Webtechnológia és webalkalmazás-fejlesztés - Auth

Authentikáció és autorizáció

Magda Donát

Széchenyi István Egyetem, Győr

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

2026. február 24.

Mi az authentikáció?

Definíció

Folyamat, amely során a rendszer ellenőrzi egy felhasználó identitását.

- **Kérdés**: Ki vagy?
- **Cél**: Felhasználó identitásának megállapítása
- **Eredmény**: Sikeres vagy sikertelen azonosítás

Fontos!

Authentikáció $\neq$ Autorizáció!

Authentikáció vs. Autorizáció

Authentikáció

- Ki vagy?
- Bejelentkezés
- Példa: Jelszó ellenőrzés

Autorizáció

- Mit tehetsz?
- Jogosultságok
- Példa: Admin jogosultság

`Authentikáció` $\rightarrow$ `Autorizáció`

Authentikációs módszerek

1. **Tudás alapú**
   - Példa: jelszó, PIN kód
2. **Tulajdon alapú**
   - Példa: okostelefon, hardver token
3. **Inherencia alapú**
   - Példa: ujjlenyomat, arcfelismerés

Többfaktoros authentikáció (MFA)

Multi-Factor Authentication

Két vagy több független faktor kombinációja.

- **2FA példák**:
  - Jelszó + SMS kód
  - Jelszó + Authenticator app
  - Jelszó + ujjlenyomat
- **Előnyök**:
  - Magasabb biztonság
  - Védelem jelszó kompromittálódás esetén

Jelszó alapú authentikáció

Leggyakoribb módszer

Felhasználónév + jelszó párossal történő azonosítás.

- **Folyamat**:
  1. Felhasználó megadja username + password
  2. Rendszer összehasonlítja a hash-sel
  3. Sikeres/sikertelen válasz
- **Biztonság**:
  - Soha ne tároljuk plain text-ben!
  - Erős hash (bcrypt, Argon2)
  - Salt minden jelszóhoz

Token alapú authentikáció

Modern megközelítés

Bejelentkezés után szerver tokent generál, kliens minden kéréshez csatolja.

1. Bejelentkezés (username + password)
2. Szerver ellenőrzi
3. Token generálás
4. Kliens eltárolja (localStorage, cookie)
5. Token csatolása minden kéréshez
6. Szerver validálja

- **Típusok**: JWT, OAuth, API keys
- **Előnyök**: Stateless, skálázható

Session alapú authentikáció

Hagyományos megközelítés

Bejelentkezési információkat a szerver tárolja session-ökben.

1. Bejelentkezés
2. Szerver session létrehozás
3. Session ID küldése cookie-ban
4. Kliens csatolja minden kéréshez
5. Szerver azonosítja a felhasználót

Előnyök:

- Biztonságosabb
- Könnyű visszavonás

Hátrányok:

- Stateful
- Nehezebb skálázás

OAuth 2.0 / OpenID Connect

Harmadik fél authentikáció

Bejelentkezés külső szolgáltatókkal (Google, Facebook, GitHub).

- **OAuth 2.0**: Autorizációs framework
- **OpenID Connect**: Authentikációs réteg
- **Előnyök**:
  - Nincs jelszó kezelés
  - Jobb UX
  - Biztonságos protokoll
- **Használat**:
  - Social login
  - SSO megoldások

Biztonsági megfontolások

Gyakori sebezhetőségek

- Gyenge jelszavak
- Nincs rate limiting
- Token kiszivárogtatás
- Man-in-the-middle (nincs HTTPS)

Best Practices

- HTTPS mindig
- Erős jelszó policy
- Rate limiting
- MFA
- Token lejárati idő
- CSRF védelem

Mi az autorizáció?

Definíció

Meghatározza, hogy egy azonosított felhasználó milyen erőforrásokhoz férhet hozzá és milyen műveleteket végezhet.

- **Kérdés**: Mit tehetsz?
- **Cél**: Jogosultságok szabályozása
- **Eredmény**: Engedélyezett vagy tiltott művelet

Fontos!

Autorizáció az authentikáció után történik.

Autorizációs modelljei

1. **RBAC (Role-Based Access Control)**
   - Szerepkörök alapján
   - Példa: Admin, Editor, User
2. **ABAC (Attribute-Based)**
   - Attribútum alapján (user, resource, context)
   - Példa: "Department = IT és 9-17 között"
3. **ACL (Access Control List)**
   - Erőforrás-szintű lista
   - Példa: fájlrendszer jogosultságok

RBAC - Szerepkör alapú

Lényege

Felhasználók szerepköröket kapnak, ez határozza meg a jogosultságokat.

