Webtechnológia és webalkalmazás-fejlesztés - Rest API

Rest API

Magda Donát

Széchenyi István Egyetem, Győr

https://git.mdnd-it.cc/Donat/GKNB$_M$STM071

2026. január 28.

Mi a REST API?

REST - Representational State Transfer

- Architektúra stílus webes szolgáltatásokhoz
- Roy Fielding dolgozta ki 2000-ben
- HTTP protokollon alapuló kommunikáció
- Állapotmentes (stateless) kliens-szerver kapcsolat

API - Application Programming Interface

Interfész, amin keresztül különböző alkalmazások kommunikálnak egymással.

REST alapelvek

1. **Kliens-szerver architektúra**
   - Szétválasztott felelősségek
   - Kliens: felhasználói felület
   - Szerver: adatkezelés, üzleti logika
2. **Állapotmentesség (Stateless)**
   - Minden kérés független
   - Szerver nem tárol kliens állapotot
3. **Cachelhető**
   - Válaszok cache-elhetők a teljesítmény növelésére

REST alapelvek (folyt.)

4. **Egységes interfész**
   - Erőforrások URI-kon keresztül azonosíthatók
   - Reprezentációkon keresztüli manipuláció
   - Önleíró üzenetek
5. **Rétegzett rendszer**
   - Köztes rétegek (proxy, gateway) használhatók
   - Kliens nem tudja, közvetlenül a szerverhez kapcsolódik-e

Erőforrások és URI-k

Erőforrás (Resource)

Minden, amit azonosítani és kezelni akarunk (pl. felhasználó, termék, rendelés).

URI példák

- `/users` - Összes felhasználó
- `/users/123` - Adott felhasználó (ID: 123)
- `/users/123/posts` - Egy felhasználó posztjai
- `/products` - Összes termék
- `/products/456` - Adott termék

Konvenció

Használj **főneveket** és **többes számot**: `/users`, `/products`

HTTP metódusok

Metódus	Művelet	Idemp.	Példa
GET	Lekérdezés	Igen	Lista/részlet
POST	Létrehozás	Nem	Új elem
PUT	Frissítés (teljes)	Igen	Teljes csere
PATCH	Frissítés (rész)	Nem	Rész módosítás
DELETE	Törlés	Igen	Elem törlése

Idempotencia

Ugyanaz a kérés többször végrehajtva ugyanazt az eredményt adja.

HTTP metódusok - GET példák

Összes elem lekérdezése

```
GET /api/users
Response: 200 OK
[
  {"id": 1, "name": "Alice", "email": "alice@example.com"},
  {"id": 2, "name": "Bob", "email": "bob@example.com"}
]
```

Egy elem lekérdezése

```
GET /api/users/1
Response: 200 OK
{"id": 1, "name": "Alice", "email": "alice@example.com"}
```

HTTP metódusok - POST, PUT

POST - Új elem létrehozása

```
POST /api/users
Body: {"name": "Charlie", "email": "charlie@example.com"}
Response: 201 Created
{"id": 3, "name": "Charlie", "email": "charlie@example.com"}
```

PUT - Teljes frissítés

```
PUT /api/users/1
Body: {"name": "Alice Smith", "email": "alice.smith@example.com"}
Response: 200 OK
{"id": 1, "name": "Alice Smith", "email": "alice.smith@example.com"}
```

HTTP metódusok - PATCH, DELETE

PATCH - Részleges frissítés

```
PATCH /api/users/1
Body: {"email": "newemail@example.com"}
Response: 200 OK
{"id": 1, "name": "Alice", "email": "newemail@example.com"}
```

DELETE - Törlés

```
DELETE /api/users/1
Response: 204 No Content
```

HTTP státuszkódok

2xx - Sikeres

- **200** OK
- **201** Created
- **204** No Content

4xx - Kliens hiba

- **400** Bad Request
- **401** Unauthorized
- **403** Forbidden
- **404** Not Found

5xx - Szerver hiba

- **500** Internal Error
- **502** Bad Gateway
- **503** Service Unavailable

Fontos

Mindig a megfelelő státuszkódot küldd!

JSON formátum

JavaScript Object Notation

A REST API-k leggyakoribb adatformátuma.

Példa

```
{
  "id": 1,
  "name": "John Doe",
  "email": "john@example.com",
  "age": 30,
  "active": true,
  "roles": ["user", "admin"],
  "address": {
    "city": "Budapest",
    "zip": "1111"
  }
}
```

REST vs SOAP

	REST	SOAP
Protokoll	HTTP	HTTP, SMTP, TCP
Formátum	JSON, XML	Csak XML
Állapot	Stateless	Lehet stateful
Sebesség	Gyorsabb	Lassabb
Használat	Egyszerűbb	Komplexebb

Mikor REST?

