1 ES6 (JavaScript) alapok.

1.1 Egyszerű adattípusok

1.1.1 Symbol

Ha egy objektumon belül kulcsokat szeretnél létrehozni, amelyek garantáltan nem ütköznek más kulcsokkal, a Symbol erre tökéletes.

```
const azonosito = Symbol('id');
```

A paraméterként megadott szöveg csak hibakeresésre használható.

Nem iterálható.

1.1.2 Number

1.1.3 String

1.1.4 Boolean

1.2 Összetett adattípusok

1.2.1 Objektumok (Object)

Kulcs-érték párokat tárol.

Hivatkozás: objektum.név, vagy objektum['név'].

Metódusok: Object.keys(), Object.values(), Object.entries()

dinamikus és prototípus-objektumok

```
//Minta
var obj = ;

//Teszt
ok( typeof obj === 'object', 'Objektum jött létre' );
ok( Object.getPrototypeOf(obj) === Object.prototype, 'A prototípus az
Object.prototype' );

//Üres objektumliterállal kompatibilis objektum létrehozása
var obj = Object.create(Object.prototype);
//Teszt
ok( typeof obj === 'object', 'Objektum jött létre' );
ok( Object.getPrototypeOf(obj)_ === Object.prototype, 'A prototípus objektum az
Object.prototype' );

//Prototípus nélküli objektum létrehozása
var obj = Object.create(null);
//Teszt
ok( typeof obj === 'object', 'Objektum jött létre' );
ok( Object.getPrototypeOf(obj) === null, 'Nincsen prototype objektuma' );
```

1.2.2 Tömbök (Array)

Rendezett lista (indexek alapján), ahol minden elemnek van egy numerikus indexe (nullától kezdődően). Hivatkozás: array[0]

1.3 Operátorok

https://www.w3schools.com/js/js_es6.asp

1.4 Elágazó utasítások

1.5 Nyíl függvény

A function() {} helyett az ES6 vagy ECMAScript 2015-ben bevezették a () => {} jelölést, amely nem csak jelölésben tér el az elődjétől.

1.5.1 Implicit visszatérés

A nyílfüggvények implicit visszaadást tesznek lehetővé: az értékek *return a kulcsszó használata nélkül* kerülnek visszaadásra.

1.5.2 ’this’ működése

A ’this’ értékét mindig is a környezete (kontextus) határozza meg. Emiatt a nyíl függvények nem használhatóak objektum metódusként.

```
const car = {
  model: 'Fiesta',
  manufacturer: 'Ford',
  fullName: () => {
    return `${this.manufacturer} ${this.model}`
  }
}
```

Ebben a kódban car.fullName() nem fog működni, és a következőt adja vissza "undefined undefined"

Ez az események kezelése során is lehet probléma. A DOM eseményfigyelők this célelemként vannak beállítva, és ha this-re az eseménykezelőben hivatkozunk,

```
const link = document.querySelector('#link')
link.addEventListener('click', () => {
  // this === window
})
```

akkor egy hagyományos funkcióra van szükség:

```
const link = document.querySelector('#link')
link.addEventListener('click', function() {
  // this === link
})
```

1.5.3 Spread operátor ....

Az iterálható objektumokat (pl. listák, tömbök, sztringek) kibontja (*objektum destrukturálás*). Hasonló funkciókat tartalmaznak a Ruby, Python és PHP nyelvek is.

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combined = [...arr1, ...arr2];
console.log(combined); // Output: [1, 2, 3, 4, 5, 6]

function sum(a, b, c) {
    return a + b + c;
}

const numbers = [1, 2, 3];
console.log(sum(...numbers)); // Output: 6
```

1.5.4 Rest Operátor ...

Több argumentum vagy tömb elem összegyűjtése egyetlen változóba. Paraméter átadásnál rendkívül megnöveli az átláthatóságot, egyszerűbb kódolást tesz lehetővé.

```
function sum(...numbers) {
    return numbers.reduce((acc, curr) => acc + curr, 0);
}
console.log(sum(1, 2, 3, 4)); // Output: 10
```

A fenti példában a rest operátor összegyűjti a `sum` függvény összes argumentumát egy `numbers` nevű tömbbe.

```
const [first, ...rest] = [1, 2, 3, 4];
console.log(first); // Output: 1
console.log(rest); // Output: [2, 3, 4]
```

Itt a destrukturálás során az első elemet különválasztja, a többit pedig a `rest` tömbbe gyűjti.

1.6 Template Literals ~ AltGr+7 ~ backtick

Az ES2015 / ES6 újdonsága, karakterláncok kezelésére. Segítségével több soros szövegeket is tárolhatunk, a ’|n’ szekvencia begépelése nélkül. Egyszerűen interpolálhatunk kifejezéseket a szövegbe.