Előnyök:

- Könnyű adminisztráció
- Átlátható
- Gyors ellenőrzés

Hátrányok:

- Skálázhatóság
- Kevesebb rugalmasság

ABAC - Attribútum alapú

Lényege

Hozzáférési szabályok attribútumok alapján.

- **User**: szerepkör, osztály, beosztás
- **Resource**: tulajdonos, típus
- **Context**: idő, hely, eszköz

Példa szabály

```
(role=Manager) AND (resource.owner = user) AND (time < 18:00)
```

ACL - Access Control List

Lényege

Erőforrásonként tárolt lista a hozzáférési jogokról.

- **Példa**: fájlrendszer (read, write, execute)
- **Előny**: finém hozzáférés
- **Hátrány**: nagy rendszereknelnehéz kezelni

Autorizáció webalkalmazásban

Tipikus folyamat

Szerver minden kérésnél ellenőrzi a jogosultságot.

1. Authnentikáció után szerepkör kiosztás
2. Tokenben vagy session-ben tárolás
3. Middleware ellenőrzi a hozzáférést
4. Nincs jog: `403 Forbidden`

HTTP státuszkód

401 = nincs auth, 403 = nincs jogosultság

Autorizációs middleware (RBAC)

Egyszerű szerepkör ellenőrzés

```
const authorize = (...roles) => {
  return (req, res, next) => {
    const { role } = req.user; // pl. JWT-ből
    if (!roles.includes(role)) {
      return res.status(403).json({ error: 'Forbidden' });
    }
    next();
  };
};

app.get('/admin', auth, authorize('Admin'), (req, res) => {
  res.json({ data: 'Admin content' });
});
```

Policy alapú autorizáció (ABAC)

Szabály központú hozzáférés

```
const canAccess = (user, resource) => {
  return user.department === resource.department &&
         user.level >= resource.requiredLevel &&
         new Date().getHours() < 18;
};

app.get('/reports/:id', auth, async (req, res) => {
  const report = await getReport(req.params.id);
  if (!canAccess(req.user, report)) {
    return res.status(403).json({ error: 'Forbidden' });
  }
  res.json(report);
});
```

Legkisebb jogosultság elve

Principle of Least Privilege

Mindenki csak a feladatohoz szükséges minimális jogokat kapja.

- Csökkenti a kockázatot
- Korlátozza a hibák hatását
- Egyszerűbb auditálás

Gyakori hiba

Mindenki admin = gyors fejlesztés, de veszélyes élesben!

Autorizáció Best Practices

- Minden érzékeny végpontot ellenőrizz
- Middleware vagy policy réteg
- Role/policy központi kezelés
- Logolás
- Admin funkciók külön kezelése
- RBAC + ABAC kombináció

Összefoglalás - Autorizáció

- Autorizáció = hozzáférések és jogosultságok kezelése
- 401 vs 403: authentikáció hiánya vs jogosultság hiánya
- Fő modellek: RBAC, ABAC, ACL
- Middleware-ekkel megvalósítható a gyakorlatban
- Least Privilege elv alkalmazása

Mi az a hash?

Definíció

Tetszőleges hosszúságú bemenetet rögzített méretű lenyomattá alakít.

- Egyirányú függvény (nem visszafejthető)
- Gyorsan számolható
- Kis változás teljesen más hasht eredményez

Fontos

Hash $\neq$ titkosítás: nem visszafejthető!

Hash tulajdonságok

- **Deterministic**: azonos bemenet = azonos kimenet
- **Preimage resistance**: nehéz visszafejteni
- **Second preimage**: nehéz másik bemenetet találni
- **Collision resistance**: nehéz két különböző bemenetet találni azonos hash-sel

Példa

```
hash("jelszo") = 5f4dcc3b5aa765d61d8327deb882cf99
```

Gyenge vs. erős hash

Gyenge hash

- MD5, SHA1 (elavult)
- Gyors → brute force könnyebb
- Ütközések ismertek

Erős hash

- bcrypt, Argon2, scrypt
- Lassú, konfigurálható
- Salt + work factor támogatás

Jelszavakhoz

Soha ne használj általános hash-t (MD5, SHA1) jelszavak tárolásához!

Salt és Pepper

Miért fontos?