Webes és mobil alkalmazások, publikus API-k, mikroszolgáltatások

Hibakezelés

Hibaválasz struktúra

```
{
  "error": {
    "code": "USER_NOT_FOUND",
    "message": "A megadott felhasználó nem található",
    "details": "User with ID 123 does not exist",
    "timestamp": "2026-01-27T20:30:00Z"
  }
}
```

Best Practices

- Mindig küldd a megfelelő HTTP státuszkódot
- Adj értelmes hibaüzeneteket
- Ne adj ki érzékeny információkat

Összefoglalás - REST Alapok

- **REST** = Representational State Transfer (Reprezentációs Állapotátvitel)
- Architektúra stílus HTTP-n keresztül
- **Erőforrások** (Resources) URI-kon keresztül azonosíthatók
- **HTTP metódusok**: GET (olvasás), POST (létrehozás), PUT/PATCH (módosítás), DELETE (törlés)
- **Státuszkódok** (Status Codes) jelzik a kérés eredményét
- **JSON** a leggyakoribb adatformátum
- **Állapotmentes** (Stateless) kommunikáció

Miért van szükség architektúrára?

A probléma: "Spagetti kód"

Kezdő fejlesztők gyakran mindent egy helyre írnak:

- Adatbázis lekérdezések a route-okban
- Üzleti logika a controller-ekben
- Validáció szétszórva mindenhol

Eredmény: Áttekinthetetlen, nehezen karbantartható kód

A megoldás: Rétegezett architektúra

Minden funkciónak megvan a saját helye és felelőssége.

5-rétegű architektúra

Separation of Concerns - Felelősségek szétválasztása

- **Karbantarthatóság** - Könnyebb módosítás, hibakeresés
- **Tesztelhetőség** - Rétegek külön tesztelhetők
- **Skálázhatóság (Scalability)** - Független fejlesztés és skálázás
- **Újrafelhasználhatóság** - Rétegek több helyen használhatók

Alapelv

Minden réteg csak a saját felelősségével foglalkozik!

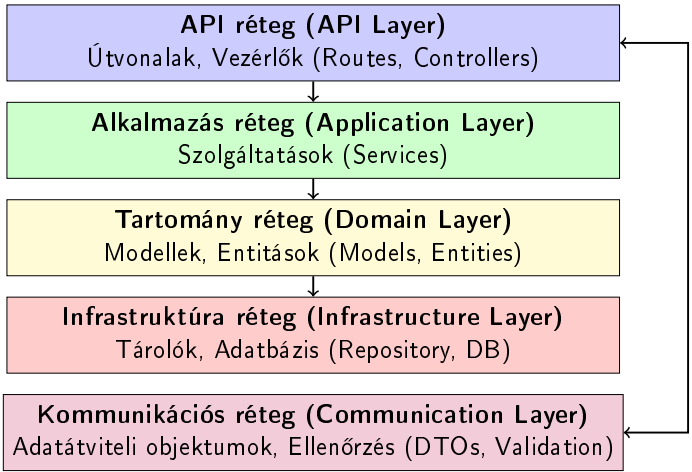
Az 5 réteg

1. **API Layer (Presentation Layer)**
   - HTTP kérések fogadása
   - Route-ok, Controller-ek
2. **Application Layer (Service Layer)**
   - Üzleti logika koordinálása
   - Service-ek
3. **Domain Layer (Business Logic)**
   - Üzleti szabályok
   - Model-ek, Entity-k

Az 5 réteg (folyt.)

4. **Infrastructure Layer**
   - Adatbázis hozzáférés
   - Repository-k, ORM
5. **Communication Layer**
   - Adatok átalakítása
   - DTO-k, Validáció

Rétegek áttekintése - Diagram



API Layer - Route és Controller

Route példa

```
// routes/user.routes.js
const express = require('express');
const router = express.Router();
const userController = require('../controllers/user.controller');

router.get('/', userController.getAllUsers);
router.get('/:id', userController.getUserById);
router.post('/', userController.createUser);

module.exports = router;
```

API Layer - Controller

Controller példa

```
// controllers/user.controller.js
const userService = require('../services/user.service');

exports.getAllUsers = async (req, res) => {
  try {
    const users = await userService.getAllUsers();
    res.json(users);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
};

exports.getUserById = async (req, res) => {
  try {
    const user = await userService.getUserById(req.params.id);
    if (!user) return res.status(404).json({ error: 'Not found' });
    res.json(user);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
};
```

