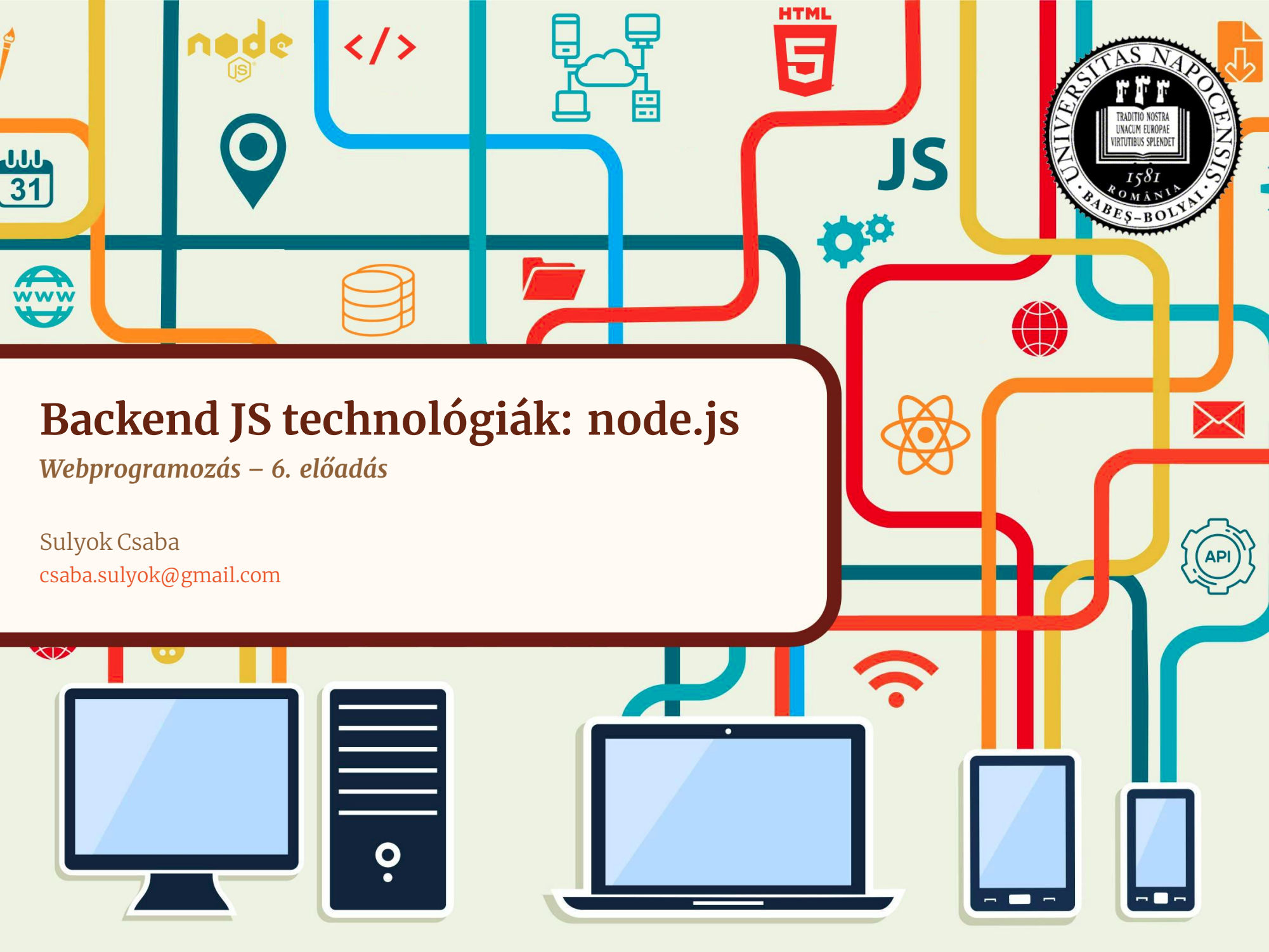


Backend JS technológiák: node.js

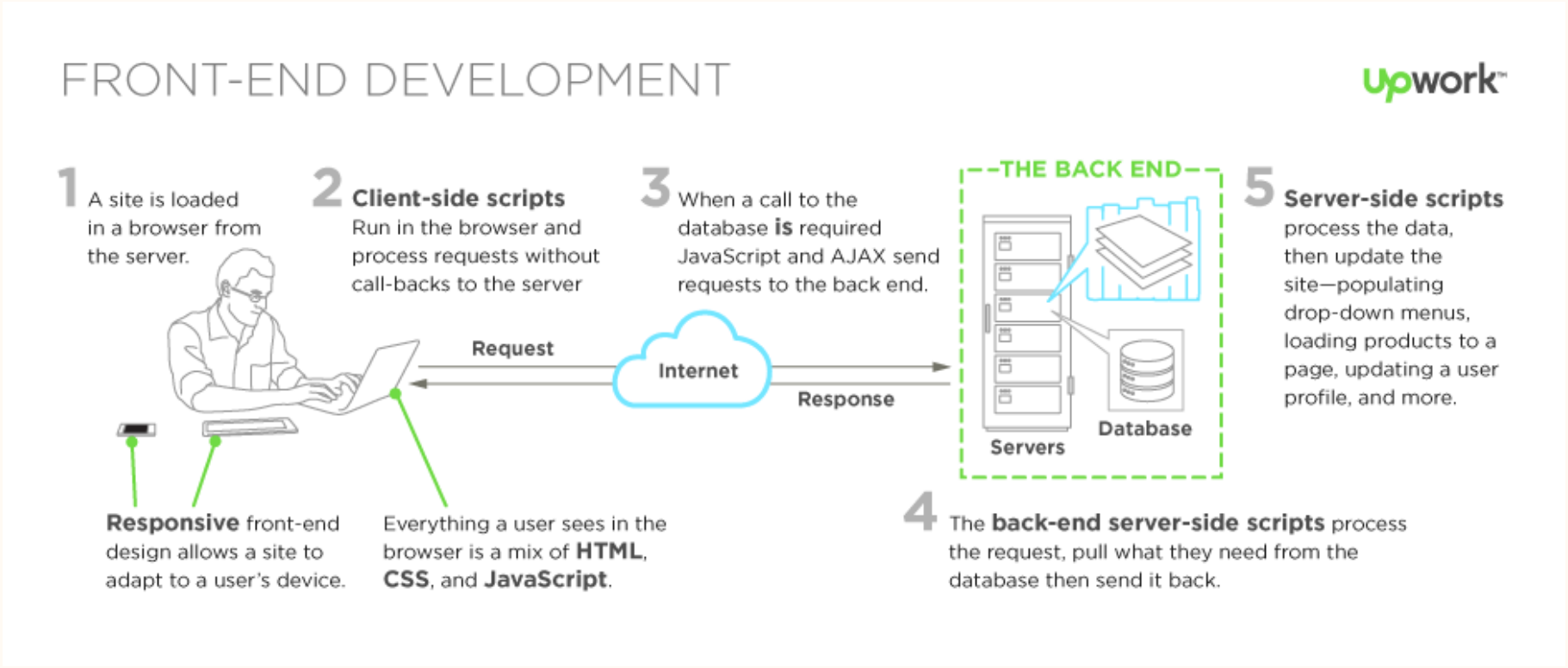
Webprogramozás – 6. előadás

Sulyok Csaba
csaba.sulyok@gmail.com

1. rész

Szerver oldali technológiák

Frontend-backend



forrás: upwork.com

Frontend-backend

Frontend

- kliensoldali technológiák
- minden, amit a felhasználó lát, és amivel interakcióba léphet
- publikus (nem biztonságos)
- HTML, CSS illetve kliensoldali JavaScript ide tartoznak

Backend

- szerveroldali technológiák
- feldolgozzák a kliensoldalról érkezett kéréseket
- biztonságosabb; nem publikus a működése

Statikus vs. dinamikus szerveroldal

Statikus

- a válaszok nem változnak többszörös kérésre
- gyorsak
- általában egy-egy állomány tartalmát térítik vissza változatlanul
- nem lehet őket programozni
- példák: *nginx*, *Apache*

