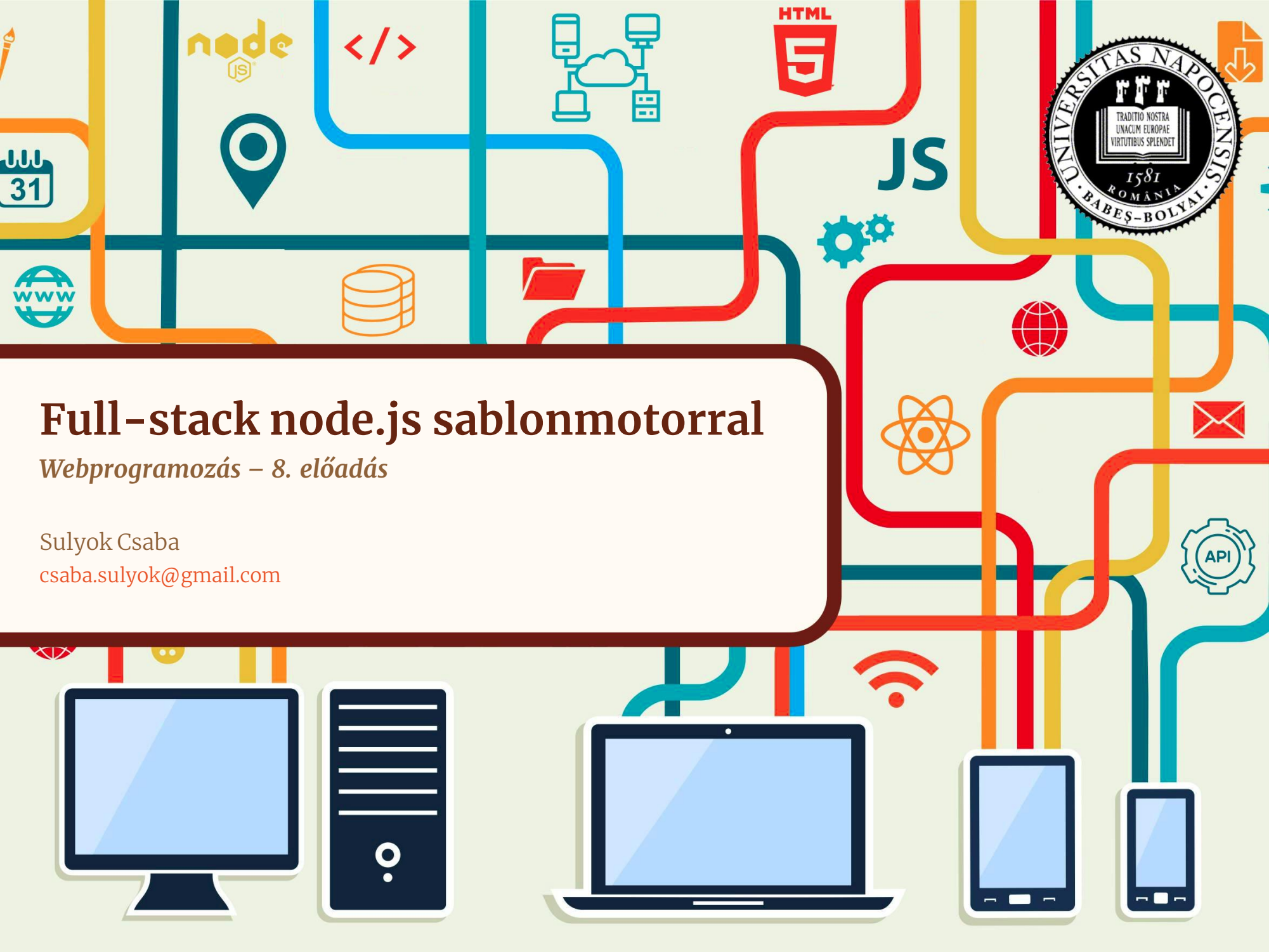
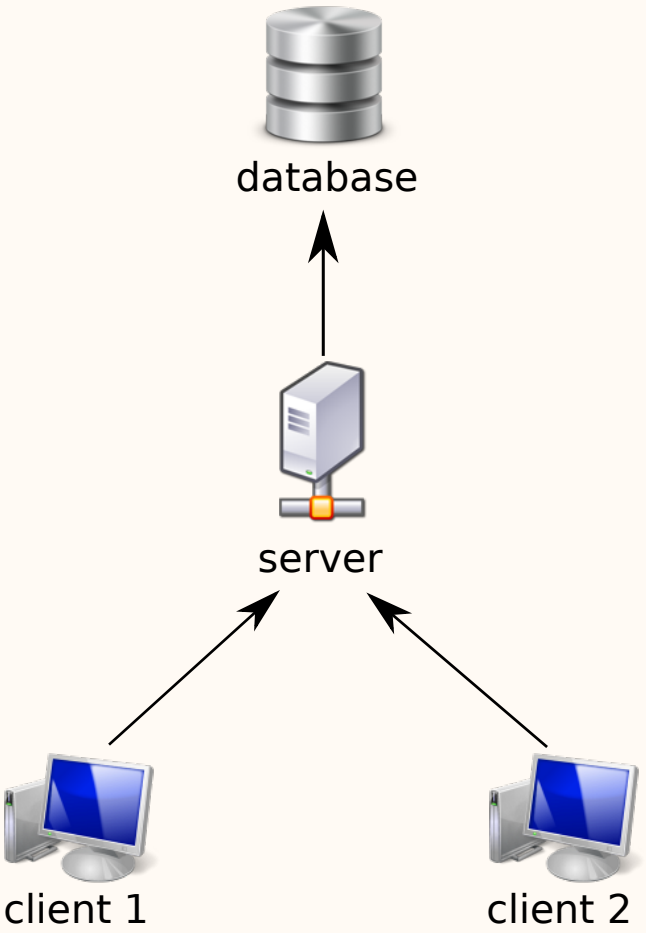


