

Webtechnológia és webalkalmazás-fejlesztés

1. hét – Alapok, Node.js, HTTPS, NPM Package Manager

Magda Donát

Széchenyi István Egyetem, Győr

https://git.mdnd-it.cc/Donat/GKLB_INTM131.git

2026. szeptember 7.

Oktató

Üdvözlünk a *Webtechnológia és webalkalmazás-fejlesztés* tárgy első óráján! Ezt a diasort úgy állítottuk össze, hogy akkor is jól követhető legyen, ha valaki lemaradt az óráról – minden fogalmat igyekszünk a lehető legrészletesebben, magyarázó szöveggel is ellátni.

Elérhetőségek

Magda Donát, mérnökinformatikus BSc

- **E-mail:** magda.donat@sze.hu – bármilyen kérdéssel, technikai problémával nyugodtan keress meg
- **Konzultáció:** előre egyeztetve, Google Meet – ha személyesen szeretnéd átbeszélni a haladásodat

Tantárgy célja

A tárgy célja, hogy végigjárjuk egy modern, Javascript/Node.js alapú backend rendszer felépítésének teljes útját: a nyers HTTP-kérések kezelésétől egészen egy biztonságos, adatbázissal és külső szolgáltatásokkal kommunikáló API-ig.

- Javascript alapú backend fejlesztés elsajátítása – ugyanazt a nyelvet használjuk, mint amit sokan már a frontendről ismernek
- HTTP/HTTPS alapok, REST API-k és architektúrais felépítések megértése
- Adatbázis-kezelés, ORM (Object-Relational Mapping) és a CQRS (Command Query Responsibility Segregation) megközelítés
- Authentikáció, autorizáció és külső szolgáltatások (pl. email küldés, cache) integrálása
- Gyakorlati tapasztalat egy önállóan futtatható backend rendszer elkészítésével

Cél

Az alapoktól (HTTP, Node.js, csomagkezelés) egy teljes, biztonságos backend API-ig – lépcsőről lépésre, minden héten egy újabb réteggel bővítve a tudásunkat.

Tematika – 1-6. hét

Az első hat hét az alapozásról szól: megismerjük a HTTP protokollt, felépítünk egy REST API-t, és megtanuljuk, hogyan lehet azt egy adatbázissal összekötni.

- 1. hét** Alapok, Node.js, HTTPS, NPM package manager
- 2. hét** REST API, architekturális felépítések, CQRS
- 3. hét** Express.js, Routing, Postman, TypeScript
- 4. hét** ORM, database connection, CRUD operations
- 5. hét** TypeORM, Aggregates, Repositories
- 6. hét** ZH 1: Database CRUD, ORM, CQRS + DTOs, Mappers

Tematika – 6-13. hét

A második félidőben a biztonságra és a valós, éles rendszerekben elengedhetetlen kiegészítőkre fókuszálunk: bejelentkezés, jogosultságkezelés, emailküldés, teljesítmény.

- | | |
|----------------|---|
| 7. hét | AUTH, OAUTH, OAUTH2 |
| 8. hét | Session, JWT, Cookie |
| 9. hét | Hash, Middleware, Services |
| 10. hét | Email sending, Templating |
| 11. hét | Dependency Injection, Logging |
| 12. hét | CORS, Redis, External API |
| 13. hét | Gyakorlás – teljes API írása (NAGY ZH) |

Teljesítés

A tárgyat két zárthelyi dolgozat (ZH) alapján teljesítjük. A pontszámok összeadódnak, és az alábbi táblázat szerint alakul belőlük az érdemjegy:

Pontszámítás

Jegy	Pont
1	0–50
2	51–62
3	63–74
4	75–86
5	87–100

Megajánlott jegy

Megajánlott jegy ZH pontokból szerezhető, mely a vizsga pontozásával megegyezően kerül értékelésre – vagyis ha a féléves munkád elég jó, a vizsgaidőszakban nem kell feltétlenül vizsgáznod.

Mi a HTTP?

