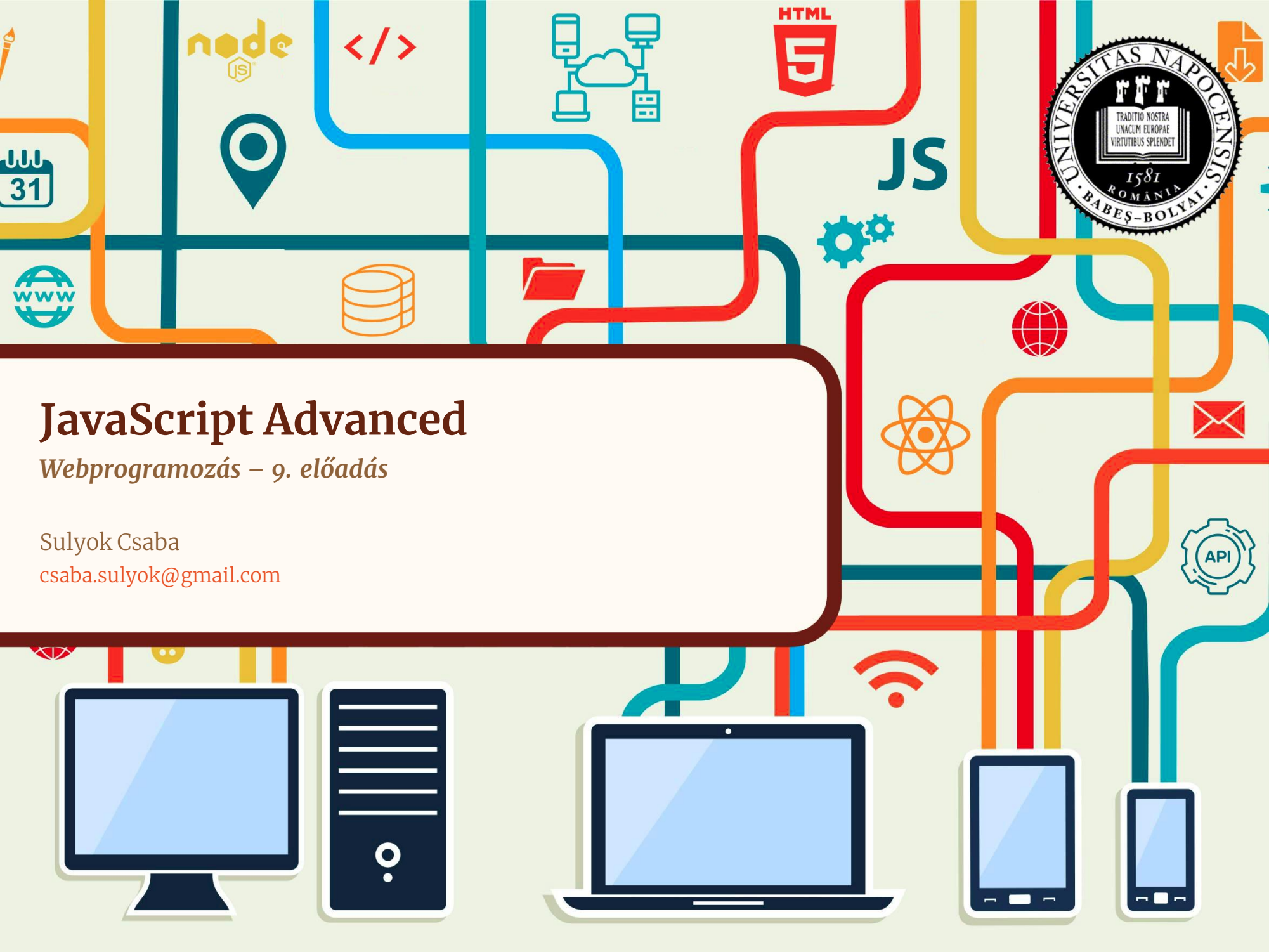

JavaScript Advanced

Webprogramozás – 9. előadás

Sulyok Csaba
csaba.sulyok@gmail.com

1. rész

Konstruktor-függvények, prototype, class

Saját objektumok létrehozása

- Saját objektum létrehozása "konstruktor"-szerű függvény segítségével:

```
function MyObject(name, size) {
    this.name = name;
    this.size = size;
}
```