```
const myVariable = 'test'
const string = `something ${myVariable}` //something test
```

vagy kifejezéseket

```
const string = `something ${1 + 2 + 3}`
const string2 = `something ${doSomething() ? 'x' : 'y'}`
```

1.6.1 Callback function

Egy callback függvény egy olyan függvény, amelyet egy másik függvény paramétereként adunk át, és amelyet az adott függvény belsejében hívnak meg, hogy valamilyen műveletet vagy rutint hajtsanak végre. A callback függvények egyaránt lehetnek szinkron vagy aszinkron jellegűek.

1.7 Iterációk

Iterációs utasítás	Milyen típusokon használható	Visszatérési érték	Mikor használjuk
for	Bármilyen	Nincs	Általános célú iteráció
while	Bármilyen	Nincs	Feltételes iteráció, nem ismert vég
do...while	Bármilyen	Nincs	Legalább egyszer futnia kell
for...in	Objektumok, tömbök	Kulcsok	Objektum tulajdonságok bejárása
for...of	Iterálható objektumok	Értékek	Iterálható értékek bejárása
forEach	Tömbök	Nincs	Művelet végrehajtása minden elemre
map	Tömbök	Új tömb	Új tömb generálása átalakított értékekkel
filter	Tömbök	Új tömb	Szűrés adott feltétel alapján
reduce	Tömbök	Egyetlen érték	Összesítés egyetlen értékre
some	Tömbök	Logikai érték	Legalább egy elem megfelel-e
every	Tömbök	Logikai érték	Minden elem megfelel-e
find	Tömbök	Elem értéke	Első elem keresése adott feltétellel
findIndex	Tömbök	Elem indexe	Első elem indexének keresése

2 Node.js alapok

2.1 Telepítés

A Node.js honlapról le lehet tölteni az alapot. További kiegészítőket vagy az npm vagy az npx segítségével érhetsz el.

2.1.1 npm (Node Package Manager)

Az npm egy csomagkezelő, amelyet a Node.js csomagok telepítésére, frissítésére és eltávolítására használnak.

Az npm segítségével telepíthetsz csomagokat globálisan vagy lokálisan a projektedben.

2.1.2 npx (Node Package Runner)

Az npx egy eszköz, amely lehetővé teszi csomagok futtatását anélkül, hogy előzetesen telepítenénk őket.

Az npx segítségével futtathatsz csomagokat közvetlenül az npm registry-ből, vagy a projektedben lokálisan telepített csomagokat.

2.2 Modulok közötti hivatkozások

A Node.js-ben minden fájl egy külön modulnak tekinthető. A modulok lehetővé teszik a kód szervezését és újrahasználhatóságát. Törekedni kell arra, hogy egy fájlban ne legyen száz sor. Az alkalmazást területekre kell bontani és az adott területhez tartozó függvényeket és változókat egy fájlban elhelyezni.

Nagyobb méretű alkalmazásoknál sok fájl keletkezhet, ezért nagyon fontos, hogyan nevezzük el a fájlokat. Használjunk kisbetűket, a szavakat válasszuk el kötőjellel a jobb olvashatóság miatt.

A Node.js támogatja a CommonJS és az ECMAScript modulokat is. A CommonJS modulok esetében a require és module.exports szintaxist használjuk, míg az ECMAScript moduloknál az import és export szintaxist.

fájl	*CommonJS*	*ECMAScript*

index.js	`const adatbazis = require('./adatbazis');`
	
	`adatbazis.connect();`
		`import { connect } from './adatbazis.js';`
		
		`connect();`

adatbazis.js	`function connect() {`
	`  console.log('Csatlakozás az adatbázishoz...');`
	`}`
	
	`module.exports = { connect };`
		`export function connect() {`
		`  console.log('Csatlakozás az adatbázishoz...');`
		`}`

Az ECMAScript modulok esetében a fájlok kiterjesztése általában .mjs, vagy a package.json fájlban meg kell adni a "type": "module" beállítást, hogy a .js fájlokat modulokként kezelje

A CommonJS modulok szinkron módon töltődnek be. Amikor egy modult require-rel betöltünk, a kód végrehajtása megáll, amíg a modul teljesen be nem töltődik és ki nem értékelődik.

Az ECMAScript modulok aszinkron módon töltődnek be. Az import utasítás nem blokkolja a kód végrehajtását, és a modulok betöltése párhuzamosan történik.

Import esetén megkülönböztetünk:

- **Named export**: Ha több exportált elemre van szükséged, vagy moduláris kódot szeretnél.
- **Default export**: Ha egyetlen fő funkciót vagy elemet akarsz exportálni.
- *** as import**: Ha az összes exportált elemet egy helyen szeretnéd kezelni, például névtérként.

2.2.1 Named export

modul.js:

```
export function fuggvenyEgy() {
  console.log('Ez az első függvény');
}

export function fuggvenyKetto() {
  console.log('Ez a második függvény');
}

export const konstansErtek = 42;
```

index.js:

```
import { fuggvenyEgy, fuggvenyKetto, konstansErtek } from './modul.js';

fuggvenyEgy(); // Kimenet: Ez az első függvény
fuggvenyKetto(); // Kimenet: Ez a második függvény
console.log(konstansErtek); // Kimenet: 42
```

2.2.2 Default Export és Named Export együtt

modul.js:

```
export default function alapertelmezettFuggveny() {
  console.log('Ez az alapértelmezett függvény');
}

export function masikFuggveny() {
  console.log('Ez egy másik függvény');
}
```

index.js:

```
import alapertelmezettFuggveny, { masikFuggveny } from './modul.js';

alapertelmezettFuggveny(); // Kimenet: Ez az alapértelmezett függvény
masikFuggveny(); // Kimenet: Ez egy másik függvény
```

2.2.3 Mindent importálni egy objektumba

modul.js:

```
export function fuggvenyEgy() {
  console.log('Ez az első függvény');
}

export function fuggvenyKetto() {
  console.log('Ez a második függvény');
}

export const konstansErtek = 42;
```

index.js:

```
import * as modul from './modul.js';

modul.fuggvenyEgy(); // Kimenet: Ez az első függvény
modul.fuggvenyKetto(); // Kimenet: Ez a második függvény
console.log(modul.konstansErtek); // Kimenet: 42
```

2.3 Promise használata

A Node.js filozófiájának alapja az aszinkron működés. A promise-t egy olyan érték helyettesítőjeként definiáljuk, amely végül elérhetővé válik. Az ES2015- ben vezették be, most pedig az ES2017- ben az aszinkron funkciók váltották fel őket.

Amint egy ígéret létrejön, függő állapotba kerül . Ez azt jelenti, hogy a hívó függvény folytatja a végrehajtást, miközben várja az ígéretet, hogy elvégezze a saját feldolgozását, és visszajelzést adjon a hívó függvénynek.

Egy ponton a hívó függvény arra vár, hogy **feloldott** vagy **elutasított állapot**ban adja vissza az ígéretet, de a függvény folytatja a végrehajtást, amíg az ígéret működik .

3 Express telepítése és első alkalmazás

3.1 Express telepítése

Kezdd egy új Node.js projekt létrehozásával, majd telepítsd az Express-t
npm install express
paranccsal.

3.2 Első szerver létrehozása

Készíts egy alapvető Express szervert, ami egy egyszerű "Hello World" üzenetet ad vissza.

4 URL paraméterek és lekérdezési paraméterek kezelése

Cél: Megérteni, hogyan lehet URL paraméterekkel és lekérdezési paraméterekkel dolgozni.

Az Express-ben az adatok fogadásának főbb módjai a következők:

1. **Lekérdezési paraméterek** (req.query)

```
// GET /search?name=John&age=30

app.get('/search', (req, res) => {
   const name = req.query.name; // "John"
   const age = req.query.age;   // "30"
   res.send(`Name: ${name}, Age: ${age}`);
});
```

2. **Útvonal paraméterek** (req.params)

```
// GET /users/123

app.get('/users/:id', (req, res) => {
   const userId = req.params.id; // "123"
   res.send(`User ID: ${userId}`);
});
```

3. **Törzs (body) paraméterek** (req.body) – JSON és URL-kódolt adatok.

HTML

```
<form method="POST" action="/users">
  <input type="text" name="name">
  <input type="number" name="age">
  <button type="submit">Submit</button>
</form>
```

JS:

```
app.use(express.urlencoded({ extended: true })); // Middleware az URL-kódolt adatok kezeléséhez

app.post('/users', (req, res) => {
   const { name, age } = req.body;
   res.send(`Received user: ${name}, Age: ${age}`);
});
```

4. **Fájlok feltöltése** (multer middleware-rel)

HTML:

```
<form action="/upload" method="POST" enctype="multipart/form-data">
  <input type="file" name="myfile">
  <button type="submit">Upload</button>
</form>
```

JS:

```
const multer = require('multer');
const upload = multer({ dest: 'uploads/' }); // Mappába menti a fájlokat

app.post('/upload', upload.single('myfile'), (req, res) => {
    res.send(`File uploaded: ${req.file.originalname}`);
});
```

5. **Fejléc adatok** (req.headers)

```
app.get('/headers', (req, res) => {
    const userAgent = req.headers['user-agent'];
    res.send(`User Agent: ${userAgent}`);
});
```

6. **Cookie-k fogadása** (cookie-parser middleware-rel)

```
const cookieParser = require('cookie-parser');
app.use(cookieParser());

app.get('/cookies', (req, res) => {
    const myCookie = req.cookies.myCookieName;
    res.send(`Cookie value: ${myCookie}`);
});
```

Gyakorlati feladat:

- Hozz létre egy útvonalat, amely egy URL paraméter alapján fogad be egy adatot, például: /users/:id, és visszaadja az adott felhasználót.
- Készíts egy olyan útvonalat, amely lekérdezési paramétereket (*query parameters*) kezel, például: */search?name=John*.

5 Statikus fájlok kiszolgálása

Hogyan szolgálhatók ki statikus fájlok (HTML, CSS, képek stb.) Express alkalmazásból.

```
const express = require('express'); //-- synchronus csatlakozás
const app = express();
const port = 3000;
const cors = require('cors');
app.use(cors({origin: 'http://localhost:3000'}));
const fs = require('fs'); //-- lehetővé teszi a képek betöltése
const path = require('path'); //- lehetővé teszi a könyvtár betöltése
app.use(express.static(path.join(__dirname, 'public'))); //-- statikus fájlok betöltése

app.get('/', (req, res) => {
   res.header('Content-Type', 'text/html; charset=utf-8');
   res.status(201).sendFile(__dirname + '/public/index.html');
});