Salt: véletlen érték, amit a jelszóhoz adunk hash előtt.

- Védelem rainbow table ellen
- Minden jelszónál egyedi salt
- Salt-et tárolhatjuk adatbázisban

Pepper

Közös titkos kulcs, szerver konfigban tárolva.

Password hashing algoritmusok

- **bcrypt**
  - Beépített salt és work factor
- **scrypt**
  - Memória-igényes → GPU támadások ellen jobb
- **Argon2**
  - Modern szabvány (PHC winner)
  - Paraméterezhető (time, memory, parallelism)

bcrypt használat (Node.js)

Hash készítés

```
const bcrypt = require('bcrypt');
const saltRounds = 12;

const hashPassword = async (password) => {
  const salt = await bcrypt.genSalt(saltRounds);
  return bcrypt.hash(password, salt);
};
```

Ellenőrzés

```
const isValid = await bcrypt.compare(password, storedHash);
```

Hash-elés felhasználása

- Jelszó tárolás
- Fájl integritás ellenőrzés
- Digitális aláírások
- Cache kulcsok

Megjegyzés

Authentikációban: jelszó soha ne legyen visszaolvasható.

Hash Best Practices

- Jelszhoz **bcrypt/Argon2/scrypt**
- Egyedi salt minden jelszóhoz
- Work factor beállítása (pl. bcrypt 10-12)
- Soha ne plain text
- Rate limiting + lockout

Összefoglalás - Hash

- Hash = egyirányú lenyomat
- Jelszavakhoz speciális algoritmus kell
- Salt és pepper növeli a biztonságot
- Argon2 a legajánlottabb modern választás
- Hash nem titkosítás!

Mi az a Cookie?

HTTP Cookie definíció

Kis méretű adat (max 4KB), amit szerver küld böngészőnek, automatikusan visszaküldődik.

- Session kezelés, preferenciák, tracking
- Beállítható élettartam

Cookie beállítása Express-ben

```
// Egyszerű cookie
res.cookie('session', 'abc123');

// Opciókkal
res.cookie('user', 'john', {
  maxAge: 900000,  // 15 perc ms-ban
  httpOnly: true,  // JS nem fér hozzá
  secure: true,    // csak HTTPS
  sameSite: 'strict' // CSRF védelem
});
```

Cookie attribútumok

- **Domain**: `.example.com` - subdomain-ek
- **Path**: `/api` - URL path scope
- **Expires/Max-Age**: Lejárat ideje
- **HttpOnly**: JS hozzáférés blokkolása (XSS védelem)
- **Secure**: Csak HTTPS-en küldve
- **SameSite**: `Strict|Lax|None` - CSRF védelem

Cookie típusok

1. **Session Cookie**: Nincs Expires/Max-Age, böngésző bezáráskor törlődik
2. **Persistent Cookie**: Van lejárati idő, túléli böngésző bezárást
3. **Secure Cookie**: Csak HTTPS-en
4. **HttpOnly Cookie**: JS nem fér hozzá
5. **SameSite Cookie**: CSRF támadás ellen
6. **Third-party Cookie**: Más domain-ről

SameSite Cookie attribútum

Strict	Cookie csak saját site kérésekkel küldve. Legjobb CSRF védelem.
Lax	GET kéréseknél küldi cross-site. Default modern böngészőkben.
None	Mindig küldi (Secure kötelező). Third-party használathoz.

CSRF véd

SameSite=Strict/Lax megakadályozza cross-site támadásokat.

Cookie vs localStorage vs sessionStorage

	Cookie	localStorage	sessionStorage
Kapacitás	4KB	5-10MB	5-10MB
Lejárat	Beállítható	Nincs	Tab bezárás
HTTP-ben	Igen	Nem	Nem
XSS véd	HttpOnly	Nem	Nem

Ajánlás

Érzékeny: HttpOnly Cookie, **Publikus**: localStorage

Cookie olvasása Express-ben

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

app.get('/profile', (req, res) => {
  const sessionId = req.cookies.session;
  res.json({ sessionId });
});
```

Cookie törlése

```
app.post('/logout', (req, res) => {
  res.clearCookie('session');
  res.json({ message: 'Logged out' });
});
```

Signed Cookies

Integritás védelem

HMAC aláírással ellátott cookie, megakadályozza manipulációt.