Application Layer - Service

Service példa

```
// services/user.service.js
const userRepository = require('../repositories/user.repository');

exports.getAllUsers = async () => {
  return await userRepository.findAll();
};

exports.getUserById = async (id) => {
  return await userRepository.findById(id);
};

exports.createUser = async (userData) => {
  // Üzleti logika validálás
  if (!userData.email || !userData.name) {
    throw new Error('Email and name are required');
  }
  return await userRepository.create(userData);
};
```

Domain Layer - Model/Entity

User Model

```
// models/user.model.js
class User {
  constructor(id, name, email, createdAt) {
    this.id = id;
    this.name = name;
    this.email = email;
    this.createdAt = createdAt || new Date();
  }

  // Üzleti logika metódusok
  isActive() {
    return this.active === true;
  }

  hasRole(role) {
    return this.roles.includes(role);
  }
}

module.exports = User;
```

Infrastructure Layer - Repository

Repository példa

```
// repositories/user.repository.js
const db = require('../config/database');

exports.findAll = async () => {
  return await db.query('SELECT * FROM users');
};

exports.findById = async (id) => {
  const result = await db.query('SELECT * FROM users WHERE id = ?', [id]);
  return result[0];
};

exports.create = async (userData) => {
  const result = await db.query(
    'INSERT INTO users (name, email) VALUES (?, ?)',
    [userData.name, userData.email]
  );
  return { id: result.insertId, ...userData };
};
```

Communication Layer - DTO

Data Transfer Object

```
// dtos/user.dto.js
class UserDTO {
  constructor(user) {
    this.id = user.id;
    this.name = user.name;
    this.email = user.email;
    // Érzékeny mezők nem kerülnek át (pl. password)
  }
}

class CreateUserDTO {
  constructor(data) {
    this.name = data.name;
    this.email = data.email;
    this.password = data.password;
  }

  validate() {
    if (!this.email || !this.name || !this.password) {
      throw new Error('All fields required');
    }
  }
}
```

Projekt struktúra - Mappa felépítés

Az 5-rétegű architektúra a mappákban

- `src/` - Forráskód gyökér
- `routes/` - API réteg: útvonalak definiálása
- `controllers/` - API réteg: HTTP kérések kezelése
- `services/` - Alkalmazás réteg: üzleti logika
- `models/` - Tartomány réteg: adatmodellek
- `repositories/` - Infrastruktúra réteg: DB műveletek
- `dtos/` - Kommunikáció réteg: adatátviteli objektumok
- `middlewares/` - Köztes réteg (autentikáció, naplózás)
- `config/` - Konfigurációs fájlok

Projekt struktúra - Függőségek

Kommunikáció a rétegek között

- **Routes** → Controllers (útvonal hívja a vezérlőt)
- **Controllers** → Services (vezérlő hívja a szolgáltatást)
- **Services** → Models + Repositories (szolgáltatás használja az adatréteget)
- **Repositories** → Database (tároló eléri az adatbázist)
- **DTOs** ↔ Minden réteg (minden réteg használhat DTO-kat)

Fontos szabály

A belső rétegek NEM függhetnek a külső rétegektől! A függőségek mindig befelé mutatnak.

Clean Architecture

Dependency Rule

Belső rétegek nem függhetnek külső rétegektől. A függőségek mindig befelé mutatnak.

Előnyök

- Független a framework-től
- Tesztelhető
- Független az UI-tól
- Független az adatbázistól
- Független külső szolgáltatásoktól

Összefoglalás - Architektúra

- 5-rétegű architektúra a clean code érdekében
- API Layer (API réteg): Route-ok és Controller-ek (útvonalak, vezérlők)
- Application Layer (Alkalmazás réteg): Service-ek (szolgáltatások - üzleti logika)
- Domain Layer (Tartomány réteg): Model-ek és Entity-k (modellek, entitások)
- Infrastructure Layer (Infrastruktúra réteg): Repository-k és DB (tárolók, adatbázis)
- Communication Layer (Kommunikációs réteg): DTO-k és validáció (adatátvitel, ellenőrzés)

Mi a CQRS?

Command Query Responsibility Segregation

Magyarul: Parancs-Lekérdezés Felelősség Szétválasztása

Az adatok **írásának** (módosítás) és **olvasásának** (lekérdezés) szétválasztása külön modellekre.

Command (Parancs)

Írási műveletek:

- POST, PUT, PATCH, DELETE
- Adatok módosítása
- Validáció, üzleti logika
- Pl: felhasználó létrehozása

Query (Lekérdezés)

Olvasási műveletek:

- GET
- Adatok lekérdezése
- Optimalizált olvasás
- Pl: felhasználók listázása

Miért van szükség CQRS-re?

