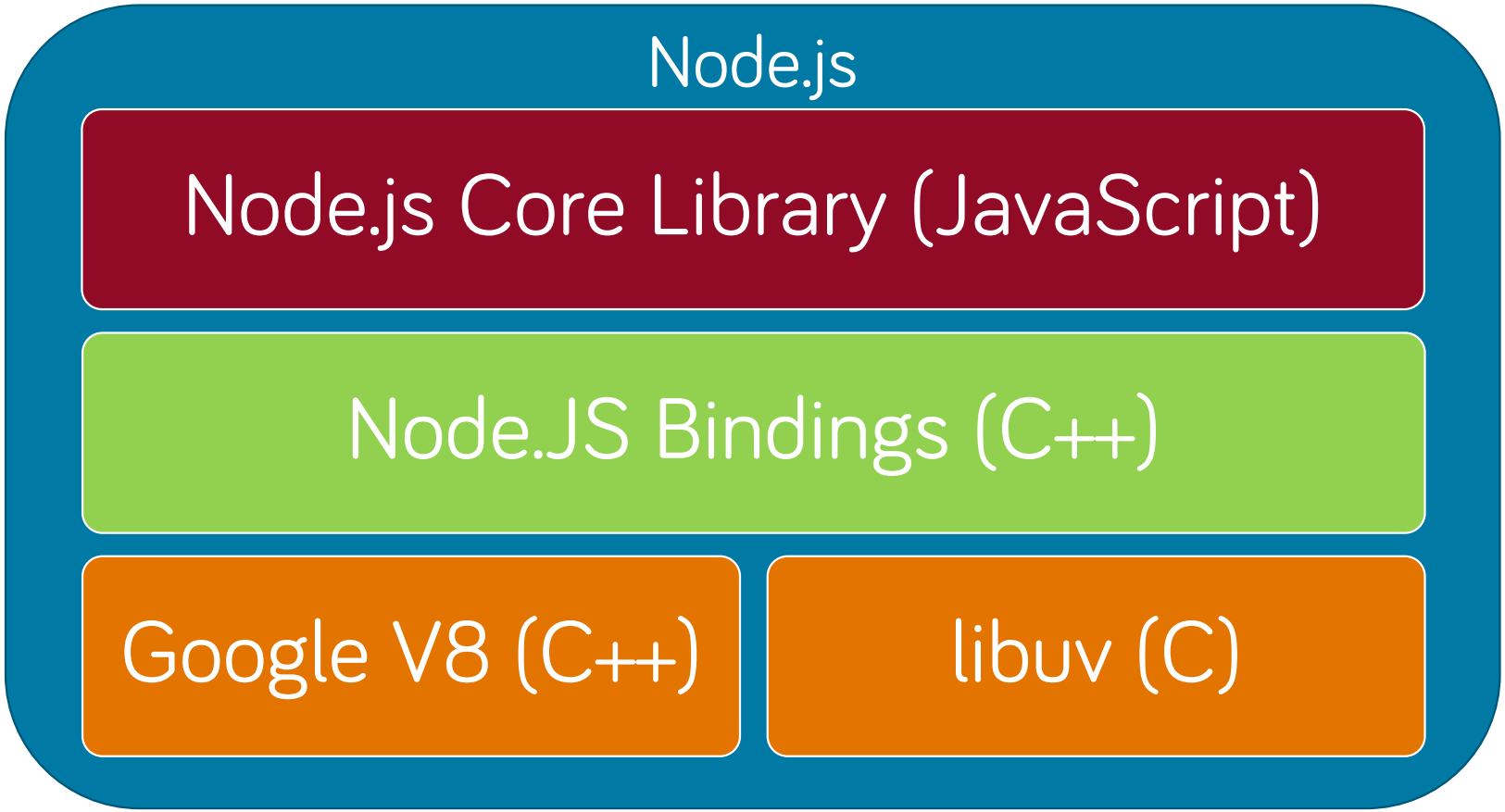
Szerver oldalon JavaScript?