```
app.use(cookieParser('secret-key'));

app.get('/set', (req, res) => {
  res.cookie('userId', '123', { signed: true });
  res.send('OK');
});

app.get('/get', (req, res) => {
  const id = req.signedCookies.userId;
  res.json({ id });
});
```

XSS vs CSRF

XSS (Cross-Site Scripting)

- JS injection támadás
- Cookie lopás `document.cookie`
- **Védelem:** HttpOnly cookie

CSRF (Cross-Site Request Forgery)

- Hamis kérés küldés
- Cookie auto-send kihasználása
- **Védelem:** SameSite, CSRF token

CSRF Token védelem

```
const csrf = require('csurf');
app.use(csrf({ cookie: true }));

app.get('/form', (req, res) => {
  res.render('form', { csrfToken: req.csrfToken() });
});

app.post('/submit', (req, res) => {
  res.send('Valid!');
});
```

Cookie Security Best Practices

1. **HttpOnly**: Érzékeny cookie-khoz mindig
2. **Secure**: Éles környezetben HTTPS-el
3. **SameSite**: Strict vagy Lax CSRF ellen
4. **Short expiration**: Session token-ekhez rövid lejárat
5. **Domain scope**: Csak szükséges domain-hez
6. **Signed cookies**: Integritás ellenőrzéshez

Mi az a session?

Definíció

Szerver-oldali állapot, amely felhasználói adatokat tárol bejelentkezés után.

- **Stateful** megközelítés
- Kliens csak **session ID**-t tárol (cookie)
- Tartalom a szerveren (memória/DB/Redis)

Fontos

Session auth nem token-alapú, szerver kezeli az állapotot.

Session vs Token

Session

- Szerver tárolja
- Cookie-ban ID
- Könnyű visszavonás
- Nehezebb skálázás

Token (JWT)

- Stateless
- Kliens tárolja
- Nehezebb visszavonás
- Könnyebb skálázás

Session $\leftrightarrow$ JWT: kontroll vs skálázhatóság

Session folyamat

1. Bejelentkezés (username + password)
2. Adatok ellenőrzése
3. **Session** létrehozás
4. Session ID cookie-ban
5. Kliens csatolja minden kéréshez
6. Szerver ID alapján azonosít

Megjegyzés

HTTPS kötelező, különben session ID elliopható!

Session tárolás

Hol tároljuk?

Session adatokat szerveren, több lehetőség:

- **In-memory** - gyors, nem skálázható
- **Redis** - gyors, skálázható, TTL
- **DB** - tartós, lassabb
- **Distributed cache** - nagy rendszerekhez

Best Practice

Élesben ne használj in-memory store-t!

Redis Session Store - Telepítés

Redis előnyei session tároláshoz

Gyors, skálázható, beépített TTL (Time To Live) támogatás

```
npm install express-session connect-redis redis

// Redis kliens és session store
const session = require('express-session');
const RedisStore = require('connect-redis').default;
const { createClient } = require('redis');

const redisClient = createClient({
  host: 'localhost',
  port: 6379
});
redisClient.connect();
```

Redis Session - Konfiguráció

```
app.use(session({
  store: new RedisStore({ client: redisClient }),
  secret: process.env.SESSION_SECRET,
  resave: false,
  saveUninitialized: false,
  cookie: {
    httpOnly: true,
    secure: true,  // HTTPS
    maxAge: 1000 * 60 * 60 * 24  // 1 nap
  }
}));

app.post('/login', async (req, res) => {
  const user = await authenticateUser(req.body);
  req.session.userId = user.id;
  req.session.role = user.role;
  res.json({ success: true });
});
```

Redis Session - Műveletek

```
// Session olvasása
app.get('/profile', (req, res) => {
  if (!req.session.userId) {
    return res.status(401).json({ error: 'Not logged in' });
  }
  res.json({ userId: req.session.userId });
});

// Session módosítása
app.post('/settings', (req, res) => {
  req.session.theme = req.body.theme;
  res.json({ success: true });
});

// Session törlése (logout)
app.post('/logout', (req, res) => {
  req.session.destroy((err) => {
    if (err) return res.status(500).send('Error');
    res.clearCookie('connect.sid');
    res.json({ message: 'Logged out' });
  });
});
```

Redis Session előnyei

Előnyök:

- Villámgyors (in-memory)
- Automatikus TTL támogatás
- Skálázható (cluster mode)
- Persistence opcionális
- Pub/Sub támogatás

Használati esetek:

- Session store
- Rate limiting
- Real-time analytics
- Cache layer
- Message queue

Session élettartam

Session Timeout

Session-ök lejárnak idő vagy inaktivitás után.

- **Absolute**: fix lejárat (pl. 24h)
- **Idle**: aktivitás hiánya
- **Sliding**: aktivitás meghosszabbítja

Példa

Online bank: 5 perc inaktivitás → kiléptetés.

Express session példa

express-session használata