Dinamikus

- programozhatóak
- így bármilyen aspektust figyelembe vehetnek, hogy kérésekre válaszokat generáljanak (kliens elhelyezkedése, be van-e jelentkezve, stb.)
- lassabbak
- példák: $\longrightarrow$

*A static website is a **vending machine**.*
*A dynamic website is a **restaurant**.* (Noah Veltman)

Dinamikus szerveroldali technológiák

- Bármely programozási nyelv, amely képes natív hálózati kommunikációra, használható dinamikus szerveroldali technológiaként
  - pl. C-ben socketek segítségével
  - pl. Java-ban Socketek segítségével
- Ilyen alacsony szintű megoldások könnyen vezetnek hibákhoz, mivel a teljes protokoll ismeretét le kell programozni.
- Ezért *általános célú nyelvek*kel együtt használunk **könyvtárakat** és **keretrendszereket**.
- Így kihasználhatjuk a nyelvek minden webtől független adottságát (pl. kapcsolódás adatbázishoz, függőségkezelő rendszer, stb.).

Dinamikus szerveroldali technológiák

- **PHP** (PHP Hypertext Preprocessor) - saját programozási nyelv
- **Servlet/JSP** (Java Server Pages) - Java webalkalmazások készítésére
- **Java EE** és **Spring Web** - Java-alapú komplex keretrendszerek, melyek ugyancsak Servleteket alkalmaznak
- **ASP.NET** (Active Server Pages) - Microsoft által fejlesztett, a .NET keretrendszer része, C# nyelvvel használják
- **Node.js**
- **Django/flask** - Pythonhoz
- **Ruby on Rails**

Dinamikus szerveroldali technológiák

- Kombinálva a dinamikus szervereket adatbázissal és/vagy más keretrendszerrel, különböző elterjedt *technology stack*et kapunk:

MAP/XAMP/LAMP:

- *MySQL* (adatbázis)
- *Apache* (statikus webszerver)
- *PHP* (szerveroldali szkript)

MEAN/MERN:

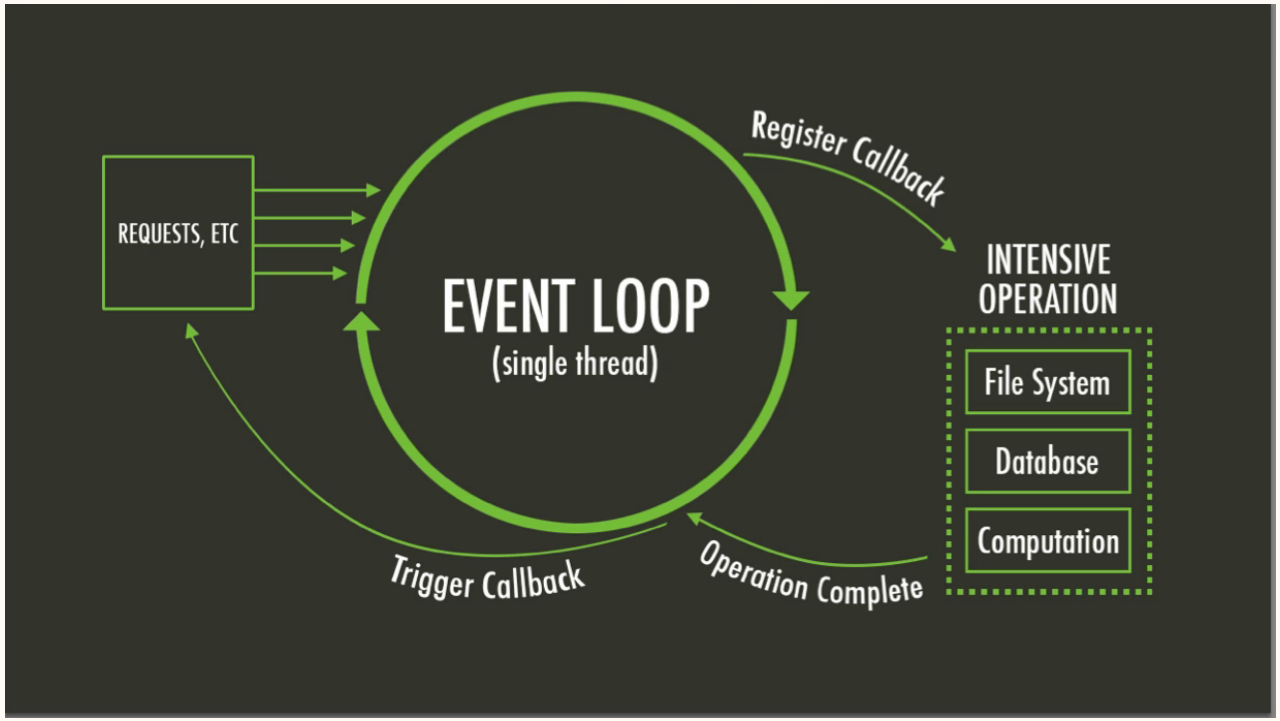
- *MongoDB* (adatbázis)
- *Express* (webszerver - node.js függőség)
- *Angular/React* (kliensoldali keretrendszer)
- *node.js* (szerveroldali szkriptmotor)