- Böngészőben a JavaScript DOM eseményekkel dolgozik
- Egy szál használata
- Csak nem blokkoló I/O műveletek

```
$("#mySubmitButton").click(function(event){
  event.preventDefault();
  alert("My form is submitted!");
});
```

```
$.get("http://nodejs.org", function(data){
  document.body.innerHTML = data;
});
```

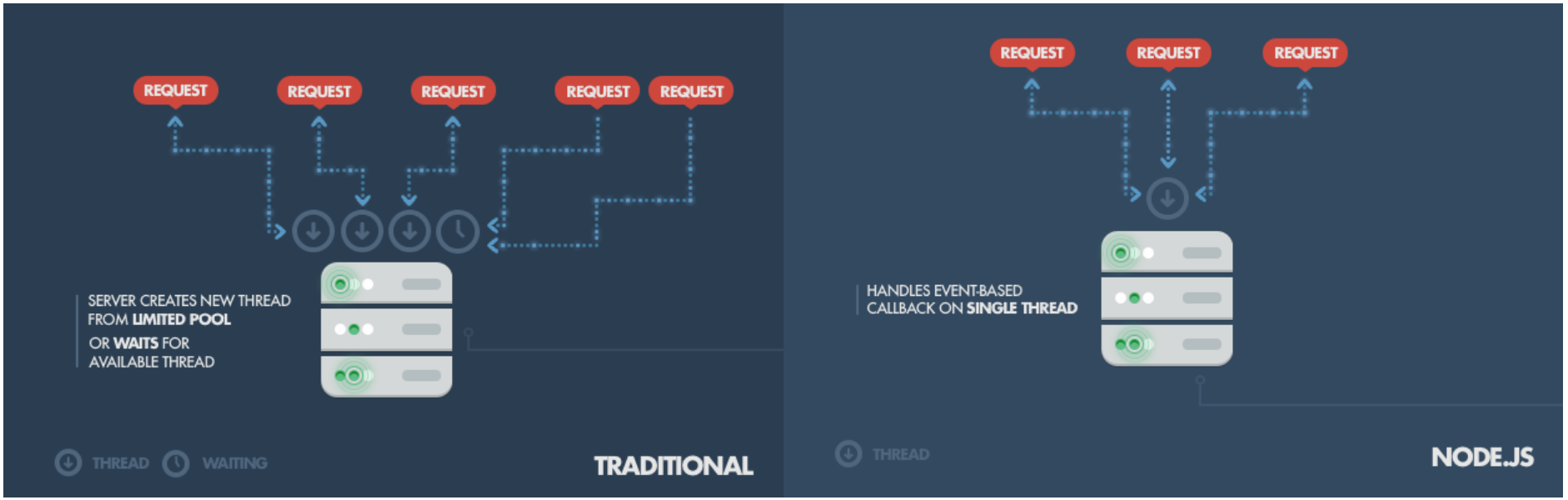
Hogyan működik a Node.js



Beépített osztálykönyvtárak

Buffer Child Processes Cluster
Console Crypto Debugger DNS
Domain Events File System HTTP
HTTPS Modules Net OS Path
Process QueryStrings Readline REPL
Smalloc Stream String Decoder
Timers TLS/SSL TTY UDP/Datagram
URL Utilities VM ZLIB