Amikor a böngésződ betölt egy weboldalt, vagy amikor egy mobilalkalmazás adatot kér egy szervertől, a háttérben szinte mindig a **HTTP** (HyperText Transfer Protocol) protokoll zajlik. Ez egy egyszerű, szövegalapú szabályrendszer arra, hogyan kérjen adatot egy kliens (pl. böngésző) egy szervertől, és hogyan válaszoljon arra a szerver.

- Kliens-szerver kommunikáció alapja a weben
- **Állapotmentes (stateless)** protokoll: a szerver alapból nem emlékszik az előző kérésekre – minden kérés önmagában, a többitől függetlenül értelmezhető
- **Request-Response** modell: a kliens kér, a szerver válaszol
- Alapértelmezett port: 80 (HTTP), 443 (HTTPS)

HTTP verziók

A HTTP protokoll az évek során több jelentős verzióváltáson esett át, mindig azzal a céllal, hogy gyorsabbá és hatékonyabbá tegye a weben zajló kommunikációt. Ma leggyakrabban HTTP/1.1-gyel és HTTP/2-vel találkozunk, de egyre több nagy szolgáltatás áll át a legújabb HTTP/3-ra is.

- **HTTP/1.1** (1997) – perzisztens kapcsolat, azaz nem kell minden kéréshez új TCP-kapcsolatot nyitni
- **HTTP/2** (2015) – multiplexing (több kérés egyszerre, egy kapcsolaton), bináris protokoll
- **HTTP/3** (2022) – a QUIC protokollra épül, UDP alapú, még gyorsabb kapcsolatfelépítés

HTTP vs HTTPS

A sima HTTP-n keresztül küldött adatokat bárki, aki hozzáfér a hálózati forgalomhoz (pl. egy nyílt wifin), el tudja olvasni, sőt módosítani is. A **HTTPS** ezt a problémát oldja meg azzal, hogy a HTTP forgalmat egy titkosítási réteg (SSL/TLS) fölé csomagolja.

HTTP

- Titkosítatlan kommunikáció
- Lehallgatható, módosítható (Man-in-the-Middle támadás)
- Port 80

HTTPS = HTTP + SSL/TLS

- Titkosított kommunikáció
- SSL/TLS tanúsítvány szükséges hozzá
- Port 443

Miért kötelező ma a HTTPS?

Régebben csak a bejelentkezős, kényes adatokat kezelő oldalak használtak HTTPS-t, ma viszont gyakorlatilag minden weboldalnak ezt kellene tennie – több okból is.

Fontos

Modern webalkalmazásoknál a HTTPS gyakorlatilag kötelező: a böngészők figyelmeztetést adnak a sima HTTP oldalakra, a keresőmotorok (SEO) hátrébb sorolják őket, és sok modern böngésző-funkció (pl. helymeghatározás, kamera) is csak HTTPS mellett engedélyezett.

- **Bizalom:** a felhasználó látja a lakat ikont, tudja, hogy az adatai védve utaznak
- **Integritás:** útközben senki nem tudja módosítani a forgalmat (pl. reklámot beszúrni)
- **Hitelesség:** a tanúsítvány igazolja, hogy tényleg azzal a szerverrel beszélünk, akivel hisszük

HTTP metódusok

Minden HTTP kérésnek van egy **metódusa**, ami megmondja, milyen műveletet szeretnénk végrehajtani az adott erőforráson. A REST API-k tervezésénél ez a metódusválasztás lesz az egyik legfontosabb döntésünk.

Metódus	Jelentés
GET	Erőforrás lekérése (csak olvasás, idempotens)
POST	Új erőforrás létrehozása (nem idempotens)
PUT	Erőforrás teljes frissítése (idempotens)
PATCH	Erőforrás részleges frissítése (nem idempotens)
DELETE	Erőforrás törlése (idempotens)
HEAD	Mint GET, de csak fejlécek, body nélkül
OPTIONS	Támogatott metódusok lekérése (CORS)

Idempotencia – miért számít?

Egy művelet **idempotens**, ha többszöri végrehajtása ugyanazt az eredményt adja, mint az egyszeri. Ennek nagy jelentősége van a gyakorlatban: ha egy kérés elakad a hálózaton és a kliens újraküldi, idempotens metódusnál (GET, PUT, DELETE) nem történik semmi baj – a végállapot ugyanaz marad.