2. rész

node.js

- A **node.js** a Google V8-as JavaScript értelmező motra elválasztva a böngészőtől.
- Megengedi a JS kód futtatását parancssorból mind állományokból, mind interaktív konzolon keresztül.
- Önmagában **nem** szerveroldali technológia (csak annyira, amennyire a C is az).
- Aszinkron, eseményvezérelt, nem használ blokkoló műveleteket $\longrightarrow$ *hatékonyság*
- Skálázható hálózati alkalmazások készítésére lett tervezve
- Egyszálú végrehajtás (*single threaded*)
- Előnyös adatintenzív alkalmazások esetén *sok I/O művelet*
- A gyors és gyakori változások miatt ajánlott a legújabb **LTS** (long-term support) verzió használata

node.js event loop



- **Eseménykezelő ciklus** - események kiváltódása esetén a node.js runtime végrehajtja az illető eseményhez rendelt visszahívandó (*callback*) függvényt
- A *callback* függvényeket a programozó írja és csatolja eseményekhez.
- Event Emitter dokumentáció

node.js modulok

- a standard JavaScript értelmező motor mellett tartalmaz alap JavaScript csomagokat (`core`)
- **modulok szüksége** - HTML `script` elemek hiányában a node.js-nek szükséges külső könyvtárcsomagokat (modulokat) importálni
- node.js modularitási opciók:
  - ECMAScript 2015-ben bevezetett *modulok* (node.js 13.2-től arrafele alapértelmezett és preferált):

```
import module from 'modulename';
import module2 from './mymodulename.js';
```

  - a CommonJS konvenció - node.js-specifikus:

```
const module = require('modulename');
const module2 = require('./mymodulename');
```

  - hivatalos node.js modulok
- importálhatunk általunk írt modulokat relatív elérési útvonal megadásával
- külső modulok elérhetőek az `npm` csomagkezelő rendszer segítségével

node.js beépített modulok

- Pár példa beépített modulokra:
  - `fs` (file system) - fájlrendszerrel való interakció, állományok olvasása, írása, figyelése
  - `path` - fájl útvonalak kezeléséhez
  - `http`, `https` - HTTP(S) protokollt támogató low-level szerverek készítéséhez
  - `os` - információk a futó oprendszerről
  - `child_process` - alfolyamatok forkolása és monitorizálása
  - `stream` - adatfolyamok kezeléséhez

node.js ES2015 modulok

Példa: `3-nodejs/es2015modules`

Saját függvénykönyvtárak

`es2015module_default.js`:

```
// alapértelmezett export
export default function libFunction() {
  console.log('I am libfunction');
}
```

`es2015module_multi.js`:

```
export function libFunction1() {
  console.log('I am libfunction1');
}

export const libFunction2 = () => {
  console.log('I am libfunction2');
}
```