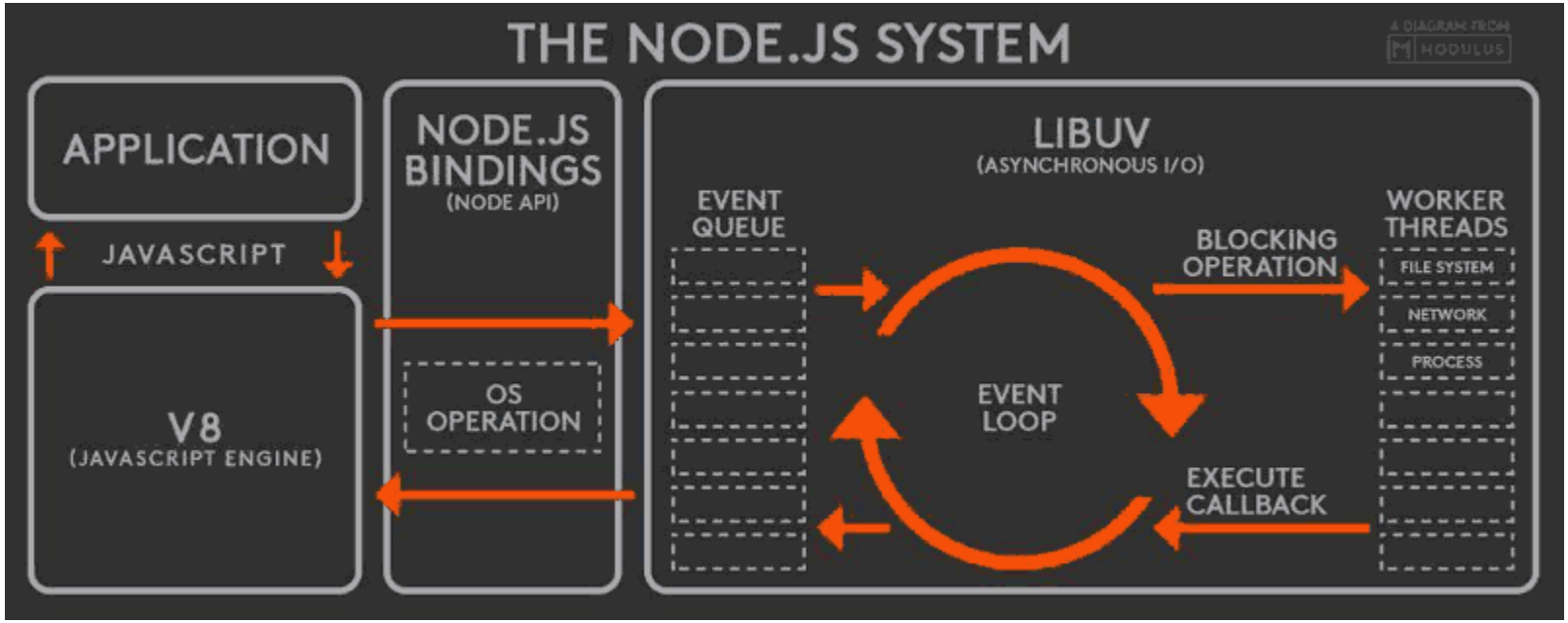
Hogyan működik a Node.js



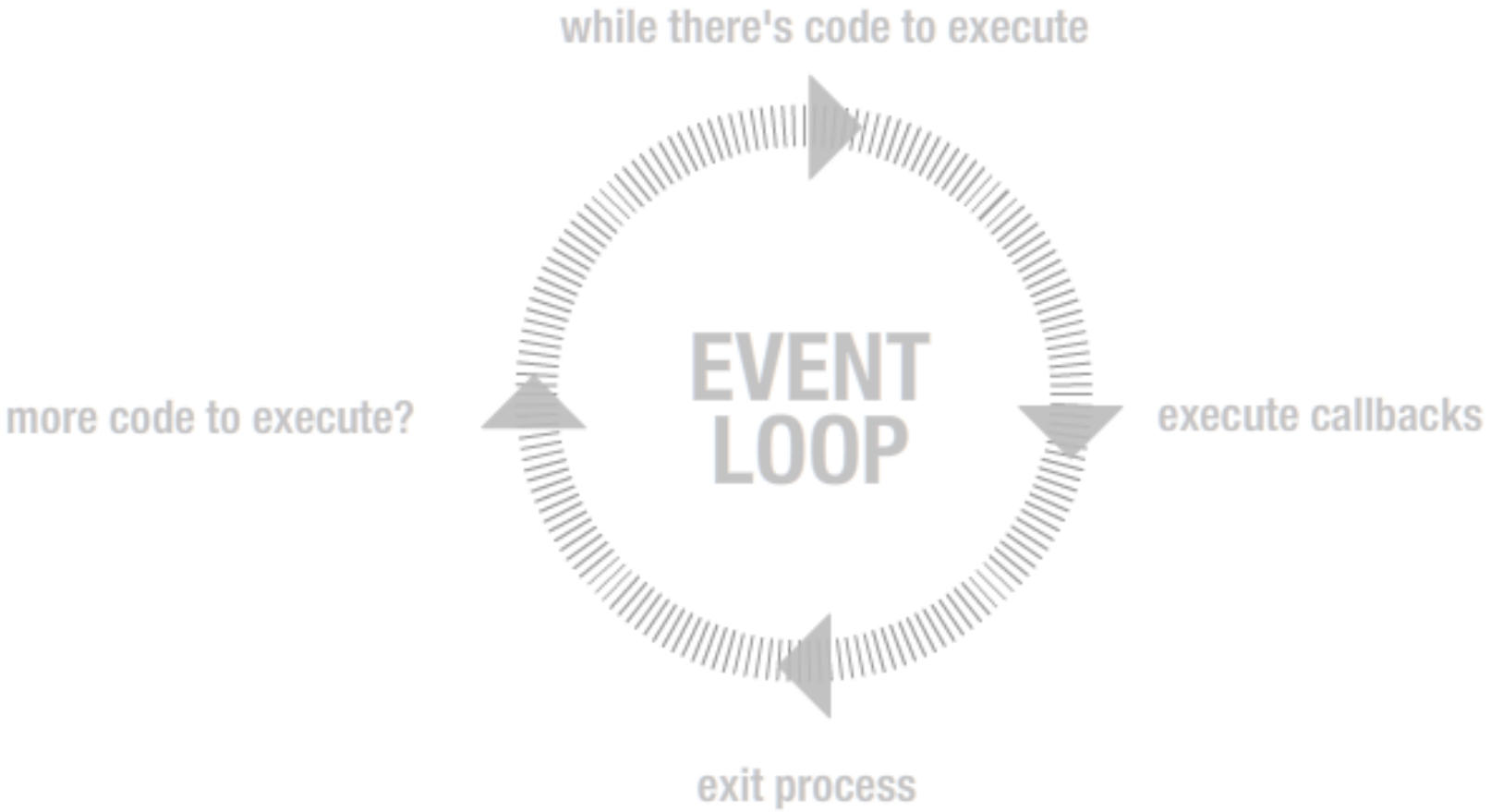
Node.js teljesítménye

- Hagyományos kiszolgáló esetén:
  - Egy kérés kiszolgálására indított thread mérete kb 1 MB
  - 16 GB memória esetén ~16000 konkurens kapcsolatot lehet maximum kiszolgálni
- Node.js szerver esetén:
  - 16 GB memóriával rendelkező VPS-en 1 millió konkurens kapcsolatot sikerült elérni
  - http://blog.caustik.com/2012/08/19/node-js-w1m-concurrent-connections/

Node.js Event loop



Node.js Event loop



Callbacks

- Legfontosabb konvenció Node.js-ben a callback függvények használata
- Probléma:
  - Minden I/O, komplex függvény aszinkron kerül végrehajtásra
  - Viszont egy programban tipikusan több ilyen műveletet hajtunk végre, amelyek egymás kimenetére van szüksége
- Megoldás:
  - Egy aszinkron metódusban nem értékkel térünk vissza hanem, egy paraméterül kapott függvényt hívunk meg az eredményt átadva

Callbacks

```
var fs = require('fs')
var myNumber = undefined

function addOne(callback) {
  fs.readFile('number.txt', function(err, fileContents) {
    myNumber = parseInt(fileContents);
    myNumber++;
    callback();
  })
}

function logMyNumber() {
  console.log(myNumber)
}

addOne(logMyNumber)
```

Callback hell

```
fs.readdir(source, function(err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function(filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function(err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function(width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(destination + 'w' + width + '_' + filename,
              function(err) {
                if (err) console.log('Error writing file: ' + err)
              })
          }.bind(this))
        }
      })
    })
  }
})
```

Promises/A+

- Egy megoldás a callback hell problémára
- Minden aszinkron függvény visszaad egy Promise objektumot
  - Szinkron módon
- Promise objektumnak ezután át tudunk adni egy függvényt ami végrehajtódik amint az előző aszinkron művelet végez
  - Nem kell explicit callback paramétereket megadni
  - Promise egységes felületet ad
  - Egyszerűbb hibakezelésre ad lehetőséget
  - Promise-ok összeköthetők, különböző control flow-k alapján (láncban, párhuzamosan, stb.)
- ES 6 nyelvi elem, de számos ES 5 programkönyvtár tartalmazza

