

# Webtechnológia és webalkalmazás-fejlesztés

## 2. hét – Express.js, Routing, Postman, TypeScript

Magda Donát

Széchenyi István Egyetem, Győr

[https://git.mdnd-it.cc/Donat/GKLB\\_INTM131.git](https://git.mdnd-it.cc/Donat/GKLB_INTM131.git)

2026. szeptember 8.



## Miről volt szó múlt héten?

Az első héten megismertük a webes kommunikáció alapjait: a HTTP protokollt, a Node.js futtatókörnyezetet, és elindítottunk egy egyszerű szervert a beépített `http` modullal. Emellett megtanultuk, hogyan kezeljük `npm`-mel a projekt függőségeit.

- **HTTP:** Request-Response modell, metódusok (GET, POST, PUT, PATCH, DELETE), státuszkódok
- **Node.js:** `http.createServer` – egy függvény, ami minden kérést kiszolgál
- **npm:** `package.json`, csomagok telepítése, `npm scripts`

### A hiányzó darab

A nyers `http` modullal minden egyes útvonalat, minden HTTP metódust és a kérés törzsének feldolgozását is kézzel, `if-else` ágakkal kellene megírunk. Ez pár végpontnál még kezelhető, de gyorsan átláthatatlanná válik.

## Mi a célja a mai órának?

A mai órán egy valódi keretrendszert vezetünk be, amivel a múlt heti kézi munkát nagyrészt kiváltjuk, majd megtanuljuk, hogyan teszteljük és hogyan típusozzuk a kódunkat.

- **Express.js** – a legelterjedtebb Node.js webes keretrendszer, ami elveszi a routing és a kérelmfeldolgozás nagy részének terhét
- **Routing** – hogyan szervezzük az útvonalainkat, hogyan olvassuk ki a kérés adatait
- **Postman** – hogyan teszteljük a saját API-nkat anélkül, hogy frontendet kellene hozzá írunk
- **TypeScript** – hogyan adjunk típusbiztonságot a JavaScript kódunkhoz

### Fontos

Ma még mindig egyszerű, tárolás nélküli (memóriában tartott) példákkal dolgozunk – az adatbázis-kapcsolat csak a 4. héten kerül képbe. Most a keretrendszerre és az eszközökre fókuszálunk.

## Miért nem elég a nyers http modul?

Emlékezzünk vissza az 1. héten írt szerverre: egyetlen callback-ben kellett eldöntenünk, milyen útvonalra és milyen metódussal érkezett a kérés.

```
const server = http.createServer((req, res) => {  
  if (req.url === '/users' && req.method === 'GET') {  
    // ...  
  } else if (req.url === '/users' && req.method === 'POST') {  
    // ...  
  } else if (req.url.startsWith('/users/') && req.method === 'GET') {  
    // egy adott felhasználó -- id kezzel kivagva az URL-ből  
  } // es így tovább...  
});
```

### A probléma

Tíz-húsz végpontnál ez a lánc kezelhetetlenné válik: nincs beépített segítség a paraméterek kiolvasásához, a JSON törzs feldolgozásához, vagy az útvonalak csoportosításához.

# Mi az Express.js?

Az **Express.js** a legelterjedtebb Node.js webes keretrendszer (web framework): a nyers `http` modulra épül, és eltünteti a kézi `if-else` láncokat egy sokkal kényelmesebb, deklaratív routing API-val.

- Egyszerű, minimalista – nem kényszerít ránk egy előírt projektszerkezetet
- Beépített segítség: útvonal-paraméterek, query stringek, JSON válaszok
- Hatalmas ökoszisztéma: middleware-ek szinte minden problémára (autentikáció, naplózás, CORS, stb.)
- Ez lesz a tárgy hátralévő részének elsődleges keretrendszere

## Fontos

Az Express *nem* helyettesíti a Node.js-t – ráépül. Amit az 1. héten a `http` modulról tanultunk (request/response ciklus), az Express alatt is pontosan úgy működik.

# Az első Express alkalmazás

Néhány sorból már működő szerverünk van – hasonlítsd össze az előző dia if-else láncával!

## Telepítés

```
npm install express
```

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

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

app.listen(3000, () => {
  console.log('Szerver fut: http://localhost:3000');
});
```

## A middleware fogalma

Az Express legfontosabb koncepciója a **middleware**: egy függvény, ami a kérés és a válasz között fut le, és vagy továbbbengedi a folyamatot (`next()`), vagy lezárja azt egy válasszal.

```
app.use((req, res, next) => {  
  console.log(`${req.method} ${req.url}`);  
  next(); // tovább a kovetkezo middleware-hez/route-hoz  
});
```

```
app.use(express.json()); // JSON body-t olvaso middleware
```

- A kérések egy "csövön" (pipeline) haladnak végig, middleware-eken keresztül
- `express.json()` nélkül a `req.body` üres lenne POST/PUT kéréseknél
- Naplózás, autentikáció, hibakezelés – mind middleware-ként valósul meg

# Összefoglalás – Express.js

## Express.js alapok

- Node.js-re épülő, minimalista webes keretrendszer
- Kiváltja a kézi `if-else` útvonal-elágazásokat
- `app.listen()` – szerver indítása, ugyanaz mint a nyers `http`-nél
- Middleware: `app.use()` – a kérés-válasz "csövének" építőeleme
- `express.json()` – beépített middleware a JSON törzs feldolgozásához

# HTTP metódusok Express-ben

Minden HTTP metódushoz külön Express függvény tartozik, ami pontosan a metódus nevét viseli – ez sokkal olvashatóbb, mint egy `if (req.method === ...)` lánc.