Full-stack node.js sablonmotorral

Webprogramozás – 8. előadás

Sulyok Csaba
csaba.sulyok@gmail.com

Full-stack webfejlesztés



- hibrid megközelítés: mind a frontend, mind a backend ugyanazon kezekben van
- kliens, (alkalmazás)szerver és adatbázisszerver
- eddig tárgyalt szerver hiányosságai:
  - adatbáziskapcsolat
  - komplex dinamikus weboldalak felépítése

1. rész

Adatbázis hozzáférés

Adatbázis hozzáférés

- Legtöbb adatbázis kezelő rendszerhez létezik dedikált node.js modul, s ezáltal függőség.
- Pár elterjedtebb példa
- Kiválasztás szempontjai:
  - relációs adatbázis (`mysql`, `mysql2`, `pg` a PostgreSQL-hez, stb.) vs. dokumentumalapú (`mongodb`)
  - klasszikus (fentiek) vs. in-memory (`sqlite3`, `lokijs`, `redis`)
- ORM/ODM keretrendszerek automatikus leképezést végeznek adatbázis bejegyzések és lokális objektumok között:
  - `mongoose` - ODM a MongoDB-hez
  - `typeorm` - ORM a legtöbb SQL-alapú relációs adatbázishoz

A `mysql2` modul

- Kétféleképpen használhatjuk:
  - Egyedülálló kapcsolatok létrehozásával
  - **Connection pooling** használatával - optimálisabb legtöbb esetben
- Lépések:
  - Kapcsolat vagy pool létrehozása
  - Lekérdezések definiálása
  - `callback` függvények definiálása és illesztése

A `mysql2` modul

Példa: `4-templatedb/mysql` – minden beérkezett HTTP kérésnél készít egy bejegyzést az adatbázisba a kérés információival

```
import express from 'express';
import mysql from 'mysql2';

// létesít egy kapcsolatot az adatbázissal
// a megadott DB-nek/felhasználónak léteznie kell
// létre lehet hozni a mellékelt setup.sql szkript segítségével
const connection = mysql.createConnection({
  database: 'webprog',
  host: 'localhost',
  port: 3306,
  user: 'webprog',
  password: 'VgJUjBd8',
});
```

A `mysql2` modul

```
connection.connect((err) => {
  if (err) {
    console.error(`Connection error: ${err.message}`);
    process.exit(1);
  }

  // létrehozzunk a táblázatot, ha még nem létezik
  connection.query(`CREATE TABLE IF NOT EXISTS requests (
        method varchar(20),
        url varchar(50),
        date datetime);`, (createErr) => {
    if (createErr) {
      console.error(`Create table error: ${err.message}`);
      process.exit(1);
    } else {
      console.log('Table created successfully');
    }
  });
  console.log(`Connected: ${connection.threadId}`);
});
```

A `mysql2` modul

```
const app = express();

app.use((req, res) => {
  const { method, url } = req;
  const date = new Date();

  // felépítjük a végrehajtandó SQL lekérdezést
  const query = `INSERT INTO requests VALUES ("${method}", "${url}", "${date}");`;
  console.log(`Executing query ${query}`);

  // végrehatjuk a lekérdezést
  connection.query(query, (err) => {
    if (err) {
      res.status(500).send(`Insertion unsuccessful: ${err.message}`);
    } else {
      res.send('Insertion successful');
    }
  });
});

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

Connection pooling

- **pooling**
  - általános fogalom/tervezési minta
  - egy erőforrásból egy meghatározott számú példányt előre előkészítünk s készenlétbe helyezünk
  - szükség esetén egy felhasználó kér egy példányt, s mikor végez a használatával, visszatéríti a poolba, hogy más is használhassa
  - üres pool esetén a felhasználó hibát kap vagy várakozó állapotba kerül
  - *előnyök:*
    - inicializáló folyamatok nem váratják meg a hívó felet
    - mivel az erőforrások száma előre meghatározott, nem lép fel túlterheltség veszélye
  - *használati területek*:
    - egy osztály példányai OO környezetben (object pooling)
    - szálak (thread pooling)
    - memóriablokkok (memory pooling)
    - **kapcsolatok egy külső forráshoz (connection pooling)**
- bármely adatbázis elérése esetén hasznos connection poolingot alkalmazni, főként biztonsági és teljesítményi okokból
- minden nagyobb adatbázishoz kapcsolódó könyvtár támogatja

Connection pooling a `mysql2` könyvtárral

- hivatalos dokumentáció
- a `mysql.createPool` metódus segítségével megadott számú `connection` létrejön memóriában
- a visszatérített poolból kérhetünk kapcsolatokat (`pool.getConnection`)
- **vigyázat:** az elkért kapcsolatokat muszáj visszatérítsük a poolba minden esetben (hibák esetén is): `connection.release()`
- végrehajthatunk egy DB parancsot egyenesen a pool példányon a `query` metódussal – ebben az esetben nem kell visszahelyeznünk kapcsolatot a poolba, ez automatikusan megtörténik
- minden felsorolt metódus *callback* paraméterekkel használatos

Pooling példa

Példa: `4-templatedb/mysql_connectionpool`

