Webtechnológia és webalkalmazás-fejlesztés - Adatbázis

Adatbázis

Magda Donát

Széchenyi István Egyetem, Győr

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

2026. február 17.

CRUD műveletek

Mi az a CRUD?

- **C**reate - Létrehozás
- **R**ead - Olvasás
- **U**pdate - Módosítás
- **D**elete - Törlés

Adatbázis műveletek

Az alapvető adatbázis műveletek, amelyek minden adatkezelő rendszerben megtalálhatók.

Create - Létrehozás

SQL - INSERT

```sql
INSERT INTO users (name, email, age)
VALUES ('John Doe', 'john@example.com', 30);
```

REST API - POST

```
POST /api/users
Content-Type: application/json

{
  "name": "John Doe",
  "email": "john@example.com",
  "age": 30
}
```

Read - Olvasás

SQL - SELECT

```sql
-- Összes rekord
SELECT * FROM users;

-- Egy adott rekord
SELECT * FROM users WHERE id = 1;

-- Szűrés és rendezés
SELECT name, email FROM users
WHERE age > 25
ORDER BY name ASC;
```

Read - REST API

REST API - GET

```
// Összes felhasználó
GET /api/users

// Egy adott felhasználó
GET /api/users/1

// Szűrés query paraméterekkel
GET /api/users?age=30&sort=name
```

Update - Módosítás

SQL - UPDATE

```sql
UPDATE users
SET email = 'newemail@example.com', age = 31
WHERE id = 1;
```

REST API - PUT/PATCH

```
// Teljes frissítés
PUT /api/users/1
{ "name": "John Doe", "email": "new@example.com", "age": 31 }

// Részleges frissítés
PATCH /api/users/1
{ "email": "new@example.com" }
```

Delete - Törlés

SQL - DELETE

```sql
-- Egy adott rekord törlése
DELETE FROM users WHERE id = 1;

-- Minden rekord törlése (óvatosan!)
DELETE FROM users;
```

REST API - DELETE

```
DELETE /api/users/1
```

CRUD vs HTTP metódusok

CRUD	HTTP	SQL	Leírás
Create	POST	INSERT	Új erőforrás létrehozása
Read	GET	SELECT	Erőforrás lekérdezése
Update	PUT/PATCH	UPDATE	Erőforrás módosítása
Delete	DELETE	DELETE	Erőforrás törlése

Mi az ORM?

Object-Relational Mapping

- Objektumok és relációs táblák közötti leképezés
- Osztályok → táblákat, objektumok → sorokat jelentenek
- Átfedő absztrakció az SQL felett

Példa

- `User` osztály → `users` tábla
- `user.email` mezők → `email` oszlop

Miért használunk ORM-et?

- Kevesebb boilerplate SQL, gyorsabb fejlesztés
- Típusbiztonság, IDE támogatás, egységes API
- Adatbázis-portabilitás (pl. PostgreSQL → MySQL)
- Biztonság: paraméterizált lekérdezések alapból

Korlátai és kockázatai

- Teljesítmény: rosszul írt lekérdezések, N+1 probléma
- "Leaky abstraction": komplex SQL-t nehéz elrejteni
- Nem minden adatbázis funkció elérhető el könnyen
- Extra tanulási görbe és konfiguráció

Alap fogalmak

Entity

Egy tábla sorát leíró objektum.

Repository

Adathozzáférési réteget kapszuláz.

Unit of Work

Változások követése és egyben mentése.

Session/Context

Kapcsolat az adatbázissal, cache és tranzakciók.

Kapcsolatok leképezése

- One-to-One: `User` → `Profile`
- One-to-Many: `Author` → `Post[]`
- Many-to-Many: `Student` → `Course[]`

Felelősség

A helyes kulcsok, indexek és megszorítások ugyanúgy fontosak, mint ORM nélkül.

SQL vs ORM lekérdezés

SQL

```sql
SELECT id, name, email
FROM users
WHERE active = true
ORDER BY name ASC;
```

ORM (példa)

```
List<User> users = db.users()
  .where(u -> u.active == true)
  .orderBy(u -> u.name)
  .select("id", "name", "email")
  .toList();
```

Lazy vs Eager betöltés

Lazy loading

Csak akkor tölt be kapcsolt adatokat, amikor elérjük őket.

Eager loading

Előre betölt kapcsolatokat egy lekérdezéssel (JOIN).

N+1 probléma

Túl sok lekérdezés keletkezik, ha minden kapcsolt adat külön jön le.