```
app.get('/users', (req, res) => { /* lista */ });  
app.post('/users', (req, res) => { /* létrehozás */ });  
app.get('/users/:id', (req, res) => { /* egy elem */ });  
app.put('/users/:id', (req, res) => { /* teljes frissítés */ });  
app.patch('/users/:id', (req, res) => { /* részleges frissítés */ });  
app.delete('/users/:id', (req, res) => { /* törlés */ });
```

## Emlékeztető

Ez pontosan a 2. héten (a jövő héten, amikor visszatérünk rá) tárgyalt CRUD → HTTP metódus táblázatot követi – nem véletlen az egyezés.

## Route paraméterek és query stringek

Az útvonalban lévő `:id`-hez hasonló szegmenseket **route paramétereknek** hívjuk, a kérdőjel utáni `?key=value` részt pedig **query stringnek**. Express mindkettőt készen, feldolgozva adja át.

```
// GET /users/123?includeOrders=true
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;           // "123"
  const includeOrders = req.query.includeOrders; // "true"
  res.json({ id: userId, includeOrders });
});
```

- `req.params`: az útvonal része, egy konkrét erőforrás azonosítására (`/users/:id`)
- `req.query`: opcionális finomítás (lapozás, szűrés, rendezés)

## A kérés törzse: req.body

POST, PUT és PATCH kéréseknél az adat jellemzően a kérés törzsében (body) érkezik JSON formátumban – ehhez szükséges az előző órán megismert `express.json()` middleware.

```
app.use(express.json());

app.post('/users', (req, res) => {
  const { name, email } = req.body;
  const newUser = { id: Date.now(), name, email };
  users.push(newUser);
  res.status(201).json(newUser);
});
```

### Emlékeztető az 1. hétről

A 201 Created a helyes státuszkód sikeres POST esetén – ezt már az 1. héten megismertük.

## Útvonalak szervezése: `express.Router()`

Ahogy nő az alkalmazás, egyetlen fájlban tartani az összes route-ot áttekinthetetlenné válik. Az `express.Router()` lehetővé teszi, hogy erőforrásonként külön fájlba szervezzük az útvonalakat.

### routes/users.js

```
const router = express.Router();

router.get('/', listUsers);
router.post('/', createUser);

module.exports = router;
```

### app.js

```
const usersRouter =
  require('./routes/users');

app.use(
  '/users',
  usersRouter
);
```

# Hibakezelés alapjai

Express két speciális esetet kezel middleware-ekkel: amikor egy útvonal nem létezik, és amikor egy route handler hibát dob.

```
// 404 - ha semelyik route nem illeszkedett
app.use((req, res) => {
  res.status(404).json({ error: 'Nem található' });
});

// hibakezelo middleware -- negy parameter jelzi Expressnek!
app.use((err, req, res, next) => {
  console.error(err);
  res.status(500).json({ error: 'Szerver hiba' });
});
```

## Fontos

A hibakezelő middleware-t *a legvégén* kell regisztrálni, az összes route után – Express a négy paraméter alapján ismeri fel.

# Összefoglalás – Routing

## Routing Express-ben

- `app.get/post/put/patch/delete` – metódusonként külön, olvasható route definíció
- `req.params` – útvonal paraméterek, `req.query` – query string, `req.body` – kérés törzse
- `express.Router()` – útvonalak szervezése külön fájlokba, erőforrásonként
- 404 és hibakezelő middleware a route-ok végén

## Mi a Postman, és miért használjuk?

Eddig minden végpontunkat vagy böngészőből (csak GET-hez alkalmas), vagy egyáltalán nem próbáltuk ki. A **Postman** egy grafikus alkalmazás, amivel bármilyen HTTP kérést összeállíthatunk és elküldhetünk – anélkül, hogy előbb frontendet kellene írunk hozzá.

- Bármilyen metódus (GET, POST, PUT, PATCH, DELETE) kipróbálható
- Egyedi fejlécek (headers) és JSON törzs (body) is összeállítható
- A választ (státuszkód, headerek, body) azonnal, formázva látjuk
- Elmenthető, újra futtatható, csapattagokkal megosztható kérésorozatok

### Miért nem elég a böngésző?

A böngésző címsorába írt URL mindig GET kérést küld – POST, PUT, DELETE kéréshez, vagy egyedi fejlécekhez (pl. `Authorization`) külön eszköz kell.

## Egy kérés összeállítása

Egy Postman-kérés pontosan azokból az elemekből áll, amiket az 1. héten a HTTP kérés felépítéseként már megismertünk: metódus, URL, fejlécek, törzs.

### Amit beállítunk

- **Metódus:** pl. POST
- **URL:** `http://localhost:3000/users`
- **Headers:** `Content-Type: application/json`
- **Body (raw, JSON):**