```
// Létrehozunk egy connection poolt
const pool = mysql.createPool({
  connectionLimit: 10,
  database: 'webprog',
  host: 'localhost',
  port: 3306,
  user: 'webprog',
  password: 'VgJUjBd8',
});

// létrehozzunk a táblázatot, ha még nem létezik
// a pool.query kér egy kapcsolatot a poolból
pool.query(`CREATE TABLE IF NOT EXISTS requests (
    method varchar(20), url varchar(50), date datetime);`, (err) => {
  if (err) {
    console.error(`Create table error: ${err.message}`);
    process.exit(1);
  } else {
    console.log('Table created successfully');
  }
});
```

Pooling példa

```
const app = express();

app.use((req, res) => {
  // kérünk egy kapcsolatot a pooltól
  pool.getConnection((err, connection) => {
    if (err) {
      connection.release(); // Fontos visszaadni itt
      res.status(500).send(`Error in DB connection: ${err.message}`);
      return;
    }

    // ... query felépítése ...

    connection.query(query, (queryErr) => {
      if (queryErr) {
        res.status(500).send(`Insertion unsuccessful: ${err.message}`);
      } else {
        res.send('Insertion successful');
      }
      // visszahelyezi a kapcsolatobjektumot a készletbe, így az ismét használható mások által is
      connection.release();
    });
  });
});
```

SQL injection

- **Forrása:** Ha SQL lekérdezések felépítésénél olyan adatokat használunk, melyek nem voltak megfelelően levédve (escape-elve).
- Segítségével titkos adatot bányászhatunk ki adatbázisokból.
- Még mindig sok gyenge weboldal támadható a segítségével.
- Mivel query-k futtatásakor egyik nyelvből (*JavaScript*) hívunk egy másikat (*SQL*) egyszerű karakterláncok segítségével, nem tudjuk egyértelműen kezelni a célpont nyelvben használatos kontrollkaraktereket (idézőjelek, kommentjelzések, százalékjel, stb.)
- **Védekezés**: Minden ilyen tranzició esetén a forrásnyelvben le kell kezelni azokat a karaktereket, melyek a célpontnyelvben begolyásolják a futást, egy olyan módon ahogy a célpontnyelv megengedi (pl. levédés a \ karakter segítségével).
- **Nem elégséges kliensen elvégezni egy validációt** (pl. form validálással).
- Minden kliens által megadott adat veszélyes (*guilty until proven innocent*).

SQL injection

- Veszélyes kód:

```
queryString = `select * from myTable where myColumn='${queryValue}'`;
pool.query(queryString, (err, results) => { ... });
```

- Ha a `queryValue` értékét egy klienstől kapjuk, bármilyen karaktereket használhat, beleértve az idézőjelet, mely bezárná a fenti keresőstringet, vagy kommentet jelző karaktereket.
- Ezáltal egy valid bemenet a `queryValue`-nak: `something'--`
- A fenti behelyettesítés eredménye pedig egy valid SQL parancs kommenttel a végén:

```
select * from myTable where myColumn='something'--'
```

- Így egyszerű `queryValue` értékekkel adatokat törölhetünk (`something'; DROP ALL DATABASES;--`) vagy bányászhatunk (`UNION` parancs segítségével).

SQL injection példa

Példa: `7-security/sqlinjection`

```
// helytelen, támadható!!!
query = `select * from blogPosts where title like '%${titleQuery}%'`;

// helyes!!!
query = 'select * from blogPosts where title like ?';
options = [titleQuery];

// ugyancsak helyes
query = `select * from blogPosts where title like '%${db.escape(titleQuery)}%'`