Jó gyakorlatok

- Használj migrációkat a séma kezelésére
- Mérj teljesítményt és ne félj nyers SQL-t írni
- Állíts be indexeket a gyakori szűrésekhez
- Tranzakciókkal biztosítsd a konzisztenciát

Mi az a Prisma?

Prisma

Modern ORM (Object-Relational Mapping) Node.js és TypeScript környezethez

Főbb jellemzők

- Típusbiztos adatbázis kliens
- Deklaratív séma definíció
- Automatikus migráció kezelés
- Intuitív API CRUD műveletekhez
- Támogatott adatbázisok: PostgreSQL, MySQL, SQLite, MongoDB, stb.

Prisma telepítése

NPM telepítés

```
# Prisma CLI telepítése dev dependencyként
npm install prisma --save-dev

# Prisma Client telepítése
npm install @prisma/client

# Prisma inicializálása
npx prisma init
```

Létrejövő fájlok

- `prisma/schema.prisma` - Adatbázis séma
- `.env` - Környezeti változók (pl. DATABASE_URL)

Prisma Schema alapok

schema.prisma

```
generator client {
  provider = "prisma-client-js"
}
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}
model User {
  id        Int      @id @default(autoincrement())
  email     String   @unique
  name      String?
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}
```

Prisma migrációk

Adatbázis migráció létrehozása és futtatása

```
# Migráció létrehozása
npx prisma migrate dev --name init

# Prisma Client újragenerálása
npx prisma generate

# Adatbázis vizualizáció
npx prisma studio
```

Prisma Client inicializálása

prisma.js

```
const { PrismaClient } = require('@prisma/client');

const prisma = new PrismaClient();

module.exports = prisma;
```

Singleton pattern

Érdemes egyetlen Prisma Client példányt használni az alkalmazásban.

Create - Létrehozás

Egy rekord létrehozása

```
const prisma = require('./prisma');

async function createUser() {
  const user = await prisma.user.create({
    data: {
      email: 'john@example.com',
      name: 'John␣Doe'
    }
  });

  console.log(user);
  // { id: 1, email: 'john@example.com', name: 'John Doe', ... }
}
```

Create - Több rekord

createMany használata

```
async function createManyUsers() {
  const users = await prisma.user.createMany({
    data: [
      { email: 'alice@example.com', name: 'Alice' },
      { email: 'bob@example.com', name: 'Bob' },
      { email: 'charlie@example.com', name: 'Charlie' }
    ]
  });

  console.log(`${users.count} felhasználó létrehozva`);
}
```

Read - Összes rekord

findMany használata

```
async function getAllUsers() {
  const users = await prisma.user.findMany();
  console.log(users);
}

// Rendezéssel
async function getUsersSorted() {
  const users = await prisma.user.findMany({
    orderBy: {
      name: 'asc'
    }
  });
}
```

Read - Egy rekord

findUnique és findFirst

```
// Egyedi mező alapján (pl. id, email)
async function getUserById(id) {
  const user = await prisma.user.findUnique({
    where: { id: id }
  });
  return user;
}
// Első találat
async function getUserByName(name) {
  const user = await prisma.user.findFirst({
    where: { name: name }
  });
  return user;
}
```

Read - Szűrés

Where feltételek

```
async function filterUsers() {
  const users = await prisma.user.findMany({
    where: {
      email: {
        contains: '@example.com'
      },
      name: {
        startsWith: 'J'
      }
    }
  });
  return users;
}
```

Read - Mezők kiválasztása

Select használata

```
async function getUserEmails() {
  const users = await prisma.user.findMany({
    select: {
      id: true,
      email: true,
      name: true
    }
  });
  return users;
  // [{ id: 1, email: '...', name: '...' }, ...]
}
```

Update - Módosítás

Egy rekord frissítése

```
async function updateUser(id) {
  const user = await prisma.user.update({
    where: { id: id },
    data: {
      name: 'Jane Doe',
      email: 'jane@example.com'
    }
  });
  return user;
}
```

Update - Több rekord

updateMany használata

```
async function updateManyUsers() {
  const result = await prisma.user.updateMany({
    where: {
      email: {
        contains: '@old-domain.com'
      }
    },
    data: {
      email: 'updated@new-domain.com'
    }
  });

  console.log(`${result.count} rekord frissítve`);
}
```

Delete - Törlés

Egy rekord törlése

```
async function deleteUser(id) {
  const user = await prisma.user.delete({
    where: { id: id }
  });
  return user;
}
```

Delete - Több rekord

deleteMany használata

```
async function deleteInactiveUsers() {
  const result = await prisma.user.deleteMany({
    where: {
      createdAt: {
        lt: new Date('2020-01-01')
      }
    }
  });

  console.log(`${result.count} felhasználó törölve`);
}
```