A probléma

Hagyományos CRUD alkalmazásokban ugyanazt a modellt használjuk írásra és olvasásra. Ez problémás lehet:

- Az írási és olvasási igények eltérőek
- Az olvasás gyakran bonyolultabb (join-ok, aggregációk)
- A teljesítmény-optimalizálás nehéz

A megoldás: CQRS

Külön modellek külön optimalizálással:

- **Write Model** (Író modell): validáció, üzleti szabályok
- **Read Model** (Olvasó modell): gyors lekérdezések, cache

CQRS előnyei

Előnyök

- **Független optimalizálás** - Az írás és olvasás külön optimalizálható
  - Íráshoz: tranzakciók, validáció, konzisztencia
  - Olvasáshoz: cache, denormalizált adatok, gyors lekérdezések
- **Skálázhatóság (Scalability)** - Read és Write modellek külön skálázhatók
  - Több olvasó szerver, kevesebb író szerver
- **Egyszerűség** - Komplexitás csökken
  - Minden modell egy feladatra fókuszál
- **Teljesítmény (Performance)** - Gyorsabb lekérdezések

Mikor használjuk a CQRS-t?

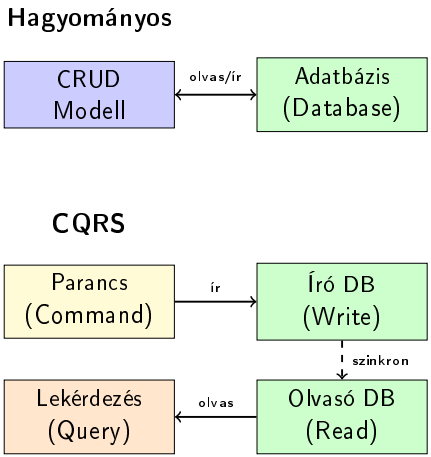
Jó választás, ha:

- **Komplex domain logika** van az alkalmazásban
- **Nagy terhelés** várható (sok párhuzamos felhasználó)
- **Eltérő írás/olvasás igények**
  - Pl: ritkán írunk, sokszor olvasunk
- **Különböző adatformátumok** kellenek íráshoz és olvasáshoz

NEM használjuk, ha:

- Egyszerű CRUD alkalmazás
- Kis terhelés
- Az írás és olvasás hasonló

CRUD vs CQRS - Architektúra összehasonlítás



Command - Írás példa

CreateUserCommand

```
// commands/create-user.command.js
class CreateUserCommand {
  constructor(name, email, password) {
    this.name = name;
    this.email = email;
    this.password = password;
  }

  validate() {
    if (!this.email || !this.name || !this.password) {
      throw new Error('All fields required');
    }
    if (this.password.length < 8) {
      throw new Error('Password too short');
    }
  }
}

module.exports = CreateUserCommand;
```

Command Handler

CreateUserCommandHandler

```
// handlers/create-user.handler.js
const userRepository = require('../repositories/user.repository');

class CreateUserCommandHandler {
  async handle(command) {
    command.validate();

    // Üzleti logika
    const existingUser = await userRepository.findByEmail(command.email);
    if (existingUser) {
      throw new Error('Email already exists');
    }

    // Password hash
    const hashedPassword = await bcrypt.hash(command.password, 10);

    // User létrehozás
    const user = await userRepository.create({
      name: command.name,
      email: command.email,
      password: hashedPassword
    });

    return user;
  }
}
```

Query - Olvasás példa

GetUsersQuery

```
// queries/get-users.query.js
class GetUsersQuery {
  constructor(filters = {}) {
    this.page = filters.page || 1;
    this.limit = filters.limit || 10;
    this.role = filters.role;
    this.active = filters.active;
  }
}

// handlers/get-users.handler.js
class GetUsersQueryHandler {
  async handle(query) {
    // Optimalizált lekérdezés
    const users = await userReadModel.find({
      role: query.role,
      active: query.active,
      skip: (query.page - 1) * query.limit,
      limit: query.limit
    });

    return users;
  }
}
```

CQRS REST API-ban (1/2)

Command végpontok

```
// routes/user.routes.js
const CreateUserCommand = require('../commands/create-user.command');
const createUserHandler = require('../handlers/create-user.handler');

router.post('/users', async (req, res) => {
  try {
    const command = new CreateUserCommand(
      req.body.name,
      req.body.email,
      req.body.password
    );
    const user = await createUserHandler.handle(command);
    res.status(201).json(user);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});
```

CQRS REST API-ban (2/2)