// ...
pool.query(query, options, callback);
```

- node.js-en belüli *védekezések*:
  - bemenetek levédése az `escape` metódussal - elérhető a connection és a pool objektumokon
  - előkészített hívások (**prepared statements**) - a `mysql2` npm csomagban `?`-lel helyettesítünk minden érzékeny bemenetet, s ezek tömbjét átadjuk a query-hívásnak paraméterként (háttérben a fenti `escape` metódust használja, de a szintaxis elegánsabb)
- minden jobb nyelvnek/könyvtárnak megvan a *prepared statements* megfelelője

SQL injection példa

A helytelen használat esetén használhatjuk a következő query stringeket:

1. `dummy%'-- comment`
   Teszteli hogy lehet-e SQL injectiont végezni.
2. `dummy%' union select 1,2,3,4-- comment`
   Megkeressük a használt tábla oszlopainak számát (amíg meg felel meg, hibákat kapunk), s megnézzük melyik információ van kivetítve a képernyőre (pl. 2).
3. `dummy%' union select 1,database(),3,4 from dual-- comment`
   Megkeressük a használt adatbázis nevét.
4. `dummy%' union select 1,table_name,3,4 from information_schema.tables where table_schema='webprog'-- comment`
   Lekérjük az adatbázis összes tábláját s választunk egyet, melyet ki akarunk bányászni (pl. `users`).

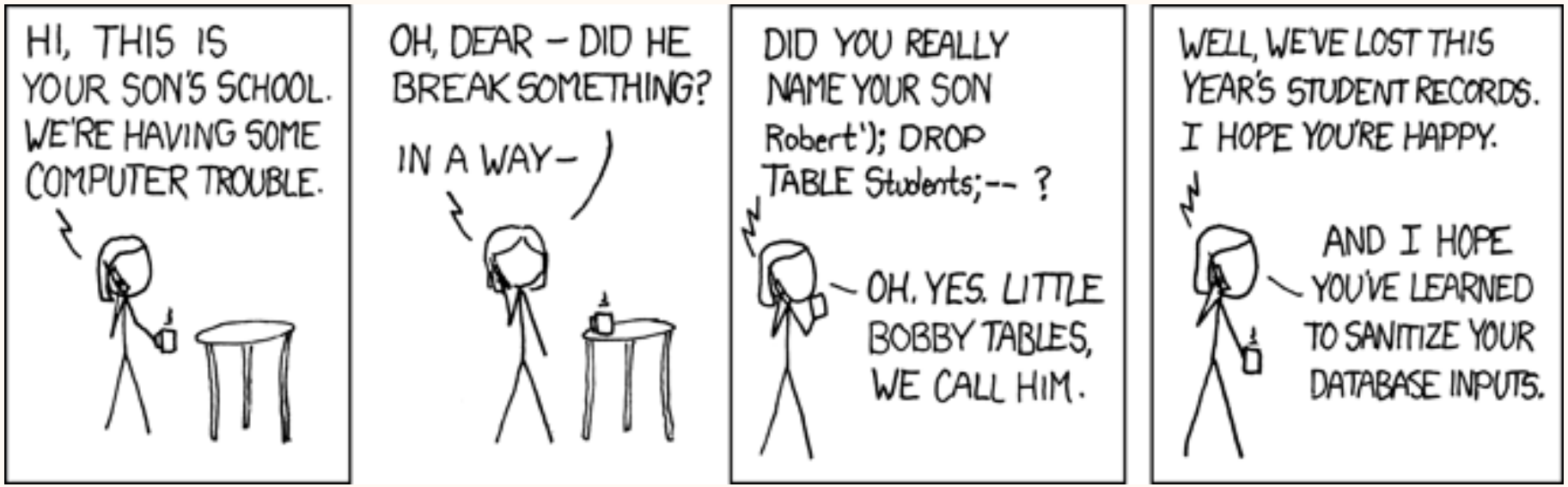
SQL injection példa

5. `dummy%' union select 1,column_name,3,4 from information_schema.columns where table_name='users' and table_schema='webprog'-- comment`
   Lekérjük a tábla összes oszlopnevét.
6. `dummy%' union select 1,username,password,4 from users-- comment`
   Lekérjük a tábla tartalmát.

Élő SQL injection futtatás:

`https://www.youtube.com/watch?v=ciNHn38EyRc`

SQL injection vicc



Forrás: xkcd

2. rész

Template rendering (sablonalapú megjelenítés)

Dinamikus tartalom generálása

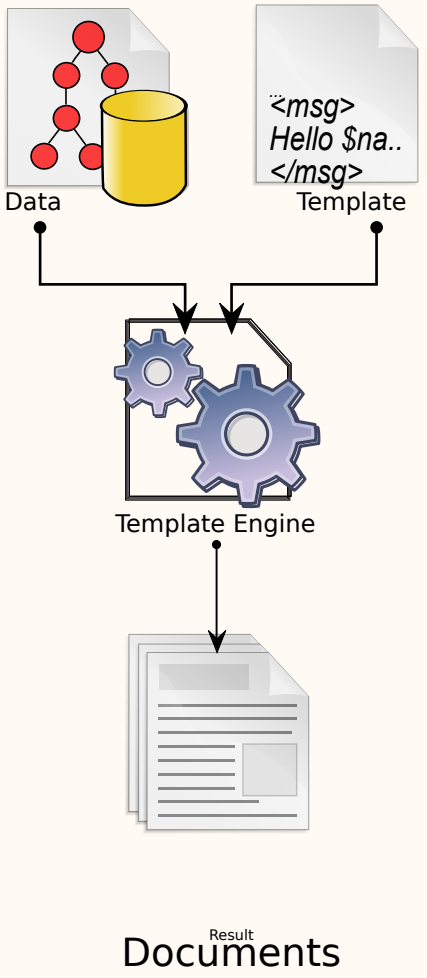
- Eddig ismert módszerek HTML küldésére válaszban:
  - statikus HTML állományok küldése az express `static` middleware-jének használatával
  - a `response` objektum `write`, `end`, `send` metódusok
- Példa:

```
app.get('/', (req, resp) => {
  response.send(`
    <ul>
      <li>This is a static string</li>
      <li>This is a variable: ${data}
    </ul>
  `);
});
```

- Noha az ES6 sok segítséget nyújt (sablonliterálok és többsoros karakterláncok), hátrányok maradnak:
  - Megakadályozza a kérésfeldolgozás **logikájának** és **megjelenésének** elkülönítését (ellehetetleníti az MVC elv betartását)
  - Számos hibalehetőség - a fenti HTML nincs validálva és a szintaxisa sincs megjelölve
  - Megnehezíti a komplexebb műveleteket, pl. feltételek, bejárások

Dinamikus tartalom

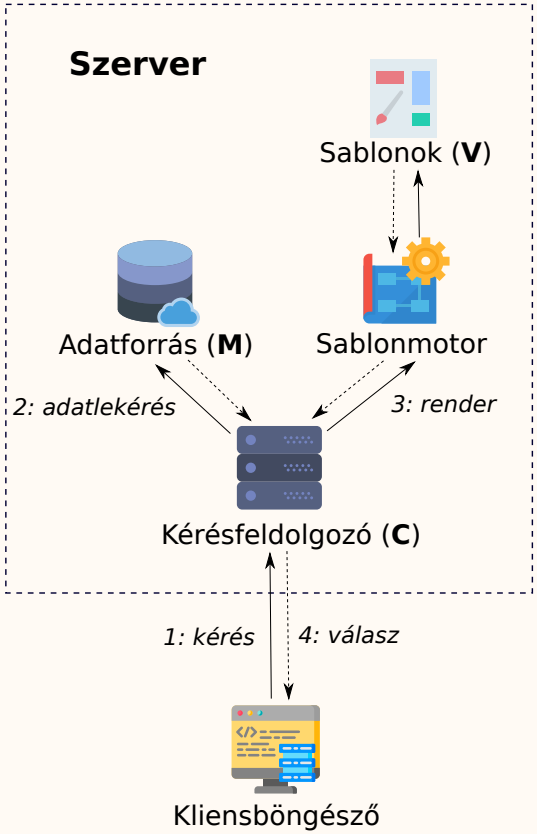
- *Megoldás:* **sablonmotrok/sablonfeldolgozó motrok** (*template rendering engine*) használata
- Lehetővé teszi egy HTML (vagy bármely más) sablon megadását, amelyet dinamikusan egészíthetünk ki tartalommal
- Segít a dinamikus tartalom szintaktikai helyességének garanciájában
- Külső állományok használatával megmarad az MVC elv (a sablonok képezik a **V**iewt)



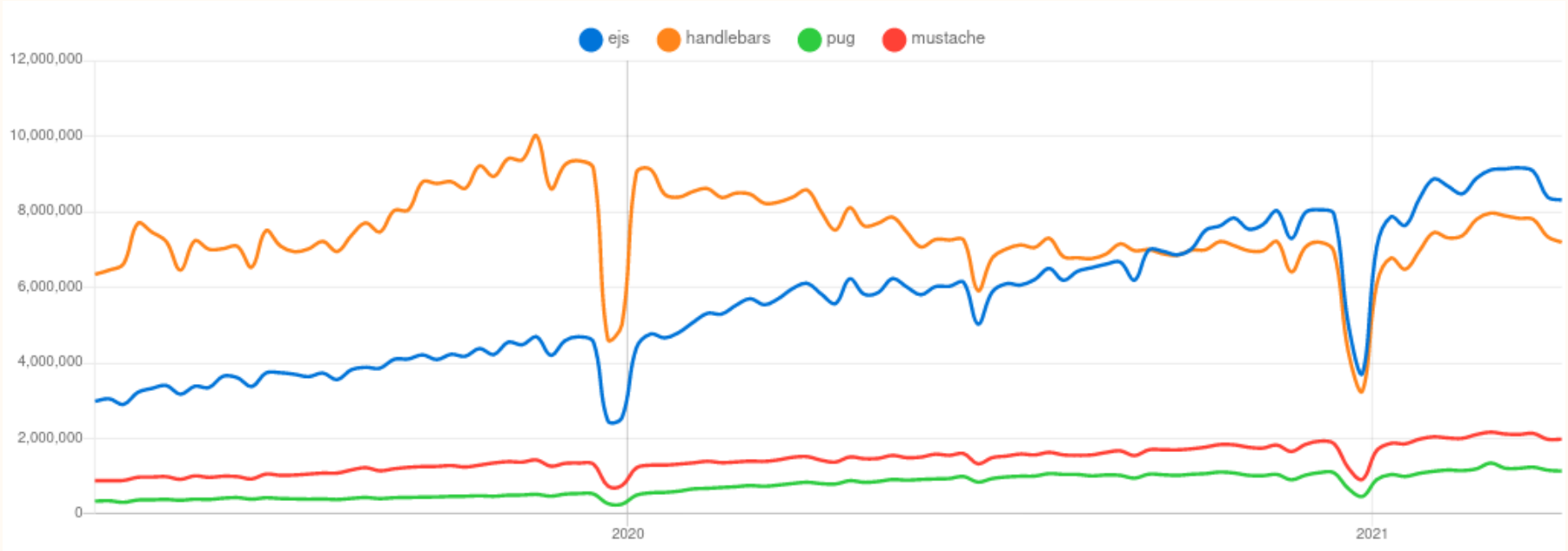
Thin vs. Rich kliensek

Kliens-szerver alkalmazások esetén a klienst megkülönböztetjük aszerint, hogy az MVC logikából mennyit vállal át:

- **Thin client**
  - A böngésző csupán minimális feladatot lát el (megjelenítés)
  - Az MVC komponensek (modell, nézet és kontroller logika) a szerveren léteznek/futnak
  - Szerveroldali **sablonmotrot** alkalmaznak
- Példák (több részlet itt):