Relációk - Schema

Egy-a-többhöz kapcsolat

```
model User {
  id     Int      @id @default(autoincrement())
  name   String
  posts  Post[]
}

model Post {
  id        Int     @id @default(autoincrement())
  title     String
  content   String?
  authorId  Int
  author    User    @relation(fields: [authorId],
                              references: [id])
}
```

Relációk - Include

Kapcsolt adatok lekérdezése

```
async function getUserWithPosts(userId) {
  const user = await prisma.user.findUnique({
    where: { id: userId },
    include: {
      posts: true
    }
  });

  console.log(user);
  // { id: 1, name: 'John', posts: [...] }
}
```

Relációk - Nested Write

Kapcsolt rekordok létrehozása

```
async function createUserWithPosts() {
  const user = await prisma.user.create({
    data: {
      name: 'John Doe',
      email: 'john@example.com',
      posts: {
        create: [
          { title: 'First Post', content: 'Hello!' },
          { title: 'Second Post', content: 'World!' }
        ]
      }
    }
  });
}
```

Tranzakciók

Több művelet egy tranzakcióban

```
async function transferPosts(fromUserId, toUserId) {
  const result = await prisma.$transaction([
    prisma.post.updateMany({
      where: { authorId: fromUserId },
      data: { authorId: toUserId }
    }),
    prisma.user.update({
      where: { id: fromUserId },
      data: { name: 'Archived User' }
    })
  ]);

  return result;
}
```

Express integráció példa

REST API endpoint Prisma-val

```
const express = require('express');
const prisma = require('./prisma');
const app = express();
app.use(express.json());
app.get('/users', async (req, res) => {
  const users = await prisma.user.findMany();
  res.json(users);
});
app.post('/users', async (req, res) => {
  const user = await prisma.user.create({
    data: req.body
  });
  res.json(user);
});
```

Hibakezelés

Try-catch használata

```
async function safeCreateUser(data) {
  try {
    const user = await prisma.user.create({
      data: data
    });
    return { success: true, user };
  } catch (error) {
    if (error.code === 'P2002') {
      return { success: false,
               error: 'Email már létezik' };
    }
    return { success: false, error: error.message };
  }
}
```

Prisma előnyei

Miért használjuk?

- **Típusbiztonság** - Auto-complete és típus ellenőrzés
- **Egyszerű API** - Intuitív, könnyen tanulható
- **Migráció kezelés** - Automatikus séma szinkronizáció
- **Teljesítmény** - Optimalizált query-k
- **Prisma Studio** - Vizuális adatbázis böngésző
- **Jó dokumentáció** - Részletes guide-ok és példák

Kapcsolat alapok

Mi az adatbázis kapcsolat?

- A kliens és az adatbázis szerver közötti kommunikáció
- Hozzáférés a lekérdezésekhez és tranzakciókhoz
- Hitelesítés, jogosultság és titkosítás része a kapcsolatnak

Kapcsolati paraméterek

Alapvető beállítások

- **Host**: Szerver címe (localhost, IP, domain)
- **Port**: Adatbázis portja
  (PostgreSQL: 5432, MySQL: 3306, MongoDB: 27017)
- **Database**: Adatbázis neve
- **User/Password**: Hitelesítési adatok
- **SSL/TLS**: Titkosított kapcsolat (production kötelező!)

Connection String formátumok

PostgreSQL

```
postgresql://user:password@localhost:5432/mydb?schema=public
```

MySQL

```
mysql://user:password@localhost:3306/mydb
```

MongoDB

```
mongodb://user:password@localhost:27017/mydb
```

Konfiguráció alkalmazásban

Környezeti változók (.env fájl)

```
# Alapvető kapcsolat
DATABASE_URL="postgresql://user:pass@localhost:5432/appdb"
```

SOHA ne commitold a .env fájlt!

Használj .env.example sablont, a valódi értékeket .gitignore-ba!

Prisma specifikus konfiguráció

schema.prisma fájl

```
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}
```

Connection URL extended példa

```
DATABASE_URL="postgresql://user:pass@localhost:5432/mydb?schema=public\
&connection_limit=10&pool_timeout=20"
```

Mi az adatbázis replikáció?

Definíció

Az adatok **több adatbázis példány között** vannak másolva annak érdekében, hogy növekedjen a **rendelkezésre állás**, **skálázhatóság** és **hibatűrés**.