Főállomány `index.js`:

```
// külső (node-hoz tartozó) modul betöltése
import fs from 'fs';

// belső (relatív útvonalon elhelyezkedő)
// modul betöltése
import libFunction from './es2015module_default.js';
import { libFunction1, libFunction2 } from './es2015module_mult

// az összes exportált függvény
// közös néven való betöltése
import * as libMulti from './es2015module_multi.js';

// függvények használata

console.log('Contents of current directory',
  fs.readdirSync('.'));

libFunction();
libFunction1();
libFunction2();
```

node.js példa: állomány figyelése

Példa: `3-nodejs/watcher`

- Figyeljük egy állomány változásait. Változás esetén üzenünk a console-on keresztül, s kiírjuk az állomány tartalmát.

```
// az fs (file sytem) beépített modul interakciót enged a fájlrendszerrel
// https://nodejs.org/api/fs.html
import fs from 'fs';

const fileName = 'valami.txt';

// a watch függvény figyeli a változásokat - 2. paramétere egy callback függvény
fs.watch(fileName, async () => {
  console.log(`A ${fileName} állomány módosult!`);
  // a readFile olvassa az állomány tartalmát, majd meghívja a callbacket
  fs.readFile(fileName, (err, data) => {
    if (err) throw err;
    console.log(`Az állomány tartalma:\n${data}`);
  });
});

// egyből kiíródik ez az üzenet (mert a fenti függvények nem hívódnak meg csak események nyomán)
console.log(`Figyelem a ${fileName} állomány tartalmát`);
```

node.js webszerver

- Készítsünk egy egyszerű webszervert, amely:
  - Fogadja a HTTP kéréseket egy adott porton
  - Kiírja a konzolra a kérések adatait
  - Válaszként visszaküld egy egyszerű üzenetet, miután beállítja a fejlécet
- A szerver létrehozása alacsony szinten megoldható a beépített `http` modul használatával. Lépések:
  - szerverobjektum létrehozása: `http.createServer`
  - callback függvény csatolása a `request` eseményhez
  - a kérés kezelése *ugyancsak callback függvények segítségével*
  - válasz készítése és küldése
  - szerver aktiválása: `server.listen`

node.js webszerver

- szerverobjektum létrehozása:

```
const server = http.createServer((request, response) => {
  // handling a request happens here!
});
```

- egyenértékű hívás callback függvénnyel:

```
const server = http.createServer();
server.on('request', (request, response) => {
  // handling a request still happens here!
});
```

- `request instanceof http.IncomingMessage`
  - implementálja a `ReadableStream`-et
  - elérhető a kérés metódusa (`method`), URL-je (`url`) s fejlécinfó objektumformában (`headers`)
- `response instanceof http.ServerResponse`
  - implementálja a `WritableStream`-et
  - beleírhatóak válaszfejlécek, státuszkód, s a válasz törzse

ReadableStream, WritableStream

- Stream dokumentáció
- Működési módozatok:
  - flowing mode – *adatok folyamatos továbbítása* - ha nincsenek eseménykezelők rendelve az adatok érkezését jelző eseményekhez, akkor az adatok elvesznek.
  - paused mode – *adatszerzés érdekében explicit meg kell hívni a* `stream.read()` *függvényt.* Ez az alap állapot.
- Legfontosabb események:
  - `data` - adat-töredék *(chunk)* érkezését jelzi
  - `end` - adatok végét jelzi
  - `error` - hibát jelez.
- Eseménykezelő hozzárendelése a `data` eseményhez (`stream.on('data', callback)`) a folyamot automatikusan flowing módba kapcsolja.

