SZERVER OLDALI JAVASCRIPT

3. hét

Express keretrendszer

ELŐADÁS TÉMÁJA

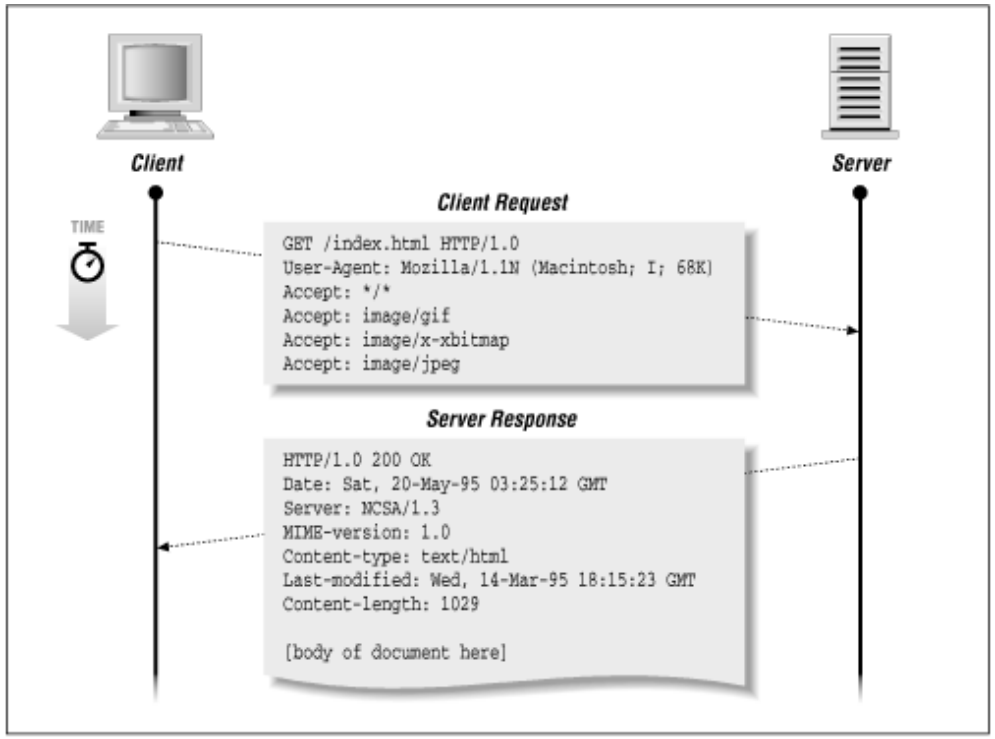
- Express keretrendszer bemutatása
- Gyakran használt modulok, azok kommunikációja

RECEPT EGY WEBALKALMAZÁSHOZ

Kell nekünk:

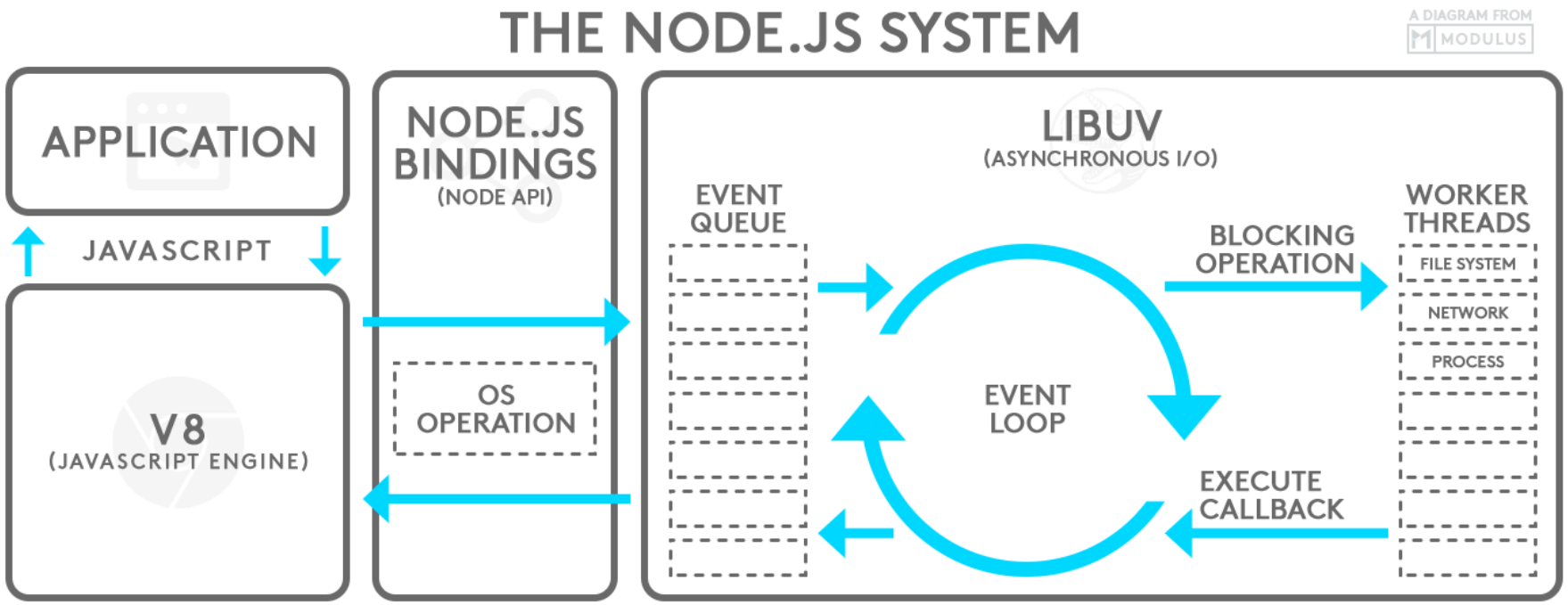
- Webszerver (bejövő http kapcsolat kezelése)
- Routing (ami eldönti, hogy a bejövő kapcsolat milyen logikai részhez kapcsolódik)
- "Controller" (ami a tényleges logikát megvalósítja)
- "Model" (valami ami adatot tárol)
- "View" (template alapú dolgok vannak mostanában, ami html-t generál)