```
const session = require('express-session');

app.use(session({
  secret: 'super-secret-key',
  resave: false,
  saveUninitialized: false,
  cookie: {
    httpOnly: true,
    secure: true,
    maxAge: 1000 * 60 * 60 // 1 óra
  }
}));

app.post('/login', (req, res) => {
  req.session.userId = user.id;
  res.json({ message: 'Logged in' });
});
```

Cookie biztonság

Mitől biztonságos?

Session ID cookie biztonságos beállításokkal.

- **HttpOnly**: JS nem fér hozzá
- **Secure**: csak HTTPS
- **SameSite**: CSRF védelem
- **Rotálás**: ID frissítés (fixation ellen)

Session Fixation

Támadó előre beállított ID-t "ráerőltet" a felhasználóra.

Session alapú autorizáció

Jogosultságok session-ben

A session tárolhatja a felhasználó szerepköreit és jogosultságait.

- **Példa**: `req.session.role = 'Admin'`
- Middleware ellenőrzi a role-t
- Ha nincs jogosultság: `403 Forbidden`

Minta ellenőrzés

```
if (req.session.role !== 'Admin') return res.status(403);
```

Session Best Practices

- HTTPS használata mindenhol
- HttpOnly + Secure + SameSite cookie beállítások
- Session store: Redis vagy DB
- Rövid idle timeout érzékeny rendszereknél
- Session ID rotálás bejelentkezéskor
- Rate limiting és brute force védelem

Összefoglalás - Session

- Session = szerver-oldali állapot (cookie csak ID)
- Kényelmes, de skálázásnál kihívás
- Cookie biztonsági beállítások kritikusak
- **Redis a leggyakoribb session store** - gyors, skálázható
- Timeout és rotáció védelem session támadások ellen
- Redis TTL automatikus session lejárathoz

Produkciós környezet

Redis cluster + persistence + backup = megbízható session kezelés

Mi az a JWT?

JSON Web Token (RFC 7519)

Kompakt, önálló token információ biztonságos továbbításához.

- Digitálisan aláírt, Base64URL kódolt
- Használat: authentikáció, stateless session
- **NEM** titkosított - payload olvasható!

JWT formátum

HEADER.PAYLOAD.SIGNATURE

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZ
SI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P

- Header: Algoritmus + token típus
- Payload: Claims (user adatok)
- Signature: Integritás védelem

JWT Header & Payload

Header:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Payload:

```
{
  "sub": "1234",
  "name": "John",
  "iat": 1516239022,
  "exp": 1516242622
}
```

Fontos

Ne tároljunk jelszót payload-ban!

JWT Signature

```
HMACSHA256(
  base64(header) + "." + base64(payload),
  secret
)
```

HMAC: Egy kulcs, gyors (HS256)

RSA: Privát+publikus, biztonságosabb (RS256)

JWT generálás Node.js

```
const jwt = require('jsonwebtoken');

const payload = { userId: user.id, role: user.role };

const token = jwt.sign(
  payload,
  process.env.JWT_SECRET,
  { expiresIn: '1h' }
);

res.json({ token });
```

JWT validálás Node.js

```
const token = req.headers.authorization?.split(' ')[1];

if (!token) return res.status(401).json({ error: 'No token' });

try {
  const decoded = jwt.verify(token, process.env.JWT_SECRET);
  req.user = decoded;
  next();
} catch (err) {
  res.status(403).json({ error: 'Invalid' });
}
```

JWT middleware

```
const authJWT = (req, res, next) => {
  const token = req.headers.authorization?.split(' ')[1];
  if (!token) return res.status(401).send('Token required');

  jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
    if (err) return res.status(403).send('Invalid');
    req.user = user;
    next();
  });
};

app.get('/protected', authJWT, (req, res) => {
  res.json({ data: 'Secret', user: req.user });
});
```

JWT vs Session

JWT:

- Stateless
- Skálázható
- Cross-domain
- Nehéz visszavonni
- Token méret

Session:

- Stateful
- Szerver tárolja
- Könnyű visszavonás
- Nehezebb skálázás
- Kis cookie

Access & Refresh Token

Két token stratégia

Access token: Rövid élet (15 perc), API hozzáférés
Refresh token: Hosszú élet (7 nap), új access token generálás

1. Login → Access + Refresh token
2. API kérés Access token-nel
3. Access lejár → Refresh-el új Access-t kér
4. Refresh lejár → Újra login

Refresh Token példa