Példa

Ha kétszer küldöd el ugyanazt a DELETE /api/users/123 kérést, a felhasználó csak egyszer törlődik – a második kérés egyszerűen nem talál már semmit, amit törölnie kellene.

POST nem idempotens

Ha egy POST-tal új erőforrást hozunk létre, és véletlenül kétszer küldjük el (pl. dupla kattintás egy "Rendelés leadása" gombon), két különálló erőforrás (pl. két rendelés) jöhet létre!

HTTP Request felépítése

Egy HTTP kérés mindig ugyanabból a három részből épül fel: egy sorból, ami leírja, mit kérünk, néhány fejlécből, és opcionálisan egy törzsből (body).

Request struktúra

```
GET /api/users/123 HTTP/1.1
Host: example.com
Accept: application/json
Authorization: Bearer token123
```

- **Request Line:** metódus (GET), útvonal (/api/users/123), protokoll verzió
- **Headers:** metaadatok kulcs-érték párok formájában (pl. milyen formátumot fogadunk el, kik vagyunk)
- **Body:** opcionális adatok, jellemzően POST, PUT, PATCH kéréseknél (pl. egy új felhasználó adatai JSON-ban)

HTTP Response felépítése

A szerver válasza (response) hasonló szerkezetű, csak itt a kérés helyett egy **státuszkód** mondja meg, hogy sikerült-e a kérés.

Response struktúra

```
HTTP/1.1 200 OK  
Content-Type: application/json  
  
{"id": 123, "name": "Alice"}
```

- **Status Line:** protokoll, státuszkód (200), státusz szöveg (OK)
- **Headers:** metaadatok (pl. milyen formátumú a válasz törzse)
- **Body:** a válasz tényleges tartalma (HTML, JSON, XML, stb.)

HTTP státuszkódok

A státuszkód egy háromjegyű szám, aminek az *első számjegye* határozza meg a kategóriáját. Nem kell mind a száznál is több kódot fejből tudni – elég, ha felismered, melyik kategóriába tartozik egy adott kód.

- **1xx – Informational:** ritkán találkozunk vele, ideiglenes válasz (pl. 101 Switching Protocols)
- **2xx – Success:** a kérés sikeres volt (200 OK, 201 Created, 204 No Content)
- **3xx – Redirection:** a kliensnek egy másik URL-re kell mennie (301 Moved Permanently, 302 Found, 304 Not Modified)
- **4xx – Client Error:** a kliens hibázott (400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 429 Too Many Requests)
- **5xx – Server Error:** a szerver oldalán történt hiba, a kliens kérése önmagában helyes volt (500 Internal Server Error, 502 Bad Gateway, 503 Service Unavailable)

REST API-ban gyakori státuszkódok

Egy jól megtervezett REST API mindig következetesen használja a státuszkódokat – ez az egyik legfontosabb szokás, amit már most érdemes elsajátítani, hiszen a következő heteken folyamatosan ezekkel fogunk dolgozni.

Metódus	Sikeres	Hiba
GET	200 OK	404 Not Found
POST	201 Created	400, 409 Conflict
PUT	200 OK / 204 No Content	400, 404 Not Found
PATCH	200 OK	400, 404 Not Found
DELETE	204 No Content	404 Not Found

Két gyakori, metódusfüggetlen eset

Autentikáció: 401 Unauthorized (nincs bejelentkezve), 403 Forbidden (be van jelentkezve, de nincs jogosultsága)

Szerver hiba: 500 Internal Server Error, 503 Service Unavailable

HTTPS – SSL/TLS Handshake

Mielőtt egy HTTPS kapcsolaton bármilyen adat átmenne, a kliens és a szerver egy rövid "kézfogáson" (handshake) esik át, aminek célja, hogy közösen kialakítsanak egy titkos kulcsot, amit csak ők ketten ismernek.