HTTP TLDR


Client
Server
TIME
Client Request
GET /index.html HTTP/1.0
User-Agent: Mozilla/1.1N (Macintosh; I; 68K)
Accept: */*
Accept: image/gif
Accept: image/x-xbitmap
Accept: image/jpeg
Server Response
HTTP/1.0 200 OK
Date: Sat, 20-May-95 03:25:12 GMT
Server: NCSA/1.3
MIME-version: 1.0
Content-type: text/html
Last-modified: Wed, 14-Mar-95 18:15:23 GMT
Content-length: 1029
[body of document here]

NODE.JS

Mit old meg ebből a node.js?



EXPRESS.JS

Egy általános keretrendszer node.js felett, ami (az adattárolás problémáján kívül) megpróbál általános megoldást adni a többi kérdésre.

EXPRESS.JS MINT KERETRENDSZER

Az express app létrehozása rendkívül egyszerű:

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

app.get('/', (req, res, next) => {
  res.send('Hello World!');
});

app.listen(3000, () => {
    console.log('Running on :3000');
});
```

ROUTING

Azt mondja meg, hogy egy lekérdezés milyen szerver oldali logika fusson le egy adott url meghívására.

Pl:

```
/               = Főoldal
/books/new     = Új task könyv oldal
/books         = Könyv lista oldal
/book/12     = 12-es számú könyv oldala
```

ROUTING

HTTP metódus alapján is szétválaszthatóak az egyes routok, de általánosan is fel lehet rájuk iratkozni

```
app.use('/books', (req, res, next) => {
    console.log('hello /books');
});

app.get('/', (req, res, next) => {
    console.log('hello / with GET');
});

app.post('/', (req, res, next) => {
    console.log('hello / with POST');
});
```

PARAMÉTEREK HASZNÁLATA

A routoknál megadható paraméter is, erre külön paraméter függvény is illeszthető:

```
app.param('bookid', (req, res, next, id) => {
  console.log('Bookid: ' + id);
  return next(); // can throw here as well
});

app.get('/book/:bookid', (req, res, next) => {
  console.log('Valid book id!!');
  return res.end();
});
```

ROUTING EXTRÁK

Egy routera egy lépésben több middlewaret is fel lehet iratkoztatni.

```
app.get('/book/:bookid', (req, res, next) => {
  console.log('Itt is csinalunk valamit');
  return next();
}, (req, res, next) => {
  console.log('Meg itt is');
  return res.end();
});
```

ROUTING EXTRÁK

Több routera is feliratkozhatunk egyszerre, regexpet is megadhatunk, de ezek nem ajánlottak. A /* elhagyható az első példában, a route nélküli app.use meghívás esetében az összes lekérdezésre feliratkozunk.