```
app.post('/refresh', (req, res) => {
  const { refreshToken } = req.body;
  if (!refreshToken) return res.sendStatus(401);

  jwt.verify(refreshToken, REFRESH_SECRET, (err, user) => {
    if (err) return res.sendStatus(403);

    const accessToken = jwt.sign(
      { userId: user.userId },
      ACCESS_SECRET,
      { expiresIn: '15m' }
    );
    res.json({ accessToken });
  });
});
```

JWT Security Best Practices

1. Rövid lejárat (15-60 perc)
2. Erős secret kulcs (min 256 bit)
3. HTTPS kötelező
4. Ne tároljunk érzékeny adatot payload-ban
5. Validáld mindig az `exp` és `iat` claim-eket
6. Használj RS256-ot HS256 helyett production-ben
7. Refresh token rotation
8. Token blacklist megfontolása

Mi az OAuth 2.0?

Definíció

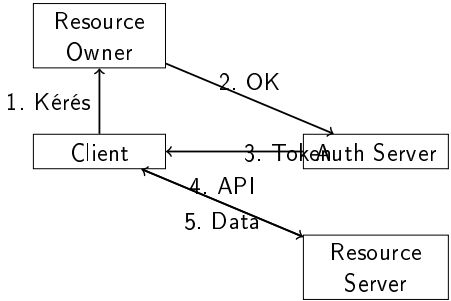
Autorizációs protokoll - korlátozott hozzáférés jelszó megosztása nélkül.

- RFC 6749 (2012)
- Google, Facebook, GitHub használja
- Példa: App hozzáfér Google Drive-hoz jelszó nélkül

OAuth 2.0 szerepkörök

1. **Resource Owner**: Felhasználó (birtokolja erőforrást)
2. **Client**: Alkalmazás (hozzáférést kér)
3. **Authorization Server**: Token kiállító
4. **Resource Server**: Védett erőforrás tároló

OAuth 2.0 Flow



Grant Types

1. **Authorization Code**: Szerver app, legbiztonságosabb
2. **Implicit**: Elavult!
3. **Resource Owner Password**: Közvetlen jelszó, megbízható app
4. **Client Credentials**: Machine-to-machine
5. **PKCE**: Modern mobil/SPA

Authorization Code Flow lépések

1. Kliens átirányít: `/authorize?client_id=&redirect_uri=&scope=`
2. Felhasználó bejelentkezik és hozzájárul
3. Auth Server visszairányít: `redirect_uri?code=AUTH_CODE`
4. Kliens kicseréli code-ot: POST `/token`
5. Auth Server ad access token-t (+ refresh)
6. Kliens használja tokent: `Authorization: Bearer <token>`

Authorization Code - Auth kérés

```
GET /oauth/authorize?
  response_type=code&
  client_id=YOUR_CLIENT_ID&
  redirect_uri=https://app.com/callback&
  scope=read write&
  state=xyz123


// Callback
GET https://app.com/callback?
  code=AUTH_CODE&
  state=xyz123
```

Authorization Code - Token csere

```
POST /oauth/token
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&
code=AUTH_CODE&
redirect_uri=https://app.com/callback&
client_id=YOUR_ID&
client_secret=YOUR_SECRET


// Válasz
{
  "access_token": "eyJ...",
  "expires_in": 3600,
  "refresh_token": "tGzv..."
}
```

PKCE (Proof Key for Code Exchange)

Miért?

Authorization Code Flow biztonságosabbá tétele mobil/SPA app-okhoz.

1. Kliens generál `code_verifier` (random string)
2. Hash: `code_challenge = SHA256(code_verifier)`
3. Auth kéréshez csatol: `code_challenge`
4. Token kérésnél küldi: `code_verifier`
5. Server validálja: `SHA256(verifier) == challenge`

Védelem

Megakadályozza authorization code ellopását.

Scope és Permission

Scope

Jogosultság amit app kér. Példa: `read:user write:repo`

- Felhasználó látja mit kér az app
- Csak kért scope-okat kapja meg
- Token tartalmazza scope-okat

GitHub példa

`repo, user, gist, notifications`

Access Token használata

```
// API kérés Bearer token-nel
GET /api/user/profile
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...