- A `MyObject` függvény **előállít** egy JS objektumot, melyre a függvényben a `this` kulcsszóval hivatkozunk
- A `new` operátor segítségével létrehozhatjuk a `MyObject` típusú objektumot:

```
const myobj = new MyObject("nadragszij", 5);
console.log(`A ${myobj.name} merete ${myobj.size}m.`);
```

- A `new` operátor implicit módon létrehozza a `this` objektumot a függvény hívásának kezdetén, majd visszatéríti ezt a végén:

```
function MyObject(name, size) {
    // this = {}; <--- new esetén implicit
    this.name = name;
    this.size = size;
    // return this; <--- new esetén implicit
}
```

Saját objektum + függvény = metódus

- A konstruktoron belül bármilyen mezőt rendelünk hozzá a `this`-hez, az elérhető lesz az objektum mezőjeként
- Megadhatunk függvényeket is, amelyek ugyancsak elérik a `this` kulcsszót (vigyázat: ne használjunk arrow function-eket, mivel azok fölülírják a `this`-t):

```
function MyObject(name, size) {
    this.name = name;
    this.size = size;
    this.report = function() {
        console.log(`A ${this.name} merete ${this.size}m.`);
    }
}

const myobj = new MyObject("nadragszij", 5);
myobj.report();
```

Prototype

- Ha módosítanánk egy (konstruktor) függvény *kitermelt* objektumain, módosítanunk kell a függvény **prototípus**át (prototype)
- A **prototípus** elképzelhető úgy, mint egy "tervrajz" objektum, amely szerint a konstruktor-függvény a `new` operátorral elkészíti az új objektumokat

```
MyObject.prototype.unit = "m";
// ...
const myobj = new MyObject("nadragszij", 5);
console.log(`A ${myobj.name} merete ${myobj.size}${myobj.unit}.`);
```

- Függvény-típusú mezőt is adhatunk hasonlóan hozzá:

```
MyObject.prototype.report = function() {
    console.log(`a(z) ${this.name} merete ${this.size}`);
};
```

- Alapértelmezett tartalma a `constructor`, amelyben önmagát beállítja, hogy később típus-verifikációt lehessen végezni:

```
MyObject.prototype.constructor === MyObject; // true
```

- Az elkészült objektum eredeti prototípusát elérjük a `__proto__` kulcs segítségével:

```
const myobj = new MyObject("nadragszij", 5);
MyObject.prototype === myobj.__proto__; // true
```

`class`?

- A `prototype` és konstruktor-függvényekkel létrehozott objektumok esetén alkalmazhatóak az objektum-orientált megnevezések, pl. **adattagok** s **metódusok**, mivel ezek hozzá vannak kötve (bind) az objektumhoz a `this` segítségével.
- De mivel a létrehozott objektumhoz hozzárendelhetünk új kulcsokat (attribútumokat) és függvényeket (metódusokat), csak *mímeljük* a klasszikus objektum-orientáltságot.
- Ennek ellenére az ECMAScript 2017-től további szintaktikai megfeleltetéseket vezet be OO nyelvekhez a `class` kulcsszó által, amely a háttérben az említett szintaktikai elemeket használja. A következőek egyenlőek:

```
function User(name) {
  this.name = name;
  this.sayHi = function() {
    console.log(`Hi, my name is ${this.name}`);
  };
}

const user = new User("John");
user.sayHi();
```

```
class User {
  constructor(name) {
    this.name = name;
  }
  sayHi() {
    console.log(`Hi, my name is ${this.name}`);
  }
}

const user = new User("John");
user.sayHi();
```

class?