Webszerver létrehozása

- a választ akkor építjük fel, ha a teljes kérést parse-oltuk - vagyis ha a `request` folyam `end` vagy `error` eseményt dob
- válasz fejlécek beállítása:

```
// explicit fejléc beírása - egyszer szabad meghívni a response.end() előtt
response.writeHead(200, { 'Content-Type': 'text/plain' });
// fejlécek hozzáadása egyesével
response.setHeader('Content-Type', 'text/plain');
```

- törzs beállítása és válasz küldése:

```
response.end(data);
// vagy
response.write(data);
response.end();
```

- szerver elindítása (ugyancsak callbackkel):

```
// callback nélkül
server.listen(8080);
// callbackkel
server.listen(8080, () => { console.log('Server listening...'); });
// az alábbi HELYTELEN
server.listen(8080);
console.log('Server listening...'); // except not really...
```

Webszerver létrehozása

Példa: `3-nodejs/simpleserver/simpleserver_lowlevel.js`

```
import { createServer } from 'http';

const server = createServer((req, res) => {
  let body = [];

  req.on('data', chunk => body.push(chunk));
  req.on('error', err => console.error(err));

  req.on('end', () => {
    body = Buffer.concat(body).toString();

    const info = `You have requested:
      - method: ${req.method}
      - URL: ${req.url}
      - headers: ${JSON.stringify(req.headers)}
      - body: ${body}
    `;
    console.log(info);

    // megadjuk a státuszkódot és fejlécinfót
    res.writeHead(200, {
      'Content-Type': 'text/plain; charset=utf-8',
    });
    // befejezi a választ
    // és visszaküldi a kliensnek
    res.end(info);
  });
});

// A listen metódus hívásának hatására aktiválódik
// a szerver és fogadja a 8080-as porton
// beérkező HTTP kéréseket
server.listen(8080, () => {
  console.log('Server listening...');
});
```

URL alapú *routing*

- Különböző útvonalak/paraméterek esetén a szerver különböző eredményeket adhat
- Használjuk az `url` beépített modult az elérési út feldolgozására

```
import { createServer } from 'http';
import { URL, URLSearchParams } from 'url';

function showParsedUrl(urlToParse) {
  const parsedUrl = new URL(urlToParse, 'http://localhost:8080/');
  const queryParams = new URLSearchParams(parsedUrl.search);

  console.log(`URL:
    href: ${parsedUrl.href}
    protocol: ${parsedUrl.protocol}
    hostname: ${parsedUrl.hostname}
    port: ${parsedUrl.port}
    pathname: ${parsedUrl.pathname}
    query: ${queryParams.toString()}
  `);
}

// Próba teljes URL-re
showParsedUrl('http://localhost:8080/myPath?param1=value1&param2=value2');
```

URL alapú *routing*

- **Példa:** `3-nodejs/pathparsing`
- Kimenet:

```
URL:
    href: http://localhost:8080/myPath?param1=value1&param2=value2
    protocol: http:
    hostname: localhost
    port: 8080
    pathname: /myPath
    query: param1=value1&param2=value2
```

- a `request.url` átadható a függvénynek, bár a `host` és `port` nem feltétlenül lesznek beállítva esetükben

URL alapú *routing*

- a `pathname` vagyis elérési út alapján döntjük el, hogy egy adott kérést hogyan szolgálunk ki, így komplex rendszer alakítható ki
- **Példa:** `3-nodejs/simplerouter` – karban tart egy számot, amelyet le lehet kérni, csökkenteni, növelni vagy újrainicializálni.

```
import { createServer } from 'http';
import { URL } from 'url';
import { parse } from 'querystring';

// számunk kiinduló értéke
let myNumber = 42;

const server = createServer((request, response) => {
  const body = [];

  request.on('data', chunk => body.push(chunk));
  request.on('error', err => console.error(err));

  request.on('end', () => {
    // lekérjük a pathname-et
    const parsedUrl = new URL(request.url, 'http://localhost:8080/');
    const query = parse(parsedUrl.search.substr(1));
    console.log(`Request with pathname ${parsedUrl.pathname} arrived`);
```