- Primary–Replica (master–slave) felépítés a leggyakoribb
- A replikáció lehet **szinkron** vagy **aszinkron**
- Read-heavy alkalmazásoknál különösen hasznos

Miért jó a replikáció?

Főbb előnyök

- **Nagyobb rendelkezésre állás**: hiba esetén átváltás (failover)
- **Skálázás olvasásra**: több replica olvasási terhelést visz el
- **Biztonsági másolat**: adatvesztés kockázatának csökkentése
- **Földrajzi közelség**: alacsonyabb késleltetés regionális replikákkal

Replikációs topológiák

- **Primary–Replica**: egy írható csomópont, több olvasó
- **Multi-primary**: több írható csomópont (konfliktuskezelés szükséges)
- **Chain/Cascade**: láncolt replikák nagy földrajzi területre

Gyakorlatban

A legtöbb backend szolgáltatásnál a Primary–Replica modell az alapértelmezett.

Szinkron replikáció

Hogyan működik?

- Az írás **csak akkor sikeres**, ha a replika is visszaigazolta
- Erős **konzisztencia** (nincs adatvesztés írás után)
- A primary **vár** a replikákra → nagyobb latency

Mikor jó?

Pénzügyi, tranzakciós rendszerek, ahol az adatvesztés nem elfogadható.

Szinkron replikáció előnyök / hátrányok

Előnyök

- Erős konzisztencia
- Minimális adatvesztés
- Egyszerűbb olvasási modell

Hátrányok

- Nagyobb válaszidő írásnál
- Érzékenyebb hálózati késleltetésre
- Replikák kiesése lassítja az írást

Aszinkron replikáció

Hogyan működik?

- Az írás **azonnal sikeres**, a replika később frissül
- Jobb **teljesítmény** és alacsonyabb latency
- **Időbeli eltérés** lehetséges (eventual consistency)

Mikor jó?

Nagy forgalmú rendszerek, ahol az olvasási skálázás fontosabb, mint a tökéletes konzisztencia (pl. hírportál, analytics).

Aszinkron replikáció előnyök / hátrányok

Előnyök

- Alacsonyabb írási késleltetés
- Jobb skálázhatóság
- Replikák nem lassítják az írást

Hátrányok

- Ideiglenes inkonzisztencia
- Failover esetén adatvesztés kockázata
- Komplexebb olvasási logika

Szinkron vs. aszinkron – döntési szempontok

- **Konzisztencia igény**: kritikus adatok → szinkron
- **Teljesítmény igény**: magas throughput → aszinkron
- **Hálózat megbízhatósága**: nagy latency → aszinkron
- **Üzleti kockázat**: adatvesztés megengedett? → aszinkron

Ökölszabály

Ha az írások számítanak jobban, válaszd a szinkront. Ha az olvasási skálázás és az alacsony latency fontos, válaszd az aszinkront.

Hogyan használjuk a gyakorlatban?

Tipikus architektúra

- **Primary** kezeli az írásokat
- **Replica** kiszolgálja az olvasásokat
- **Load balancer** irányítja az olvasást a replikákhoz

ORM/Prisma oldal

A legtöbb ORM-ben (így Prismában is) külön connection stringet használsz az író és az olvasó adatbázisokhoz, vagy read-replica routert konfigurálsz.

Mi az aggregáció?

Aggregált lekérdezések

Adatok összesítése és elemzése számítások segítségével

Prisma aggregáció műveletek

- `count` - Rekordok számlálása
- `avg` - Átlag számítás
- `sum` - Összegzés
- `min` - Minimum érték
- `max` - Maximum érték
- `groupBy` - Csoportosítás

Példa séma

schema.prisma - Product

```
model Product {
  id          Int      @id @default(autoincrement())
  name        String
  price       Float
  stock       Int
  categoryId  Int
  category    Category @relation(fields: [categoryId],
                                references: [id])
  createdAt   DateTime @default(now())
}
```

Példa séma (folyt.)

schema.prisma - Category

```
model Category {
  id       Int       @id @default(autoincrement())
  name     String
  products Product[]
}
```

Count - Számlálás

Összes rekord számolása

```
const prisma = require('./prisma');
async function countProducts() {
  const count = await prisma.product.count();
  console.log(`Összes termék: ${count}`);
}
```

Count - Számlálás (szűréssel)

Szűrt számlálás

```
async function countExpensiveProducts() {
  const count = await prisma.product.count({
    where: {
      price: {
        gt: 1000
      }
    }
  });
  console.log(`Drága termékek: ${count}`);
}
```

Aggregate - Alapok

Egyszerű aggregáció

```
async function getProductStats() {
  const stats = await prisma.product.aggregate({
    _count: true,
    _avg: {price: true},
    _sum: {stock: true},
    _min: {price: true},
    _max: {price: true}
  });
```

Aggregate - Alapok (folyt.)