Query végpontok

```
const GetUsersQuery = require('../queries/get-users.query');
const getUsersHandler = require('../handlers/get-users.handler');

router.get('/users', async (req, res) => {
  try {
    const query = new GetUsersQuery({
      page: req.query.page,
      limit: req.query.limit,
      role: req.query.role,
      active: req.query.active
    });
    const users = await getUsersHandler.handle(query);
    res.json(users);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

CQRS előnyei és hátrányai

Előnyök

- Jobb teljesítmény
- Független skálázás
- Egyszerűbb modellek
- Optimalizált lekérdezések

Hátrányok

- Komplexitás növekedés
- Több kód
- Szinkronizáció problémák
- Eventual consistency

Event Sourcing + CQRS

Event Sourcing

Az állapotváltozások eseményekként tárolása, nem a végállapot mentése.

Események

- UserCreatedEvent
- UserEmailChangedEvent
- UserDeactivatedEvent

CQRS + ES kombinációja

Command-ok eseményeket generálnak, Query modellek eseményekből épülnek fel.

Összefoglalás - CQRS

- CQRS = Command és Query szétválasztása
- Command: adatok módosítása (POST, PUT, DELETE)
- Query: adatok lekérdezése (GET)
- Előnyök: teljesítmény, skálázhatóság
- Hátrányok: komplexitás
- Event Sourcing jól kombinálható CQRS-sel

Mi az Express.js?

Express

Gyors, minimalista web framework Node.js-hez.

Jellemzők

- Egyszerű API
- Middleware-alapú
- Routing támogatás
- Template engine-ek
- Nagy közösség és sok plugin

Express telepítése

NPM inicializálás és telepítés

```
npm init -y
npm install express
```

package.json

```
{
  "name": "my-api",
  "version": "1.0.0",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

Első Express szerver

server.js

```
const express = require('express');
const app = express();
const PORT = 3000;

// Middleware
app.use(express.json());

// Egyszerű route
app.get('/', (req, res) => {
  res.json({ message: 'Hello, Express!' });
});

// Szerver indítása
app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

GET végpont

Összes elem lekérdezése

```
let users = [
  {id: 1, name: 'Alice', email: 'alice@example.com'},
  {id: 2, name: 'Bob', email: 'bob@example.com'}
];

app.get('/api/users', (req, res) => {
  res.json(users);
});
```

Egy elem lekérdezése

```
app.get('/api/users/:id', (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (!user) {
    return res.status(404).json({ error: 'User not found' });
  }
  res.json(user);
});
```

POST végpont

Új elem létrehozása

```
app.post('/api/users', (req, res) => {
  const { name, email } = req.body;

  if (!name || !email) {
    return res.status(400).json({ error: 'Name and email required' });
  }

  const newUser = {
    id: users.length + 1,
    name,
    email
  };

  users.push(newUser);
  res.status(201).json(newUser);
});
```

PUT és PATCH végpontok

PUT - Teljes frissítés

```
app.put('/api/users/:id', (req, res) => {
  const idx = users.findIndex(u => u.id === parseInt(req.params.id));
  if (idx === -1) return res.status(404).json({ error: 'Not found' });

  users[idx] = { id: parseInt(req.params.id), ...req.body };
  res.json(users[idx]);
});
```

PATCH - Részleges frissítés

```
app.patch('/api/users/:id', (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (!user) return res.status(404).json({ error: 'Not found' });

  Object.assign(user, req.body);
  res.json(user);
});
```

DELETE végpont

Elem törlése

```
app.delete('/api/users/:id', (req, res) => {
  const idx = users.findIndex(u => u.id === parseInt(req.params.id));

  if (idx === -1) {
    return res.status(404).json({ error: 'User not found' });
  }

  users.splice(idx, 1);
  res.status(204).send();
});
```

204 No Content

Sikeres törlés esetén nem kell response body.

Middleware-ek

Mi a Middleware?

Függvények, amelyek hozzáférnek a request és response objektumokhoz, és a `next()` függvényhez.

Logger middleware

```
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});
```

Auth middleware

```
const authMiddleware = (req, res, next) => {
  const token = req.headers.authorization;
  if (!token) return res.status(401).json({ error: 'Unauthorized' });
  next();
};

app.get('/api/protected', authMiddleware, (req, res) => {
  res.json({ data: 'Protected data' });
});
```

Error Handling

Error handling middleware

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({
    error: {
      message: err.message,
      code: 'INTERNAL_ERROR'
    }
  });
});
```

Try-catch async

```
app.get('/api/users/:id', async (req, res, next) => {
  try {
    const user = await getUserById(req.params.id);
    if (!user) throw new Error('User not found');
    res.json(user);
  } catch (error) {
    next(error);
  }
});
```

Validáció Joi-val (1/2)

Joi telepítése

```
npm install joi
```

Schema definiálás

```
const Joi = require('joi');

const userSchema = Joi.object({
  name: Joi.string().min(3).max(50).required(),
  email: Joi.string().email().required(),
  age: Joi.number().integer().min(0).max(120)
});
```