Event emitters

- Node.js eseménykezelő modul
  - Beépített I/O modulok is ezt használják
- Legfontosabb függvényei
  - emitter.on(event, listener) – event bekövetkezésekor meghívja a listener cb-t
  - emitter.once(event, listener) – csak egyszer hívja meg a listener-t
  - emitter.removeListener(event, listener) – esemény kezelő leiratkozás
  - emitter.emit(event[, arg1][, arg2][, ...]) – esemény kiváltás

Streams

- Probléma
  - Node.js I/O műveletek aszinkronok
  - Egy nagy fájlt csak a teljes beolvasás után lehetne a callback-nek odaadni
  - Ez nagyon pazarló és memória igényes
- Megoldás
  - Adatfolyamok használata
    - Folyamatos olvasás és feldolgozás, transzformáció
    - Streamnek adható callback függvény mindig adat szeleteket kap amiknek elkezdődhet a párhozamos feldolgozása
    - Ha szükség van rá egy adatfolyamot lehetséges teljesen a memóriába is olvasni
- Példa
  - Log állományok feldolgozása hatékonyan

Stream írás és olvasás

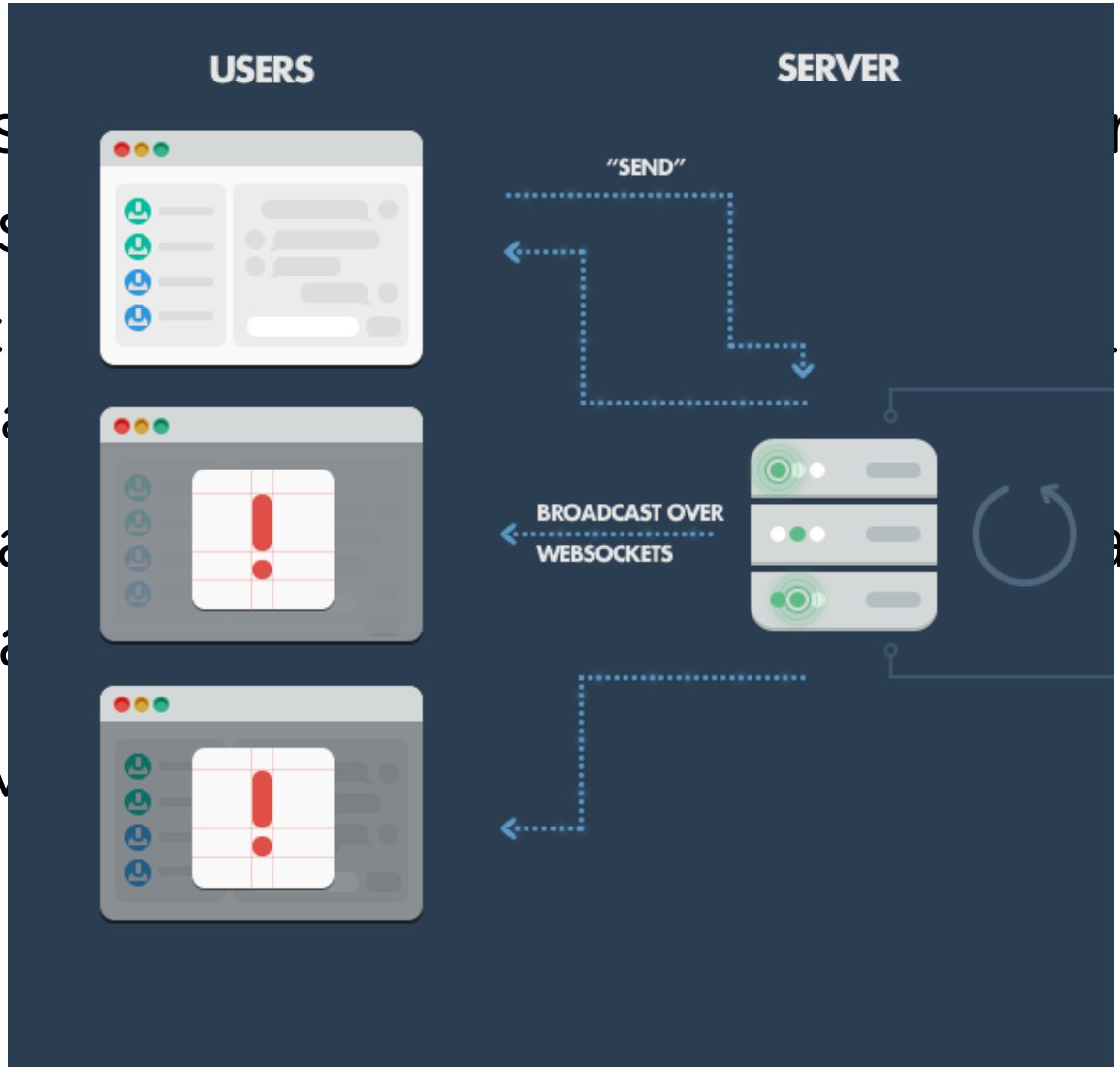
- Olvasható stream
  - Meghatározott feldolgozott adatmennyiség után egy eseményt süt el, benne a beolvasott adatszelettel
  - Az adatforrás vége esetén egy *end* eseményt küld
  - Erre az eseményre feliratkozhatunk
  - Olvasás leállítható és újraindítható
- Írható stream
  - Hasonló működés az olvasható streamhez, de itt explicit hívjuk a *write* függvényt
    - Ennek visszatérési értéke egy *bool* érték, amely azt jelzi tud e még fogadni írást