Egyszerű aggregáció

```
  console.log(stats);
}
```

Aggregate - Eredmény

Visszatérési érték példa

```
{
  _count: 150,
  _avg: {
    price: 1250.50
  },
  _sum: {
    stock: 5420
  },
  _min: {
    price: 99.99
  },
  _max: {
    price: 9999.99
  }}
```

Aggregate - Szűréssel

Where feltétellel

```
async function getCategoryStats(categoryId) {
  const stats = await prisma.product.aggregate({
    where: {
      categoryId: categoryId
    },
    _count: true,
    _avg: {
      price: true
    },
    _sum: {
      stock: true
    }
  });
  return stats;
}
```

GroupBy - Csoportosítás

Kategóriánkénti csoportosítás

```
async function groupByCategory() {
  const result = await prisma.product.groupBy({
    by: ['categoryId'],
    _count: true,
    _avg: {
      price: true
    },
    _sum: {
      stock: true
    }
  });

  return result;
}
```

GroupBy - Eredmény

Csoportosított adatok

```
[
  {
    categoryId: 1,
    _count: 45,
    _avg: { price: 599.99 },
    _sum: { stock: 1200 }
  },
  {
    categoryId: 2,
    _count: 67,
    _avg: { price: 1299.50 },
    _sum: { stock: 890 }
  }
]
```

GroupBy - Having feltétel

Szűrés az aggregált eredményen

```
async function getCategoriesWithHighAvgPrice() {
  const result = await prisma.product.groupBy({
    by: ['categoryId'],
    _avg: {price: true},
    _count: true,
    having: {
      price: {
        _avg: {gt: 1000}
      }
    }
  });
  return result;
}
```

GroupBy - Rendezés

OrderBy aggregált mezőn

```
async function getCategoriesOrderedByAvgPrice() {
  const result = await prisma.product.groupBy({
    by: ['categoryId'],
    _avg: {price: true},
    _count: true,
    orderBy: {
      _avg: {price: 'desc'}
      }
  });
  return result;
}
```

Több mező szerinti csoportosítás

Összetett GroupBy

```
// Bővített séma szükséges: inStock boolean mező
async function groupByCategoryAndStock() {
  const result = await prisma.product.groupBy({
    by: ['categoryId', 'inStock'],
    _count: true,
    _avg: {price: true},
    orderBy: {categoryId: 'asc'}
  });
  return result;
}
```

Relációk és aggregáció

Kapcsolt táblák aggregálása

```
async function getCategoriesWithProductCount() {
  const categories = await prisma.category.findMany({
    include: {
      _count: {
        select: {
          products: true
        }
      }
    }
  });
  // Eredmény: [{ id: 1, name: 'Electronics',
  //              _count: { products: 45 } }, ...]
  return categories;
}
```

Relációs szűrés aggregációval

Több mint N termékkel rendelkező kategóriák

```
async function getCategoriesWithManyProducts() {
  const categories = await prisma.category.findMany({
    where: {
      products: {some: {}}
    },
    include: {
      _count: {
        select: {products: true}
      }
    }});
  // Csak azok a kategóriák, amikhez van termék
  return categories.filter(c => c._count.products > 10);
}
```

Express API - Statisztika endpoint

Aggregált adatok REST API-n keresztül

```
const express = require('express');
const prisma = require('./prisma');
const app = express();

app.get('/api/stats/products', async (req, res) => {
  const stats = await prisma.product.aggregate({
    _count: true,
    _avg: { price: true },
    _sum: { stock: true },
    _min: { price: true },
    _max: { price: true }
  });
  res.json(stats);
});
```

Express API - Csoportosított adatok

GroupBy endpoint

```
app.get('/api/stats/by-category', async (req, res) => {
  const stats = await prisma.product.groupBy({
    by: ['categoryId'],
    _count: true,
    _avg: { price: true },
    _sum: { stock: true },
    orderBy: {
      _avg: {
        price: 'desc'
      }
    }
  });
  res.json(stats);
});
```

Egyedi aggregáció - Raw Query

SQL használata komplex esetekben

```
async function customAggregation() {
  const result = await prisma.$queryRaw`
    SELECT "categoryId",
      COUNT(*) as count,
      AVG(price) as avgPrice,
      SUM(stock) as totalStock
    FROM "Product"
    WHERE price > 100
    GROUP BY "categoryId"
    HAVING AVG(price) > 500
    ORDER BY avgPrice DESC
  `;
  return result;
}
```