```
app.use('/*', (req, res, next) => {
  console.log('Mindenhol');
  return next();
});

app.use(['/book/:valami','/book/*'], (req, res, next) => {
  console.log('Tobbhelyen');
  return next();
});

app.use(/^\/commits\/(\w+)(?:\.\.(\w+))?$/, (req, res, next) => {
  console.log('Lol regexp');
  return next();
});
```

MIDDLEWARE

Eddig csak körbejártuk, most nézzük meg mi ez:

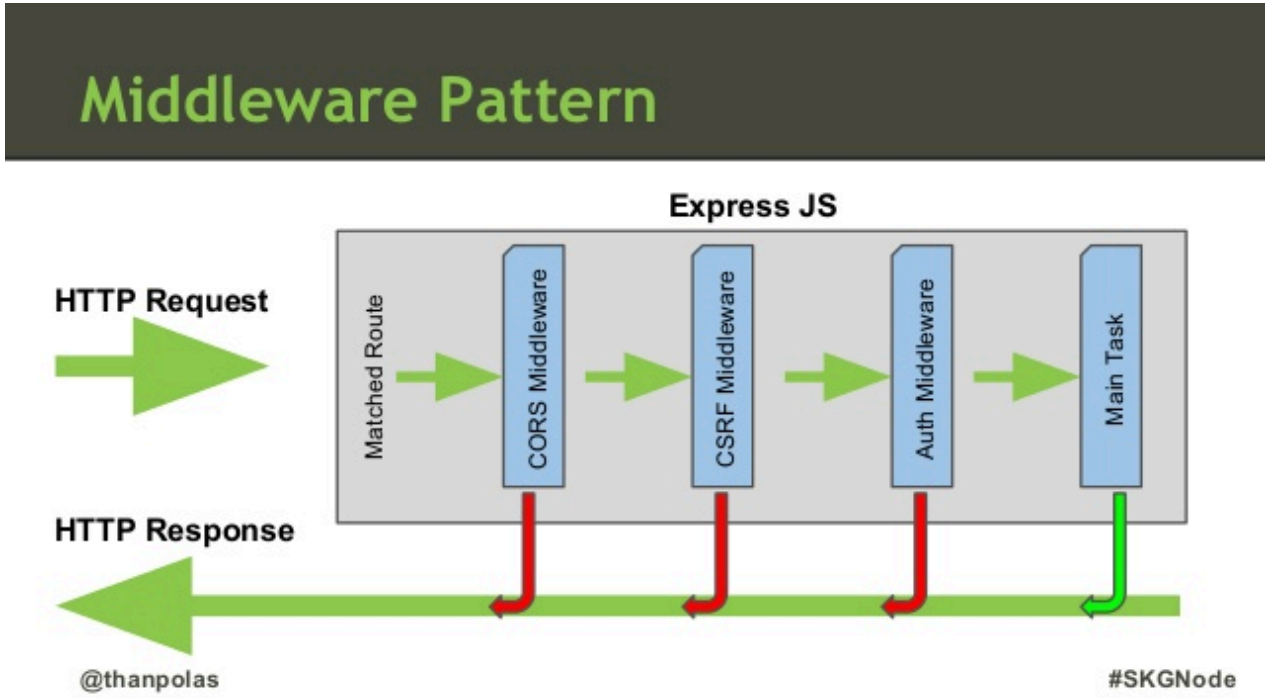
```
function mw(req, res, next) {
  // ...
  return next();
}
```

A middleware egy egyszerű függvény, ami hozzáfér a request (req) és response (res) objektumokhoz, illetve hatással lehet az őt meghívó végrehajtási láncra is.

Miven minden async, a végrejatási lánc next() hatására lép tovább (a return nem működik, csak egyszerűsítő hatása lesz)

VÉGREHAJTÁSI LÁNC

Egy adott routera feliratkozott middlewarek egy láncot alkotnak:



Ennek a láncnak az első elemét hívja meg az express.

VÉGREHAJTÁSI LÁNC

Ez gyakorlatilag a "Controller" rétegünk.

```
app.get('/book/*', (req, res, next) => {
    console.log('Valami tortenik');
    return next();
});

app.get('/book/:userid', (req, res, next) => {
    console.log('Meg itt is');
    return res.end();
});
```

Akkor miért nem controller? Miért két függvény és nem egy?

MIDDLEWARE ÚJRAFELHASZNÁLÁS

```
const authMW = (req, res, next) => {
    //ellenorizzuk, hogy be van-e jelentkezve a user
    // ha nincs, atiranyitjuk a bejelentkezo oldalra
    // ha be van, akkor nextet hivunk
    return next();
}
app.get('/user/:userid',
        authMW,
        (req, res, next) => {
            console.log('Mar biztosan be van lepve a user.');
        })
app.get('/books',
        authMW,
        (req, res, next) => {
            console.log('Mar biztosan be van lepve a user.');
        })
```

Ezért kell mindent értelmes méretű middlewarekre darabolni.

MIDDLEWARE

Mi van, ha nem hívok nextet?