  - `ejs`
  - `handlebars`
  - `pug` (korábban `jade`)



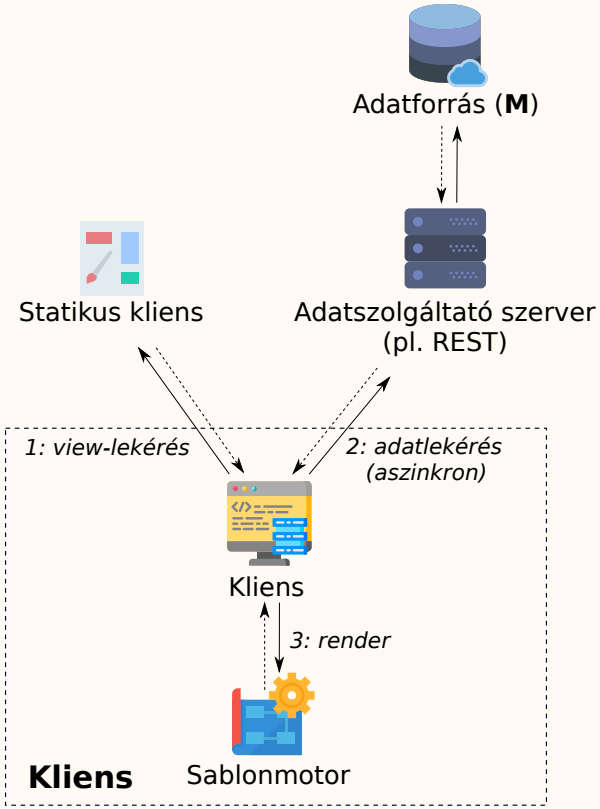
Thin kliensek: összehasonlítás



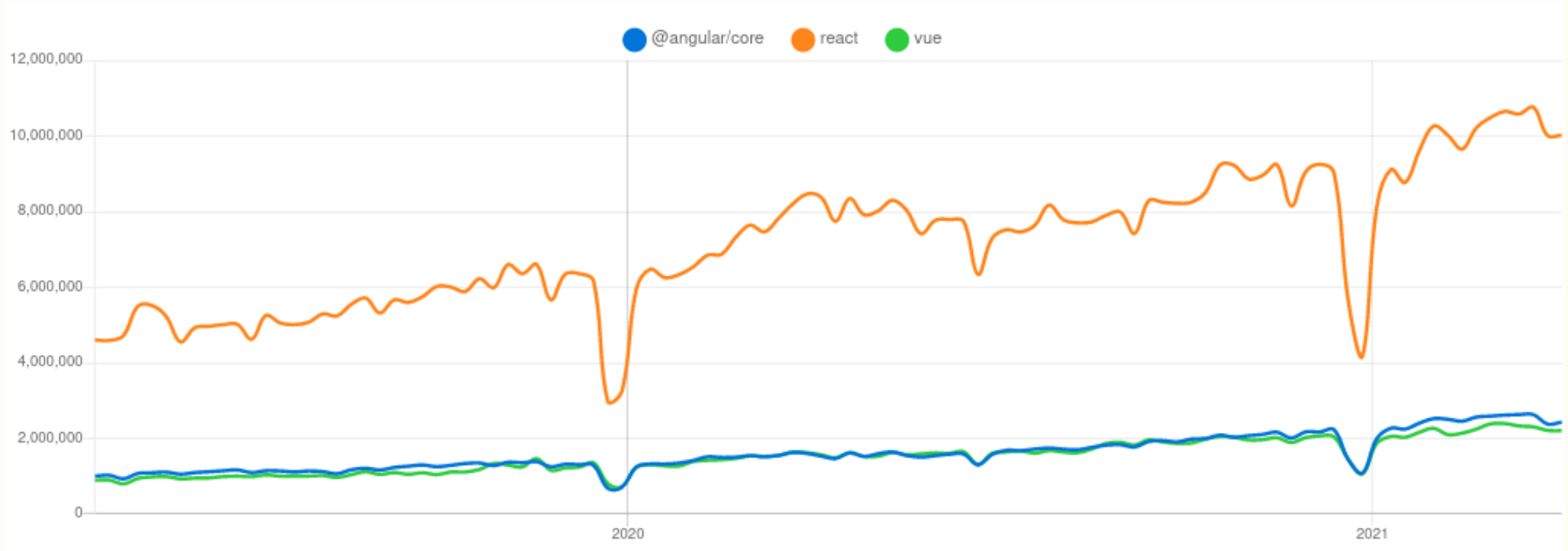
Thin vs. Rich kliensek

Kliens-szerver alkalmazások esetén a klienst megkülönböztetjük aszerint, hogy az MVC logikából mennyit vállal át:

- **Rich client**
  - MVC komponensek részben a kliens oldalon lesznek végrehajtva
  - Szorosan kapcsolódik az aszinkron kéréseket lehetővé tevő technológiákhoz
  - Számos kliens oldali keretrendszer támogatja
- Példák (összehasonlítás):
  - `React`
  - `Angular`
  - `Vue.js`



Rich kliensek: összehasonlítás



<%= EJS %>

- *Embedded JavaScript templating.*
- `https://ejs.co/`
- Sablonokat kompilálhatunk le, majd dinamikus tartalmat generálhatunk adatok későbbi megadásával.
- Hasonló a JavaScript sablonliterálokhoz, a következő előnyökkel:
  - a dinamikus jelöléseknél részleges JavaScript tartalom is megadható, így `if`-ek, `for`-ok is használhatóak sablonrészekre
  - külön állományokba választhatjuk a sablonokat a használt modelltől
  - egy sablont több különböző modellel használhatunk, az EJS elcache-eli a sablonokat
  - újrahasznosítható sablonrészek (pl. a ha navigációs menü/lábléc közös minden sablonnál)
  - integrálható `express`-szel
  - command-line eszközt is nyújt, így node.js projekten kívül konzolról is használható

EJS használat

- használat:

```
import ejs from 'ejs';
const result = ejs.render('templatestringhere', model, options);
```

- a model egy objektum, amelynek tartalma direkten használható a sablon stringben
- az `options` objektum opcionális, extra beállításokat tartalmaz (pl. `cache`, `debug`, `rmWhitespace`)
- a sablon tartalmát külső file-ból is olvashatjuk, így aszinkron a render - használhatjuk az `ejs` kiterjesztést

```
import ejs from 'ejs';
ejs.render('filename', model, options, (err, result) => {
  // result - a render eredménye
});
```

- ha gyakran újrahasználunk egy sablont, érdemes előre "kompilálni", ezzel az EJS feldolgozza a benne levő dinamikus tokeneket, mielőtt bármit renderelne:

```
import ejs from 'ejs';
const template = ejs.compile('templatestringhere', options);
const result = template(model);
```

EJS sablonok szintaxisa

- kifejezések eredménye beszúrható a `<%= expression %>` szintaxissal (az alábbinál a `name` kulcs benne kell legyen a modellben)

```
Hello <%= name %>'
```

- szkriptlet – általános JavaScript blokkok: `<% JS code %>`

```
<% if (error) { %>
  <div>Error is set!</div>
<% } %>
<% people.forEach((person) => { %>
  <div>Person name: <%= person.name %>
<% }) %>
```

- kommentek

```
Hello <%= name %>'. <%# Comment here %>
```

- egy EJS behúzása egy másikba:

```
<%- include("otherfile.ejs") %>
<%- include("otherfile2.ejs", { extraKey: "extraValue" }) %>
```

EJS példa

```
// előre kompiláljuk sablonunkat
const template = ejs.compile(`
  <p>Hello, my name is <%= name %>.</p>
  <p>I am from <%= hometown %>.</p>
  <p>I have <%= kids.length %> kids:</p>
  <ul><% kids.forEach((kid) => { if (kid.age) { %>
    <li><%= kid.name %> is <%= kid.age %></li><% } else { %>
    <li>I don't remember how old <%= kid.name %> is.</li><% }}) %>
  </ul>
`);

// megadunk egy modellt (dinamikus tartalom)
const model = {
  name: 'Alan',
  hometown: 'Somewhere, TX',
  kids: [
    { name: 'Jimmy', age: '12' },
    { name: 'Sally', age: '4' },
    { name: 'Johnny' },
  ],
};

// a sablon renderelése
const view = template(model);
```

Kimenet:

```
<p>Hello, my name is Alan.</p>
<p>I am from Somewhere, TX.</p>
<p>I have 3 kids:</p>
<ul>
  <li>Jimmy is 12</li>
  <li>Sally is 4</li>
  <li>I don't remember how old Johnny is.</li>
</ul>
```

Express és sablonmotrok

- Az express natívan támogatja a sablonmotrok bekötését az alkalmazásokba
- Szükséges beállítások:
  - melyik motrot alkalmazzuk:

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

  - hol tároljuk a sablonállományainkat (konvencionálisan a projekt gyökerében elhelyezkedő `views` mappában) - ezeket lekompilálja az alkalmazás automatikusan:

```
app.set('views', path.join(process.cwd(), 'views'));
```

- A motor kell nyújtson express-ben használható middleware-t. Az EJS nyújtja ezt natívan, míg egyeseknél (pl. a handlebars) szükség van egy összekötő csomagra.

Express és sablonmotrok

- A beállítások nyomán kérések kiszolgálásánál alkalmazható a `render` metódus:

```
response.render('viewname', [model], [callback]); // a views/viewname.ejs a sablon
```

- callback függvény megadása opcionális - ha nem adjuk meg, a HTTP válasz automatikusan kimegy
- a `model` válik a sablonban a `this` objektummá - hivatkozhatunk a *mezőire*

Express + EJS példa

Példa: `4-templatedb/ejs_express`

```
const app = express();

// beállítjuk az ejs engine-t
app.set('view engine', 'ejs');
app.set('views',
  path.join(process.cwd(), 'views'));

app.use((request, response) => {
  // beállítjuk a view modelljét
  const model = {
    method: request.method,
    url: request.url,
    headers: request.headers,
  };

  // views/requestinfo.ejs sablonnal
  // felépítjük a választ a kliensnek
  response.render('requestinfo', model);
});

app.listen(8080, () => { ... });
```

`views/requestinfo.ejs`:

```
...
<h1>Information on your request</h1>
<p>Method: <b><%= method %></b></p>
<p>URL: <code><%= url %></code></p>
<p>Request headers:</p>
<table>
  <tr><th>Key</th><th>Value</th></tr>
  <% Object.entries(headers).forEach(([key, value]) => { %>
    <tr>
      <% if (key === "user-agent") { %>
        <td><b><%= key %></b></td>
      <% } else { %>
        <td><%= key %></td>
      <% } %>
      <td><%= value %></td>
    </tr>
  <% }) %>
</table>
```

3. rész

Full-stack példa

Full-stack példa

- **Összefoglaló példa:** `4-templatedb/fullstack`
  - elmenti a bárhova érkezett HTTP kérések információit MySQL adatbázisba
  - `EJS` segítségével view-t generál korábbi kérésekkel
  - hibaoldalra irányít hibás működés esetén
  - `morgan` segítségével naplózik
- Mivel nőnek a szempontjaink száma, **modularizáljuk** az alkalmazásainkat, amennyire lehetséges
  - egy állomány csak egy dologgal foglalkozzon (**single responsibility principle**)
- Leválasztható elemek:
  - adatbázissal foglalkozó modul (`db`)
  - statikus állományok (`static`)
  - saját routinggal foglalkozó modul (`routes`)
  - saját middleware-ek (`middleware`)
  - a sablonjaink (`views`)

```
fullstack
├── db
│   └── db.js
├── middleware
│   ├── error.js
│   └── requestlogger.js
├── routes
│   └── requests.js
├── static
│   └── style.css
├── views
│   ├── partials
│   │   ├── head.ejs
│   │   └── navbar.ejs
│   ├── error.ejs
│   └── requests.ejs
├── index.js
├── package.json
└── setup.sql
```

Full-stack példa

db/db.js