Aggregáció - Best Practices

Ajánlások

- Használj `select` és `where` feltételeket a teljesítmény növelésére
- GroupBy esetén mindig gondold át az indexelést
- Nagy adathalmazoknál fontold meg a lapozást
- Komplex esetekben raw SQL lehet hatékonyabb
- Cacheld az aggregált eredményeket, ha ritkán változnak
- Használj `having` feltételt az aggregált adatok szűrésére

Összefoglalás

Prisma aggregációk

- **count** - Egyszerű és szűrt számlálás
- **aggregate** - Átlag, összeg, min, max számítások
- **groupBy** - Csoportosítás és aggregálás
- **having** - Szűrés aggregált eredményen
- **orderBy** - Rendezés aggregált mezők szerint
- **_count** - Kapcsolt rekordok számlálása

Repository minta lényege

Mire jó?

- Elválasztja az adat-hozzáférést az üzleti logikától
- Egységes API az adatok kezelésére
- Könnyebben tesztelhető kód

IRepository és konkret repository

IRepository

- Absztrakt interfész CRUD műveletekkel
- Pl. `findAll`, `findById`, `create`, `update`, `delete`

Konkret repository

Az interfész implementációja (pl. `UserRepository`).

IRepository példa (TypeScript)

```
interface IRepository<T> {
  findAll(): Promise<T[]>;
  findById(id: number): Promise<T | null>;
  create(data: T): Promise<T>;
  update(id: number, data: Partial<T>): Promise<T>;
  delete(id: number): Promise<void>;
}
```

Prisma repository példa

Prisma Client

A Prisma Client kezeli a kapcsolatot és az SQL generálást.

```
class UserRepository implements IRepository<User> {
  constructor(private prisma: PrismaClient) {}
  findAll() {
    return this.prisma.user.findMany();
  }
  findById(id: number) {
    return this.prisma.user.findUnique({ where: { id } });
  }
```

Prisma repository példa folytatás

```
  create(data: User) {
      return this.prisma.user.create({ data });
    }

    update(id: number, data: Partial<User>) {
      return this.prisma.user.update({ where: { id }, data });
    }

    async delete(id: number) {
      await this.prisma.user.delete({ where: { id } });
    }
}
```

Mikor hasznos?

- Ha több adatforrás van (SQL, API, cache)
- Ha szeretnél változtatható adat-hozzáférést
- Ha szükség van mock-olható IRepositories-re tesztekhez

Mi az a DTO?

Data Transfer Object

- Adatok szállítására szolgáló objektum
- Elválasztja az adatbázis sémát
  a kliens felőli interfésztől
- Kontroll az átadott adatok felett
- Validáció és transzformáció helye

Példa

Felhasználó entitás tartalmaz jelszó hash-t,
de a DTO nem adja ki.

Miért használjunk DTO-t?

Előnyök

- **Biztonság** - Érzékeny mezők elrejtése (pl. jelszó)
- **Verziókezelés** - API változhat a DB séma változtatása nélkül
- **Validáció** - Bejövő adatok ellenőrzése
- **Dokumentáció** - Világos szerződés a kliens és szerver között
- **Transzformáció** - Adatok átalakítása (pl. dátum formázás)

Prisma modell vs DTO

Prisma Model

```
model User {
  id          Int       @id
                  @default(autoincrement
  email       String    @unique
  password    String
  name        String?
  createdAt   DateTime  @default(now())
  updatedAt   DateTime  @updatedAt
}
```

UserDTO (kimenet)

```
{
  id: 1,
  email: "john@example.com",
  name: "John␣Doe"
  // password nincs benne!
  // createdAt, updatedAt
  // opcionális
}
```

Input DTO - Létrehozás

CreateUserDTO - Constructor

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

Input DTO - Validáció

CreateUserDTO - Validate módszer

```
class CreateUserDTO {
  // ...

  validate() {
    if (!this.email || !this.email.includes('@')) {
      throw new Error('Érvénytelen email cím');
    }
    if (!this.password || this.password.length < 8) {
      throw new Error('Jelszó túl rövid (min. 8 karakter)');
    }
    return true;
  }
}
```

Output DTO - Válasz

UserResponseDTO

```
class UserResponseDTO {
  constructor(user) {
    this.id = user.id;
    this.email = user.email;
    this.name = user.name;
    // password nincs másolva!
  }
  static fromPrismaUser(user) {
    return new UserResponseDTO(user);
  }
  static fromManyPrismaUsers(users) {
    return users.map(u => new UserResponseDTO(u));
  }
}
```

DTO használata service-ben

UserService példa (1)

```
const prisma = require('./prisma');
const bcrypt = require('bcrypt');
class UserService {
  async createUser(createUserDTO) {
    createUserDTO.validate(); // Validálás
    const hashedPassword = await bcrypt.hash(
      createUserDTO.password, 10
    ); // Jelszó hash-elés
```

DTO használata service-ben (folyt.)