- Noha a `class` majdnem teljesen egyenértékű a `prototype` és konstruktor-függvény párosával, kisebb különbségeket észlelhetünk.
- Egy `class`-szel deklarált osztály nem hívható meg a `new` kulcsszó nélkül.

```
function User(name) {
  this.name = name;
  this.sayHi = function() {
    console.log(`Hi, my name is ${this.name}`);
  };
}

User(); // undefined
```

```
class User {
  constructor(name) {
    this.name = name;
  }
  sayHi() {
    console.log(`Hi, my name is ${this.name}`);
  }
}

User(); // HIBA: Class constructor User cannot
        // be invoked without 'new'
```

- `class`-ben megadott metódusok (mint a `sayHi`) nem "enumerable"-ek, így `for`-ral való iterálás esetén nem jönnek elő, ellentétben a `prototype`-pal.

```
Object.keys(new User()); // [ 'name', 'sayHi' ]
```

```
Object.keys(new User()); // [ 'name' ]
```

Method binding

- OO rendszerekkel ellentétben az osztályokban használt metódusok nincsenek *véglegesen hozzákötve* a létrehozott példányokhoz (a használt `this` változhat), ezért ha egy metódusra való hivatkozást átadunk más függvénynek, elveszítjük a kötést (binding).

```
user = new User('James');

user.sayHi();                           // unbound, but OK
setTimeout(() => user.sayHi(), 1000); // unbound, but OK
setTimeout(user.sayHi, 1000);           // rebound, undefined
```

- A függvényeken definiált `bind` metódussal hozzákötethetünk a metódust egy objektumhoz, így az mindig a `this` kulcsszó szerepét játssza.

```
user.sayHi = user.sayHi.bind(user);
setTimeout(user.sayHi, 1000);           // bound after declaration, OK
```

Method binding

- Újrahasznosíthatóság kedvéért a metódusok a jelen példányhoz köthetőek a konstruktorban:

```
class User {
  constructor(name) {
    this.name = name;
    this.sayHi = this.sayHi.bind(this);
  }
  ...
}
setTimeout(user.sayHi, 1000); // bound in constructor, OK
```

- Hogy elkerüljük a metódusok felsorolását ilyen esetekben, alkalmazhatjuk az `auto-bind` külső npm csomagot:

```
import autoBind from 'auto-bind';

class User {
  constructor(name) {
    this.name = name;
    autoBind(this); // bind all methods automatically
  }
  ...
}
setTimeout(user.sayHi, 1000); // auto-bound in constructor, OK
```

2. rész

Promise-ok és az async/await szintaxis

Promise-ok

- klasszikus JavaScript események (`keyup`, `mousedown`, stb.) hátrányai:
  - előjöhetnek többször ugyanazon kontextusban
  - nincs beépített mechanizmus hibakezelésre—a lekezelő függvény a paraméterek szerint észlelhet hibákat
  - hozzárendelt függvények nem hívódnak meg, ha az esemény már korábban bekövetkezett
- a promise-ok hasonlóak az eseményekhez, de:
  - egyszer váltódnak ki
  - különbséget teszünk a sikeres és sikertelen végkimenetelű események közt (s egy esemény nem válthat állapotot)
  - ha egy promise kiváltódik, de callback függvényt később rendelünk hozzá, akkor az egyből meghívódik (i.e. a promise megvárja a lekezelő callback függvény létrejöttét)
- promise-ok állapota:
  - **pending** (várakozásban) - még nem derült ki a sikeresség (nem váltódott ki az esemény)
  - **settled** (eldőlt) - kiderült a sikeresség, ezen belül:
    - **fulfilled** (beteljesedett) - sikeres
    - **rejected** (visszautasított) - sikertelen/hiba lépett fel

Promise-ok

- A Promise-okat az ECMAScript 2015-ös verziójában vezették be, így modern böngészők is támogatják
- A Promise-okat a `Promise` konstruktorfüggénnyel készítjük—ez egy függvényt vár el, melynek 2 callback függvény paramétere van:

```
const promise = new Promise((resolve, reject) => {
  // fő akció végrehajtása
  if (/* sikeres végkimenetel */) {
    resolve('Siker');
  } else {
    reject(Error('Sikertelen'));
  }
});
```

- A megadott fő akció végre lesz hajtva **egyszer**, még ha nem is rendelünk callback(ek)et hozzá
- Ha a promise lejárta után rendeljük hozzá a callbacket, a megfelelő lekezelő függvény egyből lefut