Buffers

- Bináris adatok kezelésére szolgál
  - Például: fájlok, képek, stb..
- Jellemzően stream-ből tölthető fel, és stream-be írható ki

WebSockets

- Valós r és
  kliens
  - > Ez e nem
    csa
- Tiszta ack-ek
  alapj
- Szerv



Clustering

- Node.js alap esetben egy szálon fut, ahol az event loop
  - Célszerű lenne bizonyos események végrehajtását külön szálon végrehajtani
- Cluster modul segítségével a fő szálhoz kapcsolódó worker szálakat lehet indítani
  - Beépített load-balancer (round-robin) a feladatok optimális elosztásához
  - Nekünk kell a worker process-eket elindítani
  - Az event loop továbbra is egy szálon fut és érhető el
- Process managerek elintézik ezt helyettünk, pl:
  - StrongPM
  - PM2

Hibakezelés

- try catch aszinkron metódusokkal nem működik

```
function myApiFunc(callback)
{
  try {
    doSomeAsynchronousOperation(function (err) {
      if (err)
        throw (err);
      /* continue as normal */
    });
  } catch (ex) {
    callback(ex);
  }
}
```

Hibakezelés

- Kivételek és hibák a callback metódusokon keresztül kell hogy terjedjenek
- Error first konvenció
  - A callback függvén első bemeneti paramétere mindig egy *error* objektum
  - A hibát tartalmazó objektum mindi származzon a beépített *Error*-ból

```
function error(res,next) {
  return function( err, result ) {
    if( err ) {
      res.writeHead(500); res.end(); console.log(err);
    }
    else {
      next(result)
    }
  }
}
```

Hibakeresés

- Konzolba írás
  - console.log .error .info
- Node-inspector (~Chrome Developer Tools)
  - Set breakpoints (and specify trigger conditions)
  - Step over, step in, step out, resume (continue)
  - Inspect scopes, variables, object properties
  - Edit variables and object properties
  - Break on exceptions
  - Disable/enable all breakpoints
  - CPU and HEAP profiling
  - Network client requests inspection
  - Console output inspection

Hibakeresés WebStormmal

- Telepített Node.js plugin szükséges hozzá
- Run és Debug configuraition-t kell beállítani
  - Node.exe helye
  - Alkalmazás belépési pontjának megadása
  - Környezeti változók beállítása
- Támogatott hiba keresési módszerek
  - Helyi hibakeresés, hasonlóan mint például Java és C# esetén
  - Live Editing hibakeresés közben
  - Távoli hibakeresés, csatlakozás távoli debug módban indított alkalmazáshoz
    - Akár folyamatos éles működés közben lehet így tesztelni