UserService példa (2)

```
const user = await prisma.user.create({
  data: {
    email: createUserDTO.email,
    password: hashedPassword,
    name: createUserDTO.name
  } // Mentés Prisma-val
});
```

DTO használata service-ben (folyt.)

UserService példa (3)

```
    // DTO-vá alakítás visszaadás előtt
    return UserResponseDTO.fromPrismaUser(user);
  }
  async getAllUsers() {
    const users = await prisma.user.findMany();
    return UserResponseDTO.fromManyPrismaUsers(users);
  }
```

DTO használata service-ben (folyt.)

UserService példa (4)

```
  async getUserById(id) {
    const user = await prisma.user.findUnique({
      where: { id }
    });
    if (!user) return null;
    return UserResponseDTO.fromPrismaUser(user);
  }
}
```

Express integráció DTO-val

REST endpoint (POST)

```
const express = require('express');
const userService = new UserService();
const app = express();
app.post('/api/users', async (req, res) => {
  try {
    const createDTO = new CreateUserDTO(req.body);
    const userDTO = await userService.createUser(createDTO);
    res.status(201).json(userDTO);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});
```

Express integráció DTO-val (folyt.)

REST endpoint (GET)

```
app.get('/api/users', async (req, res) => {
  const usersDTO = await userService.getAllUsers();
  res.json(usersDTO);
});
```

Update DTO - Constructor

UpdateUserDTO

```
class UpdateUserDTO {
  constructor(data) {
    // Csak a megadott mezők kerülnek be
    if (data.email !== undefined)
      this.email = data.email;
    if (data.name !== undefined)
      this.name = data.name;
    if (data.password !== undefined)
      this.password = data.password;
  }
```

Update DTO - Validáció

UpdateUserDTO - Validate

```
class UpdateUserDTO {
  // ...

  validate() {
    if (this.email && !this.email.includes('@')) {
      throw new Error('Érvénytelen email');
    }
    if (this.password && this.password.length < 8) {
      throw new Error('Jelszó túl rövid');
    }
    return true;
  }
}
```

Nested DTO - Relációkkal

PostWithAuthorDTO

```
class PostWithAuthorDTO {
  constructor(post) {
    this.id = post.id;
    this.title = post.title;
    this.content = post.content;
    this.createdAt = post.createdAt;
    // Nested DTO az author-hoz
    if (post.author) {
      this.author = new UserResponseDTO(post.author);
    }
  }
  static fromPrismaPost(post) {
    return new PostWithAuthorDTO(post);
  }
}
```

Nested DTO használata

Service metódus

```
class PostService {
  async getPostWithAuthor(postId) {
    const post = await prisma.post.findUnique({
      where: { id: postId },
      include: {
        author: true
      }
    });

    if (!post) return null;

    return PostWithAuthorDTO.fromPrismaPost(post);
  }
}
```

Validációs könyvtár - Joi

Joi telepítése

```
npm install joi
```

Validációs séma

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

const createUserSchema = Joi.object({
  email: Joi.string().email().required(),
  password: Joi.string().min(8).required(),
  name: Joi.string().optional()
});
```

DTO Joi validációval

CreateUserDTO sémával

```
class CreateUserDTO {
  constructor(data) {
    const { error, value } =
      createUserSchema.validate(data);
    if (error)
      throw new Error(error.details[0].message);
    Object.assign(this, value);
  }
}
```

Paginációs DTO

PaginatedResponseDTO

```
class PaginatedResponseDTO {
  constructor(data, total, page, pageSize) {
    this.data = data;
    this.pagination = {
      total, page, pageSize,
      totalPages: Math.ceil(total / pageSize)
    };
  }
}
```

Paginációs DTO használata

PaginatedResponseDTOUsage

```
// Használat
const users = await prisma.user.findMany({
  skip: (page - 1) * pageSize, take: pageSize
});
const total = await prisma.user.count();
return new PaginatedResponseDTO(
  UserResponseDTO.fromManyPrismaUsers(users),
  total, page, pageSize
);
```

Szűrési DTO - Constructor

UserFilterDTO