Validáció Joi-val (2/2)

Validáció használata

```
app.post('/api/users', (req, res) => {
  const { error, value } = userSchema.validate(req.body);

  if (error) {
    return res.status(400).json({
      error: error.details[0].message
    });
  }

  // value a validált adat
  const newUser = { id: users.length + 1, ...value };
  users.push(newUser);
  res.status(201).json(newUser);
});
```

Express Router - Moduláris routing (1/2)

routes/users.js

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

router.get('/', (req, res) => {
  res.json({ message: 'Get all users' });
});

router.get('/:id', (req, res) => {
  res.json({ message: `Get user ${req.params.id}` });
});

router.post('/', (req, res) => {
  res.json({ message: 'Create user' });
});

module.exports = router;
```

Express Router - Moduláris routing (2/2)

app.js - Router használata

```
const express = require('express');
const app = express();
const userRoutes = require('./routes/users');

app.use(express.json());
app.use('/api/users', userRoutes);

app.listen(3000);
```

Prefix

Az `app.use('/api/users', userRoutes)` prefix-szel látja el az összes route-ot.

Összefoglalás - Express

- Express = minimalista web framework
- Routing: GET, POST, PUT, PATCH, DELETE
- Middleware-ek: kérések feldolgozása
- Error handling: hibakezelő middleware
- Validáció: Joi vagy más library
- Router: moduláris útvonalkezelés

Mi a Routing?

Routing

A kérések irányítása a megfelelő végpontokra (endpoint).

Route komponensek

- **HTTP metódus** - GET, POST, PUT, DELETE
- **URL path** - /api/users, /api/products/:id
- **Handler függvény** - (req, res) => {}

Alapvető routing

Egyszerű route-ok

```
app.get('/api/users', (req, res) => {
  res.json({ message: 'Get all users' });
});

app.post('/api/users', (req, res) => {
  res.json({ message: 'Create user' });
});

app.put('/api/users/:id', (req, res) => {
  res.json({ message: `Update user ${req.params.id}` });
});

app.delete('/api/users/:id', (req, res) => {
  res.json({ message: `Delete user ${req.params.id}` });
});
```

Route paraméterek

:id paraméter

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

// Több paraméter
app.get('/api/users/:userId/posts/:postId', (req, res) => {
  const { userId, postId } = req.params;
  res.json({ userId, postId });
});
```

req.params

URL-ben megadott dinamikus értékek.

Query paraméterek

Query string

```
// GET /api/users?page=1&limit=10&role=admin

app.get('/api/users', (req, res) => {
  const { page, limit, role } = req.query;

  res.json({
    page: page || 1,
    limit: limit || 10,
    role: role
  });
});
```

req.query

URL query string paraméterek (?key=value).

Query - Szűrés, Rendezés, Lapozás

GET /api/users?page=1&limit=10&sort=name&role=admin

```
app.get('/api/users', (req, res) => {
  let { page = 1, limit = 10, sort = 'id', role } = req.query;

  page = parseInt(page);
  limit = parseInt(limit);

  let filteredUsers = users;
  if (role) {
    filteredUsers = filteredUsers.filter(u => u.role === role);
  }

  // Rendezés
  filteredUsers.sort((a, b) => a[sort] > b[sort] ? 1 : -1);

  // Lapozás
  const start = (page - 1) * limit;
  const paginatedUsers = filteredUsers.slice(start, start + limit);

  res.json({ data: paginatedUsers, total: filteredUsers.length });
});
```

Express Router létrehozása

routes/users.js

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

router.get('/', (req, res) => {
  res.json({ message: 'Get all users' });
});

router.get('/:id', (req, res) => {
  res.json({ message: `Get user ${req.params.id}` });
});

router.post('/', (req, res) => {
  res.json({ message: 'Create user' });
});

router.put('/:id', (req, res) => {
  res.json({ message: `Update user ${req.params.id}` });
});

module.exports = router;
```

Router importálása

app.js

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

const userRoutes = require('./routes/users');
const productRoutes = require('./routes/products');

app.use(express.json());
app.use('/api/users', userRoutes);
app.use('/api/products', productRoutes);

app.listen(3000);
```

Prefix

Az `app.use('/api/users', userRoutes)` minden route-hoz hozzáadja a prefix-et.