app.get('/login', (req, res) => {
   res.header('Content-Type', 'text/html; charset=utf-8');
   res.status(201).sendFile(__dirname + '/public/login.html');
});
```

Gyakorlat:

- Készíts egy alap weboldalt, amelynek a HTML fájljait, CSS stílusait és képeit az Express a /public mappából szolgáltatja ki.
- Használd az express.static() middleware-t a statikus fájlok kiszolgálására.

6 REST API készítése

Express remekül alkalmas RESTful API-k készítésére. Tanuld meg, hogyan kezelheted a különböző HTTP metódusokat (GET, POST, PUT, DELETE), és hogyan strukturálhatod API-jaidat.

Gyakorlat:

- **Adatkezelés és JSON válaszok**: Készíts egy REST API-t, amely JSON adatokat szolgáltat.
- Hozz létre egy egyszerű CRUD (*Create, Read, Update, Delete*) alkalmazást, amely pl. felhasználók adatait kezeli.

7 Middleware használata

Az Express middleware-ek az Express alkalmazás szíve-lelke. Ezek olyan funkciók (függvények), amelyek hozzáférést biztosítanak a bejövő kérésekhez, és módosíthatják azokat, illetve válaszokat generálhatnak vagy átadhatják a vezérlést a következő middleware-nek a láncban. A middleware-ek alapvető szerepet játszanak az alkalmazás logikájának kezelésében és szervezésében.

Amikor egy HTTP kérést kap a szerver, az Express végigfut a **middleware lánc**on, és minden egyes middleware megkapja a következő három dolgot:

- **Kérés objektumot** (req): A kliens által küldött kérésről szóló információk.
- **Válasz objektumot** (res): Ezen keresztül küldheted vissza a válaszokat a kliensnek.
- **Következő middleware hívása** (next()): Ez a függvény hívja meg a következő middleware-t a láncban.

A middleware-lánc a programban való fizikai elhelyezkedés alapján épül fel.

```
app.use((req, res, next) => {
  console.log('Első middleware');
  next();
});

app.use((req, res, next) => {
  console.log('Második middleware');
  next();
});

app.get('/', (req, res) => {
  res.send('Főoldal');
});
```

A next() hívás tudja, hogy melyik a következő middleware a láncban, mert az Express a middleware-eket belsőleg sorban regisztrálja, amikor az app.use() vagy app.get(), stb. metódusokat meghívod. Amikor egy middleware meghívja a next()-et, az Express automatikusan a következő regisztrált middleware-t futtatja.

A middleware-ek rugalmas módon szervezik az alkalmazás működését, lehetőséget adva a kód újra felhasználására, tisztább logika kialakítására és a moduláris felépítésre.

7.1 Middleware típusok

7.1.1 Alkalmazás szintű middleware

Az egész alkalmazásra érvényesek, minden útvonalra és HTTP metódusra lefutnak. Ezt általában az app.use() metódussal definiálják.

```
app.use((req, res, next) => {
  console.log('Alkalmazás szintű middleware.');
  next();
});
```

7.1.2 Route (útvonal) szintű middleware

Csak egy adott útvonalon, vagy útvonalcsoporton futnak le. A middleware függvényt itt paraméterként adhatod át egy adott útvonalnak.

```
app.get('/user/:id', (req, res, next) => {
  console.log('Csak a /user/:id útvonalra fut le');
  next();
}, (req, res) => {
  res.send('Felhasználói információk');
});
```

7.1.3 Harmadik fél által készített middleware-ek

Express middleware-eket harmadik felek is készítenek, amelyek megkönnyítik például a hitelesítést, a fájlfeltöltést, a naplózást stb. Ezeket NPM csomagokon keresztül telepítheted és használhatod.

```
const morgan = require('morgan');
app.use(morgan('combined')); // Naplózza a kéréseket
```

7.1.4 Hiba middleware

Ez egy speciális típusú middleware, amelyet a hibák kezelésére használnak. Egy hiba middleware-t négy paraméterrel definiálnak: *err, req, res* és *next*. Ezek csak akkor hívódnak meg, ha valami hiba történik az alkalmazásban.

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Valami elromlott!');
});
```

7.1.5 Saját middleware

Használd az Express beépített middleware-jeit, mint a express.json() vagy a express.static(), és ha nem találsz megfelelőt, akkor készítsd el a sajátodat.

Leggyakrabban az alábbi területeken lesz rá szükséged:

- Kérések feldolgozása: *Lehetővé teszik az adatok, pl. JSON vagy form adat feldolgozását.*
- Naplózás: *Segíthetnek naplózni a kérések érkezését.*
- Hitelesítés: *Ellenőrizhetik a felhasználók jogosultságait.*
- Statikus fájlok kiszolgálása: *Statikus fájlokat szolgálhatnak ki, például HTML, CSS, JavaScript.*
- Hiba kezelés: *Kezelhetik az alkalmazásban felmerülő hibákat.*

Gyakorlatok:

- Implementálj egy middleware-t, ami minden kérés előtt naplózza a kérés időpontját.
- Hozz létre egy egyszerű logger middleware-t, amely minden kérésnél kiírja a konzolra az időbélyeget, az útvonalat és a HTTP metódust.

- Készíts egy hibakezelő middleware-t, amely kezeli a nem létező útvonalakat (404-es hibák).

```
app.use((req, res, next) => {
   res.status(404).send('Az oldal nem található!');
});
```

A 404-es hibakezelő middleware-t mindig a route-ok után kell elhelyezni, hogy a többi route ellenőrzése után fusson le.

- Alkalmazz egy body-parser middleware-t a JSON adat kezelésére a POST és PUT metódusok esetén.

```
function validateRequest(req, res, next) {
  if (!req.body.name || req.body.name.length < 5) {
    return res.status(400).send('Name must be at least 5 characters long');
  }
  next();
}
```

- Készíts a kérések számának a korlátozására egy middleware-t

```
let requestCounts = {};

function rateLimitMiddleware(req, res, next) {
  const userIp = req.ip;
  requestCounts[userIp] = (requestCounts[userIp] || 0) + 1;

  if (requestCounts[userIp] > 100) {
    return res.status(429).send('Too many requests');
  }

  next();
}
```

- Készíts a CORS kezelésre

```
function corsMiddleware(req, res, next) {
  res.setHeader('Access-Control-Allow-Origin', '*');
  res.setHeader('Access-Control-Allow-Methods', 'GET,POST,PUT,DELETE');
  next();
}
```