```
class UserFilterDTO {
  constructor(query) {
    this.email = query.email;
    this.name = query.name;
    this.page = parseInt(query.page) || 1;
    this.pageSize = parseInt(query.pageSize) || 10;
  }
```

Szűrési DTO - Prisma konverzió

toPrismaWhere módszer

```
class UserFilterDTO {
  // ...

  toPrismaWhere() {
    const where = {};
    if (this.email)
      where.email = { contains: this.email };
    if (this.name)
      where.name = { contains: this.name };
    return where;
  }
}
```

Szűrési DTO használata

Express endpoint szűréssel (1)

```
app.get('/api/users', async (req, res) => {
  const filterDTO = new UserFilterDTO(req.query);
  const where = filterDTO.toPrismaWhere();
  const users = await prisma.user.findMany({
    where,
    skip: (filterDTO.page - 1) * filterDTO.pageSize,
    take: filterDTO.pageSize
  });
```

Szűrési DTO használata (folyt.)

Express endpoint szűréssel (2)

```
  const total = await prisma.user.count({ where });
  const result = new PaginatedResponseDTO(
    UserResponseDTO.fromManyPrismaUsers(users),
    total, filterDTO.page, filterDTO.pageSize
  );
  res.json(result);
});
```

Mapper függvények

Alternatív megközelítés (1)

```
function toUserResponseDTO(user) {
  return {
    id: user.id,
    email: user.email,
    name: user.name
  };
}
function toUserResponseDTOs(users) {
  return users.map(toUserResponseDTO);
}
```

Mapper függvények (folyt.)

Alternatív megközelítés (2)

```
module.exports = {
  toUserResponseDTO,
  toUserResponseDTOs
};
```

Mapper használata

Service-ben (1)

```
const { toUserResponseDTO, toUserResponseDTOs }
  = require('./dto/user.mapper');
class UserService {
  async getAllUsers() {
    const users = await prisma.user.findMany();
    return toUserResponseDTOs(users);
  }
```

Mapper használata (folyt.)

Service-ben (2)

```
  async getUserById(id) {
    const user = await prisma.user.findUnique({
      where: { id }
    });
    if (!user) return null;
    return toUserResponseDTO(user);
  }
}
```

DTO Best Practices

Ajánlások

- Külön DTO minden use case-hez (Create, Update, Response)
- Ne add vissza közvetlenül a Prisma modellt
- Validálj minden bejövő adatot
- Használj validációs könyvtárat (Joi, Yup, Zod)
- Tartsd egyszerűnek - ne túlbonyolítsd
- Dokumentáld a DTO mezőket

DTO vs Prisma Select

Prisma Select

- Egyszerű mezők kiválasztása
- Kevesebb kód
- Nincs validáció
- DB szintű szűrés

DTO

- Komplex transzformációk
- Validáció és logika
- Típusbiztonság
- API verziókezelés

Kombináld őket!

Használd a Prisma select-et a DB szinten, és DTO-t a transzformációhoz.

Összefoglalás

DTO előnyei

- **Biztonság** - Érzékeny adatok elrejtése
- **Validáció** - Adatok ellenőrzése
- **Transzformáció** - Adatok átalakítása
- **Dokumentáció** - Világos API szerződés
- **Karbantarthatóság** - Könnyebb változtatások
- **Tesztelhetőség** - Egyszerűbb unit tesztek

Mapper minta szerepe

Mi a mapper?

- Átalakítás domain objektumok és DTO-k között
- Elválasztja az adatformátumot az üzleti modelltől
- Központi helyen tartja az átalakítás logikáját

Miért hasznos?

- Csökkenti a duplikált átalakításokat
- Stabil API-szerződések (DTO-k) a változó domain mellett
- Tesztelhető, moduláris és újrafelhasználható kód

Egyszerű mapper példa

```
// Domain objektum
const user = {
  id: 1, name: "John Doe",
  email: "john@example.com", passwordHash: "..."
};

// Mapper függvény
const toUserDto = (u) => ({
  id: u.id, name: u.name, email: u.email
});
```

Kétirányú mapper

```
const toUserEntity = (dto) => ({
  id: dto.id ?? undefined,
  name: dto.name,
  email: dto.email
});
```

Megjegyzés

Írhatsz külön mappert bejövő és kimenő adatokra.

Mapper és repository

- Repository a tárolást kezeli
- Mapper az objektumok formátumát kezeli
- Két külön felelősségi kör

Jó gyakorlatok

- Ne másolj át érzékeny mezőket (pl. jelszó)
- Tartsd egy helyen az átalakításokat
- Használj tiszta függvényeket és kis helper-eket