- 1 **Client Hello** – a kliens elküldi, milyen titkosítási módokat támogat
- 2 **Server Hello** – a szerver kiválasztja a közösen támogatott titkosítást, és elküldi a tanúsítványát
- 3 **Certificate Verify** – a kliens ellenőrzi a tanúsítványt egy megbízható hitelesítésszolgáltatónál (CA)
- 4 **Key Exchange** – létrejön egy közös, szimmetrikus titkosító kulcs
- 5 **Finished** – innentől minden adat ezzel a kulccsal titkosítva utazik

SSL/TLS verziók

A TLS 1.2 még sok helyen használatban van, de a TLS 1.3 az ajánlott: gyorsabb kapcsolatfelépítés és erősebb, biztonságosabb titkosítási módok.

Összefoglalás – HTTP/HTTPS

HTTP/HTTPS alapok

- HTTP: Request-Response modell, állapotmentes (stateless) protokoll
- HTTPS: titkosított HTTP, SSL/TLS handshake-kel
- HTTP metódusok: GET, POST, PUT, PATCH, DELETE – mindegyiknek megvan a maga szerepe
- Idempotencia: GET, PUT, DELETE biztonságosan újraküldhető
- Státuszkódok: az első számjegy adja meg a kategóriát (1xx–5xx)
- REST API-ban leggyakrabban használt kódok: 200, 201, 204, 400, 401, 403, 404, 500

Mi a Node.js?

A Javascript eredetileg kizárólag böngészőkben futott. A **Node.js** ezt változtatta meg: kiszedte a Google Chrome V8 motorját a böngészőből, és lehetővé tette, hogy Javascript kód a szerveren, önállóan is fusson – így írhatunk vele backend alkalmazásokat, parancssori eszközöket, akár build-rendszereket is.

- JavaScript futtatókörnyezet a V8 motoron alapulva
- Szerveroldali JavaScript fejlesztés lehetősége
- Eseményvezérelt, nem blokkoló (non-blocking) I/O modell
- Létrehozta: Ryan Dahl (2009)

Miért Node.js?

Ugyanazt a nyelvet használhatjuk frontend és backend oldalon is, hatalmas npm ökoszisztéma áll rendelkezésre szinte bármilyen problémára, és gyors vele a fejlesztés, prototípuskészítés.

Node.js architektúra

A Node.js két fő komponensre épül: a Javascript kódot lefordító és futtató **V8 motorra**, valamint a háttérben futó **libuv** könyvtárra, ami az aszinkron I/O műveleteket (fájl, hálózat) kezeli.

V8 JavaScript Engine

- Google Chrome motorja
- JIT (Just-In-Time) fordítás
- C++ nyelven íródott, rendkívül gyors

libuv

- Event loop implementáció
- Aszinkron I/O műveletek
- Cross-platform támogatás

Event Loop

Timers → Pending callbacks → Poll (I/O) → Check → Close callbacks. A Node.js egyetlen szálon (single-threaded) fut, de az I/O műveletek aszinkron módon, a háttérben zajlanak – így egy lassú fájlművelet nem blokkolja a többi kérés kiszolgálását.

Első Node.js program

Nézzünk egy egyszerű, de tanulságos első programot, ami megmutatja, milyen beépített, Node.js-specifikus változókhoz férünk hozzá.

hello.js – futtatás: `node hello.js`

```
console.log('Hello, Node.js!');  
console.log('Node verzio:', process.version);  
console.log('Aktualis könyvtar:', __dirname);
```

Fontos

A Node.js nem böngésző! Nincs `window`, nincs `document`, de van helyette `global` (a böngésző `window`-jának megfelelője) és `process` (a futó Node.js folyamatról ad információt).

Modulrendszer – CommonJS

Ahogy a kódunk növekszik, elkerülhetetlen, hogy több fájlra bontsuk. A Node.js hagyományos, alapértelmezett modulrendszere a **CommonJS**, ahol a `require` és a `module.exports` párossal osztunk meg kódot fájlok között.

math.js – export

```
function add(a, b) {  
  return a + b;  
}  
module.exports = { add };
```

app.js – import

```
const math = require('./math');  
console.log(math.add(2, 3));  
// 5
```

Beépített (core) modulok

A Node.js számos hasznos modult ad eleve a kezünkbe, telepítés nélkül: `fs` (file system, fájlkezelés), `path` (útvonalak kezelése), `http` (webszerver készítése).