```
app.get('/user/*', (req, res, next) => {
    console.log('Elso blokk');
}, (req, res, next) => {
    console.log('Masodik blokk');
    if (alma === korte){
        return next();
    }
    console.log('Harmadik');
});
```

AYNC - AWAIT ?

```
const promiseWrapMW = (otherMw) => {
    return (...args) => {
        const wrappedMw = otherMw(...args);
        return (req, res, next) =>
            wrappedMw(req, res, next)
            .catch(error => next(error));
    };
};

async function myMW(req,res,next) {
    await iLovePromise();
    return next();
}

app.get('/fancy',promiseWrapMW(myMW));
```

REQUEST OBJECT

A request (req) object tartalmazza a böngészőtől szerver irányba jövő kérés főbb paramétereit. Persze ezek mind feldolgozott formában, függvényeken keresztül érhetőek el.

Most csak a fontosabbakról lesz szó.

REQUEST.BODY

POST metódus esetében itt van találhatóak a postolt paraméterek

```
<form method="POST">
    <input type="text" name="username"/>
</form>
```

```
app.post('/login', (req, res, next) => {
    if (typeof req.body.username !== 'undefined'){
        console.log('Username:' + req.body.username);
    }
    return next();
})
```

REQUEST.BODY

Viszont a POST-os parsoláshoz már kell egy külső modul: **body-parser**. 4.0 vs 4.16.0

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');

app.use(bodyParser.urlencoded());
app.use(bodyParser.json());

app.post('/', (req, res, next) => {
    console.log(req.body.username);
});

app.listen(3000, () => {});
```

A bodyParser.json() egy middleware függvényt ad vissza, amit fel tudunk iratkoztatni az összes routera.

REQUEST.PARAMS

Emlékszünk még a paraméterekre? Ezek a request.params alatt laknak.

```
app.get('/book/:bookid', (req, res, next) => {
    console.log(`Ez a bookid: ${req.params.bookid}`);
    return res.end();
});
```

REQUEST.QUERY

GET paraméterek lekérdezhetőek ezen keresztül.

/books?search=alma&category[type]=sci-fi lekérdezéssel

```
app.get('/books', (req, res, next) => {
    console.log(`Search: ${req.query.search}`);
    console.log(`Category type: ${req.query.category.type}`);
    return res.end();
});
```

Itt is mindig érdemes ellenőrizni, hogy létezik-e a paraméter (undefined).

REQUEST.PARAM()

Az egyszerűség kedvéér a params, body és query paraméterek (ez a prioritás is) elérhetőek a param() hívással is.

```
app.get('/book/:bookid', (req, res, next) => {
    console.log(`Ez a bookid: ${req.param('userid')}`);
    return res.end();
});
```

Ezt tényleg csak egyszerű esetekben használjuk, ahogy a manual is mondja: Direct access to req.body, req.params, and req.query should be favoured for clarity - unless you truly accept input from each object.

RESPONSE OBJECT

A response objektum segítségével adunk vissza adatot / állapotot a böngészőnek. Itt is csak a fontosabb függvényeket fogom bemutatni.

RESPONSE.STATUS()

A status hívással tudtok http kódokkal jelezni a kliensnek.
Ha nem hívtok statust-t, 200-as lesz a státusz (minden ok).

```
app.get('/book/:bookid', (req, res, next) => {
    return res.status(404).end();
});
```

RESPONSE.REDIRECT()

A böngészőnek küldhetünk egy jelzést, hogy az oldal "máshol található". Ilyenkor a böngészőbe az url ténylegesen megváltozik, és mi kapunk egy 2. http hívást az új urlre.

```
app.get('/book/:bookid', (req, res, next) => {
    return res.redirect('/login');
});
```

RESPONSE.JSON()

Egy egyszerű hívással küldhetünk json formátumú adatot kliens oldalra. Instant REST api

```
app.get('/book/:bookid', (req, res, next) => {
    return res.json({
        key: 'value'
    });
});
```

RESPONSE.SEND()

Szöveget (akár html-t) küldhetünk kliens oldalra.

```
app.get('/book/:bookid', (req, res, next) => {
    return res.send('Lorem ipsum');
});
```

Ha content típust is kell állítani: *res.set('Content-Type', 'text/html');*

RESPONSE.END()

Az end metódus meghívással le lehet zárni a http választ (ez ugye elválik a végrehajtási lánctól), így a böngésző nem fog több adatot várni. Két opcionális paramétere van: end([data] [, encoding]). Ha tényleges adat utazik, inkább használjuk a **json** és **send** metódusokat.

```
app.get('/book/:bookid', (req, res, next) => {
    return res.end('Üzenet - őűáéüóöí','utf8');
});
```