8 Adatbázis integráció

8.1 MySQL adatbázis közvetlen elérésére middleware segítségével

```
const express = require('express');
const mysql = require('mysql2');
const app = express();
const port = 3000;

// MySQL kapcsolat létrehozása
const db = mysql.createConnection({
    host: 'localhost',   // Az adatbázis szerver címe
    user: 'root',        // MySQL felhasználó
    password: '',        // MySQL jelszó
    database: 'pizza'    // Az adatbázis neve
});

// Kapcsolat indítása
db.connect((err) => {
    if (err) {
        console.error('Hiba történt a MySQL kapcsolódás során: ', err);
        return;
    }
    console.log('Csatlakozás a MySQL adatbázishoz sikeres.');
});

// GET kérés, amely JSON formátumban küldi vissza a vevő adatokat
app.get('/vevok', (req, res) => {
    const sql = 'SELECT * FROM vevo'; // SQL lekérdezés
    db.query(sql, (err, results) => {
        if (err) {
            console.error('Hiba a lekérdezés során: ', err);
            res.status(500).json({ error: 'Adatbázis hiba' });
            return;
        }
        res.json(results); // Az eredményt JSON formátumban küldjük vissza
    });
});

// Az alkalmazás elindítása
app.listen(port, () => {
    console.log(`Az alkalmazás fut a http://localhost:${port} címen.`);
});
```

8.2 ORM használata

8.2.1 Mongoose

8.2.2 Sequelize

8.2.3 Prisma

8.2.4 Objection.js

8.2.5 TypeORM[1]

Gyakorlat:

- Hozz létre egy alkalmazást, amely adatokat kér le és tárol egy adatbázisban.
- Integrálj egy MongoDB vagy MySQL adatbázist az alkalmazásba.
- Töltsd le az adatokat az adatbázisból (pl. felhasználói adatokat) és jelenítsd meg őket a REST API-n keresztül.
- Adatok beillesztése és frissítése az adatbázisban a POST és PUT műveletek segítségével.

9 Sablonmotorok használata

Sablonmotor jelentősen megkönnyíti a kód és az adatok elválasztását, a weboldalak dinamikussá tételét, valamint a fejlesztés gyorsítását.

Express támogat több sablonmotort.

9.1 EJS (Embedded JavaScript)

<% %> és <%= %> jelölésekkel dolgozik.

- A <% %> kódot futtat, de nem jelenít meg tartalmat.
- A <%= %> kódot futtat, és megjeleníti a kifejezés értékét HTML-ben, HTML-eszképelt formában.

Egyszerű és közvetlen, támogatja a vezérlési szerkezeteket, mint a ciklusok és feltételek (például for, if).

9.2 Handlebars.js

Használja a „mustache” stílusú kódot {{ }}, amely egyszerű és tiszta megjelenést ad. Támogatja a segédfüggvényeket (helpers) és a részsablonokat, amik segítenek a sablon felépítésében és újrafelhasználhatóságában.

[1] Ebben a tanévben (2024/25) ezt fogjuk tanulni

9.3 Pug (korábban Jade)

Különleges és tömör, HTML-típusú kóddal dolgozik, ahol a behúzások számítanak, így kevesebb zárójelet használ. Egyszerű logikai műveletek, ciklusok és feltételek elérhetők benne, valamint részletes CSS és JavaScript támogatás.

9.4 Mustache

„Mustache" stílusú, {{ }} jelölés használatával, ami tiszta, de logikamentes sablonokat eredményez. Minimális logika, mivel a sablonmotor nem támogat ciklusokat vagy feltételeket – ezek JavaScript-ből jönnek. Egyszerű adatmegjelenítéshez használják, mivel a sablonok nem bonyolíthatók túlságosan.

9.5 Nunjucks

Használja a Django-hoz hasonló {% %} és {{ }} szintaxist. Kifejezetten erős sablonkezelés, sok beépített funkcióval (pl. szűrők, feltételek, ciklusok). Frontend és backend sablonoknál egyaránt jól működik, ahol szükségesek a komplexebb sablonstruktúrák.

Moduláris és támogat segédfüggvényeket, komplex feltételeket, ciklusokat, melyek nagyobb rugalmasságot nyújtanak.

Gyakorlat:

- **Sablonmotor beállítása és használata**: Készíts egy egyszerű sablont egy adott motorral, és adj vissza dinamikus tartalmat.
- Integrálj egy templating motort, például **Pug**-ot vagy **EJS**-t az alkalmazásba.
- Készíts egy dinamikus HTML oldalt, amely a szerverről kapott adatokat jeleníti meg *(pl. felhasználók listája*).

10 Routing (útvonalkezelés)

Routing: Tanuld meg, hogyan kezelheted a különböző útvonalakat (app.get(), app.post(), stb.) az Express-ben.

- **Dinamikus útvonalak**: Ismerd meg a dinamikus útvonalak használatát, például hogyan lehet paramétereket átadni az URL-ben (/users/:id).

Gyakorlati feladat:

- Készíts egy szervert, amely különböző útvonalakon különböző adatokat szolgáltat.
- Készíts egy egyszerű "felhasználó" (user) API-t, amelyben a felhasználók adatait egy tömbben tárolod (kezdetben az adatok lehetnek statikusak).
    - GET /users – adjon vissza egy listát az összes felhasználóról.

