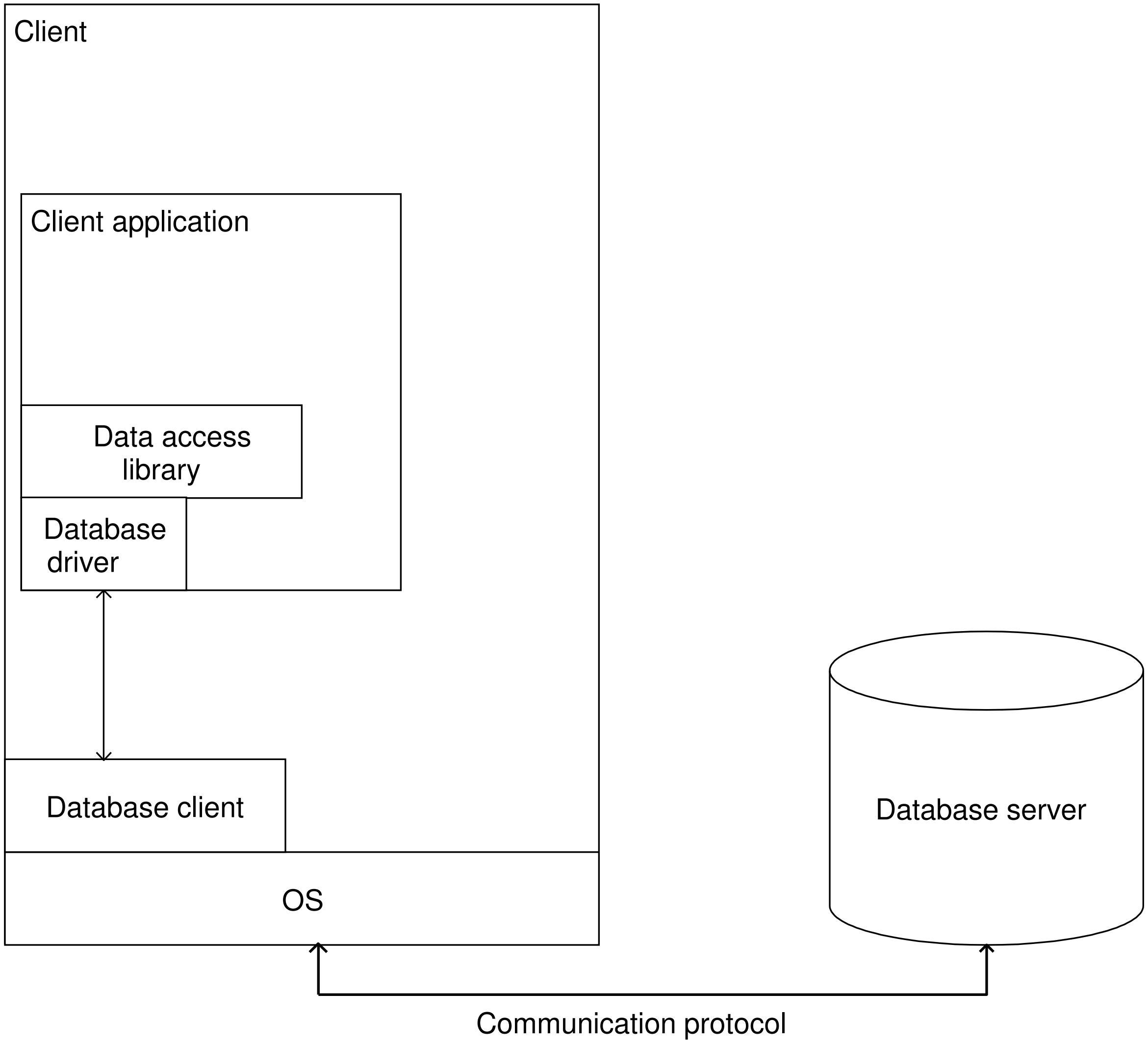
Data-driven systems

ADO.NET, Entity Framework

BME AUT

Automatizálási és
Alkalmazott
Informatikai Tanszék

Data Access Layer


Client

Client application

Data access library

Database driver

Database client

OS

Database server

Communication protocol

Role of data access layer

- Abstract database access
  - Access relational databases
  - Database independent
  - Unified, database independent coding
- Typical elements
  - Connection
    - Database drivers
  - Command
    - Parameters
  - ResultSet
  - Exception

ADO.NET

Concept of ADO.NET

- Unified, database independent coding
  - Database independent interface
  - Interfaces and abstract classes
- Implementations
  - Implement the basic functions
  - Extend the basic functions
  - Part of the .NET framework
    - OleDb, Microsoft SQL Server
  - Other implementations from Microsoft
    - ODBC, Oracle
  - Third-party implementations
    - MySQL

Establishing database connection

- IDbConnection interface
  - Open: Open
  - Close: Close
  - BeginTransaction: Start a transaction
- Connection Pooling
  - Supported by OleDB and MSSQL providers
  - Cached connetions, reusable
  - Min Pool Size → per ConnectionString
  - Connection leak

Creating a Connection String

- Syntax depends on the type of database
- `User ID=LOGIN;Password=PASSWD;Persist Security Info=false;Initial Catalog=AdventureWorks;Data Source=DATASOURCE;Packet Size=4096`
- http://www.connectionstrings.com/
- Connection String-based attacks
  - ConnectionStringBuilder

Establish a connection - example

```
var builder = new
SqlConnectionStringBuilder();
builder.UserID = "User";
builder.Password = "Pw";
builder.DataSource = "database.server.hu";
builder.InitialCatalog = “DataDriven";

var con = new
SqlConnection(builder.ConnectionString));
con.Open();
…
con.Close();
```