// Express validálás
app.get('/api/user', (req, res) => {
  const token = req.headers.authorization?.split(' ')[1];
  // Token validálás...
  res.json({ user: userData });
});
```

OAuth 2.0 Security

1. **HTTPS kötelező**: Minden kommunikáció
2. **State parameter**: CSRF védelem
3. **Redirect URI validation**: Előre regisztrált URI
4. **Short token lifetime**: Access token max 1 óra
5. **PKCE használata**: Mobil/SPA app-okhoz
6. **Scope limitation**: Csak szükséges jogok

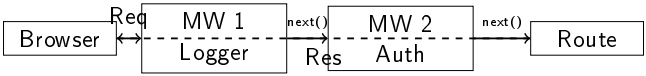
Mi az a middleware?

Definíció

Függvény HTTP kérés-válasz között, hozzáfér `req`, `res`, `next`-hez.

- Láncolható
- Kérés előfeldolgozás, válasz utófeldolgozás
- Kérés megszakítható

Middleware működése



Fontos

`next()` nélkül kérés megáll!

Middleware típusok

1. **Application-level**: `app.use()`
2. **Router-level**: `router.use()`
3. **Error-handling**: `(err, req, res, next)`
4. **Built-in**: `express.json()`, `express.static()`
5. **Third-party**: `cors`, `helmet`, `morgan`

Logger middleware

```
const logger = (req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
  next();
};

app.use(logger);

app.get('/', (req, res) => {
  res.json({ msg: 'Hello' });
});

// Kimenet:
// [2026-02-24T10:30:15.000Z] GET /
```

Request timing middleware

```
const timer = (req, res, next) => {
  req.startTime = Date.now();
  res.on('finish', () => {
    const duration = Date.now() - req.startTime;
    console.log(`${req.method} ${req.url} - ${duration}ms`);
  });
  next();
};

app.use(timer);

// GET /api/users - 145ms
```

Validation middleware

```
const validateEmail = (req, res, next) => {
  const { email } = req.body;
  const re = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

  if (!email || !re.test(email)) {
    return res.status(400).json({ error: 'Invalid email' });
  }
  next();
};

app.post('/register', validateEmail, (req, res) => {
  res.json({ success: true });
});
```

Auth middleware

```
const auth = (req, res, next) => {
  const token = req.headers.authorization?.split(' ')[1];

  if (!token) {
    return res.status(401).json({ error: 'Token required' });
  }

  try {
    const decoded = jwt.verify(token, SECRET);
    req.user = decoded;
    next();
  } catch (err) {
    res.status(403).json({ error: 'Invalid token' });
  }
};

app.get('/protected', auth, (req, res) => {
  res.json({ user: req.user });
});
```

Role-based middleware

```
const requireRole = (role) => {
  return (req, res, next) => {
    if (req.user.role !== role) {
      return res.status(403).json({ error: 'Forbidden' });
    }
    next();
  };
};

app.delete('/user/:id', auth, requireRole('admin'), (req, res) => {
  res.json({ deleted: true });
});
```

Error handling middleware

```
// Mindig utolsóként!
app.use((err, req, res, next) => {
  console.error(err.stack);

  res.status(err.status || 500).json({
    error: err.message || 'Internal Server Error'
  });
});

// Használat
app.get('/error', (req, res, next) => {
  const err = new Error('Something broke!');
  err.status = 500;
  next(err);
});
```

CORS middleware

```
const cors = require('cors');

// Minden origin engedélyezése
app.use(cors());

// Specifikus beállítás
app.use(cors({
  origin: 'https://frontend.com',
  credentials: true,
  methods: ['GET', 'POST', 'PUT', 'DELETE']
}));
```

Built-in middleware

- `express.json()`: JSON body parsing
- `express.urlencoded()`: URL-encoded body parsing
- `express.static()`: Static fájlok kiszolgálása

Használat