URL alapú *routing*

```
    // aszerint különböző műveleteket végzünk
    switch (parsedUrl.pathname) {
      case '/query':
        response.end(`The number is ${myNumber}`);
        break;
      case '/reset':
        myNumber = 42;
        response.end(`Reset, now the number is ${myNumber}`);
        break;
      case '/increment':
        myNumber += Number(query.with || 1);
        response.end(`Incremented, now the number is ${myNumber}`);
        break;
      case '/decrement':
        myNumber -= Number(query.with || 1);
        response.end(`Decremented, now the number is ${myNumber}`);
        break;
      default: // ismeretlen útvonal esetén hiba
        response.writeHead(404);
        response.end();
    }
  });
});
server.listen(8080, () => { console.log('Server listening...'); });
```

URL alapú *routing*

- Tesztkérések:
  - `http://localhost:8080/query`
  - `http://localhost:8080/reset`
  - `http://localhost:8080/increment`
  - `http://localhost:8080/increment?with=10`
  - `http://localhost:8080/decrement`
  - `http://localhost:8080/decrement?with=10`

Statikus webszerver példa node.js-szel

- Készítsünk egy statikus HTTP webszervert, amely:
  - Megnézi, hogy a kért erőforrás létezik-e
  - Ha igen, visszaküldi a tartalmát
  - Ha nem, visszaküld egy 404 hibakódot
- **Példa:** `3-nodejs/staticserver/staticserver_lowlevel.js`

Statikus webszerver node.js-szel

```
import http from 'http';
import fs from 'fs';
import path from 'path';
import mimeTypes from 'mime-types';

// a mappa ahonnan statikus tartalmat szolgálunk
// SOSE a gyökeret tegyünk publikussá
const staticDir = path.join(process.cwd(), 'static');

const indexFile = 'index.html';

const server = http.createServer((request, response) => {
  request.on('data', () => {});
  request.on('error', (err) => console.error(err));
  request.on('end', () => {
    console.log(`Received request for ${request.url}`);
    let filename = path.join(staticDir, request.url);
```

Statikus webszerver node.js-szel

```
    fs.stat(filename, (err, stats) => {
      // hiba = az állomány nem létezik
      if (err) {
        console.error(`  ${filename} does not exist`);
        response.statusCode = 404;
        response.end();
        return;
      }

      if (stats.isDirectory()) {
        filename = path.join(filename, indexFile);
      }

      // az állomány nevéből megpróbálunk Content-Type-ot deriválni
      const mimeType = mimeTypes.lookup(filename) || 'text/plain';
      response.setHeader('Content-Type', mimeType);
      console.log(`  ${filename} exists, content-type=${mimeType}`);

      const readStream = fs.createReadStream(filename);
      readStream.pipe(response);
    });
  });
});

server.listen(8080, () => { console.log('Server listening...'); });
```

3. rész

npm

- **Node Package Manager** - függőségkezelő rendszer a node.js-hez
- külső függőségek letöltésére ad lehetőséget
- nyílt forráskódú könyvtárakat tölt le
- command-line elérés: `npm -v`
- külső csomagok böngészése: `https://www.npmjs.com/`
- csomagok használtságának összehasonlítása: `https://www.npmtrends.com`
- `npm install <packagename>` vagy `npm i <packagename>`
  - letölti a `packagename` nevű csomagot
  - a lokális mappát tekinti a projekt gyökerének
  - minden függőséget a lokális `node_modules` mappában cache-eli, s csak a lokális projekt használhatja
  - minden node.js alkalmazásbeli `require` automatikusan keresi a függőségeket ebben a mappában

npm projektfüggőségek

- egy-egy projektnek sok függősége lehet, amelyet nem kellene minden fejlesztőnek kézzel telepítenie
- a megoldás egy **projektdeszkriptor** állomány: `package.json`, mely körülírja a projektet
- **mindig állítsuk be a modultípust az alábbi szerint, másképp az ES6 modulok nem működnek**

```
{
  "name": "npmuppercase",                          <-- projekt neve
  "version": "1.0.0",                              <-- projekt verziója
  "description": "WebProg npm example",            <-- projekt leírása
  "type": "module",                                <-- modulok típusa (CommonJS vagy ES6)
  "scripts": {                                     <-- futtatható szkriptek
    "start": "node index.js"
  },
  "author": "Csaba Sulyok",
  "license": "ISC",
  "dependencies": {                                <-- függőségek listája
    "upper-case": "^1.1.3"
  }                                                .
}
```