Nested (beágyazott) route-ok

Hierarchikus erőforrások

```
// GET /api/users/:userId/posts
router.get('/:userId/posts', (req, res) => {
  const { userId } = req.params;
  res.json({ message: `Posts for user ${userId}` });
});

// GET /api/users/:userId/posts/:postId
router.get('/:userId/posts/:postId', (req, res) => {
  const { userId, postId } = req.params;
  res.json({ message: `User ${userId}, Post ${postId}` });
});

// POST /api/users/:userId/posts
router.post('/:userId/posts', (req, res) => {
  const { userId } = req.params;
  res.json({ message: `Create post for user ${userId}` });
});
```

Controller-ek használata (1/2)

Separation of Concerns

Route-ok csak az útvonalat definiálják, Controller-ek tartalmazzák a logikát.

controllers/user.controller.js

```
exports.getAllUsers = async (req, res) => {
  try {
    const users = await User.find();
    res.json(users);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
};

exports.getUserById = async (req, res) => {
  try {
    const user = await User.findById(req.params.id);
    if (!user) return res.status(404).json({ error: 'Not found' });
    res.json(user);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
};
```

Controller-ek használata (2/2)

routes/users.js

```
const express = require('express');
const router = express.Router();
const userController = require('../controllers/user.controller');

router.get('/', userController.getAllUsers);
router.get('/:id', userController.getUserById);
router.post('/', userController.createUser);
router.put('/:id', userController.updateUser);
router.delete('/:id', userController.deleteUser);

module.exports = router;
```

Tiszta kód

Route fájlok rövidek, Controller-ekben a logika.

Middleware route-okban

Auth middleware route-ra

```
const auth = require('../middlewares/auth');

router.get('/protected', auth, (req, res) => {
  res.json({ message: 'Protected data' });
});

router.post('/users', auth, validateUser, userController.createUser);

// Minden route-ra
router.use('/admin', auth, adminRoutes);
```

Több middleware

Több middleware függvényt is alkalmazhatsz sorban.

API verziókezelés (1/2)

Verziókezelés URL-ben

```
// V1 routes
const v1Routes = require('./routes/v1');
app.use('/api/v1', v1Routes);

// V2 routes
const v2Routes = require('./routes/v2');
app.use('/api/v2', v2Routes);
```

Struktúra

- routes/v1/index.js
- routes/v1/users.js
- routes/v1/products.js
- routes/v2/index.js
- routes/v2/users.js

API verziókezelés (2/2)

routes/v1/index.js

```
const express = require('express');
const router = express.Router();
const userRoutes = require('./users');
const productRoutes = require('./products');

router.use('/users', userRoutes);
router.use('/products', productRoutes);

module.exports = router;
```

Használat

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

Route Best Practices

Konvenciók

1. Használj **főneveket**: `/users`, `/products`
2. Használj **többes számot**: `/users` ✓
3. **Hierarchia**: `/users/:userId/posts/:postId`
4. **Kisbetűs URL**: `/api/users` ✓
5. **Kötőjel**: `/user-profiles` ✓
6. **Verziókezelés**: `/api/v1/users`
7. **Szűrés query-vel**: `/users?role=admin`
8. **Ne igék**: `POST /users` ✓, `/createUser` ×

Összefoglalás - Routing

- Routing = kérések irányítása végpontokra
- Route paraméterek: `/users/:id` → `req.params.id`
- Query paraméterek: `/users?page=1` → `req.query.page`
- Express Router: moduláris route szervezés
- Controller-ek: logika szétválasztása
- Middleware: route-specifikus middleware-ek
- Verziókezelés: /api/v1, /api/v2

Mi a Postman?

Postman

API fejlesztéshez és teszteléshez használt platform.

Funkciók

- HTTP kérések küldése
- Collection-ök létrehozása
- Environment változók
- Automatikus tesztek
- API dokumentáció generálás
- Csapatmunka támogatás

Postman telepítése

1. **Weboldal**: `https://www.postman.com/downloads/`
2. **Letöltés**: Desktop app (Windows, Mac, Linux)
3. **Telepítés**: Installer futtatása
4. **Bejelentkezés**: Ingyenes fiók létrehozása (opcionális)

Web verzió

Használható böngészőben is, de a desktop app gyorsabb.

Első kérés Postman-ben

1. **New** → **HTTP Request**
2. **Metódus** kiválasztása: GET, POST, PUT, DELETE
3. **URL** megadása: `http://localhost:3000/api/users`
4. **Send** gomb
5. **Response** megjelenik alul

Példa

GET `http://localhost:3000/api/users`

HTTP metódusok Postman-ben

GET kérés

- Metódus: GET
- URL: /api/users
- Body nem kell

PUT kérés

- Metódus: PUT
- URL: /api/users/:id
- Body tab → JSON

POST kérés

- Metódus: POST
- URL: /api/users
- Body tab → JSON

DELETE kérés

- Metódus: DELETE
- URL: /api/users/:id
- Body nem kell

Request Body típusok

1. **none** - Nincs body (GET, DELETE)
2. **form-data** - Multipart form (fájl feltöltés)
3. **x-www-form-urlencoded** - URL encoded form
4. **raw** - Nyers adat
   - JSON (leggyakoribb)
   - Text, XML, HTML
5. **binary** - Fájl feltöltés
6. **GraphQL** - GraphQL query

Headers beállítása

Headers tab

```
Content-Type: application/json
Authorization: Bearer YOUR_TOKEN_HERE
Accept: application/json
```

Fontos header-ek

- **Content-Type** - Body formátuma
- **Authorization** - Autentikáció
- **Accept** - Milyen formátumot várunk

Collection-ök

Mi a Collection?