    - POST /users – fogadjon el egy új felhasználót JSON formátumban és adja hozzá a tömbhöz.
    - GET /users/:id – adja vissza az adott felhasználót a megadott id alapján.
    - PUT /users/:id – frissítse az adott felhasználót az id alapján.
    - DELETE /users/:id – törölje a felhasználót az id alapján.
- Implementálj egyszerű validációkat, például azt, hogy
    - a felhasználó neve kötelező legyen POST esetén.
    - születési dátuma két dátum közötti legyen (18 és 140 év közötti)
    - fizetése a havi minimálbérnél nagyobb legyen (teljes munkaidőben foglalkoztatott munkavállalónak 266 800 Ft, szakképzettséget igénylő munkakörben 326 000 Ft.)

11 Biztonság - CORS

A **CORS (Cross-Origin Resource Sharing)** egy olyan biztonsági mechanizmus, amelyet a böngészők alkalmaznak annak érdekében, hogy ellenőrizzék és szabályozzák a weboldalak közötti adatcserét, különösen akkor, amikor egy weboldal egy másik domain, protokoll vagy port erőforrásaihoz próbál hozzáférni.

11.1 Alapfogalmak

11.1.1 Mi az az "origin"?

- Egy weboldal **origin**-je az alábbi három részből áll:
    1. **Domain** (például: example.com)
    2. **Protokoll** (például: http:// vagy https://)
    3. **Port** (alapértelmezés szerint a 80-as port az HTTP-nél, és a 443-as port az HTTPS-nél)

Például a https://example.com:3000 URL originje:

- Protokoll: https
- Domain: example.com
- Port: 3000

11.1.2 Mi az a "cross-origin"?

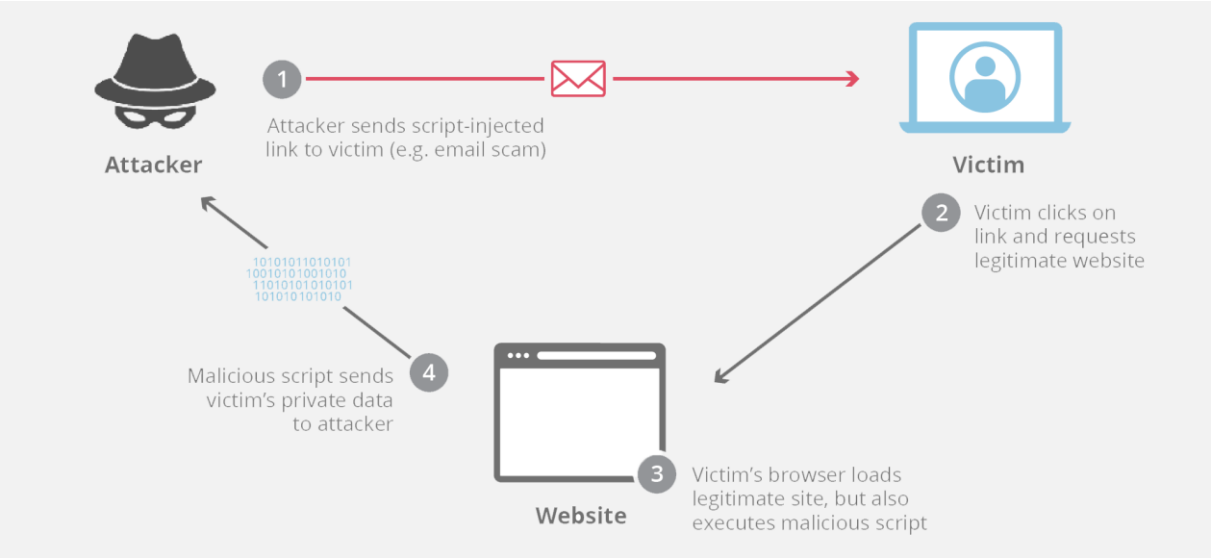
Egy "cross-origin" kérésről akkor beszélünk, ha egy weboldal egy másik originről (*domain-ről, portból vagy protokollból*) próbál betölteni adatokat. Például, ha a http://example.com weboldalról egy API-kérést küldünk a http://api.example.com címre, az már egy "cross-origin" kérés.

11.2 CORS szükségessége

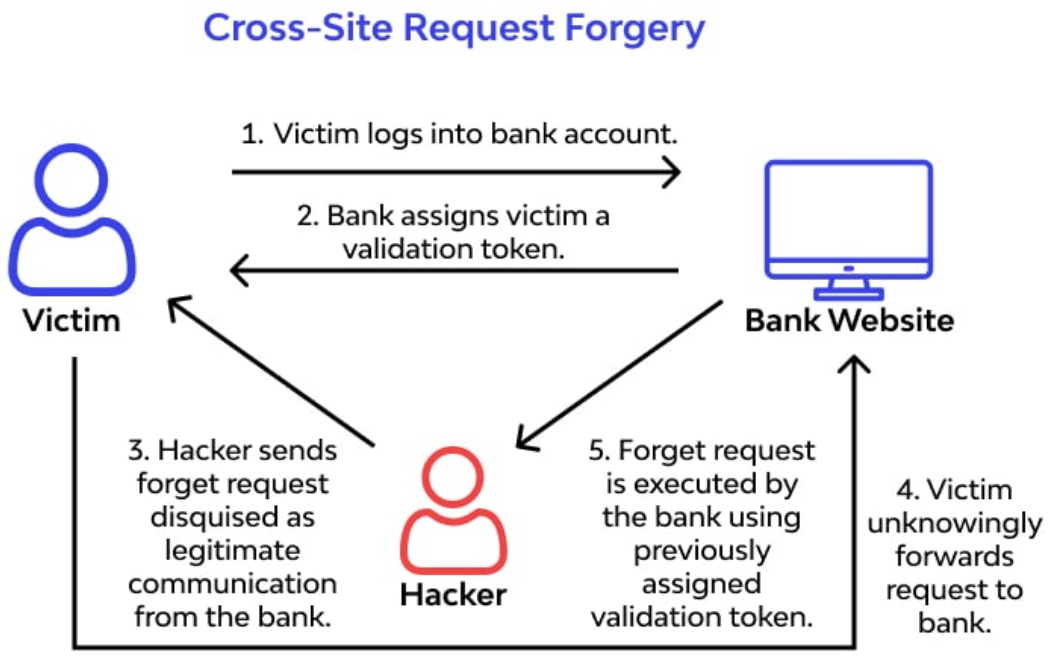
A böngészők alapvetően korlátozzák a "cross-origin" kéréseket, hogy megakadályozzák a **Cross-Site Scripting (XSS)** vagy **Cross-Site Request Forgery (CSRF)** típusú támadásokat. A CORS mechanizmus lehetővé teszi a szerverek számára, hogy megadják, melyik originről érkezhetnek biztonságosan kérések.