```
import mysql from 'mysql2';

const pool = mysql.createPool({
  connectionLimit: 10,
  database: 'webprog',
  host: 'localhost',
  port: 3306,
  user: 'webprog',
  password: 'VgJUjBd8',
});
// létrehozzunk a táblázatot, ha még nem létezik
// a pool.query kér egy kapcsolatot a poolból
pool.query(`CREATE TABLE IF NOT EXISTS requests (
    method varchar(20),
    url varchar(50),
    date datetime);`, (err) => {
  if (err) {
    console.error(`Create table error: ${err}`);
    process.exit(1);
  } else {
    console.log('Table created successfully');
  }
});
```

```
// service metódus - beszúr egy DB sort
// majd callback-re reagál
export const insertRequest = (req, callback) => {
  const date = new Date();
  const query = 'INSERT INTO requests VALUES (?, ?, ?)
  pool.query(query, [req.method, req.url, date], callb
};

// service metódus - lekéri az összes sort
export const findAllRequests = (callback) => {
  const query = 'SELECT * FROM requests';
  pool.query(query, callback);
};

// service metódus - törli az összes sort
export const deleteAllRequests = (callback) => {
  const query = 'DELETE FROM requests';
  pool.query(query, callback);
};
```

Full-stack példa

middleware/requestlogger.js

```
import { insertRequest } from '../db/db.js';

// loggoljunk minden kérést az adatbázisba, mint middleware
export default function requestLogger(req, res, next) {
  insertRequest(req, (err) => {
    if (err) {
      res.status(500).render('error', { message: `Insertion unsuccessful: ${err.message}` });
    } else {
      next();
    }
  });
}
```

middleware/error.js

```
// nem talált oldalakra használjunk másik sablont
// vigyázat hogy ez a middleware a lánc VÉGÉRE kerüljön
export default function handleNotFound(req, res) {
  // kirajzoljuk a hibaoldalunk sablonját
  res.status(404).render('error', { message: 'The requested endpoint is not found' });
}
```

Full-stack példa

routes/requests.js

```
import express from 'express';
import * as db from '../db/db.js';

const router = express.Router();

// gyökérre vagy /indexre érkezett GET kérésre
// rendereljük a korábbi hívásokat
router.get(['/', '/index'], (req, res) => {
  db.findAllRequests((err, [requests]) => {
    if (err) {
      res.status(500).render('error', { message: `
        Selection unsuccessful: ${err.message}` });
    } else {
      // kirajzoljuk a requests sablont
      // a model az query eredménye
      res.render('requests', { requests });
    }
  });
});

// /delete-re érkezett POST esetén töröljük
// a tábla tartalmaát
router.post('/delete', (req, res) => {
  db.deleteAllRequests((err) => {
    if (err) {
      res.status(500).render('error', { message: `
        Deletion unsuccessful: ${err.message}` });
    } else {
      // siker esetén visszairányítunk
      // az eredeti lekérési oldalra
      res.redirect('index');
    }
  });
});

export default router;
```

Full-stack példa

views/error.ejs

```
<!DOCTYPE html>
<html>
  <%- include("partials/head.ejs", { title: "Error" }) %>
  <body>
    <%- include("partials/navbar.ejs") %>

    <h1>We have encountered an error</h1>
    <p>Message:</p>
    <p class="message"><%= message %></p>
    <p>Maybe you would like to visit the
      <a href="/requests">requests page</a>
    </p>

  </body>
</html>
```

Full-stack példa

views/requests.ejs

```
...
<h1>Requests</h1>
<% if (requests.length) { %>
  <table>
    <tr><th>Method</th><th>URL</th><th>Date</th></tr>
    <% requests.forEach((request) => { %>
      <tr>
        <td><%= request.method %></td>
        <td><code><%= request.url %></code></td>
        <td><%= request.date %></td>
      </tr>
    <% }) %>
  </table>
<% } else { %>
  <p class="message">No requests to show</p>
<% } %>

<form method="POST" action="/requests/delete">
  <input type="submit" value="Delete all requests">
</form>
...
```

Full-stack példa

views/partials/head.ejs

```
<head>
  <meta charset="utf-8">
  <title><%= title %></title>
  <link href="/style.css"
        rel="stylesheet"
        type="text/css">
</head>
```

views/partials/navbar.ejs

```
<div id="navbar">
  <span>
    <a href="/requests">Requests</a>
  </span>
  <span>
    <a href="/otherpage">Other page</a>
  </span>
</div>
```

static/style.css

```
body {
    font-family: Arial, Helvetica, sans-serif;
    text-align: center;
}
code {
    font-size: 125%;
}
.message {
    font-style: italic;
}
#navbar > span {
    display: inline-block;
    width: 25%;
}
table {
    width: 100%;
}
table, th, td {
    border: 2px solid gray;
    border-collapse: collapse;
    padding: 3px;
}
```

Full-stack példa

index.js

```
// imports...

const app = express();

// statikus állományok (pl. CSS/kliensoldali JS)
app.use(express.static(path.join(process.cwd(), 'static')));

// beállítjuk az EJS-t, mint sablonmotor
app.set('view engine', 'ejs');
app.set('views', path.join(process.cwd(), 'views'));

// naplózás (globális)
app.use(morgan('tiny'));
// kössük be a middleware-t, amely minden hívást DB-be szúr
app.use(requestLoggerMiddleware);
// kössük be a külső modulban megírt route-okat
app.use('/requests', requestRoutes);

// utolsóként kössük be a hibaoldalkezelőt globálisan
app.use(errorMiddleware);

app.listen(8080, () => { console.log('Server listening on http://localhost:8080/ ...'); });
```