npm projektfüggőségek

- `npm init`
  - `package.json` inicializálására alkalmas parancs
- `npm install` vagy `npm i` (paraméter nélkül)
  - kiolvassa a függőségek listáját a deszkriptorból, s ezeket telepíti
- `npm i --save <packagename>` – letölti a `packagename` nevű csomagot, s elmenti a deszkriptorba, hogy a projektünk függ tőle
- `npm i --save-dev <packagename>` – fejlesztői függőséget jelez (produkcióban nem szükséges, csak fejlesztés közben, pl. tesztfuttató vagy linter könyvtárak)
- `npm start` vagy `npm run start`
  - futtatja a `package.json=>scripts=>start`-ban körülírt parancsot

npm példa

- **Példa**: `3-nodejs/npmuppercase` – használja az `upper-case` függőséget
- Parancsok, melyekkel a projekt készült:

```
npm init
npm i --save upper-case
```

- `package.json` tartalma:

```
{
  "name": "npmuppercase",
  "version": "1.0.0",
  "description": "WebProg npm example",
  "scripts": {
    "start": "node index.js"
  },
  "author": "Csaba Sulyok",
  "license": "ISC",
  "dependencies": {
    "upper-case": "^1.1.3"
  }
}
```

npm példa

- `index.js` tartalma:

```
import upperCase from 'upper-case';
console.log(upperCase('I am not screaming'))
```

- Parancs, mellyel futtatni lehet:

```
npm i
npm start
```

- a statikus webszerverünk már tartalmazott egy külső függőséget: a `mime-types` segít egy állomány nevéből MIME típust következtetni

npm globális függőségek

```
npm install --global <packagename>
```

- telepíti az adott függőséget globálisan (minden projekt eléri) - vagy frissíti újabb verzióra, ha már létezik
- nem módosítja a lokális projekt `package.json`-jét
- hozzáadhatnak futtatható, mindenhonnan elérhető parancsokat a `PATH`-hez
- **ne** használjuk projektekhez szükséges függőségekhez, mivel:
  - ha más fejlesztő letölti a git tárolónkat, egy `npm i` nem lesz elegendő a beállításhoz
  - nem tudunk több különböző projektünkben használni ugyanazon függőség különböző verzióit
- használjuk inkább segédeszközök telepítésére:
  - pl. `nodemon` - automatikusan újraindítja a node.js alkalmazásunk egy mentés esetén - **jelentősen** felgyorsítja a fejlesztést
  - pl. `eslint` - statikus kódanalízis eszköz, mely jelez szemantikai-konvencionális hibákat a kódunkban (**szükséges a 3-as laborfeladattól arrafele**).
- az `npm` is egy `npm` csomag (*eat your own dog food* elv) - tehát új verziót telepíthetünk az `npm i --global npm` paranccsal

npx

- Egyes függőségek tartalmaznak parancssorból futtatható állományokat - pl. `nodemon`, `eslint`, `ng`, stb.
- Közös név: **CLI** (command-line interface)
- Ha ezeket globális függőségekként telepítjük, elérhetőek lesznek parancssorból (pl. `npm install -g eslint` után `eslint`-et hívhatunk)
- Ha projektszintű függőségként telepítjük, a futtatható állományok a `<gyökér>/node_modules/.bin`-be kerülnek. Ez a könyvtár alapbeállításból nincs `PATH`-en - a benne levő parancsokat nem futtathatjuk akárhonnan.
- Cserébe futtathatóak az `npx` paranccsal, pl.:

```
npm i --save-dev nodemon
npx nodemon index.js
```

- Ha lehet, kerüljük a globális függőségeket és használjunk inkább `npx`-et, mivel megengedi, hogy több projekt ugyanazon eszköz különböző verzióit alkalmazza.
- A `package.json` `scripts` részében definiált parancsok futtatásakor az `npm` felteszi a `.bin` mappát `PATH`-re, így nem szükséges ott is `npx`-et használni.