Automatizált tesztelés

- Miért írunk automatizált tesztet?
  - Biztos minden úgy működik-e mint ahogy szeretnénk?
  - Nem rontottam el más ezzel a változtatással?
  - ...
- JavaScript teszt keretrendszerek léteztek már Node.js előtt is, jellemzően böngészőben futottak, pl:
  - Qunit
  - Jasmine
  - ...
- Node.js-hez több teszt keretrendszer is készült, a legnépszerűbb:
  - Mocha

Mocha

- Nagyon sok beépített funkcióval rendelkező JavaScript teszt keretrendszer
- Futtatható
  - Node.js alatt
  - Böngészőben
- Segítségével egyszerűen lehet aszinkron műveletek tesztelni
- Fejlett hiba kezelő és elkapó mechanizmust tartalma
- Tesztek sorrendben, és szinkron futnak, így:
  - Determinisztikus végrehajtás
  - Konzisztens jelentés generálás

Mocha

- Példa teszt kód:

```
var assert = require("assert");
describe('Array', function() {
  describe('#indexOf()', function () {
    it('should return -1 when the value is not present', function () {
      assert.equal(-1, [1,2,3].indexOf(5));
      assert.equal(-1, [1,2,3].indexOf(0));
    });
  });
});
```

Mocha

- Eredmény a konzolon:

```
Adam@SZADAM-WORK C:\Tools
> mocha mochademo.js


  Array
    #indexOf()
      √ should return -1 when the value is not present


  1 passing (14ms)


Adam@SZADAM-WORK C:\Tools
>
```

Feltétel vizsgálat - Chai

- Tesztek írásakor jellemzően az utolsó ellenőrző fázisban komplex vizsgálatokat kell végrehajtani
- Erre kínálnak megoldást az Assertion library-k, mint például a Chai
- Mocha tesztekben tetszőleges ilyen osztálykönyvtárat használhatunk, amely egy vizsgálat sikertelensége esetén kivételt dob.
- Chai
  - BDD és TDD stílus támogatás
    - Should, expect, assert
  - Könnyen bővíthető

Chai assert stílusok

Should

```
chai.should();

foo.should.be.a('string');
foo.should.equal('bar');
foo.should.have.length(3);
tea.should.have.property('flavors')
  .with.length(3);
```

Expect

```
var expect = chai.expect;

expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.length(3);
expect(tea).to.have.property('flavors')
  .with.length(3);
```

Assert

```
var assert = chai.assert;

assert.typeOf(foo, 'string');
assert.equal(foo, 'bar');
assert.lengthOf(foo, 3)
assert.property(tea, 'flavors');
assert.lengthOf(tea.flavors, 3);
```

Node.js vs IO.js

- IO.js a Node.js fork-ja egészen idén szemptember 8-ig.
  - Node.js alkalmazások változtatás nélkül futnak IO.js alatt
- V8 motor frissítéseinek gyakoribb beemelése a platformba
  - Így az IO.js sokkal több ES 6.0 szolgáltatásokat implementált
- Verziók
  - Szeptember előtt Node.js 0.x verziókat adott ki
  - IO.js 1.0.0-tól indult és 3.3.0 az utolsó stabil verziója
- 2015 szeptember 8.-án összeolvadt a Node és IO.js 4.0.0 verziószámmal

Mi nem a Node.js, és mikor nem érdemes használni?

- Nem egy teljes MVC stack
  - Önmagában csak a legfontosabb osztálykönyvtárakat, package manager-t és futtató környezetet tartalmazza
  - Természetesen léteznek Node.js alapú open source megoldások rá
- Nem érdemes CPU intenzív feladatokra használni
  - Hiába a nagyon jó JavaScript motor, jellemzően egy jól megírt C, C++ kód nagyságrendekkel gyorsabb
  - Lehetséges C++-ban írt modulok elérése Node.js alól
- Statikus weboldalak, illetve állományok kiszolgálására jobb választás egy statikus fájlszerver, pl Apache, IIS

Node.js gyakorlat