ES Modules (ESM)

A modern Javascript szabvány, az ECMAScript saját, egységes modulrendszer vezetett be `import/export` kulcsszavakkal – ugyanezt ismerheted a frontend keretrendszerekből (React, Vue) is. A Node.js is támogatja ezt, ha bekapcsoljuk.

math.mjs – export

```
export function add(a, b) {  
  return a + b;  
}
```

app.mjs – import

```
import { add } from './math.mjs';  
console.log(add(2, 3));
```

package.json

Az ESM használatához be kell állítani: `{"type": "module"}` a `package.json` fájlban – ekkor a `.js` kiterjesztésű fájlok is ES modulként fognak viselkedni.

File System műveletek

A backend fejlesztés egyik leggyakoribb feladata fájlokkal dolgozni (konfiguráció olvasása, log írás, feltöltött fájlok kezelése). A Node.js fs modulja ehhez háromféle API-t is kínál.

Fájl olvasás – Promise API (ajánlott)

```
const fs = require('fs').promises;

async function readData() {
  const data = await fs.readFile('input.txt', 'utf8');
  console.log(data);
}
```

- `readFileSync` – szinkron, blokkolja a végrehajtást, amíg be nem fejeződik (kerülendő szerveroldalon)
- `readFile` callback-kel – aszinkron, a régebbi, callback-alapú stílus
- `fs.promises` – aszinkron, `async/await`-tel szépen, szinkron kódként olvasható – ma ez az ajánlott megoldás

HTTP szerver létrehozása

Ahogy korábban láttuk, a HTTP request-response modellt bármilyen nyelven meg lehet valósítani. Node.js-ben a beépített `http` modullal mindössze néhány sorból már működő webszervert tudunk indítani – ez az alapja mindannak, amit a következő heteken Express.js-szel fogunk csinálni.

Egyszerű HTTP szerver

```
const http = require('http');
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, Node.js szerver!\n');
});
server.listen(3000, () => {
  console.log('Szerver fut: http://localhost:3000');
});
```

Node.js vs. Böngésző

Fontos tudatosítani, hogy bár mindkét környezetben Javascript kód fut, a Node.js és a böngésző *teljesen más* környezetet biztosítanak – más API-k érhetők el, és más a rendeltetésük is.

Jellemző	Node.js	Böngésző
DOM API	Nincs	Van
window objektum	Nincs	Van
Globális objektum	global	window
File system	Van (fs)	Nincs
HTTP szerver	Készíthető	Nem
Használat	Backend, CLI	Frontend

Összefoglalás – Node.js

Node.js alapok

- V8 engine alapú JavaScript futtatókörnyezet, szerveroldalon
- Event-driven, non-blocking I/O modell – egy szálon, de aszinkron módon
- Modulrendszer: CommonJS (`require`) és ES Modules (`import`)
- Beépített modulok: `fs` (fájlok), `path` (útvonalak), `http` (szerver)
- HTTP szerver készítése pár sorban – ez az alapja a jövő heti Express.js-nek

Mi az npm?

Amint elkezdünk egy valós projekten dolgozni, hamar szükségünk lesz mások által megírt kódra – ne írjunk újra mindent nulláról! Az **npm** (Node Package Manager) ebben segít: ez a Node.js hivatalos csomagkezelője, amivel telepíthetünk, frissíthetünk és megoszthatunk kódcsomagokat.

- Node Package Manager – Node.js csomagkezelője
- A világ legnagyobb szoftver registry-je (2+ millió csomag)
- Node.js-szel együtt települ, külön telepíteni nem kell
- Projekt függőségek kezelése, verziószám ellenőrzése

Verziószám ellenőrzése

```
node --version  
npm --version
```

npm alternatívái

Bár a tárgy során mindvégig az npm-et fogjuk használni, érdemes tudni, hogy nem ez az egyetlen csomagkezelő a Node.js ökoszisztémában – a szakmában több helyen találkozatsz az alternatíváival is.