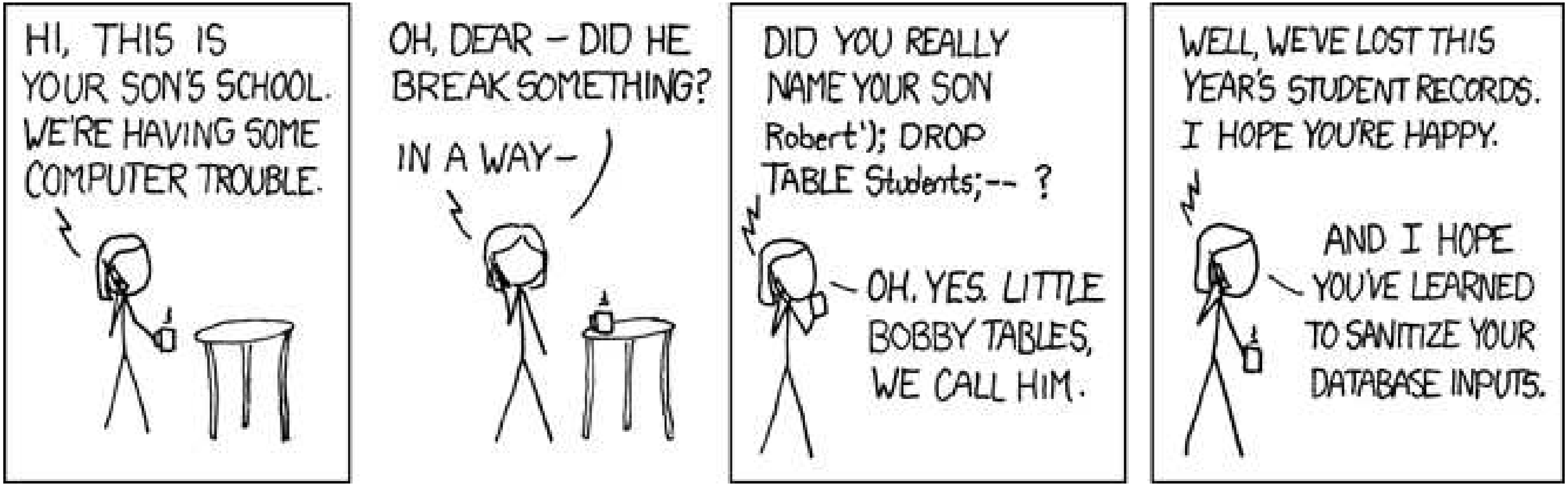
Using commands

- `IDbCommand` interface
  - Three types of commands (`CommandType`)
    - Stored procedure
    - Entire table
    - SQL query
  - The SQL command (`CommandText`)
  - Database connection (`Connection`)
  - Transaction used (`Transaction`)
  - Timeout (`CommandTimeout`)
    - 30 sec by default
  - Parameters
    - Avoiding SQL injection

Command execution

- `ExecuteReader`
  - Fetch multiple records (cursor)
- `ExecuteScalar`
  - Retrieve a scalar result
- `ExecuteNonQuery`
  - No return value (E.g. INSERT)
    - Return value: affected rows (int)
- `ExecuteXmlReader` (MS SQL Server)
  - Returns XML document (XmlReader)
  - A single record with a single XML column
    - Can use „FOR XML"
- Re-using commands
  - `Command.Prepare`()
    - Prepares the execution in the server
    - Subsequent execution is faster in exchane for an earlier overhead
    - Useful if the same command is re-used (with different arguments)

SQL Injection – 1


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

SQL Injection – 2

- Security risk
  - Use input data without checking/sanitization
- String-concatenation of a SQL query
  - `“Select * from product where name =“ + Name.Text;`
  - Named.Text can contain anything
    - 1; drop table product;--
- Major bug!
  - Using parameters

Command – query example

```
var command = new SqlCommand();
command.Connection = connection;
command.CommandText =
             "Select * from product where name = @Name";
command.CommandType = CommandType.Text;

var parameter = new SqlParameter();
parameter.ParameterName = "@Name";
parameter.SqlDbType = SqlDbType.NVarChar;
parameter.Value = productName;
command.Parameters.Add(parameter);

var reader = command.ExecuteReader();
```

Command – SP example

```
var command = new SqlCommand();
command.Connection = connection;
command.CommandText = "SalesByCategory";
command.CommandType = CommandType.StoredProcedure;

var parameter = new SqlParameter();
parameter.ParameterName = "@CategoryName";
parameter.SqlDbType = SqlDbType.NVarChar;
parameter.Value = categoryName;
command.Parameters.Add(parameter);

var reader = command.ExecuteReader();
```

Transactions int ADO.NET

- Start a transaction
  - `BeginTransaction` method
- Set transaction of the command
  - `Transaction` property
- Run the command
- Finish transaction
  - `CommitTransaction`
  - `RollbackTransaction`
- Isolation levels
  - Argument of `BeginTransaction`
  - Database specific isolation levels

Transactions int ADO.NET

- System.Transaction namespace
- TransactionScope
  - TransactionScope.Current
- 10 minutes maximum
  - Defined in system-wide MachineConfig
- One transaction – one Connection
  - Otherwise needs to use MSDTC

Error handling in ADO.NET

- Errors cause exceptions
- Readers and connections must be closed
  - Standard exception handling
  - Close in the finally block
- Dispose pattern
  - With the *using* keyword
  - Closes automatically at the end of the block

```
try
{
   var connection =
      new SqlConnection(...)

   connection.Open();
   ...
}
finnaly
{
   if (connection ! =null)
       connection.Dispose();
}
```

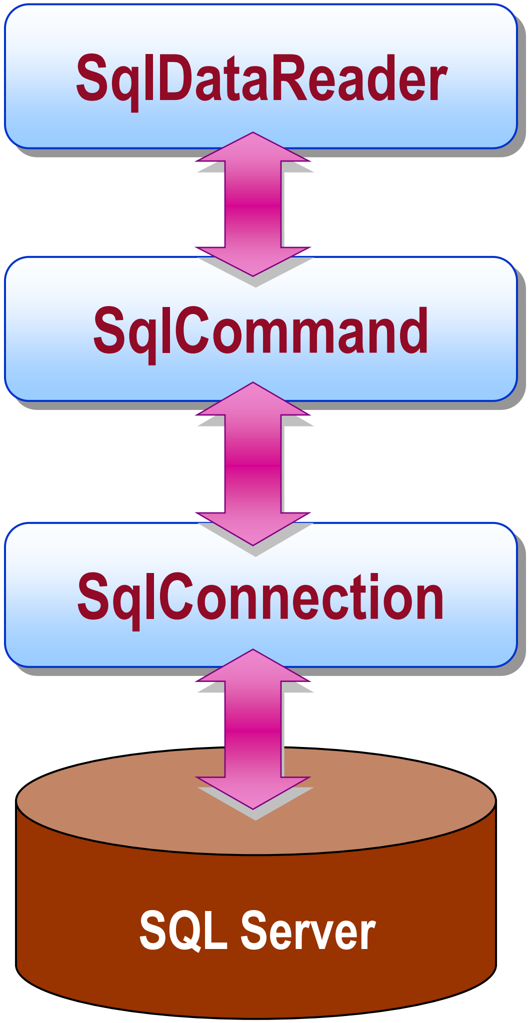
or

```
using (var connection = new
               SqlConnection(...))
{
   connection.Open();
   ...
}
```