```
app.use(express.json());
app.use(express.static('public'));
```

Third-party middleware

helmet	HTTP security headers
morgan	HTTP request logger
compression	Response compression (gzip)
cookie-parser	Cookie parsing
express-session	Session kezelés
passport	Authentication

Middleware best practices

1. Mindig hívd meg `next()`-et (vagy küldjél választ)
2. Error middleware utolsóként
3. Sorrendiség számít!
4. Global middleware-ek előre (`cors`, `helmet`)
5. Route-specifikus middleware-ek közvetlenül route-hoz
6. Async middleware-eknél használj try-catch-et

Mi az a Service Layer?

Definíció

Tervezési minta - üzleti logika elválasztása controller-ektől és data layer-től.

- Separation of Concerns
- Reusability (újrafelhasználható logika)
- Testability (könnyebb tesztelés)
- Maintainability

Architektúra

Controller → Service → Repository → Database

Háromrétegű architektúra

Presentation

Controllers, Routes
HTTP req/res
Validáció

Business Logic

Services
Üzleti szabályok
Orchestration

Data Access

Repository
ORM
Database

Fontos

Controller **NEM** tartalmaz üzleti logikát, csak delegál!

Service Layer előnyei

1. **Separation**: Tiszta felelősségi körök
2. **Reusability**: Több controller használhatja
3. **Testability**: Könnyen unit tesztelhető
4. **Maintainability**: Változások izoláltak

AuthService - Interface

```
class AuthService {
  async register(userData) { ... }
  async login(email, password) { ... }
  async logout(userId) { ... }
  async refreshToken(refreshToken) { ... }
  async verifyToken(token) { ... }
  async resetPassword(email) { ... }
}
```

AuthService - Register

```
class AuthService {
  async register(userData) {
    const exists = await this.userRepo.findByEmail(userData.email);
    if (exists) throw new Error('User exists');

    const hashed = await bcrypt.hash(userData.password, 10);
    const user = await this.userRepo.create({
      ...userData,
      password: hashed
    });

    return { id: user.id, email: user.email };
  }
}
```

AuthService - Login

```
async login(email, password) {
  const user = await this.userRepo.findByEmail(email);
  if (!user) throw new Error('Invalid credentials');

  const valid = await bcrypt.compare(password, user.password);
  if (!valid) throw new Error('Invalid credentials');

  const token = jwt.sign(
    { userId: user.id, role: user.role },
    process.env.JWT_SECRET,
    { expiresIn: '1h' }
  );

  return { token, user: { id: user.id, email: user.email } };
}
```

Controller használja Service-t

```
class AuthController {
  constructor(authService) {
    this.authService = authService;
  }

  async register(req, res) {
    try {
      const user = await this.authService.register(req.body);
      res.status(201).json(user);
    } catch (err) {
      res.status(400).json({ error: err.message });
    }
  }

  async login(req, res) {
    try {
      const result = await this.authService.login(req.body.email, req.body.password);
      res.json(result);
    } catch (err) {
      res.status(401).json({ error: err.message });
    }
  }
}
```

UserService példa

```
class UserService {
  constructor(userRepo) {
    this.userRepo = userRepo;
  }

  async getById(id) {
    const user = await this.userRepo.findById(id);
    if (!user) throw new Error('User not found');
    return user;
  }

  async update(id, data) {
    const user = await this.getById(id);
    return await this.userRepo.update(id, data);
  }

  async delete(id) {
    await this.getById(id);
    return await this.userRepo.delete(id);
  }
}
```

Dependency Injection

```
// Repository
class UserRepository {
  async findById(id) { /* DB query */ }
  async create(data) { /* DB insert */ }
}

// Service
class UserService {
  constructor(userRepository) {
    this.userRepo = userRepository;
  }
  // ...
}

// DI container
const userRepo = new UserRepository();
const userService = new UserService(userRepo);
const userController = new UserController(userService);
```

Service Layer Best Practices

1. Service-ek ne dependáljanak controller-ektől
2. Egy service = egy domain (User, Auth, Order)
3. Dependency Injection használata
4. Service-ek ne ismerjék HTTP-t (req, res)
5. Hibakezelés service-ben (throw Error)
6. Transaction logika service-ben
7. Async/await következetes használata

Transaction példa Service-ben

```
class OrderService {
  async createOrder(userId, items) {
    const transaction = await db.transaction();

    try {
      const order = await this.orderRepo.create({ userId }, transaction);

      for (const item of items) {
        await this.orderItemRepo.create({ orderId: order.id, ...item }, transaction);
      }

      await transaction.commit();
      return order;
    } catch (err) {
      await transaction.rollback();
      throw err;
    }
  }
}
```

Testing Service Layer

- Service-ek unit tesztelése mock repository-val
- Ne kelljen DB a unit teszthez
- Integration teszt valódi DB-vel