Alternatívák

yarn és **pnpm** – ugyanazt a célt szolgálják, mint az npm, csak némileg gyorsabbak és hatékonyabbak lehetnek nagyobb projekteknél. A **pnpm** például egyetlen, megosztott tárolóban tartja a csomagokat, így lemezterületet spórol.

A parancsok szinte megegyeznek egymással (`npm install` ↔ `yarn add` ↔ `pnpm add`), így ha megtanulod az egyiket, a többit is könnyen felveszed majd.

npm init – Projekt inicializálás

Minden Node.js projekt egy `package.json` fájlal kezdődik, ami leírja a projekt metaadatait (nevét, verzióját, függőségeit). Ezt a fájlt az `npm init` parancs generálja le nekünk.

Interaktív projekt létrehozás

```
npm init
```

Ekkor az npm sorra megkérdezi a projekt nevét, verzióját, leírását, stb.

Gyors inicializálás alapértelmezett értékekkel

```
npm init -y
```

npm init – mit hoz létre?

Mindkét változat ugyanazt az egy fájlt hozza létre – csak abban különböznek, hogy megkérdezik-e tőlünk előtte a projekt adatait, vagy rögtön alapértelmezett értékekkel dolgoznak.

Eredmény

Létrejön a `package.json` fájl, amely a projekt metaadatait tartalmazza, és ami a projektünk "személyi igazolványaként" működik: ebben tároljuk a nevét, verzióját, a függőségeit és a saját parancsainkat is.

Ha egy projektet klónozol githubról, az első dolgod mindig az legyen, hogy megnézed a `package.json`-t – ebből azonnal kiderül, milyen technológiákat használ, és hogyan indítható.

package.json struktúra

Érdemes megismerkedni a `package.json` legfontosabb mezőivel, hiszen ez a fájl fogja a félév során minden projektünket összefogni.

```
{
  "name": "backend-app",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js"
  },
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "nodemon": "^3.0.1"
  }
}
```

package.json mezői

A `scripts` mező alatt saját parancsainkat definiálhatjuk, a `dependencies` és `devDependencies` pedig a projekt függőségeit sorolja fel.

- `name`, `version` – a projekt azonosítója és verziója
- `main` – melyik fájl a belépési pont, ha valaki `require`-olja a projektünket
- `scripts` – rövid parancsok gyakori feladatokra (erről bővebben pár dia múlva)
- `dependencies` / `devDependencies` – a projekt függőségei (erről is bővebben mindjárt)

Csomagok telepítése – production függőségek

Amikor egy csomagra van szükségünk, az `npm install` (röviden `npm i`) paranccsal tudjuk hozzáadni a projekthez. Fontos eldönteni, hogy a csomagra csak fejlesztés közben, vagy éles működéskor is szükség lesz-e.

Production függősként

```
npm install express  
npm i express
```

Ezek a csomagok a `dependencies` alá kerülnek a `package.json`-ben – ezek nélkül az alkalmazásunk éles környezetben sem tudna elindulni, hiszen a kódunk ténylegesen használja őket futás közben (pl. az Express.js keretrendszert magát).

Csomagok telepítése – development függőségek

Vannak olyan eszközök, amikre csak a fejlesztés közben van szükségünk (pl. automatikus újraindítás, tesztelés, kódformázás) – ezeket nem érdemes az éles szerverre is felmásolni.

Development függősként

```
npm install --save-dev nodemon  
npm i -D nodemon jest eslint
```

Mi a különbség?

dependencies: éles környezetben is kell | **devDependencies:** csak fejlesztéshez (pl. tesztelő, formázó eszközök)

Semantic Versioning (SemVer)

Az npm csomagok verziószámozása egy szigorú, háromrészes konvenciót követ, amiből azonnal leolvasható, hogy egy frissítés mekkora változást hozhat.

Verzió formátum

1.2.3 → 1 (major) . 2 (minor) . 3 (patch)

- **MAJOR:** nem kompatibilis, "törő" API változás
- **MINOR:** új funkció, de visszafelé kompatibilis
- **PATCH:** hibajavítás, visszafelé kompatibilis

SemVer jelölések a package.json-ban

A `package.json`-ban a verziószám elé írt jel (`^` vagy `~`) szabja meg, hogy `npm install/npm update` során milyen új verziókat fogadunk el automatikusan.

Jelölés	Jelentés
1.2.3	Pontosan 1.2.3 verzió
<code>^1.2.3</code>	$\geq 1.2.3$ és $< 2.0.0$ (MINOR frissítések engedélyezve)
<code>~1.2.3</code>	$\geq 1.2.3$ és $< 1.3.0$ (csak PATCH frissítések)
latest	Mindig a legfrissebb verzió

Gyakori alapértelmezés

Az `npm install csomagnev` alapból a `^` jelölést használja – ez a leggyakoribb, kényelmes középút a stabilitás és a frissesség között.

npm scripts

A `package.json` `scripts` mezőjében saját, rövid parancsokat definiálhatunk a gyakran ismételt feladatokra (indítás, tesztelés, linting) – így nem kell mindig hosszú parancsokat begépelni.

```
{  
  "scripts": {  
    "start": "node index.js",  
    "dev": "nodemon index.js",  
    "test": "jest",  
    "lint": "eslint ."  
  }  
}
```

Scripts futtatása

```
npm run dev  
npm start    # 'run' nélkül is működik
```

node_modules és package-lock.json

Amikor telepítünk egy csomagot, az npm két dolgot hoz létre: egy `node_modules` mappát a tényleges kóddal, és egy `package-lock.json` fájlt. Fontos tudni, melyiket kell git-ben verziókövetni, és melyiket nem.

node_modules

- A telepített csomagok tényleges kódját tartalmazó mappa
- Nagyon nagy tud lenni (akár több száz MB)
- Ne commitoljuk git-be!
- Újragenerálás: `npm install`

package-lock.json

- Pontosan rögzíti, melyik csomagból melyik verzió lett telepítve
- Determinisztikus, mindenkinél ugyanazt eredményező telepítést biztosít
- Ezt viszont commitoljuk git-be!

Hasznos npm parancsok és npx

Néhány további parancs, amire a mindennapi munka során gyakran szükség lesz: a projekt állapotának ellenőrzésére, illetve arra, hogy egy csomagot telepítés nélkül, egyszer futtassunk le.

Karbantartás

```
npm list  
npm outdated  
npm audit  
npm audit fix
```

npx – futtatás telepítés nélkül

```
npx prisma init  
npx tsx index.ts
```

Az npx egy segédeszközt csak egyszer, ideiglenesen tölt le és futtat le, anélkül, hogy állandó jelleggel telepítené a gépünkre.

Összefoglalás – NPM

npm alapok

- npm: Node Package Manager, 2+ millió elérhető csomag
- `npm init` – projekt inicializálása, `package.json` létrehozása
- `npm install/uninstall/update` – csomagkezelés
- `dependencies` vs `devDependencies` – éles vs csak fejlesztői függőség
- Semantic Versioning: `MAJOR.MINOR.PATCH`
- `node_modules` (ne `commitold`), `package-lock.json` (ezt viszont igen)
- `npx`: csomagok egyszeri futtatása telepítés nélkül

Következő lépések

A mai órán megismertük a webes kommunikáció alapjait, elindítottunk egy Node.js szerveret, és megtanultuk, hogyan kezeljük a projekt függőségeit npm-mel. A következő alkalomig érdemes ezeket a lépéseket elvégezni:

- 1 **Fejlesztői környezet telepítése** (Node.js, npm) – ha még nincs telepítve, mindenképp pótolod otthon
- 2 **Git és verziókezelés alapjai** – a kódunkat innentől mindig verziókezelve fogjuk fejleszteni
- 3 **Első Express.js alkalmazás elkészítése** – a jövő héten a beépített http modul helyett egy sokkal kényelmesebb keretrendszert használunk
- 4 **REST API alapok megismerése** (2. hét) – ráépítünk a mai HTTP-s tudásra