RESPONSE.RENDER()

Templating engine segítségével és hozzáadott paraméterekkel hozhatunk létre webes tartalmat. Templatingről később lesz szó.

```
app.get('/book/:bookid', (req, res, next) => {
    res.render('bookpage',{
        title: 'The Hitchhiker\'s Guide to the Galaxy',
        ratings: [42, 0, 42, 0, 42]
    });
});
```

EJS TEMPLATING

Az EJS egy HTML alapú templating nyelv, pár egyszerű művelettel. Ez lényegében egy modul: **ejs**.

```
app.set('view engine', 'ejs');
```

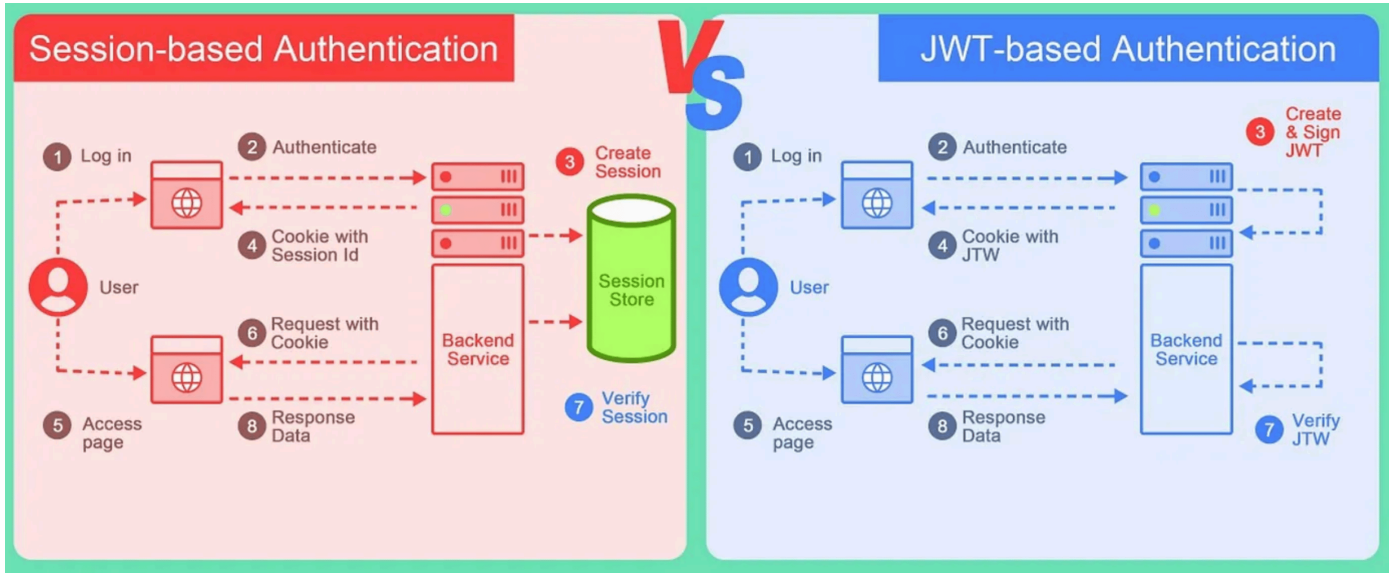
STATIC MODUL

Gyakorlaton szó volt róla, lényegében egy könyvtár **kiajánlása** adott routeon statikus tartalomként. Express része, nem kell külön package hozzá.

```
app.use(express.static('static'));
// vagy
app.use('/static', express.static('static'));
```

Utóbbi ajánlott, mert nem keveredik össze a statikus és dinamikus tartalom.

SESSION VS JWT



SESSION MODUL

A session egy böngésző session szintű szerver oldali oldalbetöltések közötti perzisztens tároló. A modul neve: **express-session**

```
const session = require('express-session');
app.use(session({
  secret: 'keyboard cat',
  resave: false,
  saveUninitialized: true
}));
```

Ezzel egy **req.session** változó válik elérhetővé, ami session szinten megtartja a tartalmát. Az alap beállítás szerint a session memóriában van, tehát ha a node.js újraindul, akkor elveszik!!

HELMET

Biztonsági modul, ami segít az express alkalmazások biztonságosabbá tételéhez. A modul neve: **helmet**

A használata nagyon egyszerű:

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

const app = express();

app.use(helmet());
```

EXPRESS ALKALMAZÁSOK SZERVEZÉSE

Gyakorlaton... :)