Mivel a fejlesztés során a szerver (*http://localhost:3000*) és kliens (*localhost:80* vagy *localhost:5500*) ugyanazon a gépen futnak, de különböző portokon, ezért a CORS problémája felmerülhet. Itt a "cross-origin" helyzet abból adódik, hogy a port számok eltérnek (3000 vagy 80 vagy 5500). A böngésző ilyenkor védi az adatokat azáltal, hogy megköveteli, hogy a szerver kifejezetten engedélyezze ezeket a kéréseket.

11.2.1 Cross-Site Scripting



11.2.2 Cross-Site Request Forgery



11.3 Hogyan működik a CORS?

Amikor egy böngésző egy cross-origin kérést próbál küldeni, a CORS mechanizmus a következő lépéseket hajtja végre:

11.3.1 Egyszerű kérés (*Simple Request*)

Egy kérés "egyszerű" (simple), ha az alábbi kritériumok mindegyike teljesül:

- **HTTP-módszerek** közül csak a GET, POST vagy HEAD van használatban.
- A kérésben csak alapvető HTTP-fejlécek vannak, mint például:
    - Accept
    - Content-Type (korlátozva text/plain, multipart/form-data, vagy application/x-www-form-urlencoded típusokra)
    - DPR, Width, Viewport-Width

Egy ilyen egyszerű kérés esetén a böngésző egyszerűen elküldi a kérést, és ha a szerver CORS fejlécekkel válaszol, akkor a böngésző ellenőrzi, hogy a válasz engedélyezett-e.

A szerver válaszában a következő fejléc szerepelhet, ami azt mondja a böngészőnek, hogy engedélyezett a kérés:

```
Access-Control-Allow-Origin: https://example.com
```

11.3.2 Előzetes ellenőrzés (*Preflight Requests*)

Ha a kérés nem "egyszerű", a böngésző egy előzetes ellenőrzést (preflight) hajt végre egy **OPTIONS** kérés küldésével, mielőtt a tényleges kérést elküldené. Ez az ellenőrzés megkérdezi a szervert, hogy az engedélyezi-e az adott típusú kérést.

Az előzetes ellenőrzés kérdésre a szerver válasza tartalmazza a következő fejléceket:

- **Access-Control-Allow-Methods**: Mely HTTP-módszereket engedélyezi a szerver (pl.: GET, POST, PUT stb.).
- **Access-Control-Allow-Headers**: Mely egyéni HTTP-fejléceket engedélyezi a szerver (pl.: Content-Type, Authorization).
- **Access-Control-Allow-Origin**: Az origin, ahonnan a kérés érkezhet (vagy * minden origin engedélyezésére).

Ha a szerver válasza megfelelő, a böngésző végrehajtja a tényleges kérést. Ha nem, a kérés elutasításra kerül.

Példa egy preflight kérésre:

```
OPTIONS /some/resource HTTP/1.1
Host: api.example.com
Origin: http://localhost:3000
Access-Control-Request-Method: POST
Access-Control-Request-Headers: Content-Type, apikey
```