Kapcsolódó API kérések csoportosítása egy helyen.

Collection struktúra

- **Users**
  - GET All Users
  - GET User by ID
  - POST Create User
  - PUT Update User
  - DELETE User
- **Products**
- **Authentication**

Collection létrehozása

1. **Collections** → + (New Collection)
2. **Név**: pl. "My REST API"
3. **Add a request** → kérések hozzáadása
4. **Mappa**: Kérések mappákba szervezése
5. **Save** - Minden kérést menteni kell

Előnyök

Gyors hozzáférés, újrafelhasználás, megosztás csapattal.

Environment Variables

Miért?

Különböző környezetek (dev, staging, prod) közötti váltás anélkül, hogy minden kérést módosítanánk.

Environment létrehozása

1. Environments → + New Environment
2. Név: `Development`
3. Változók:
   - `baseUrl`: `http://localhost:3000`
   - `apiKey`: `dev-key-123`
   - `token`: `eyJhbGc...`
4. Save

Environment használata

Változó használat

- URL: `{{baseUrl}}/api/users`
- Header: `Authorization: Bearer {{token}}`
- Body: `{"userId": "{{userId}}"}`

Váltás környezetek között

Jobb felső sarok → Environment kiválasztása

- Development
- Staging
- Production

Tests (Tesztek) (1/2)

Mi a Test?

JavaScript kód, amely a válasz után fut és ellenőrzi azt.

Tests tab - Alapvető tesztek

```
// Status code
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

// Response time
pm.test("Response time < 200ms", function () {
    pm.expect(pm.response.responseTime).to.be.below(200);
});

// JSON response
pm.test("Response is JSON", function () {
    pm.response.to.be.json;
});
```

Tests (Tesztek) (2/2)

JSON response ellenőrzés

```
const response = pm.response.json();

pm.test("Name is John", function () {
    pm.expect(response.name).to.eql("John Doe");
});

pm.test("Has email field", function () {
    pm.expect(response).to.have.property("email");
});

pm.test("Email is valid", function () {
    pm.expect(response.email).to.match(/^[^\s@]+@[^\s@]+\.[^\s@]+$/);
});
```

Test Results

Send után megjelenik a Tests panel az eredménnyel.

Pre-request Scripts

Mi az?

JavaScript kód, amely a kérés előtt fut le.

Példák

```
// Timestamp generálás
pm.environment.set("timestamp", Date.now());

// Random szám
pm.environment.set("randomNumber", Math.floor(Math.random() * 1000));

// UUID generálás
pm.environment.set("uuid", pm.variables.replaceIn('{{$guid}}'));

// Változó beállítása
pm.environment.set("userId", 123);
```

Változók mentése Tests-ből

Response-ból változó mentése

```
const response = pm.response.json();

// Token mentése
pm.environment.set("token", response.token);

// User ID mentése
pm.environment.set("userId", response.id);

// Használat következő kérésben
// URL: {{baseUrl}}/api/users/{{userId}}
// Header: Authorization: Bearer {{token}}
```

Workflow

Login → token mentése → Protected endpoint hívás

Collection Runner

Mi az?

Collection-ök automatikus futtatása sorban.

Használat

1. Collection → Run
2. Kérések sorrendje beállítása
3. Iterations: hányszor fusson
4. Delay: várakozás kérések között
5. Run Collection

Előny

Teljes API tesztelése egy gombnyomással.

Dokumentáció generálás

Automatic documentation

Postman automatikusan generál dokumentációt Collection-ből.

Lépések

1. Collection → View Documentation
2. Publish documentation (opcionális)
3. Megosztás csapattal vagy publikusan

Előny

Mindig naprakész dokumentáció a Collection alapján.

Összefoglalás - Postman

- Postman = API fejlesztési platform
- HTTP kérések küldése és tesztelése
- Collection-ök: kérések csoportosítása
- Environment: változók különböző környezetekhez
- Tests: automatikus válasz ellenőrzés
- Pre-request Scripts: kérés előtti logika
- Collection Runner: teljes API tesztelés
- Dokumentáció: automatikus generálás