ADO.NET

DataReader vs DataSet

Connection-based



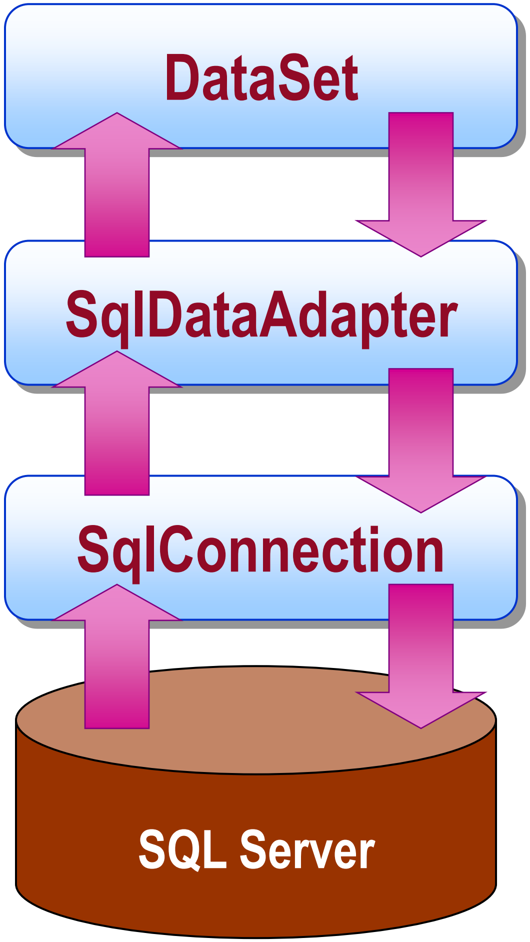
- Data comes form the database
- Steps
  - Open connection
  - Run the command
  - Process the results
  - Close the reader
  - Close the connection

Connection-based – example

```
using (var conn=new SqlConnection(connectionString))
{
    var command = new SqlCommand("SELECT ID, NAME
                               FROM Product", conn);
    connection.Open();
    using(var reader = command.ExecuteReader())
    {
       while (reader.Read()) {
           Console.WriteLine("{0}\t{1}",
                    reader["ID"], reader["Name"]);
       }
    }
}
```

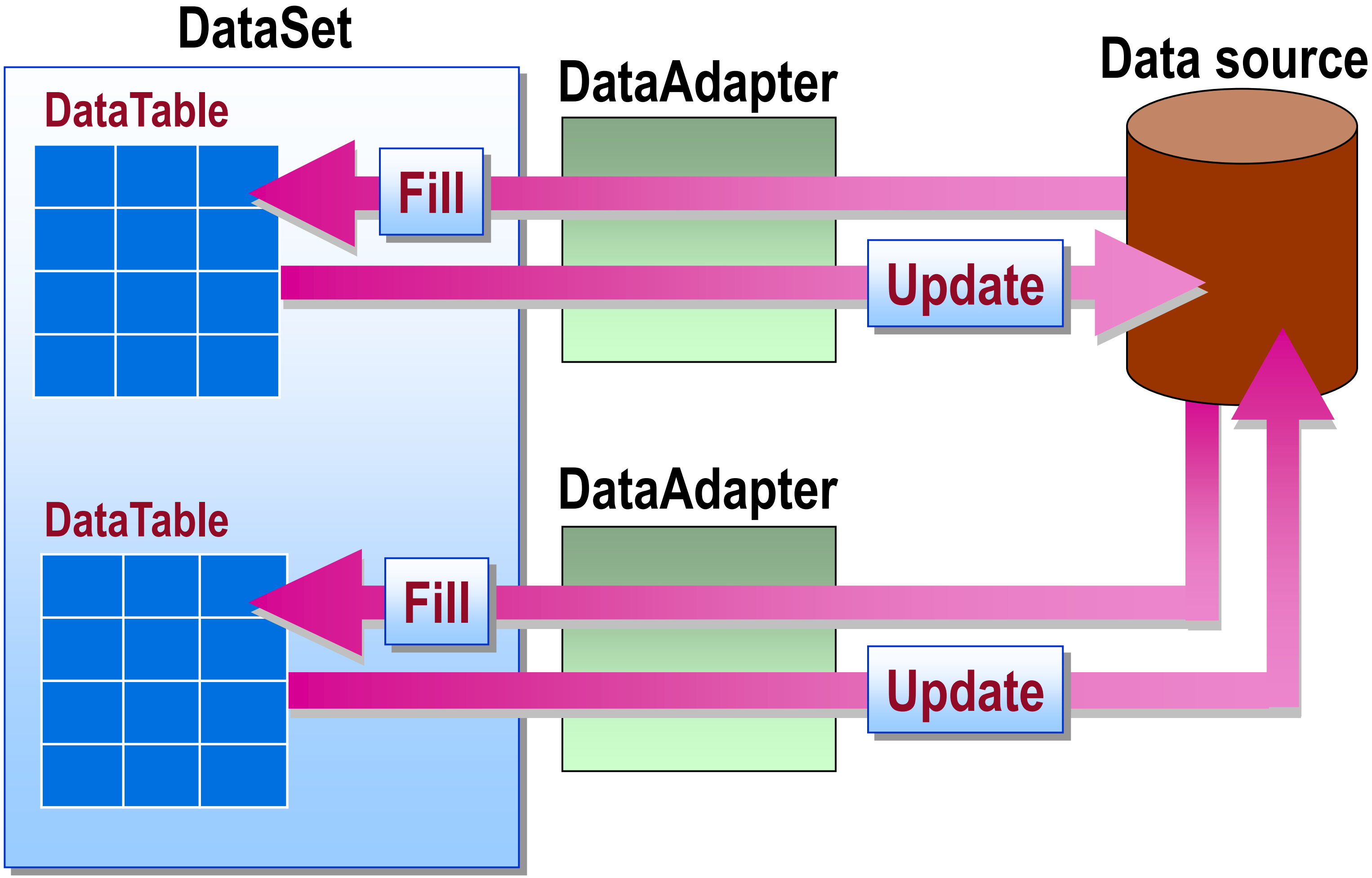
object, not string or int
alternative: reader.GetInt()
runtime error if types do not match
runtime error if value is NULL -> reader.IsDbNull()

Connectionless model



- A type of cache
- Other can modify concurrently
- Steps
  - Open the connection
  - Populate the DataSet
  - A close the connection
  - Use the DataSet and manipulate data inside
  - Open the connection
  - Send changes to the database
  - A close the connection

Role of the DataAdapter



Connectionless model – example

```
var dataSet = new DataSet();
var adapter = new SqlDataAdapter();

using (var conn = new SqlConnection(connectionString))
{
   adapter.SelectCommand = new SqlCommand ("SELECT *
                                     FROM Product", conn);
   connection.Open();
   adapter.Fill(dataSet);
}

---------------------------------------------------------

foreach(var row in dataSet.Tables[“Product"].Rows)
    Console.WriteLine("{0}\t{1}", row["ID"], row["Name"]);

---------------------------------------------------------

using (var conn= new SqlConnection(connectionString))
{
   connection.Open();
   adapter.Update(dataSet);
   dataSet.AcceptChanges();
}
```

Connection models

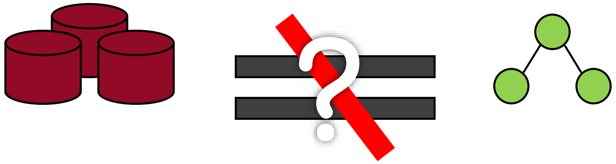
- DataReader
  - Although for a short time, but maintains a connection with the database
  - Advantages
    - Simpler concurrency handling
    - Data is always up to date
    - Requires less memory
  - Drawbacks
    - Requires active connection
    - Scalability
  - Used typically: web applications

- DataSet
  - Connections only during data manipulatoin
  - Advantages
    - Does not require constant connection
    - Scalability
  - Drawbacks
    - Data may be out of date
    - Conflicts can occur
    - Requires memory in the client
  - Typically: used in desktop applications

Entity Framework

Problem statement

- Data ≠ Object
  - > ORM



- SQL language
  - + Data oriented
  - + Easy to formulate a query
  - - No concept of objects
  - - Not strongly typed
  - - Does not integrate as a language element
- ADO.NET DataReader/DataSet
  - + Efficient
  - - Not strongly typed

Data access with LINQ

```
from product in db.Products
   where product.Name == "Lego"
   select product;
```

C# code!

Advantages

- Strongly typed
- Based on classes/objects
- Type checks in compile time

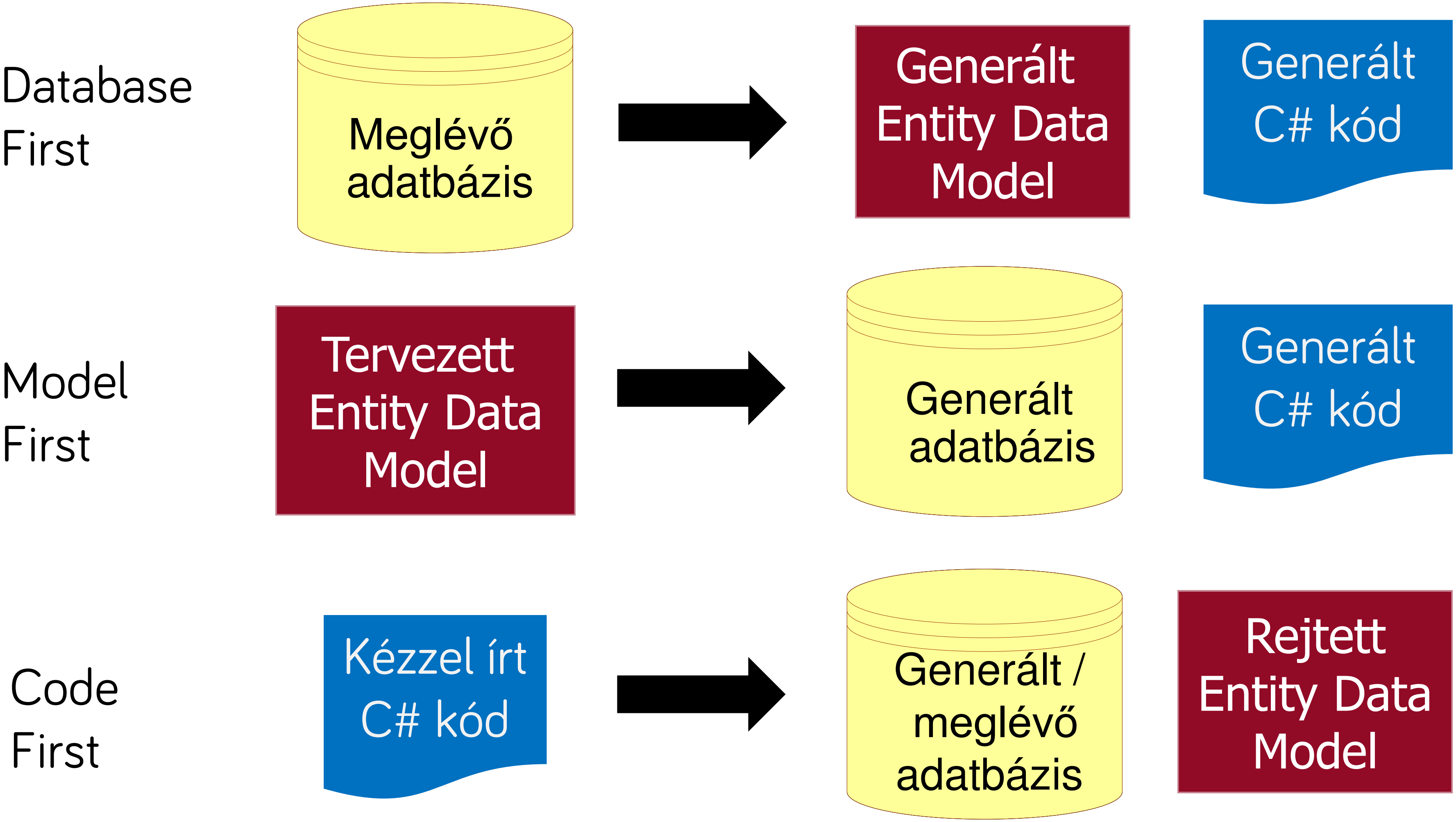
Entity Framework (EF)

- Object Relational Mapping
- Separates the **logical** (*database*) and the **conceptual** (*business*) models
- Detaches the application from the database engine

- Entity Framework VS Entity Framework ***Core***

EF	EF Core
.NET Framework (Windows)	.NET Core (platform-independent)
Stable, no longer developed	Counitnuous development

Leképezési módszerek



Database First

Meglévő adatbázis

Generált Entity Data Model

Generált C# kód

Model First

Tervezett Entity Data Model

Generált adatbázis

Generált C# kód

Code First

Kézzel írt C# kód

Generált / meglévő adatbázis

Rejtett Entity Data Model

Code first

- Generate database from C# code
- Or connect existing database with C# entities
- *Instead of* the EDMX file and EDM Designer: C# model
- C# + Attributes + FluentAPI

- This is the only option when using EF Core

Code First

```
class MyDbContext : System.Data.Entity.DbContext
{
   public DbSet<Product> Products { get; set; }
   public DbSet<Category> Categories { get; set; }

   protected override void OnModelCreating(
                       ModelBuilder modelBuilder)
   {
       modelBuilder.Entity<Product>()
          .Property(b => b.Name).IsRequired();

       modelBuilder.Entity<Product>()
         .HasOne(p => p.Category).WithMany(c => c.Products);
   }
}
```

Code First

```
[Table("Product")]
class Product
{
    [Key]
    public int Id { get; set; }

    [Required]
    [StringLength(1000)]
    public string Name { get; set; }

    public VAT VAT { get; set; }

    public ICollection<Category>
                     Categories { get; set; }
}
```

Navigation property

- „Database join automatically”

```
from p in db.Products
join c in db.Categories
      on p.CategoryID = c.ID
select c.Name
```

instead

```
from p in db.Products
select p.Category.Name
```

Entity Framework

Query

Query - example

- Customers with a main site of business in Budapest

```
from c in db.Customers
where c.MainSite.Address.City == "Budapest"
select c;
```

Query - example

- List the products and the number of oreders for each. Make sure to include products that have no orders yet.

```
from p in db.Products
select new
{
    Name = p.Name,
    OrderedAmount =
        p.OrderItems.Sum(oi => oi.Amount)
};
```

Query - example

- Which products have never been ordered?

```
from p in db.Products
where !p.OrderItems.Any()
select p.Name;
```

Query - example

- Which are the most expensive products?

```
from p in db.Products
where p.Price == db.Products.Max(pp => pp.Price)
select p.Name
```

Entity Framework

DbContext

Role of DbContext

- Used to access the database
- Tracks the entities and the changes and save them back to the database: `SaveChanges()`
- Short lifespan
  - If instantiated manually: in a *using* block
  - When using a DI system: Transient / Request scope (see later…)
- SaveChanges: usually transactional
  - The DbContext can be thought of as a “unit of work”

DbContext – typical usage

- DbContext is not thread-safe
- An instance generally means an open database connection
- Not designed for large amounts of data (not an in-memory database)
- The DbContext is created for a single method, then released
  - No concurrency issues
  - Only a few entities
  - Database connection is open for a short time

DbContext – Keys

- Usually mapped by convention
  - public string Id { get; set; }
  - public string ProductId { get; set; }
- Can use alternative field names as well
  - [Key]
    public string UniqueName { get; set; }
  - modelBuilder.Entity<Product>().HasKey(c => c.UniqueName);
- Compose key is also available
  - modelBuilder.Entity<Product>()
        .HasKey(c => new { c.CategoryName, c.UniqueName});
- Mapping tables without PK
  - EF cannot map them
  - But EF Core can since v2.1 (see [KeyLess])

Query

- A DbContext instance is needed to query the database
- The DbContext
  - Registers new and deleted entities
  - Keeps track of modifications made on the entities
  - Registers each retrieved entity
    - Respects the changes made to the entities when querying
- AsNoTracking() -> turn off tracking to save the overhead
  - db.Products.AsNoTracking().Where(...)
  - If the entities are not modified, use this

Inserting and saving new entities

1. Creating a new entity: new Class()
   - var newEntity = new Class()
   - the key is not specified
2. Attach the entity to a DbContext
   - DbContext.DbSet.Add(newEntity)
   - Or attach to an existing entity via a property: someEntity.Property = newEntity
3. DbContext.SaveChanges
4. Entify Framework executes the insert SQL command
5. The database generated values are retrieved and the C# in-memory entities are updated with them
   - newEntity.Id now has a valid value

Modifying entities

- Alter the property
- Changes are recorded by DbContext
- When calling SaveChanges the changes are propagated to the database

```
var course = context.Course.Single(q => q.Neptun=="VIAUAC01");
var aut = context.Department.Single( q => q.Code=="AUT");
course.Name = “Data-driven systems";
course.Department = aut;
context.SaveChanges();
```

Deleting entities

- Only loaded entities can be deleted
- DbSet.Remove(…)
- Must call SaveChanges at the end

```
var c = context.Course.Single(q => q.Neptun=="VIAUAC01");
context.Course.Remove(c);
context.SaveChanges();
```

Entities types: POCO

- „Plain old CLR object”
- Simple C# classes
- Change tracking
  - Automatic, based on comparison DbContext stores the original state of the entities and compares it to the current state (performed when calling SaveChanges)
    - No lazy loading
  - Related entities need to be loaded explicitly (see later)

Entities types: POCO Proxy

- POCO with change tracking and lazy loading (Code first)
- Usually enabled by default, unless we turn this off
- Requirements
  - Public, non-abstract, non-sealed class
  - Public/protected parameter-less constructor
  - Property get: public, non sealed, virtual

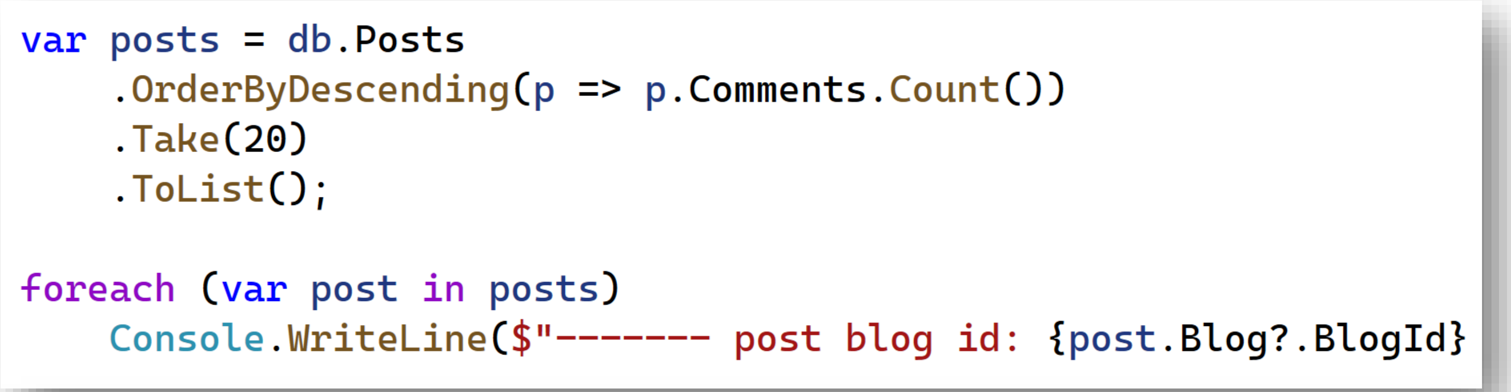
Eager loading / lazy loading

- Eager loading
  - Triggering loading referenced data
  - `context.Products.Include(entity => entity.NavProp)`
  - Yields a single SQL query
- Lazy loading – DO NOT USE!
  - Through navigation properties the related entity is loaded **on-demand when first accessed**
  - Yields a new SQL query
  - EF: usually supported, unless proxy is disabled
  - EF Core: Separate NuGet package since v2.1; proxy-based too

    -
      ```
      void OnConfiguring(DbContextOptionsBuilder ob)
          => ob.UseLazyLoadingProxies().Use...();
      ```

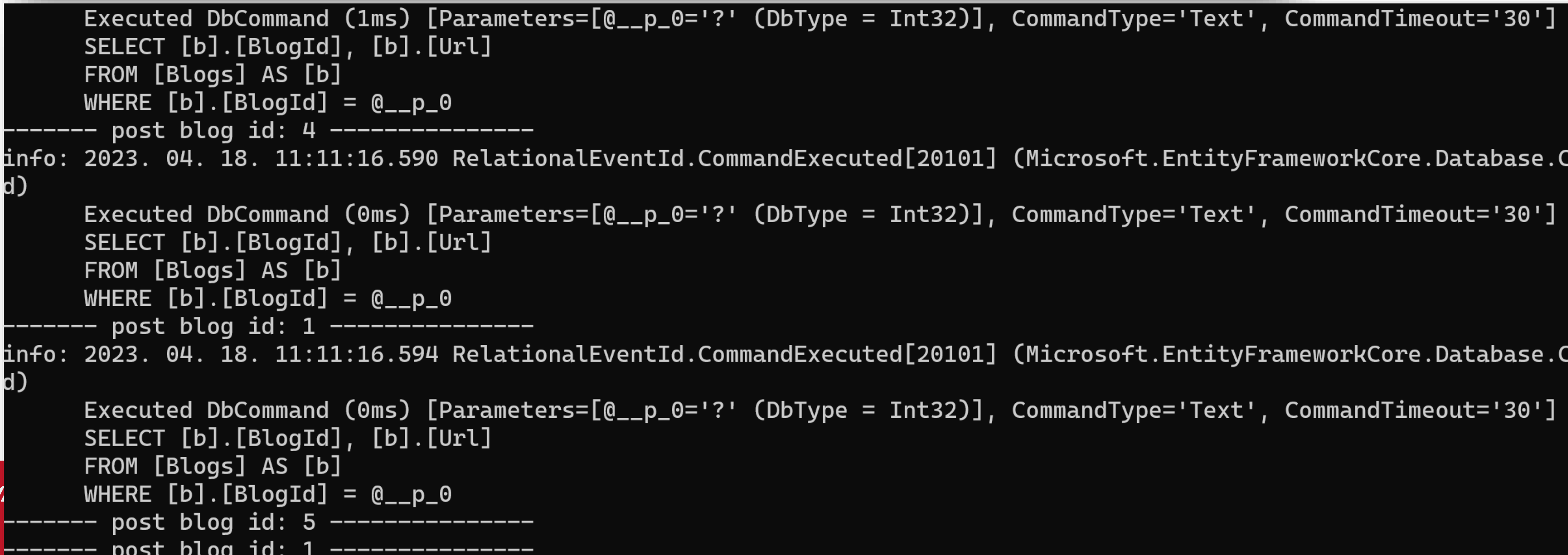
Lazy loading problem

- Too many database requests



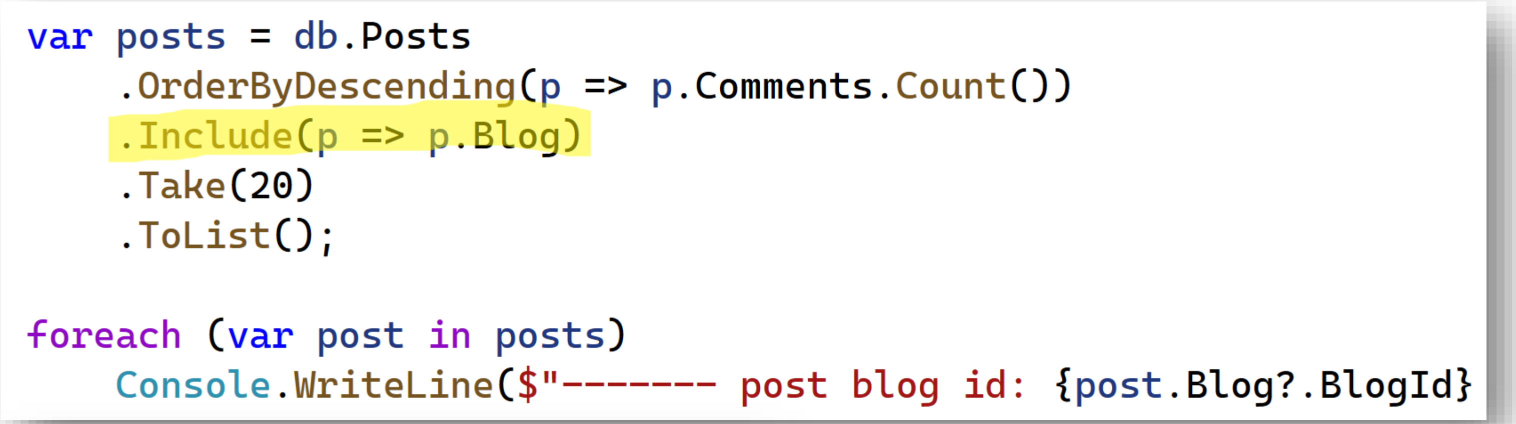
```
var posts = db.Posts
    .OrderByDescending(p => p.Comments.Count())
    .Take(20)
    .ToList();

foreach (var post in posts)
    Console.WriteLine($"------- post blog id: {post.Blog?.BlogId}
```



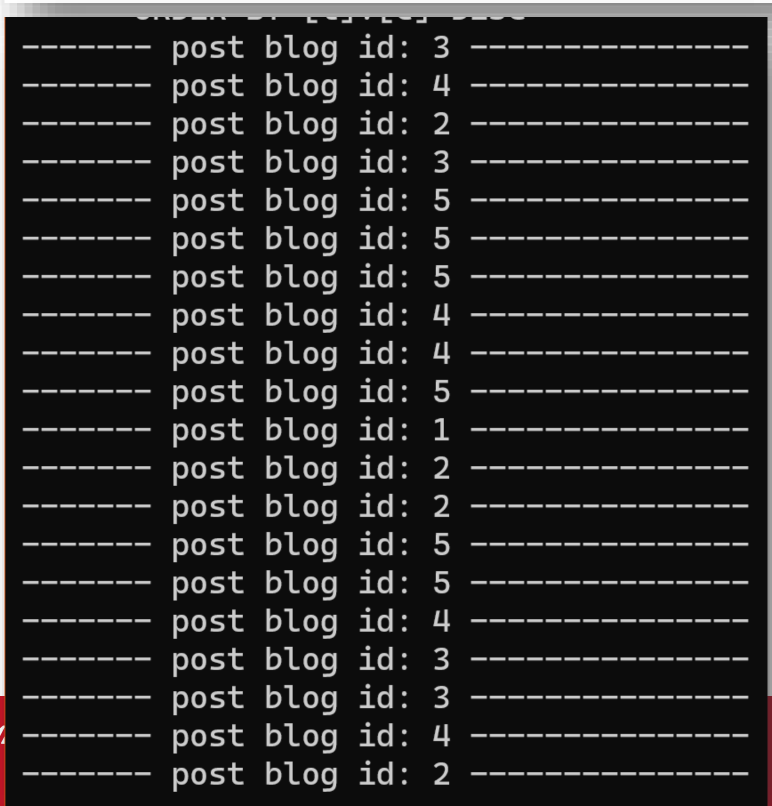
```
      Executed DbCommand (1ms) [Parameters=[@__p_0='?' (DbType = Int32)], CommandType='Text', CommandTimeout='30']
      SELECT [b].[BlogId], [b].[Url]
      FROM [Blogs] AS [b]
      WHERE [b].[BlogId] = @__p_0
------- post blog id: 4 ---------------
info: 2023. 04. 18. 11:11:16.590 RelationalEventId.CommandExecuted[20101] (Microsoft.EntityFrameworkCore.Database.C
d)
      Executed DbCommand (0ms) [Parameters=[@__p_0='?' (DbType = Int32)], CommandType='Text', CommandTimeout='30']
      SELECT [b].[BlogId], [b].[Url]
      FROM [Blogs] AS [b]
      WHERE [b].[BlogId] = @__p_0
------- post blog id: 1 ---------------
info: 2023. 04. 18. 11:11:16.594 RelationalEventId.CommandExecuted[20101] (Microsoft.EntityFrameworkCore.Database.C
d)
      Executed DbCommand (0ms) [Parameters=[@__p_0='?' (DbType = Int32)], CommandType='Text', CommandTimeout='30']
      SELECT [b].[BlogId], [b].[Url]
      FROM [Blogs] AS [b]
      WHERE [b].[BlogId] = @__p_0
------- post blog id: 5 ---------------
------- post blog id: 1 ---------------
```

Solution: eager loaing, Include



```
var posts = db.Posts
    .OrderByDescending(p => p.Comments.Count())
    .Include(p => p.Blog)
    .Take(20)
    .ToList();

foreach (var post in posts)
    Console.WriteLine($"------- post blog id: {post.Blog?.BlogId}
```



```
------- post blog id: 3 ---------------
------- post blog id: 4 ---------------
------- post blog id: 2 ---------------
------- post blog id: 3 ---------------
------- post blog id: 5 ---------------
------- post blog id: 5 ---------------
------- post blog id: 5 ---------------
------- post blog id: 4 ---------------
------- post blog id: 4 ---------------
------- post blog id: 5 ---------------
------- post blog id: 1 ---------------
------- post blog id: 2 ---------------
------- post blog id: 2 ---------------
------- post blog id: 5 ---------------
------- post blog id: 5 ---------------
------- post blog id: 4 ---------------
------- post blog id: 3 ---------------
------- post blog id: 3 ---------------
------- post blog id: 4 ---------------
------- post blog id: 2 ---------------
```

Transaction

- SaveChanges is usually transactional
  - (If the database provider supports it.)

- To save multiple changes in one transaction

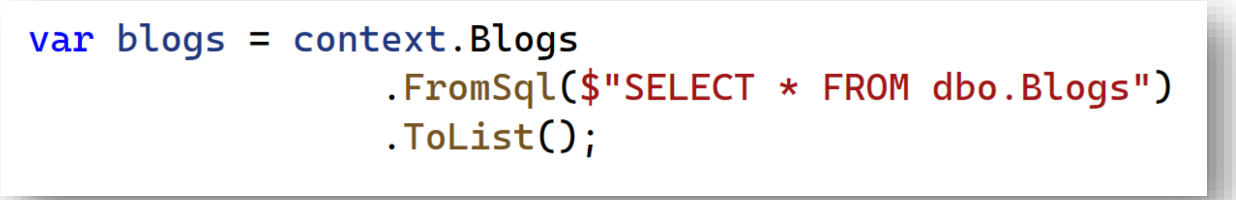
```
using (var context = ...)
using (var dbTran = context.Database.BeginTransaction( ))
{
    // ...
    context.SaveChanges();

    // ...
    context.SaveChanges();

    dbTran.Commit();
}
```

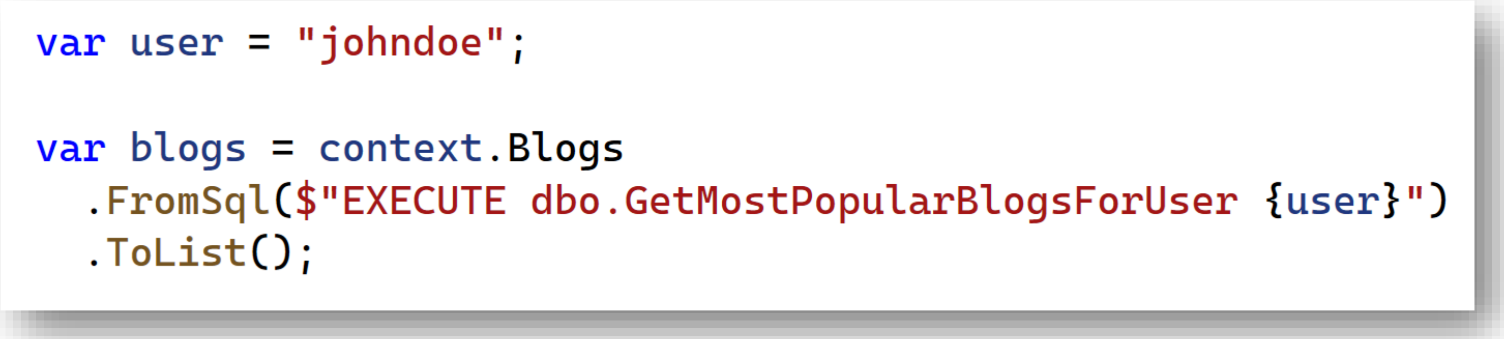
Executing SQL queries

- Simple query



```
var blogs = context.Blogs
                .FromSql($"SELECT * FROM dbo.Blogs")
                .ToList();
```

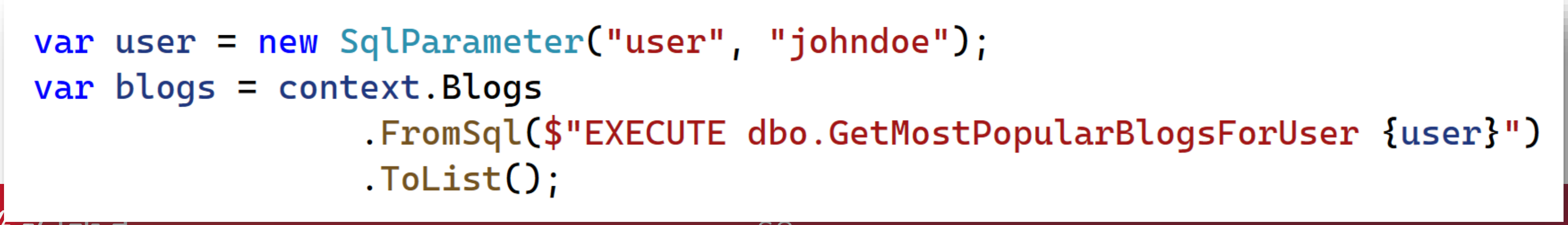
- Parametrized (safe, not concatenated), SP



```
var user = "johndoe";

var blogs = context.Blogs
  .FromSql($"EXECUTE dbo.GetMostPopularBlogsForUser {user}")
  .ToList();
```

- Parameters manually passed



```
var user = new SqlParameter("user", "johndoe");
var blogs = context.Blogs
                .FromSql($"EXECUTE dbo.GetMostPopularBlogsForUser {user}")
                .ToList();
```

Migration

- Changing the schema of the database following the data model in the application

1. Building from an empty database: Create / Delete

2. Preservation of database content: migration

Create / delete

- EnsureCreated and EnsureDeleted methods
  - Appropriate authorization is required
  - In development cycle

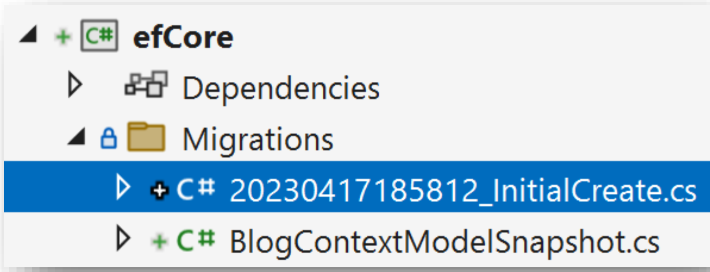
- GenerateCreateScript: SQL script to create the database
  - For example, give it to the installer, etc.

- If you use migration, don't use these!

Migration points

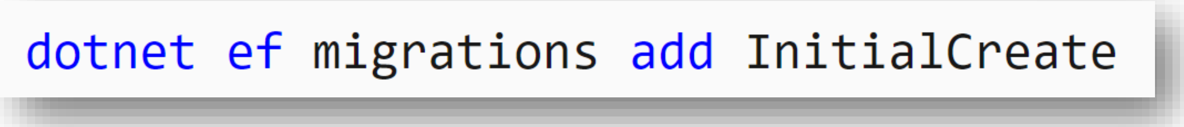
- Migration point
  - A snapshot of the database schema at a point in time
  - You can migrate "up" and "down" between migration points
  - During migration, the schema is changed and the data content is preserved
  - The migration point is referred to by its name
  - The schema description is stored in the source code programmatically with ModelBuilder
  - Source file -> is in source control
- The migration name of the current schema is stored in the database
- We typically create migration points for released versions, we do not use them in the development cycle

Migration -> cmd line



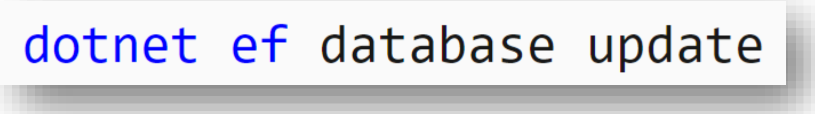
efCore
- Dependencies
- Migrations
  - 20230417185812_InitialCreate.cs
  - BlogContextModelSnapshot.cs

- Creating migration files



```
dotnet ef migrations add InitialCreate
```

  - The name of the migration point is InitialCreate
- Update database for current code

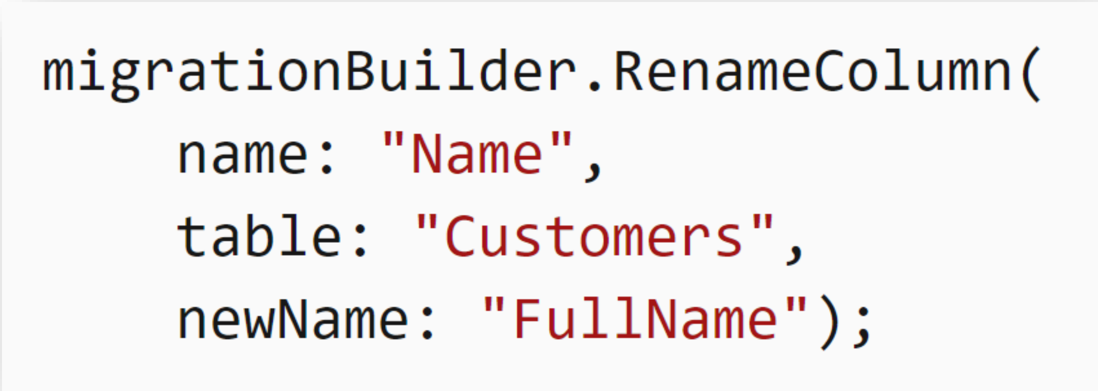


```
dotnet ef database update
```

- The database has the name of the current migration point -> perform the migration from that point
- Given its name for a given migration point

Migration – verify!

- The migration steps must be checked!
  - For example, renaming a column will be delete and create, EF can't figure out by itself that that information should be preserved
  - You can also use your own SQL



```
migrationBuilder.RenameColumn(
    name: "Name",
    table: "Customers",
    newName: "FullName");
```

Reverse engineering (DB first)

- Based on the finished database, we create the entity classes and the dbContext class
- From the command line or with EF Core Power Tools
- Partial classes are created
  - By supplementing it in other files, a new database can be generated on top of existing files
- The generation is based on a T4 template and can be easily customized