És a szerver válasza:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://localhost:3000
Access-Control-Allow-Methods: GET, POST, OPTIONS
Access-Control-Allow-Headers: Content-Type, apikey
```

11.3.3 A tényleges kérés (Actual Request)

Ha az előzetes ellenőrzés sikeres, a böngésző elküldi a tényleges kérést az adatok lekéréséhez vagy módosításához.

11.4 Gyakori CORS fejlécek

- **Access-Control-Allow-Origin**: Meghatározza, hogy mely originről érkező kérések engedélyezettek. Ha minden origin számára engedélyezett, az értéke lehet *, de biztonsági okokból ez nem mindig ajánlott.
- **Access-Control-Allow-Methods**: A szerver által engedélyezett HTTP metódusok (pl. GET, POST, PUT, DELETE).
- **Access-Control-Allow-Headers**: Mely egyéni HTTP-fejléceket engedélyez a szerver.
- **Access-Control-Expose-Headers**: Fejlécek, amelyeket a kliens olvashat a válaszból (alapértelmezetten nem minden fejléc érhető el a böngésző számára).

- **Access-Control-Allow-Credentials**: Ha a szerver megköveteli, hogy a kérések tartalmazzanak hitelesítési adatokat (pl. sütiket), ez a fejlécnek true értéket kell tartalmaznia.

11.5 CORS gyakori problémák

- **Nincs megfelelő CORS fejléc**: Ha a szerver nem válaszol a megfelelő CORS fejléc nélkül, a böngésző automatikusan blokkolja a kérést.
- **Előzetes ellenőrzés hibája**: Ha a böngésző egy preflight kérést küld, de a szerver nem adja meg a megfelelő válaszfejléceket (*például hiányzik az Access-Control-Allow-Headers vagy Access-Control-Allow-Methods*), a kérés blokkolódik.
- **Wildcard * használata Access-Control-Allow-Origin-ben**: Ha hitelesítési adatokkal (*pl. sütik*) dolgozol, nem használhatsz * értéket az Access-Control-Allow-Origin fejlécben, mivel biztonsági okokból csak konkrét origin lehet megadva.

11.6 Hogyan oldhatod meg a CORS problémákat?

Több megoldás is lehetséges a böngészők beépített védelmének kikapcsolására az alkalmazásunk tesztelésénél

11.6.1 Szerver oldali változtatások

A CORS szabályok beállítása a szerveren történik. Győződj meg arról, hogy a szerver válaszai tartalmazzák a megfelelő CORS fejléceket.

11.6.1.1 CORS Middleware használata

```
const cors = require('cors');
app.use(cors({
   origin: ['http://localhost:80', 'http://localhost:5500'],  // Allowed origins
   methods: ['GET', 'POST', 'OPTIONS'],  // Allowed methods
   allowedHeaders: ['Content-Type']  // Allowed headers
}));
```

11.6.1.2 Kézi fejléc beállítása

```
app.use((req, res, next) => {
  res.header('Access-Control-Allow-Origin', 'http://localhost:80');
  res.header('Access-Control-Allow-Methods', 'GET, POST');
  res.header('Access-Control-Allow-Headers', 'Content-Type');
  next();
});
```

11.6.2 CORS proxy használata

Ingyenes CORS proxy szerverek:

- CorsProxy.io (https://corsproxy.io/)
- CORS.SH (https://proxy.cors.sh/)
- HTMLDriven (https://cors-proxy.htmldriven.com/)

- ...

11.6.3 Böngésző bővítmények

Kikapcsolják a CORS ellenőrzést, de ez csak fejlesztés alatt ajánlott.

12 Biztonság - input validáció

Az npm-el nagyon sok már sokak által használt ellenőrző könyvtárat vehetünk használatba. Ezek közül néhány:

- Yup
- Zod
- Joi
- Validator.js
- Ajv (Another JSON Validator)
- Class-validator
- Superstruct
- Vesta
- stb.

Mi a Yup-nak a használatával fogunk ismerkedni. (https://www.npmjs.com/package/express-yup-middleware és https://github.com/wgrisa/express-yup-middleware)

A rugalmasság, áttekinthetőség, könnyű javítás miatt célszerű az egyedekhez tartozó sémákat létrehozni és ezeket saját készítésű és „gyári" middleware-ek segítségével ellenőriztetni.

13 titkosítás.

Gyakorlat:

- Implementálj alapvető hibakezelést és biztonsági middleware-t (pl. Helmet).

14 Autentikáció és jogosultságkezelés

Hitelesítés: Tanuld meg, hogyan implementálhatsz hitelesítést (pl. JSON Web Token vagy session alapú hitelesítés).

- **Jogosultságok kezelése**: Kezeld a felhasználói jogosultságokat a különböző API végpontokon.

Gyakorlat:

- Hozz létre egy bejelentkezési rendszert, ahol felhasználók tokenek segítségével tudnak autentikálni.

- Készíts egy egyszerű bejelentkezési rendszert, ahol a felhasználók regisztrálhatnak és bejelentkezhetnek.
- Használj JWT-t (JSON Web Token) a hitelesítéshez és a védett útvonalakhoz.
- Védj le egyes útvonalakat, hogy csak bejelentkezett felhasználók férhessenek hozzájuk.
- Kezeld a session-öket az **express-session** csomag segítségével.

15 File feltöltés kezelése

Cél: Megismerni, hogyan kezelhetünk fájlokat Express alkalmazásban.

- Implementálj egy fájlfeltöltést, amely lehetővé teszi képek vagy dokumentumok feltöltését a szerverre a **multer middleware** segítségével.
- A feltöltött fájlokat tárold egy dedikált mappában, és jelenítsd meg őket egy oldalon.

16 Paginating és Sorting egy API-ban

Cél: Adatok lapozásának és rendezésének kezelése.

- Készíts egy útvonalat, ahol nagy mennyiségű adatot kell lapozni (pl. felhasználók listája), és adj hozzá lapozást (pagination).
- Add hozzá a rendezési (sorting) lehetőséget egy lekérdezési paraméter alapján (pl. ?sort=name).

17 Tesztelés és telepítés

- **Tesztelés**: Tanuld meg, hogyan tesztelheted az Express alkalmazásod automatikusan (pl. Mocha, Chai vagy Jest segítségével).
- **Telepítés**: Ismerkedj meg a telepítési folyamatokkal, pl. hogyan telepíthetsz alkalmazást Heroku-ra vagy más cloud platformra.

Gyakorlat:

- Írj teszteket az API végpontjaidhoz, majd telepítsd az alkalmazásodat egy felhőszolgáltatóra.

18 További források: