Webtechnológia és webalkalmazás-fejlesztés

2. hét – REST API, Architekturális felépítések, CQRS

Magda Donát

Széchenyi István Egyetem, Győr

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

2026. szeptember 8.

Miről volt szó múlt héten?

Az első héten megismertük a webes kommunikáció alapköveit: a HTTP protokollt, a Node.js futtatókörnyezetet és az npm csomagkezelőt. Ezek nélkül nem tudnánk elindulni – de önmagukban még nem elegendők egy jól használható backend rendszer megépítéséhez.

- **HTTP/HTTPS**: Request-Response modell, állapotmentes protokoll, metódusok (GET, POST, PUT, PATCH, DELETE), státuszkódok
- **Node.js**: szerveroldali JavaScript futtatókörnyezet, beépített `http` modul egy egyszerű szerver elindítására
- **npm**: csomagkezelés, `package.json`, függőségek

A hiányzó darab

Múlt héten a `http.createServer`-rel csak egyetlen, mindent kiszolgáló függvényt írtunk. Éles alkalmazásoknál ez gyorsan kezelhetetlenné válik – szükségünk van egy *tervezési logikára*, ami megmondja, hogyan szervezzük az útvonalainkat, adatainkat és a kódunk rétegeit.

Mi a célja a mai órának?

A mai anyag három, egymásra épülő témát dolgoz fel: hogyan tervezzünk jól használható API-kat, hogyan szervezzük a backend kódunkat rétegekbe, és hogyan válasszuk szét az írási és olvasási műveleteket, ha a rendszerünk ezt megkívánja.

- **REST API alapok** – hogyan tervezzünk következetes, könnyen használható API-t
- **Architekturális felépítések** – hogyan szervezzük a backend kódunkat rétegekbe, hogy karbantartható maradjon
- **CQRS** – mikor éri meg szétválasztani az írási és olvasási műveleteket

Fontos

A mai témák még mindig *tervezési* kérdések – konkrét kódot Express.js-ben csak a jövő héttől írunk. Most a "miért így?" kérdésre keressük a választ, hogy a következő heti kód már értelmes döntéseken alapuljon.

Mi az API?

Az **API** (Application Programming Interface) egy megállapodás arról, hogyan kommunikálhat egymással két szoftverkomponens – anélkül, hogy az egyiknek ismernie kellene a másik belső működését. Amikor a telefonodon megnyitod egy időjárás-alkalmazást, az alkalmazás egy API-n keresztül kér adatot egy távoli szervertől; neked (és az alkalmazásnak) fogalma sincs, hogyan tárolja a szerver az adatokat belül – csak azt kell tudnia, milyen kérést küldjön, és milyen választ várhat.

- Egyértelmű, dokumentált "szerződés" a kliens és a szerver között
- Elrejti a belső implementációs részleteket (adatbázis típusa, kód szerkezete, stb.)
- Web API-k esetén jellemzően HTTP felett kommunikálunk – ez az, amit az elmúlt órán megismertünk

REST – egy konkrét API-stílus

A **REST** (Representational State Transfer) nem egy protokoll vagy szabvány, hanem egy architekturális stílus, amit Roy Fielding írt le 2000-ben a doktori disszertációjában. Ma a legelterjedtebb módja a web API-k tervezésének.

Mitől "RESTful" egy API?

Fielding öt fő elvet fogalmazott meg, amiknek egy API-nak meg kell felelnie ahhoz, hogy igazán RESTfulnek nevezzük. A gyakorlatban a legtöbb "REST API"-nak nevezett rendszer csak részben követi ezeket – de minél többet betartunk belőlük, annál kiszámíthatóbb és skálázhatóbb lesz az API-nk.

- **Kliens-szerver**: a kliens és a szerver különálló felelősséggel bír, egymástól függetlenül fejleszthetők
- **Állapotmentesség (statelessness)**: minden kérés önmagában tartalmaz minden szükséges információt – ezt már az 1. héten megismertük a HTTP-nél
- **Cache-elhetőség**: a válaszoknak jelezniük kell, hogy gyorsítótárazhatók-e (pl. `Cache-Control` header)
- **Egységes interfész (uniform interface)**: az erőforrásokat egységes módon, azonosítható URL-eken keresztül érjük el
- **Rétegzett rendszer**: a kliens nem tudja (és nem is kell tudnia), hogy közvetlenül a szerverrel, vagy egy köztes rétegen (pl. load balancer, cache) keresztül kommunikál-e

Erőforrás-orientált URL tervezés

Az URL-ek *erőforrásokat* (főneveket) azonosítsanak, ne *műveleteket* (igéket) – a műveletet úgyis a HTTP metódus fejezi ki.

Rossz gyakorlat (igealapú URL)

```
GET /getAllUsers
POST /createNewUser
POST /deleteUser?id=123
```

Jó gyakorlat (erőforrás-alapú, többes szám)

```
GET    /users          # osszes felhasznalo
POST   /users          # uj felhasznalo letrehozasa
GET    /users/123      # egy adott felhasznalo
DELETE /users/123      # felhasznalo torlese
```

CRUD → HTTP metódus megfeleltetés

Ez a táblázat gyakorlatilag a REST API tervezés "alaptörvénye" – ha ezt következetesen betartod, az API-d azonnal ismerős és kiszámítható lesz bárki számára, aki már használt REST API-t.

Művelet	**HTTP metódus**	**Végpont példa**
Create (létrehozás)	POST	`POST /users`
Read (egy elem)	GET	`GET /users/123`
Read (lista)	GET	`GET /users`
Update (teljes)	PUT	`PUT /users/123`
Update (részleges)	PATCH	`PATCH /users/123`
Delete (törlés)	DELETE	`DELETE /users/123`

Emlékeztető az 1. hétről

GET, PUT, DELETE idempotensek – többszöri hívás ugyanazt az állapotot eredményezi. A POST nem az.

Richardson Maturity Model

Leonard Richardson egy szintekre bontott modellt alkotott, amivel megmérhető, mennyire "RESTful" valójában egy API. A legtöbb valós API a 2. szinten áll – ez teljesen elfogadható, a 3. szint (HATEOAS) ritkán éri meg a többletbonyolultságot.

- **0. szint**: egyetlen végpont, minden művelet ugyanazon keresztül megy (pl. RPC-szerű, SOAP-hoz hasonló stílus)
- **1. szint**: külön erőforrások, saját URL-lel, de a HTTP metódusokat még nem használjuk következetesen
- **2. szint**: erőforrások + helyesen használt HTTP metódusok és státuszkódok – *ez a gyakorlatban elterjedt "REST API" szint*
- **3. szint (HATEOAS)**: a válasz linkeket is tartalmaz a kapcsolódó műveletekhez, így a kliensnek nem kell előre tudnia az összes URL-t

API verziózás

Az API-k idővel változnak – új mezők, módosított viselkedés. Ha egy változás nem visszafelé kompatibilis, szükség van egy módra, hogy a régi klienseket ne törjük el. Erre szolgál a verziózás.

URL-alapú verziózás (leggyakoribb)

```
GET /api/v1/users/123
GET /api/v2/users/123
```

Egyszerű, jól látható, könnyen tesztelhető Postmanben is.

Header-alapú verziózás

```
GET /api/users/123
Accept: application/vnd.myapp.v2+json
```

"Tisztább" URL-eket eredményez, de nehezebb böngészőből tesztelni.

Lapozás, szűrés, rendezés

Amikor egy erőforrásból (pl. felhasználók) több ezer is lehet, nem küldhetjük vissza mindet egyszerre – ezt oldják meg a query paraméterek, amikkel a kliens finomíthatja, pontosan mit szeretne kapni.

```
GET /users?page=2&limit=20
GET /users?role=admin
GET /users?sort=-createdAt
GET /users?search=alice
```

- **Lapozás (pagination)**: `page/limit` vagy `offset/limit` párokkal
- **Szűrés (filtering)**: egy adott mező szerinti szűkítés (`role=admin`)
- **Rendezés (sorting)**: `sort=mezonev`, mínusz előjellel csökkenő sorrend

Összefoglalás – REST API

REST API tervezés

- API: szerződés két rendszer kommunikációjához; REST: egy elterjedt architekturális stílus rá
- Fielding elvei: kliens-szerver, statelessness, cache-elhetőség, egységes interfész, rétegzettség
- URL-ek: erőforrások (főnevek, többes szám), nem műveletek (igék)
- CRUD → HTTP metódus: POST/GET/PUT/PATCH/DELETE következetesen
- Richardson Maturity Model: a legtöbb API a 2. szinten van, ez rendben van
- Verziózás: URL vagy header alapon; lapozás/szűrés/rendezés: query paraméterekkel

Mi a szoftverarchitektúra, és miért számít?

A szoftverarchitektúra azt írja le, hogyan van a kódunk nagy egységekre (rétegekre, modulokra, szolgáltatásokra) felosztva, és ezek hogyan kommunikálnak egymással. Egy kis, egyszemélyes prototípusnál szinte mindegy, hova írjuk a kódot – de amint a projekt nő, csapatban dolgozunk rajta, vagy hónapok múlva kell hozzányúlni, a jó architektúra dönti el, hogy karbantartható marad-e a rendszer, vagy "spagettikóddá" válik.

- Könnyebb megérteni: világos, hogy melyik kód mit csinál, és hol keresendő egy adott logika
- Könnyebb tesztelni: a rétegek külön-külön, elszigetelten tesztelhetők
- Könnyebb bővíteni: új funkció hozzáadásakor tudjuk, hova illesztjük be
- Könnyebb csapatban dolgozni: a felelősségek szét vannak választva

Rétegzett architektúra – az öt réteg (1/2)

A rétegzett felépítésnél a kódot vízszintes rétegekbe szervezzük, ahol minden réteg csak a közvetlenül alatta lévővel kommunikál. Ezen a tárgyon a következő öt réteges felosztást fogjuk következetesen használni – elsőként a "felső" három réteg:

- **API / Controller**: fogadja a HTTP kéréseket, validálja a bemenetet, meghívja az Application réteget, visszaadja a választ – ezt Express.js-ben írjuk meg a jövő héten
- **Application**: az egyes use case-eket (parancsokat/lekérdezéseket) orchesztrálja – ide kerülnek majd a CQRS command/query handlerek
- **Domain**: a tiszta üzleti logika, entitások, aggregates – semmilyen technikai függősége nincs, nem "tudja", hogy Express-ben vagy adatbázissal fut

Rétegzett architektúra – az öt réteg (2/2)

A maradék két réteg a rendszer "peremén" helyezkedik el: az egyik a kifelé irányuló adatszerződéseket, a másik a technikai megvalósítást fogja össze.

- **Communication**: a rétegek/külső rendszerek közti szerződések – DTO-k, mapperek (6. hét)
- **Infrastructure**: a konkrét technikai megvalósítás – adatbázis-kapcsolat, külső szolgáltatások

Miért éri meg?

Ha lecseréljük az adatbázisunkat, csak az Infrastructure réteget kell módosítani – a Domain és az API réteg változatlan marad.

MVC minta

Az MVC (Model-View-Controller) a rétegzett architektúra egyik legismertebb konkrét megvalósítása, eredetileg felhasználói felületekhez, de a backend világban is elterjedt elnevezés lett a hasonló felosztásra.

- **Model**: az adatokat és az azokhoz tartozó szabályokat képviseli (nálunk: a Domain + Infrastructure réteg)
- **View**: a megjelenítés – backend API-nál ez jellemzően a JSON válasz formázása
- **Controller**: fogadja a bejövő kérést, meghívja a megfelelő Model-műveletet, és visszaküldi a View-t

Backend kontextusban

Amikor a jövő héten Express.js-ben "route handler"-t írunk, az gyakorlatilag az API/Controller réteg szerepét tölti be: fogadja a HTTP kérést, és delegálja a munkát az Application rétegnek.

Monolit vs mikroszolgáltatás

Ez a döntés arról szól, hogy a teljes backend rendszerünket egyetlen, összefüggő alkalmazásként (monolit), vagy sok, önállóan futtatható kis szolgáltatásra bontva (mikroszolgáltatások) építjük-e fel. Egyik sem "jobb" alapból – a projekt méretétől és a csapat helyzetétől függ.

Monolit

- Egyetlen kódbázis, egyetlen futó folyamat
- Egyszerűbb fejleszteni és deployolni kisebb csapatban
- Egyszerűbb tranzakciókezelés

Mikroszolgáltatások

- Sok kis, önálló szolgáltatás, saját adatbázissal
- Külön-külön skálázhatók és deployolhatók
- Nagyobb komplexitás: hálózati kommunikáció, elosztott tranzakciók

Gyakorlati tanács

Kezdj monolittal! A tárgy során épített API is monolit lesz – a mikroszolgáltatásokra való szétbontás egy már működő, jól rétegzett monolitból sokkal könnyebb, mint elölről tervezni.

Separation of Concerns

A "Separation of Concerns" (felelősségek szétválasztása) az az elv, ami az összes eddigi mintát (rétegzés, MVC) megalapozza: minden kódegységnek egyetlen, jól körülhatárolt felelőssége legyen.

- Egy függvény/osztály ne csinálja egyszerre a HTTP kérés kezelését *és* az adatbázis-lekérdezést *és* az email küldést
- Ha egy dolog megváltozik (pl. az adatbázis lekérdezésének módja), az csak egy helyen igényeljen módosítást
- Könnyebb egyesével, elszigetelten tesztelni az egyes felelősségeket

Ökölszabály

Ha egy függvény nevét csak "és"-sel tudod leírni (pl. "validál ÉS ment ÉS emailt küld"), valószínűleg érdemes szétbontani.

Kitekintés – mi jön még ebben a témában

A rétegzett architektúra és a Separation of Concerns elve két fogalmat is megalapoz, amikkel a félév további részében részletesen foglalkozunk majd – most csak annyit érdemes megjegyezni, hogy hova illeszkednek.

- **Repositories és Aggregates (5. hét)**: az Infrastructure réteg formalizált, ORM-alapú megvalósítása – a repository "elrejti" az adatbázis-lekérdezéseket egy egyszerű interfész mögé
- **Dependency Injection (11. hét)**: technika arra, hogy a rétegek közötti függéseket (pl. "a controllernek szüksége van a service-re") ne kézzel drótozzuk be, hanem egy keretrendszer intézze

Összefoglalás – Architekturális felépítések

Architektúra

- A jó architektúra karbantarthatóvá, tesztelhetővé és bővíthetővé teszi a rendszert
- Rétegzett architektúra (öt réteg): API/Controller – Application – Domain – Communication – Infrastructure
- MVC: Model-View-Controller – a Controller felel meg a jövő heti Express.js route handlernek
- Monolit vs mikroszolgáltatás: kezdj monolittal, bontsd szét később, ha indokolt
- Separation of Concerns: minden kódegységnek egy jól körülhatárolt felelőssége legyen

A probléma: egy modell, két teljesen más terhelés

A legtöbb egyszerű alkalmazásban ugyanaz a kód, ugyanaz az adatmodell szolgálja ki mind az adatok módosítását (írás), mind a lekérdezését (olvasás). Ez sokáig teljesen jól működik – de bizonyos rendszereknél ez a közös modell egyre inkább korlátozó tényezővé válik.

- Az írási műveleteknél fontos a szigorú validáció és az üzleti szabályok betartatása
- Az olvasási műveleteknél viszont gyakran *más* adatszerkezet lenne ideális (pl. több táblából összefésült, "lapos" nézet egy listázó felülethez)
- Ha ugyanaz a modell szolgál ki mindkét célt, gyakran kompromisszumot kell kötni – egyik oldal sem lesz igazán optimális

Példa

Egy webshopnál a rendelés *leadása* (írás) sok validációt igényel (készlet, fizetés, kedvezmények), míg a rendelések *listázása* (olvasás) egyszerűen csak gyors, jól szűrhető megjelenítést igényel – a kettő igénye nagyon eltérő.

A CQRS lényege

A **CQRS** (Command Query Responsibility Segregation, magyarul "parancs-lekérdezés felelősség szétválasztás") azt mondja ki: válasszuk szét élesen az írási (command) és olvasási (query) műveleteket – akár külön modellel, külön kóddal, szélsőséges esetben külön adatbázissal is.

- **Command (parancs)**: egy szándékot fejez ki valaminek a megváltoztatására (pl. "hozz létre egy rendelést") – nem ad vissza adatot, legfeljebb egy azonosítót vagy sikerességet
- **Query (lekérdezés)**: adatot kér le, *nem* változtat semmin – ez pontosan megfelel a HTTP GET idempotens, csak-olvasó jellegének, amit már az 1. héten megismertünk

Fontos megkülönböztetés

A CQRS nem ugyanaz, mint a HTTP metódusok GET/POST felosztása – az egy protokollszintű konvenció, a CQRS pedig egy *architekturális döntés* arról, hogy a kódunk belső szerkezete is kövesse ezt a felosztást.

Command és Query – egyszerű példa

A lényeg pszeudokódban jól látszik: a Command egy műveletet hajt végre és validál, a Query csak lekérdez – a kettő felelőssége, sőt akár a visszatérési típusa is más.

Command – írás

```
function createOrder(data) {
  validate(data);
  checkStock(data.items);
  const order = save(data);
  return order.id;
}
```

Query – olvasás

```
function getOrderById(id) {
  return db
    .orders
    .findOne({ id });
}
```

Mi változik igazán CQRS-sel?

Nem feltétlenül a fájlok fizikai elválasztása a lényeg, hanem az, hogy a Command sosem ad vissza lekérdezési adatot, és a Query sosem módosít állapotot – ez a szigorú szétválasztás teszi átláthatóvá a rendszert.

Miért éri meg a CQRS-t bevezetni?

A szétválasztás nem öncélú bonyolítás – konkrét, mérhető előnyöket ad olyan rendszereknél, ahol az írási és olvasási terhelés jelentősen eltér egymástól.

- **Független skálázhatóság**: ha egy rendszerben sokkal több az olvasás, mint az írás (ez a leggyakoribb eset), az olvasási oldalt külön lehet skálázni
- **Optimalizált olvasási modellek**: a lekérdező oldalon szabadon kialakíthatunk a megjelenítéshez ideális, denormalizált nézeteket, anélkül hogy az írási oldal adatmodelljét kompromisszumba kényszerítenénk
- **Egyszerűbb, célzottabb kód**: egy Command csak a validációra és a módosításra koncentrál, egy Query csak a lekérdezésre – egyik sem próbál "mindent tudni"

Mikor túlzás a CQRS?

A CQRS egy erős eszköz, de nem ingyen jön – minden architekturális mintánál fontos feltenni a kérdést: *a probléma, amit megold, valóban jelen van-e nálunk?*

Ökölszabály

Egyszerű CRUD alkalmazásoknál (ahol az írás és olvasás igénye hasonló, és nincs komoly skálázási probléma) a CQRS csak felesleges réteget és bonyolultságot ad hozzá – ilyenkor egy egyszerű, közös modell teljesen megfelelő.

- Kis csapatnál, kis projektnél a többletkód karbantartása nem éri meg a hasznot
- Csak akkor vezessük be, ha ténylegesen tapasztaljuk a "két különböző igény" problémáját
- A tárgy hátralévő részében épített API-nk egyszerű CRUD marad – a CQRS itt főleg mint *fogalom* és *gondolkodásmód* fontos

Kitekintés: Event Sourcing

A CQRS-t gyakran együtt említik egy másik mintával, az **Event Sourcing**-gal, bár a kettő nem jár automatikusan együtt.

Az alapötlet

Ahelyett, hogy csak az adat *jelenlegi állapotát* tárolnánk, minden változást (eseményt) elmentünk – pl. "OrderCreated", "OrderPaid", "OrderShipped". A jelenlegi állapotot ezekből az eseményekből lehet bármikor újraszámolni.

Ez a tárgy keretein túlmutat, de érdemes tudni, hogy létezik: teljes körű auditnaplót ad ingyen, és jól illeszkedik a CQRS command-oldalához. Ára viszont jelentős többletkomplexitás.

Összefoglalás – CQRS

CQRS

- A probléma: egy közös modell néha rosszul szolgálja ki mind az írást, mind az olvasást
- CQRS: Command (írás, nem ad vissza adatot) és Query (olvasás, nem módosít) szigorú szétválasztása
- Előny: független skálázhatóság, optimalizált olvasási modellek, egyszerűbb, célzottabb kód
- Csapda: egyszerű CRUD alkalmazásoknál felesleges többletkomplexitás
- Kapcsolódó fogalom: Event Sourcing (események tárolása állapot helyett) – ez már a tárgy keretein túlmutat

Következő lépések

Ma megismertük, hogyan tervezzünk jól használható REST API-t, hogyan szervezzük a backend kódunkat rétegekbe, és mikor érdemes szétválasztani az írási és olvasási műveleteket. A jövő héten ezekre a tervezési elvekre már konkrét kódot építünk.

1. **Express.js megismerése** – a jövő heti keretrendszer, amivel az API/Controller réteget ténylegesen megírjuk
2. **Routing gyakorlása** – a mai órán tervezett erőforrás-orientált URL-eket Express.js route handlerekként fogjuk implementálni
3. **Postman telepítése** – API-k manuális teszteléséhez, hogy a saját végpontjainkat is ki tudjuk próbálni
4. **TypeScript alapok** – a jövő héttől fokozatosan áttérünk típusos JavaScriptre

Javaslat

Nézd át még egyszer a CRUD → HTTP metódus táblázatot – a jövő héten ezt fogjuk közvetlenül route definíciókra fordítani.