Promise-ok

- A Promise-hez hozzárendelhetünk lekezelési callback függvényeket a `then` metódussal:

```
promise.then((result) => { console.log(result); /* fulfilled */ },
             (error)  => { console.log(error);  /* rejected */ });
```

- Mindkét paraméter opcionális, lekezelhető csak a sikeres vagy csak a sikertelen végkimenet
- Ha csak a hibát szeretnénk lekezelni, használhatjuk a `catch` metódust:

```
promise.catch((error) => { ... });
```

- Ez egyenértékű a `then`-nel az első paraméter kihagyásával:

```
promise.then(undefined, (error) => { ... });
```

Promise-ok láncolása

- Mind a `then`, mind a `catch` metódusok ugyancsak `Promise` típusú objektumokat térítenek vissza:

```
const firstPromise = new Promise((resolve, reject) => { ... });
console.log(firstPromise.constructor);  // function Promise() ...
const secondPromise = firstPromise.then((result) => { ... });
console.log(secondPromise.constructor); // function Promise() ...
```

- Így több lekezelő függvényt egymás után köthetünk–minden láncszem az őt megelőző láncszemfüggvény visszatérítési értékét kapja meg paraméterként:

```
const promise = new Promise((resolve, reject) => {
  resolve(1);
});

promise.then((val) => {
  console.log(val); // 1
  return val + 2;
}).then((val) => {
  console.log(val); // 3
})
```

Promise.all

```
const promise1 = Promise.resolve(3);
const promise2 = 42;
const promise3 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('done'), 100);
});

Promise.all([promise1, promise2, promise3]).then((values) => {
  console.log(values); // expected in 100ms: Array [3, 42, "done"]
});
```

- Tekinthetünk úgy a `Promise`-okra, mint leválasztott szálak (noha ez természetesen lehetetlen a JS természete miatt). Ebben az esetben egy szálat "bevárunk" a `then` lekezelésével - de **többet** is be tudunk várni (join) a `Promise.all` metódus segítségével.
- A `Promise.all` fogad egy *tömb* Promise-t, s visszatérít egy másik Promise-t, amely akkor záródik, amikor a megadott tömb összes eleme is befejeződött. A válasza ugyancsak egy tömb az összes megadott Promise eredményével, egyező sorrendben.
- Hasznos az aszinkron taskok párhuzamosítására, ha ezek **nem függenek egymástól**.
- Hasonló aggregáló metódusok: `Promise.race`, `Promise.any`

Promise példa: `mysql` query-hívás

- **Példa:** Adatbázist elérő függvények Promise alapú használata:

```
// Promise-okba enkapszulálja a MySQL felé intézett hívásokat
const queryAsync = (query, options = []) => new Promise((resolve, reject) => {
  pool.query(query, options, (error, results) => {
    if (error) {
      reject(new Error(`Error while running '${query}: ${error}'`));
    }
    resolve(results);
  });
});
```

- Egy meglévő callback-alapú függvényt automatikusan `Promise` használatára bírhatunk a beépített `util` csomag `promisify` metódusával.
- Ez feltételezi, hogy a megadott függvény utolsó paramétere a callback, amelyet egy új `Promise` feloldásával szimulál, hasonlóan a korábbi példához.

```
import util from 'util';
const queryAsync = util.promisify(pool.query);
```

Promise példa: `mysql` query-hívás

- Használat klasszikus Promise-szintaxissal:

```
app.get('/blogPosts', (request, response) => {
  queryAsync('SELECT * FROM blogPosts')
    .then((blogPosts) => { response.render('blogPosts', { blogPosts }); })
    .catch((error) => { response.render('error', { error }); });
});

app.post('/deleteBlogPost', (request, response) => {
  const blogPostId = Number(request.query.blogPostId);
  // láncolás - a like és comment törlése párhuzamosítható
  Promise.all([
    queryAsync('DELETE FROM comments where blogPostId = ?', [blogPostId]),
    queryAsync('DELETE FROM likes where blogPostId = ?', [blogPostId])
  ])
    .then(() => queryAsync('DELETE FROM blogPosts where blogPostId = ?', [blogPostId]))
    .then(() => { console.log('All 3 queries done'); });
    .catch((error) => { console.error(`Error occurred: ${error.message}`); }); // közös hibakezelés
});
```

async/await

- `Promise`-okkal történő munkának **alternatív szintaxisa**
- Csak ECMAScript 2017-től arrafele használható, így a böngészők nem ismerik.
- Egy `async` módosítóval ellátott függvény a háttérben `Promise`-ként működik, amelyből a visszatérés (`return`) jelképezi a `Promise` helyes befejezését. Pl. a következő 2 egyenértékű:

```
// promise
const add = (a, b) => new Promise((resolve) => { resolve(a + b); });
// async
const add = async (a, b) => a + b;
```

- Egy `async` függvényen belül más Promise-ok eredményét várhatjuk be s láncolhatjuk a jelen híváshoz az `await` kulcsszóval.

```
const getAllBlogPosts = async () => {
  // bevárjuk a promise-t, s végértékét változóhoz rendeljük
  const blogPosts = await queryAsync('SELECT * FROM blogPosts');
  response.render('blogPosts', { blogPosts });
};
```

- Az `await` nem megengedett `async` függvényeken kívül, így top-level kódban sem.

async/await

- Az esetleges hibákat, melyeket egy `Promise` kiválthat (és `catch` metódussal kezelnénk le) itt `try/catch` blokkokkal kezelhetünk le:

```
// promise
const deleteBlogPost = (blogPostId) => Promise.all([
  queryAsync('DELETE FROM comments where blogPostId = ?', [blogPostId]),
  queryAsync('DELETE FROM likes where blogPostId = ?', [blogPostId])
])
  .then(() => queryAsync('DELETE FROM blogPosts where blogPostId = ?', [blogPostId]))
  .then(() => { console.log('All 3 queries done'); });
  .catch((error) => { console.error(`Error occurred: ${error.message}`); });

// async/await
const deleteBlogPost = async (blogPostId) => {
  try {
    await Promise.all([
      queryAsync('DELETE FROM comments where blogPostId = ?', [blogPostId]),
      queryAsync('DELETE FROM likes where blogPostId = ?', [blogPostId])
    ]);
    await queryAsync('DELETE FROM blogPosts where blogPostId = ?', [blogPostId]))
    console.log('All 3 queries done');
  } catch (error) {
    console.error(`Error occurred: ${error.message}`);
  }
};
```

3. rész

Full-stack példa (újra)

Full-stack példa kiegészítve

- Az alábbi elemek vannak kiegészítve a korábban bemutatott verzióhoz képest:
  - `Promise`-ok használata az adatbázis eléréshez
  - adatbázis-kapcsolat `class`-be burkolása
  - `async-await` használata a routerektől az adatelérési metódusok felé

Full-stack példa kiegészítve

db/connection.js – a pool.query promise-szá alakítása + osztályba burkolás

```
import mysql from 'mysql2';
import autoBind from 'auto-bind';

export class DbConnection {
  constructor() {
    this.pool = mysql.createPool({ ... });
    autoBind(this);
  }

  executeQuery(query, options = []) {
    return new Promise((resolve, reject) => {
      this.pool.query(query, options, (error, results) => {
        if (error) {
          reject(new Error(`Error while running '${query}: ${error}'`));
        }
        resolve(results);
      });
    });
  }
}

export default new DbConnection(); // default instance
```

Full-stack példa kiegészítve

db/requests.js – Promise-ok visszatérítése

```
// Adatbázis műveleteket végző modul
import dbConnection from './connection.js';

// service metódus - lekéri az összes sort
// Promise-t térít vissza
export const findAllRequests = () => {
  const query = 'SELECT * FROM requests';
  return dbConnection.executeQuery(query);
};
```

routes/requests.js – async/await router metódus

```
import * as db from '../db/requests.js';
// ...
router.get(['/', '/index'], async (req, res) => {
  try {
    const requests = await db.findAllRequests();
    res.render('requests', { requests });
  } catch (err) {
    res.status(500).render('error', { message: `Selection unsuccessful: ${err.message}` });
  }
});
```