```
{  
  "name": "Kis Anna",  
  "email": "anna@example.com"  
}
```

Küldés után azonnal látjuk a választ: 201 Created és a visszakapott, azonosítóval kiegészített felhasználó.

## Collections és Environment változók

Ahogy egyre több végpontot tesztlünk, hasznos ezeket rendszerezni, és elkerülni, hogy mindig kézzel írjuk be az alap URL-t.

- **Collection:** kérések egy mappába rendezett gyűjteménye – pl. minden "Users" végpont egy collection-ben
- **Environment:** névhez kötött változók (pl. `baseUrl`), amikkel a kéréseinkben `{{baseUrl}}/users` írható a teljes URL helyett
- Így egyetlen változó átírásával válthatunk fejlesztői és éles környezet között

### Gyakorlati tipp

Fejlesztés közben minden újonnan megírt végpontot azonnal próbáld ki Postmanben – ez sokkal gyorsabb visszajelzést ad, mint egy teljes frontend felület megvárása.

# Összefoglalás – Postman

## Postman

- Grafikus eszköz tetszőleges HTTP kérés összeállítására és elküldésére
- Kiválthatja a böngészőt olyan esetekben, amikor nem elég a sima GET
- Kérés = metódus + URL + headers + body – ugyanaz, mint amit HTTP-ből már tudunk
- Collections: kérések rendszerezése; Environments: újrafelhasználható változók

# Miért TypeScript?

A JavaScript **gyengén típusos** nyelv: egy változó bármikor bármilyen típusú értéket felvehet, és a hibák jó része csak futás közben derül ki – gyakran éppen egy éles felhasználónál. A **TypeScript** a JavaScript egy kiterjesztése, ami *statikus típusokat* ad hozzá.

- A típushibák nagy része már fordításkor (kódírás közben) kiderül, nem futás közben
- Az IDE sokkal pontosabb kódkiegészítést és hibajelzést tud adni
- Nagyobb csapatban, nagyobb kódbázisban különösen megtérül
- A TypeScript kód JavaScript-té fordul le – a böngésző/Node.js sosem futtat TS-t közvetlenül

## Fontos

A TypeScript *opcionális* réteg a JavaScript fölött – minden érvényes JavaScript kód (majdnem) érvényes TypeScript is, fokozatosan vezethető be egy meglévő projektbe.

## Telepítés és futtatás

A TypeScript fájlok (.ts kiterjesztéssel) nem futtathatók közvetlenül Node.js-szel – előbb JavaScript-té kell őket fordítani, vagy egy erre képes futtatót kell használnunk.

### Fordítás tsc-vel

```
npm install -D typescript
npx tsc --init          # létrehozza a tsconfig.json-t
npx tsc                 # lefordít minden .ts fájlt .js-re
```

### Fejlesztés közben: tsx

```
npx tsx index.ts
```

A tsx fordítás nélkül, közvetlenül futtatja a TypeScript fájlt – fejlesztés közben ez a kényelmesebb.

## Alaptípusok és egy egyszerű interfész

A típusokat a változó/paraméter neve után, kettősponttal adjuk meg. Több mező közös leírásához pedig egy interface-t érdemes definiálni.

```
let name: string = 'Anna';
let age: number = 25;
let isActive: boolean = true;
let tags: string[] = ['admin', 'user'];
```

```
interface User {
  id: number;
  name: string;
  email: string;
}
```

```
function greet(user: User): string {
  return 'Hello, ${user.name}!';
}
```

# Összefoglalás – TypeScript

## TypeScript alapok

- A JavaScript statikus típusokkal kiegészített változata – fordításkor szűri a típushibákat
- `tsc` – fordítás JavaScript-té; `tsx/ts-node` – közvetlen futtatás fejlesztéskor
- Alaptípusok: `string`, `number`, `boolean`, tömbök
- `interface` – több mezőből álló adatszerkezet típusának leírása

## Több is van!

Ez csak az alapok – a generikus típusokról, a `union/intersection` típusokról és más haladó fogalmakról egy *külön, önállóan is végigkövethető* kiegészítő anyagban olvashatsz, ha mélyebben el szeretnél merülni.

## Következő lépések

Ma megírtuk az első valódi Express.js alkalmazásunkat, megtanultuk, hogyan teszteljük Postmannal, és belekóstoltunk a TypeScriptbe. Eddig azonban inkább *gyakorlati megérzés* alapján dolgoztunk – a jövő héten ezt tudatos tervezéssé alakítjuk.

- 1 **REST API tervezési elvek** – visszatekintünk a ma írt útvonalainkra: jól neveztük-e el őket?
- 2 **Architekturális felépítések** – hogyan válasszuk szét a route handlert (amit ma írtunk) az üzleti logikától
- 3 **CQRS** – mikor éri meg külön kezelni az írási és olvasási műveleteket

### Javaslat

Nézd át még egyszer a ma írt route-jaidat – a jövő héten pontosan ezekre a konkrét példákra fogunk tervezési elveket alkalmazni.