

DB Apps Introduction

Intro to ADO.NET



Microsoft®
ADO.NET



SoftUni Team
Technical Trainers



SoftUni
Foundation

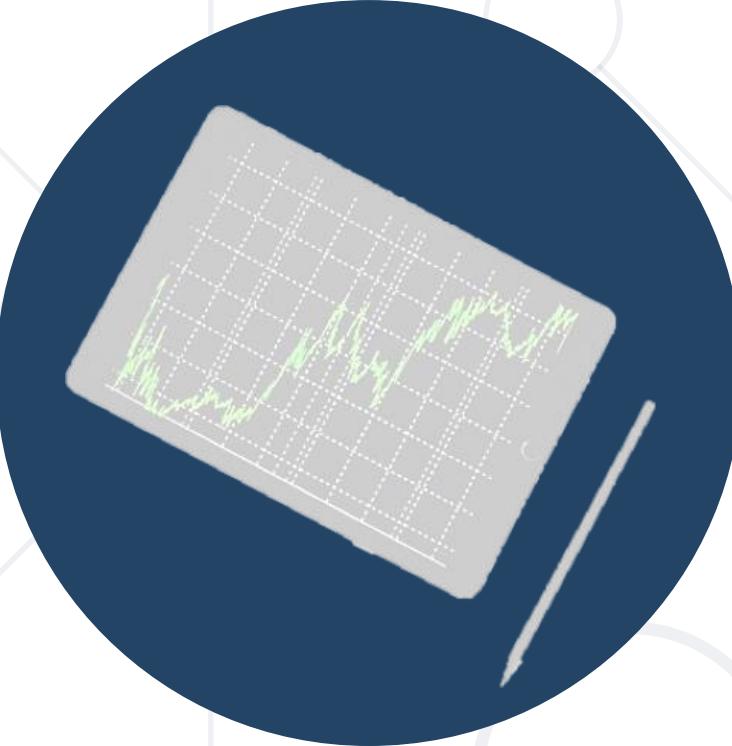


Software University
<http://softuni.bg>

Table of Contents

1. Data Access Models
2. ADO.NET Architecture
3. Accessing SQL Server from **ADO.NET**
4. SQL Injection



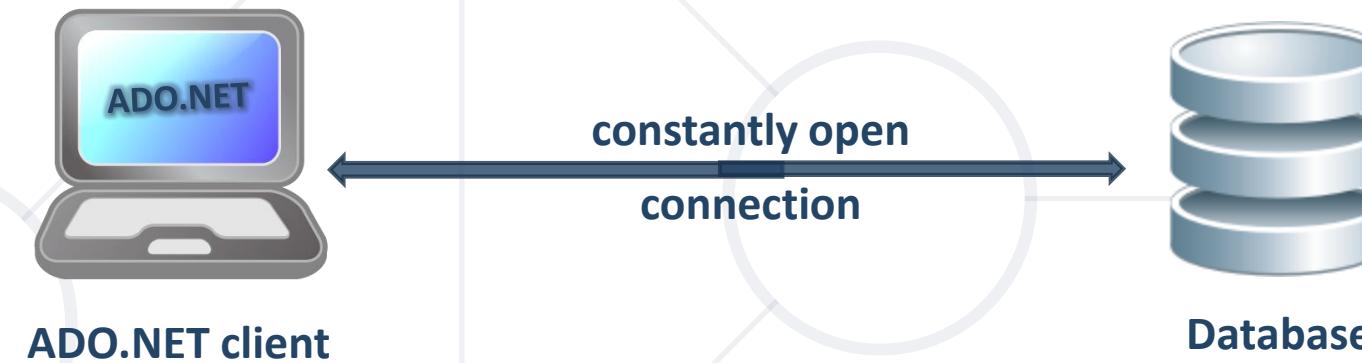


Data Access Models

Connecting to a DB through C#

Connected Model

- **Connected data access model**
 - Applicable to an environment where the database is constantly available



Connected Model: Benefits and Drawbacks

- **Connected data access model (`SqlClient`)**
 - **Benefits:**
 - Concurrency control is easier to maintain
 - Better chance to work with the most recent version of the data
 - **Drawbacks:**
 - Needs a constant reliable network
 - Problems when scalability is an issue

ADO.NET Architecture



What Is ADO.NET?

- ADO.NET is a standard **.NET class library** for accessing databases, processing data and XML
- Supports connected, disconnected and ORM data access models
 - Excellent integration with **LINQ**
 - Allows executing SQL in **RDBMS** systems
 - Allows accessing data in the **ORM** approach

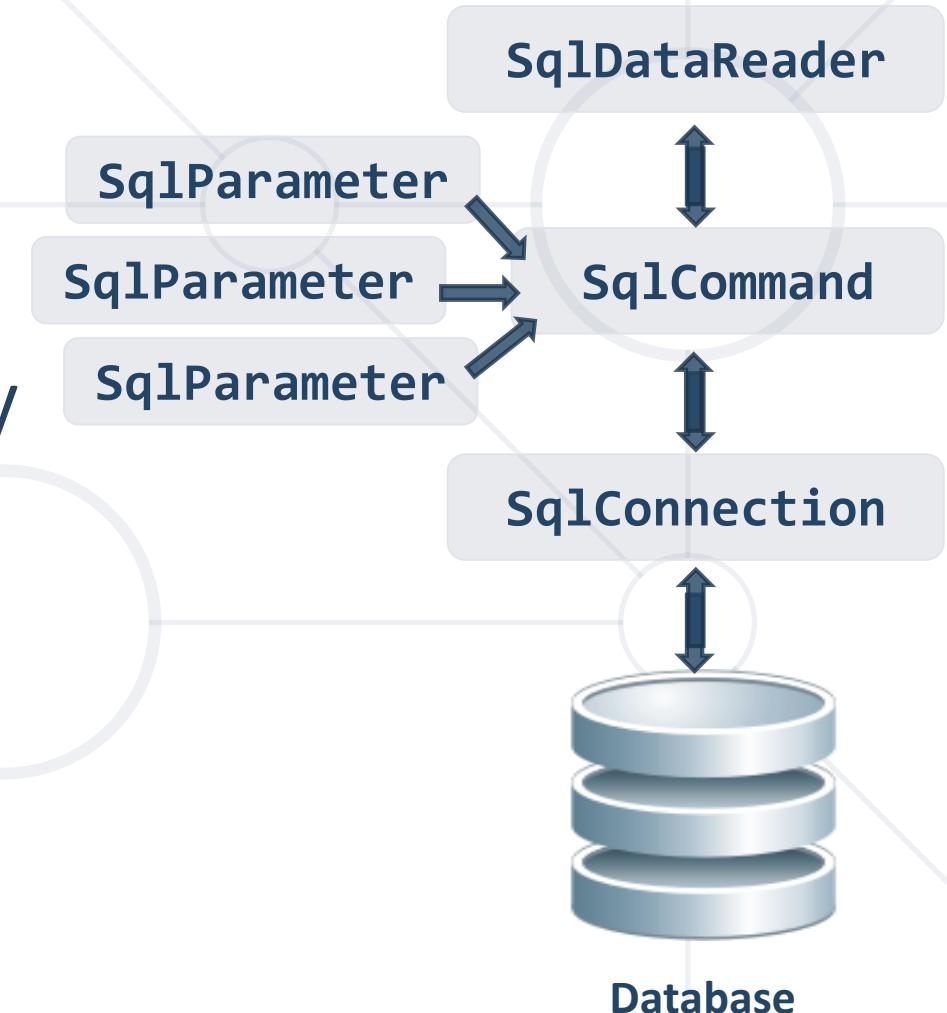
- Data Providers are collections of classes that provide access to various databases
 - For different RDBMS systems different **Data Providers** are available
- Several common objects are defined
 - **Connection** – to connect to the database
 - **Command** – to run an SQL command
 - **DataReader** – to retrieve data

Data Providers in ADO.NET (2)

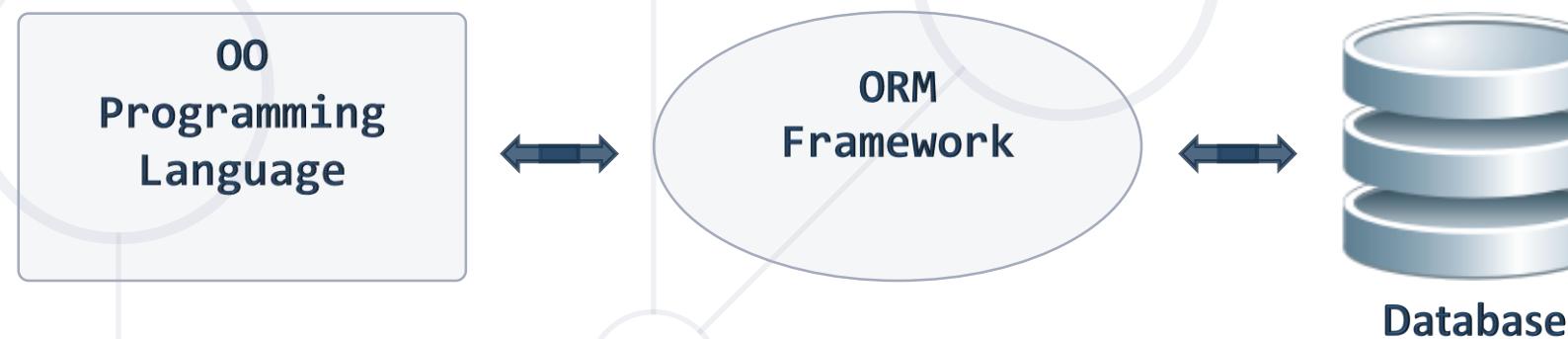
- Several standard ADO.NET Data Providers come as part of .NET Framework
 - **SqlClient** – accessing **SQL Server**
 - **OleDb** – accessing standard **OLE DB** data sources
 - **Odbc** – accessing standard **ODBC** data sources
 - **Oracle** – accessing **Oracle** databases
- Third party Data Providers are available for:
 - **MySQL, PostgreSQL, Interbase, DB2, SQLite**
 - Other RDBMS systems and data sources
 - SQL Azure, Salesforce CRM, Amazon SimpleDB, ...

SqlClient and ADO.NET Connected Model

- Retrieving data in connected model
 - Open a connection (**SqlConnection**)
 - Execute command (**SqlCommand**)
 - Process the result set of the query by using a reader (**SqlDataReader**)
 - Close the reader
 - Close the connection



- **ORM data access model** (Entity Framework Core)
 - Maps **database tables** to **classes** and **objects**
 - Objects can be **automatically persisted** in the database
 - Can operate in both connected and disconnected modes



ORM Model – Benefits and Problems

- **ORM** model benefits
 - Less code
 - Use objects with **associations** instead of tables and SQL
 - Integrated object query mechanism
- ORM model drawbacks:
 - Less flexibility
 - SQL is automatically generated
 - Performance issues (sometimes)

- **Entity Framework Core** is a generic **ORM** framework
 - Create entity data model mapping the database
 - Open an object context
 - Retrieve data with LINQ / modify the tables in the object context
 - Persist the object context changes into the DB
 - Connection is automatically managed



SQL Client Data Provider

- **SqlConnection**
 - Establish database connection to SQL Server
- **SqlCommand**
 - Executes SQL commands on the SQL Server through an established connection
 - Could accept parameters (**SqlParameter**)
- **SqlDataReader**
 - Retrieves data (record set) from SQL Server as a result of SQL query execution

The SqlConnection Class

- **SqlConnection** establishes a connection to SQL Server database
 - Requires a valid connection string
- Connection string example:

```
Data Source=(local)\SQLEXPRESS;Initial Catalog=SoftUni;Integrated Security=true;
```
- Connecting to SQL Server:

```
SqlConnection con = new SqlConnection(  
    @"Server=.;  
    Database=SoftUni;  
    Integrated Security=true");  
con.Open();
```

SqlConnection – Example

- Creating and opening connection to SQL Server (database **SoftUni**)

```
SqlConnection dbCon = new SqlConnection(  
    "Server=.\SQLEXPRESS; " +  
    "Database=SoftUni; " +  
    "Integrated Security=true");  
dbCon.Open();  
using (dbCon)  
{  
    //TODO: Use the connection to execute SQL commands here ...  
}
```

- **Database connection string**
 - Defines the parameters needed to establish the connection to the database
- Settings for **SQL Server connections**:
 - **Data Source / Server** – server name / IP address + database instance name
 - **Database / Initial Catalog** – database name
 - **User ID / Password** – credentials
 - **Integrated Security** – false if credentials are provided

Working with SqlConnection

- Explicitly opening and closing a connection
 - **Open()** and **Close()** methods
 - Works through the connection pool
- DB connections are **IDisposable** objects
 - Always use the **using** construct in C#!



The SqlCommand Class

- More important methods
 - **ExecuteScalar()**
 - Returns a single value - the value in the first column of the first row of the result set (as **System.Object**)
 - **ExecuteReader()**
 - Returns a **SqlDataReader**
 - It is a cursor over the returned records (result set)
 - **CommandBehavior** – assigns some options
 - **ExecuteNonQuery()**
 - Used for non-query SQL commands, e.g. **INSERT**
 - Returns the number of affected rows (**int**)

SqlCommand – Example

```
SqlConnection dbCon = new SqlConnection(  
    "Server=.; " +  
    "Database=SoftUni; " +  
    "Integrated Security=true");  
dbCon.Open();  
using(dbCon)  
{  
    SqlCommand command = new SqlCommand(  
        "SELECT COUNT(*) FROM Employees", dbCon);  
    int employeesCount = (int) command.ExecuteScalar();  
    Console.WriteLine("Employees count: {0}", employeesCount);  
}
```

The SqlDataReader Class

- **SqlDataReader** retrieves a sequence of records (cursor) returned as result of an SQL command
 - Data is available for reading only (can't be changed)
 - Forward-only row processing (no move back)
- Important properties and methods:
 - **Read()** – moves the cursor forward and returns false if there is no next record
 - **Indexer[]** – retrieves the value in the current record by given column name or index
 - **Close()** – closes the cursor and releases resources

SqlDataReader – Example

```
SqlConnection dbCon = new SqlConnection(...);  
dbCon.Open();  
using(dbCon)  
{  
    SqlCommand command = new SqlCommand("SELECT * FROM Employees", dbCon);  
    SqlDataReader reader = command.ExecuteReader();  
    using (reader)  
    {  
        while (reader.Read())  
        {  
            string firstName = (string)reader["FirstName"];  
            string lastName = (string)reader["LastName"];  
            decimal salary = (decimal)reader["Salary"];  
            Console.WriteLine("{0} {1} - {2}", firstName, lastName, salary);  
        }  
    }  
}
```

Fetch more rows until finished



SQL Injection

What is SQL Injection? How to Prevent It?

What is SQL Injection? (1)

```
bool IsPasswordValid(string username, string password)
{
    string sql =
        $"SELECT COUNT(*) FROM Users " +
        $"WHERE UserName = '{username}' AND" +
        $"PasswordHash = '{CalcSHA1(password)}'";
    SqlCommand cmd = new SqlCommand(sql, dbConnection);

    int matchedUsersCount = (int)cmd.ExecuteScalar();
    return matchedUsersCount > 0;
}
```

What is SQL Injection? (2)

```
bool normalLogin =  
    IsPasswordValid("peter", "qwerty123"); // true  
  
bool sqlInjectedLogin =  
    IsPasswordValid(" ' or 1=1 --", "qwerty123"); // true  
  
bool evilHackerCreatesNewUser =  
    IsPasswordValid("' INSERT INTO Users VALUES('hacker', '') --",  
    "qwerty123");
```

How Does SQL Injection Work?

- The following SQL commands are executed:

- Usual password check (no SQL injection):

```
SELECT COUNT(*) FROM Users WHERE UserName = 'peter'  
AND PasswordHash = 'X0wXWxZePV5kiyeE86Ejvb+rIG/8='
```

- SQL-injected password check:

```
SELECT COUNT(*) FROM Users WHERE UserName = ' ' or 1=1  
-- ' AND PasswordHash = 'X0wXWxZePV5kiyeE86Ejvb+rIG/8='
```

- SQL-injected INSERT command:

```
SELECT COUNT(*) FROM Users WHERE UserName = ''  
INSERT INTO Users VALUES('hacker', '')  
-- ' AND PasswordHash = 'X0wXWxZePV5kiyeE86Ejvb+rIG/8='
```

Preventing SQL Injection

- Ways to prevent the SQL injection:
 - SQL-escape all data coming from the user:

```
string escapedUsername = username.Replace("'", "''");
string sql =
    "SELECT COUNT(*) FROM Users " +
    "WHERE UserName = '" + escapedUsername + "' and " +
    "PasswordHash = '" + CalcSHA1(password) + "'";
```

- Not recommended: use as last resort only!
- Preferred approach:
 - Use **parameterized queries**
 - Separate the SQL command from its arguments

The SqlParameter Class

- What are **SqlParameters**?
 - SQL queries and stored procedures can have input and output parameters
 - Accessed through the **Parameters** property of the **SqlCommand** class
- Properties of **SqlParameter**:
 - **ParameterName** – name of the parameter
 - **DbType** – SQL type (**NVarChar**, **Timestamp**, ...)
 - **Size** – size of the type (if applicable)
 - **Direction** – input / output

Parameterized Commands – Example

```
void InsertProject(string name, string description, DateTime startDate)
{
    SqlCommand cmd = new SqlCommand(
        "INSERT INTO Projects " +
        "(Name, Description, StartDate, EndDate) VALUES " +
        "(@name, @desc, @start, @end)", dbCon);

    cmd.Parameters.AddWithValue("@name", name);
    cmd.Parameters.AddWithValue("@desc", description);
    cmd.Parameters.AddWithValue("@start", startDate);

    cmd.ExecuteNonQuery();
}
```



Connecting to Non-Microsoft Databases

Connecting to Non-Microsoft Databases

- ADO.NET supports accessing various databases via their Data Providers:
 - **OLE DB** – supported internally in ADO.NET
 - Access any OLE DB-compliant data source
 - E.g. MS Access, MS Excel, MS Project, MS Exchange, Windows Active Directory, text files
 - **Oracle** – supported internally in ADO.NET
 - **MySQL** – third party extension

ORM Fundamentals

The ORM Concept, Config, CRUD Operations

SoftUni Team
Technical Trainers



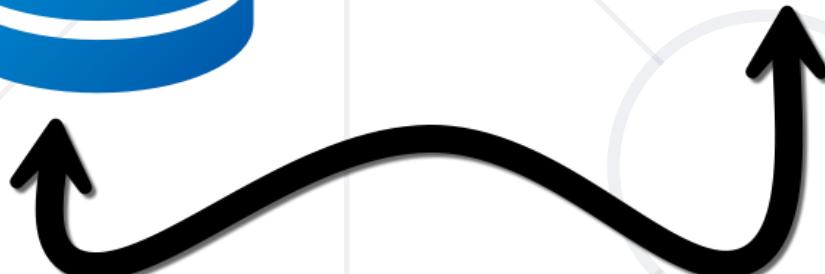
SoftUni
Foundation

Relational DB



Object

```
[Table(Name = "Customers")]
public class Customer
{}
```



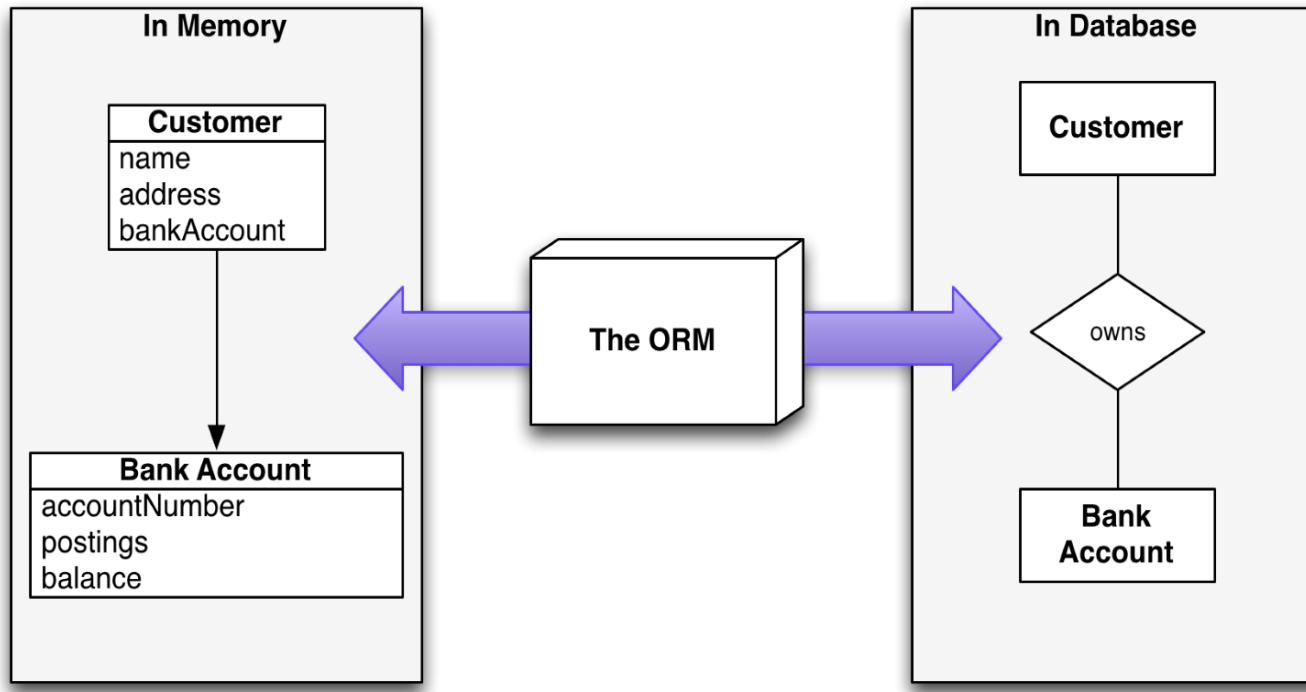
Software University
<http://softuni.bg>



Table of Contents

1. ORM Technologies: Basic Concepts
2. ORM: Advantages and Disadvantages
3. Writing an ORM Framework from Scratch
 - Retrieving Entities from Database
 - Mapping Navigation Properties
 - Change Tracking
 - Generating SQL





Introduction to ORM

Object-Relational Mapping

What is ORM?

- Object-Relational Mapping (ORM) allows manipulating databases **using common classes and objects**
 - Database Tables → C#/Java/etc. classes

Employees	
Id	
FirstName	
MiddleName	
LastName	
IsEmployed	
DepartmentId	



```
public class Employee
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public bool IsEmployed { get; set; }
    public Department Department { get; set; }
}
```

ORM Frameworks: Features

- ORM frameworks typically provide the following functionality:

- Automatically generate SQL to perform data operations

```
database.Employees.Add(new Employee  
{  
    FirstName = "Gosho",  
    LastName = "Ivanov",  
    IsEmployed = true    });
```



```
INSERT INTO Employees  
(FirstName, LastName, IsEmployed)  
VALUES  
(‘Gosho’, ‘Ivanov’, 1)
```

- Create object model from database schema (DB First model)
 - Create database schema from object model (Code First model)
 - Query data by object-oriented API (e.g. LINQ queries)

ORM Advantages and Disadvantages

- Object-relational mapping (ORM) **advantages**:
 - Developer productivity: **writing less code**
 - Abstract from differences between object and relational world
 - **Manageability of the CRUD operations** for complex relationships
 - **Easier maintainability**
- **Disadvantages**:
 - **Reduced performance** (due to overhead or autogenerated SQL)
 - **Reduces flexibility** (some operations are hard to implement)

- ADO.NET provides an interface between our apps and the database engine
- Different engines can be used with other data providers
- SQL commands must be parametrized to prevent malicious behavior





MiniORM Core

Custom ORM Framework Overview and Features

MiniORM Core: Overview

- Designed after **Entity Framework Core**
- Provides **LINQ-based data queries** and **CRUD operations**
- **Change tracking** of in-memory objects
- Maps navigation properties
- Maps collections
 - **One-to-many**, **Many-to-many**, etc.



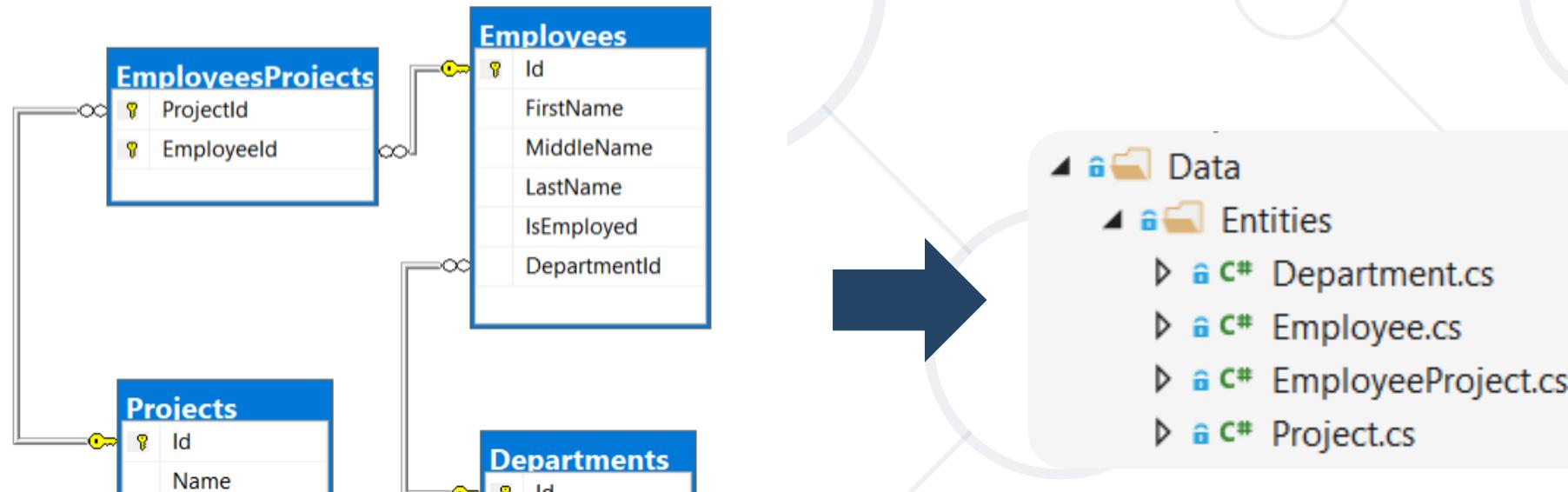
MiniORM Core

MiniORM Core Workflow: Overview

- **Define data model** (database-first)
 - Entity Classes
 - **DbContext** (with **DbSets**)
- Initialize **DbContext**
 - Using connection string
 - Query data using context
 - Manipulate data (add/remove/update entities)
 - Context gets persisted into database

Database First Model

- **Database First model** - models the entity classes after the database



- The **DbContext** class
 - Holds the **database connection** and the **DB Sets**
 - Provides **LINQ-based** data access
 - Provides **change tracking**, and an API for **CRUD** operations
- **DB Sets**
 - Hold **entities** (objects with their attributes and relations)
 - Each database **table** is typically mapped to a single **C# class**

MiniORM Components (2)

- **Associations** (relationship mappings)
 - An association is **a primary key / foreign key-based relationship** between two entity classes
 - Allows **navigation** from one entity to another

```
var courses = student.Courses.Where(...);
```
 - MiniORM **supports one-to-one, one-to-many and many-to-many relationships**

```
public class Employee
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public bool IsEmployed { get; set; }
    public Department Department { get; set; }
}
```

Entity Classes

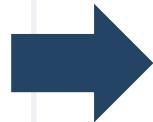
Data Holders

Entity Classes

- Entity classes are regular C# classes
- Used for storing the data from the DB in-memory



Employees	
Id	
FirstName	
MiddleName	
LastName	
IsEmployed	
DepartmentId	



```
public class Employee
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public bool IsEmployed { get; set; }
    public Department Department { get; set; }
}
```

Entity Classes: Navigation Properties

- Reference type properties
- Point to relevant object, connected by foreign key
- Set by the framework
- Example: Employee's Department:

```
public class Employee {  
    public int Id { get; set; }  
    ...  
    [ForeignKey(nameof(Department))]  
    public int DepartmentId { get; set; }  
    public Department Department { get; set; }  
}
```

Entity Classes: Navigation Properties (2)

- Navigation Properties can also be collections
- Usually of type **ICollection<T>**
- Holds all of the objects whose **foreign keys** are the same as the entity's **primary key**
- Set by the ORM framework

```
public class Department
{
    public int Id { get; set; }
    ...
    public ICollection<Employee> Employees { get; set; }
}
```

```
public class DbSet< TEntity > : ICollection< TEntity >
    where TEntity : class, new()
{
    internal DbSet([NotNull] IEnumerable< TEntity > entities) ...
    internal ChangeTracker< TEntity > ChangeTracker { get; set; }
    internal IList< TEntity > Entities { get; set; }
    public void Add(TEntity item) ...
    public void Clear() ...
    public bool Contains(TEntity item) => this.Entities.Contains(item);
```

DbSet<T>

Specialized Collections

DbSet<T> Class

- Generic collection with additional features
- Each **DbSet<T>** corresponds to a single database table
- Inherits from **ICollection<T>**
 - **foreach**-able
 - Supports **LINQ** operations
- Usually several **DbSets** are part of a **DbContext**



- Each DbSet tracks its own entities through a change tracker
- Has every other feature of an ICollection<T>
 - **Adding/Updating** elements
 - **Removing** an entity/a range of entities
 - **Checking** for element **existence**
 - Accessing the **count** of elements



DbContext

DbContext Class

- Holds several **DbSet<T>**
- Responsible for **populating** the **DbSets**
- Users create a **DbContext**, which **inherits** from **DbContext**
 - Using one DbSet per database table

```
public class SoftUniDbContext : DbContext
{
    public DbSet<Employee> Employees { get; }
    public DbSet<Department> Departments { get; }
    public DbSet<Project> Projects { get; }
    public DbSet<EmployeeProject> EmployeesProjects { get; }
}
```

```
internal class ChangeTracker<T>
    where T: class, new()
{
    private readonly List<T> allEntities;

    private readonly List<T> added;

    private readonly List<T> removed;

    public ChangeTracker(IEnumerable<T> entities) ...
    private static List<T> CloneEntities(IEnumerable<T> entities) ...
    public IReadOnlyCollection<T> AllEntities => this.allEntities.AsReadOnly();
    public IReadOnlyCollection<T> Added => this.added.AsReadOnly();
    public IReadOnlyCollection<T> Removed => this.removed.AsReadOnly();
}
```

ChangeTracker<T>

Change Tracking Class

ChangeTracker<T>

- Container for tracking changes
- Holds 3 collections:
 - **All** entities
 - **Added** entities
 - **Removed** entities
- Also can track **modified entities**
 - Through **cloning entities** at initialization



ChangeTracker<T>: Cloning Entities

- In order to check for entity modification, the change tracker **clones** all entities on initialization
- Cloning process:
 - Create **new blank instance** of entity
 - Find all **properties**, which are valid SQL types
 - Set blank instance's property **values** to existing entity values

ChangeTracker<T>: Cloning Entities (2)

- Cloning Process:

```
private static List<T> CloneEntities(IEnumerable<T> entities)
{
    var clonedEntities = new List<T>();
    var propertiesToClone =
        // TODO: get properties with SQL types

    foreach (var entity in entities) {
        var clonedEntity = Activator.CreateInstance<T>();
        foreach (var property in propertiesToClone) {
            var value = property.GetValue(entity);
            property.SetValue(clonedEntity, value);
        }
        clonedEntities.Add(clonedEntity);
    }
    return clonedEntities;
}
```

Writing an ORM Framework

Live Demo

```
internal class ChangeTracker<T>
where T : class, new()
{
    private readonly List<T> allEntities;
    private readonly List<T> added;
    private readonly List<T> removed;
    public ChangeTracker(IEnumerable<T> entities)...
    private static List<T> CloneEntities(IEnumerable<T> entities)...
    public IReadOnlyCollection<T> AllEntities => this.allEntities.AsReadOnly();
    public IReadOnlyCollection<T> Added => this.added.AsReadOnly();
    public IReadOnlyCollection<T> Removed => this.removed.AsReadOnly();
    public void Add(T item) => this.added.Add(item);
    public void Remove(T item) => this.removed.Add(item);
    public IEnumerable<T> GetModifiedEntities(HashSet<T> dbSet)...
    private static bool IsModified(T entity, T proxyEntity)...
    private static IEnumerable<object> GetPrimaryKeyValues(IEnumerable<PropertyInfo> primaryKeys, T enti
```

```
internal class ConnectionManager : IDisposable
{
    private readonly DatabaseConnection connection;
    public ConnectionManager(DatabaseConnection connection)...
    public void Dispose()...
}

public abstract class DbContext
{
    private readonly DatabaseConnection connection;
    private readonly Dictionary<Type, PropertyInfo> dbSetProperties;
    internal static readonly Type[] AllowedSqlTypes =
    {
        typeof(string),
        typeof(int),
        typeof(uint),
        typeof(long),
        typeof(ulong),
        typeof(decimal),
        typeof(bool),
        typeof(DateTime)
    };
    protected DbContext(string connectionString)...
    public void SaveChanges()...
    [UsedImplicitly]
    private void Persist< TEntity >(DbSet< TEntity > dbSet, SqlTransaction transaction)...
    private void InitializeDbSets()...
    private void MapAllRelations()...
}
```



Reading Data

Querying the DB using MiniORM

Using DbContext Class

- First create instance of the **DbContext**:

```
var context = new SoftUniDbContext(connectionString);
```

- In the constructor you can pass a database connection string
- **DbContext properties:**
 - All **entity classes** (tables) are listed as **properties**
 - e.g. **DbSet<Employee> Employees { get;}**

Reading Data with LINQ Query

- Executing **LINQ-to-Entities** query over entity:

```
var context = new  
SoftUniDbContext(connectionString)  
  
var employees = context.Employees  
.Where(e => e.JobTitle == "Design Engineer")  
.ToArray();
```

- **Employees** property in the **DbContext**:

```
public class SoftUniDbContext : DbContext  
{  
    public DbSet<Employee> Employees { get; }  
    public DbSet<Project> Projects { get; }  
    public DbSet<Department> Departments { get; }  
}
```

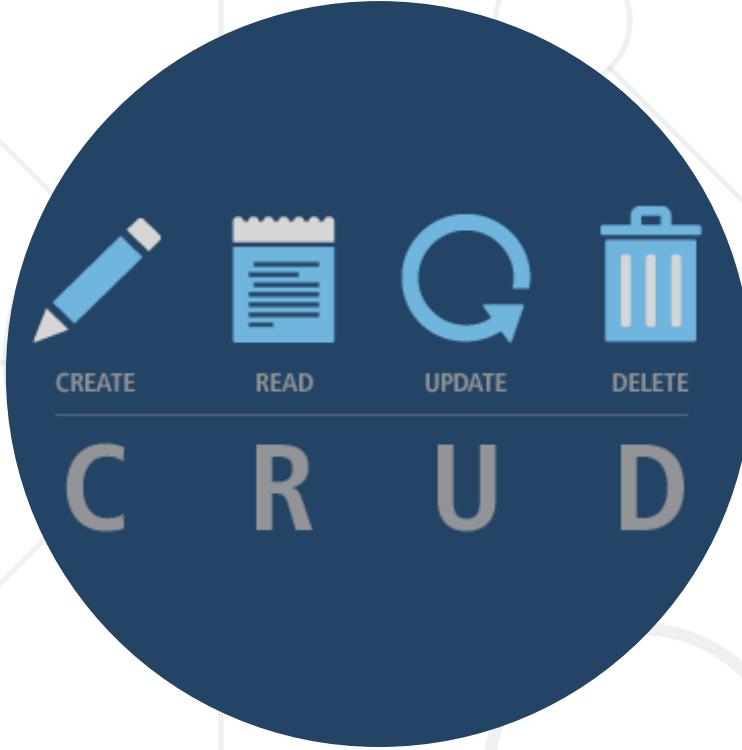
Reading Data with LINQ Query

- We can also use **extension methods** for constructing the query

```
var context = new  
SoftUniDbContext(connectionString)  
var employees = context.Employees  
.Where(c => c.JobTitle == "Design Engineering")  
.Select(c => c.FirstName)  
.ToList();
```

- Find element by **ID**

```
var context = new SoftUniEntities()  
var project = context.Projects  
.Single(e => e.Id == 2);  
Console.WriteLine(project.Name);
```



CRUD Operations With MiniORM

Creating New Entities

- To create a new database row use the method **Add(...)** of the corresponding DbSet:

```
var project = new Project()
{
    Name = "Judge System"
};

context.Projects.Add(project);
context.SaveChanges();
```

Create a new
Project object

Add the object to the DbSet

Execute SQL statements



Updating Existing Entities

- **DbContext** allows modifying entity properties and persisting them in the database
 - Just load an entity, modify it and call **SaveChanges()**
- The **DbContext** automatically tracks all **changes** made on its entity **objects**

```
Employees employee =  
    context.Employees.First();  
employee.FirstName = "Alex";  
context.SaveChanges();
```

SELECT the first
order

Execute an SQL
UPDATE

Deleting Existing Data

- Delete is done by **Remove()** on the specified entity collection
- SaveChanges()** method performs the delete action in the database

```
Employees employee =  
    softUniEntities.Employees.First();  
softUniEntities.Employees.Remove(employee);  
softUniEntities.SaveChanges();
```

Mark the entity for deleting
at the next save

Execute the SQL DELETE
command

- **ORM frameworks** map database schema to objects in a programming language
- **LINQ** can be used to query the DB through the **DB context**

```
var employees = context.Employees
    .Where(c => c.JobTitle == "Design
Engineering")
    .Select(c => c.FirstName)
    .ToList();
```



Introduction to Entity Framework Core

The ORM Concept, Config, CRUD Operations

Cre

SoftUni Team

Technical Trainers



Table of Contents

1. Entity Framework Core Overview
2. Database First Model
3. CRUD Operations Using Entity Framework Core
4. Working with LINQ





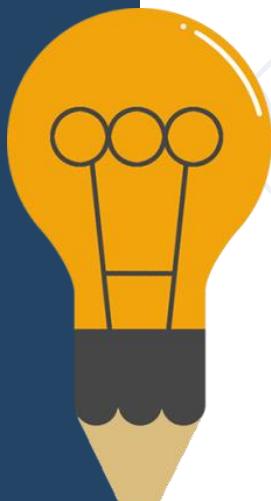
Core

Entity Framework Core

Overview and Features

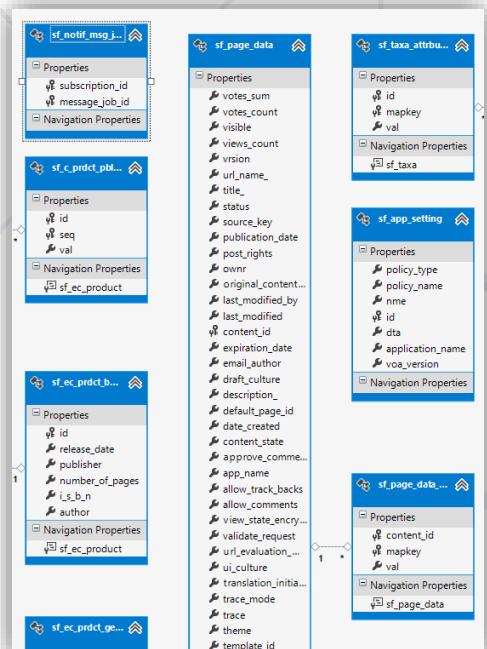
Entity Framework Core: Overview

- The standard **ORM framework** for **.NET** and **.NET Core**
- Provides LINQ-based data queries and **CRUD** operations
- Automatic **change tracking** of in-memory objects
- Works with many relational databases (with different providers)
- Open source with independent release cycle



EF Core: Basic Workflow

1. Define the data model (**Code First or Scaffold from DB**)



2. Write & execute query over **IQueryable**

```
var toolName = "";

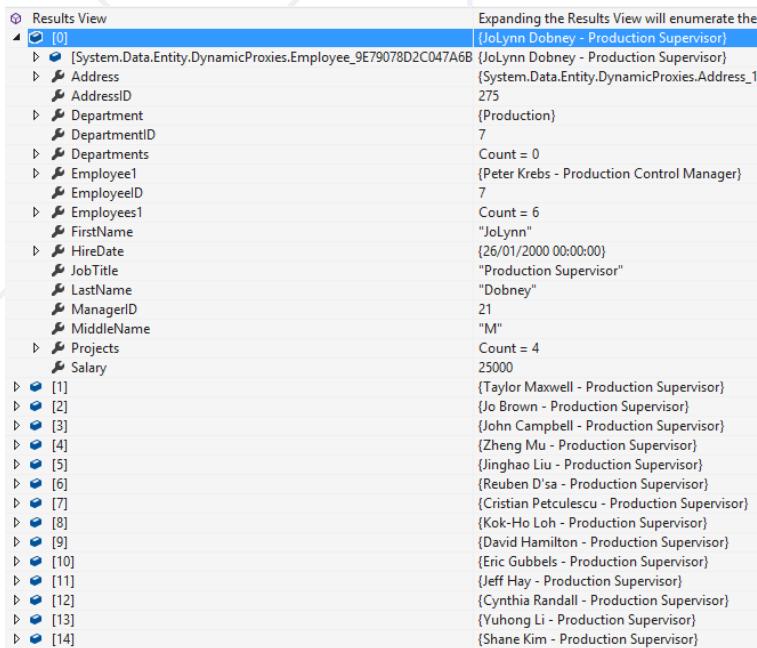
var snippetOptions = DefaultToolGroup
    .Tools
    .OfType<EditorListTool>()
    .Where(t =>
        t.Name == toolName &&
        t.Items != null &&
        t.Items.Any())
    .SelectMany(
        (t, index) =>
            t.Items
            .Select(item =>
                new {
                    text = item.Text,
                    value = item.Value
                }));
if (snippetOptions.Any())
{
    options[toolName] = snippetOptions;
}
```

3. EF generates & executes an **SQL query** in the **DB**

```
exec sp_executesql N'SELECT
[Filter2].[UserInCourseId] AS [UserInCourse]
[Filter2].[UserId] AS [UserId],
[Filter2].[CourseInstanceId] AS [CourseIns
[Filter2].[FirstCourseGroupId] AS [FirstCou
[Filter2].[SecondCourseGroupId] AS [SecondC
[Filter2].[ThirdCourseGroupId] AS [ThirdCou
[Filter2].[FourthCourseGroupId] AS [FourthC
[Filter2].[FifthCourseGroupId] AS [FifthCou
[Filter2].[IsLiveParticipant] AS [IsLivePar
[Filter2].[Accommodation] AS [Accommodatio
[Filter2].[ExcellentResults] AS [ExcellentR
[Filter2].[Result] AS [Result],
[Filter2].[CanDoTestExam] AS [CanDoTestExam
[Filter2].[CourseTestExamId] AS [CourseTest
[Filter2].[TestExamPoints] AS [TestExamPoin
[Filter2].[CanDoPracticalExam] AS [CanDoPra
[Filter2].[CoursePracticalExamId] AS [Cour
[Filter2].[PracticalExamPoints] AS [Practic
[Filter2].[AttendancesCount] AS [Attendance
[Filter2].[HomeworkEvaluationPoints] AS [Ho
FROM (SELECT [Extent1].[UserInCourseId] A
AS [SecondCourseGroupId], [Extent1].[Thirdc
[IsLiveParticipant], [Extent1].[Accommodati
[CourseTestExamId], [Extent1].[TestExamPoin
[PracticalExamPoints], [Extent1].[Attendance
FROM [courses].[UsersInCourses] AS
INNER JOIN [courses].[CoursePractic
WHERE ( EXISTS (SELECT
    1 AS [c1]
    FROM [courses].[CoursePract
    WHERE [Extent1].[UserInCourse
)) AND ([Extent2].[AllowExamFileSv
INNER JOIN [courses].[CoursePracticalExams]
WHERE ([Filter2].[UserId] = @p_linq_0) AN
```

EF Core: Basic Workflow (2)

4. EF transforms the query results into .NET objects



The screenshot shows the 'Results View' tool window in Visual Studio. It displays a list of 15 employees from the 'Employees' table. Each item in the list is a dynamic proxy object representing an employee record. The columns shown are EmployeeID, FirstName, LastName, MiddleName, JobTitle, DepartmentID, ManagerID, HireDate, and Salary.

EmployeeID	FirstName	LastName	MiddleName	JobTitle	DepartmentID	ManagerID	HireDate	Salary
1	Janet	Leverette		Accounting Manager	1	2	2003-07-01	30000
2	Renee	Kempler		Marketing Manager	2	3	2003-07-01	30000
3	Michael	Schoen		Marketing Manager	2	4	2003-07-01	30000
4	Susan	Garcia		Marketing Manager	2	5	2003-07-01	30000
5	Patricia	Wong		Marketing Manager	2	6	2003-07-01	30000
6	James	Burke		Marketing Manager	2	7	2003-07-01	30000
7	Steven	King		Marketing Manager	2	8	2003-07-01	30000
8	Robert	King		Marketing Manager	2	9	2003-07-01	30000
9	David	Chen		Marketing Manager	2	10	2003-07-01	30000
10	Christina	Parsons		Marketing Manager	2	11	2003-07-01	30000
11	Eduardo	Ferrier		Marketing Manager	2	12	2003-07-01	30000
12	Kelly	Abdou		Marketing Manager	2	13	2003-07-01	30000
13	Julia	Nayeri		Marketing Manager	2	14	2003-07-01	30000
14	Mark	Tucker		Marketing Manager	2			

5. Modify data with C# code and call "**Save Changes()**"

```
private void ChangeBlogPostName(int id,
    string newName)
{
    var db = new Context();

    var post = db.Posts
        .FirstOrDefault(x => x.Id == id);

    if (post == null)
    {
        throw new ArgumentException(
            "Item with that id was not found");
    }

    post.Name = newName;

    db.SaveChanges();
}
```

6. Entity Framework generates & executes SQL command to modify the DB

```
SELECT
[Extent1].[EmployeeID] AS [EmployeeID],
[Extent1].[FirstName] AS [FirstName],
[Extent1].[LastName] AS [LastName],
[Extent1].[MiddleName] AS [MiddleName],
[Extent1].[JobTitle] AS [JobTitle],
[Extent1].[DepartmentID] AS [DepartmentID],
[Extent1].[ManagerID] AS [ManagerID],
[Extent1].[HireDate] AS [HireDate],
[Extent1].[Salary] AS [Salary],
[Extent1].[AddressID] AS [AddressID]
FROM [dbo].[Employees] AS [Extent1]
WHERE N'Production Supervisor' = [Extent1].[JobTitle]
```

Entity Framework Core: Setup

- To add **EF Core support** to a project in **Visual Studio**:

- Install it from **Package Manager Console**

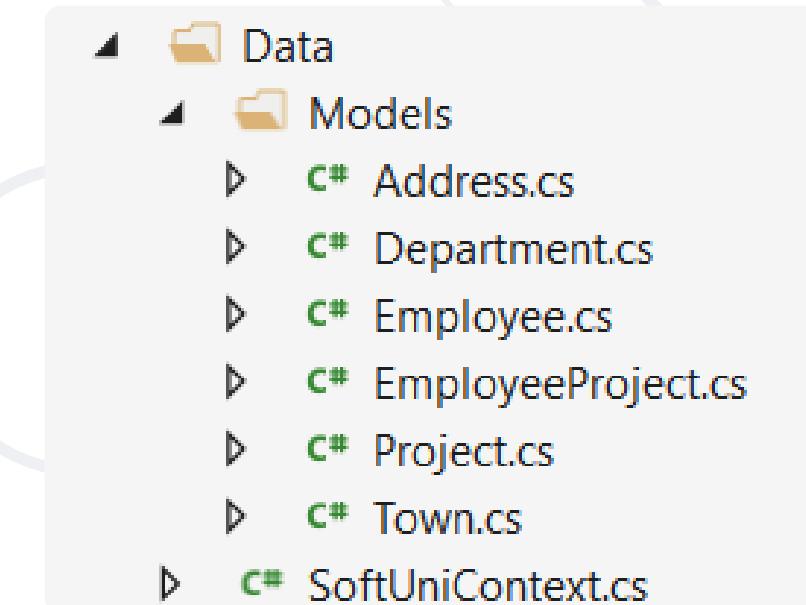
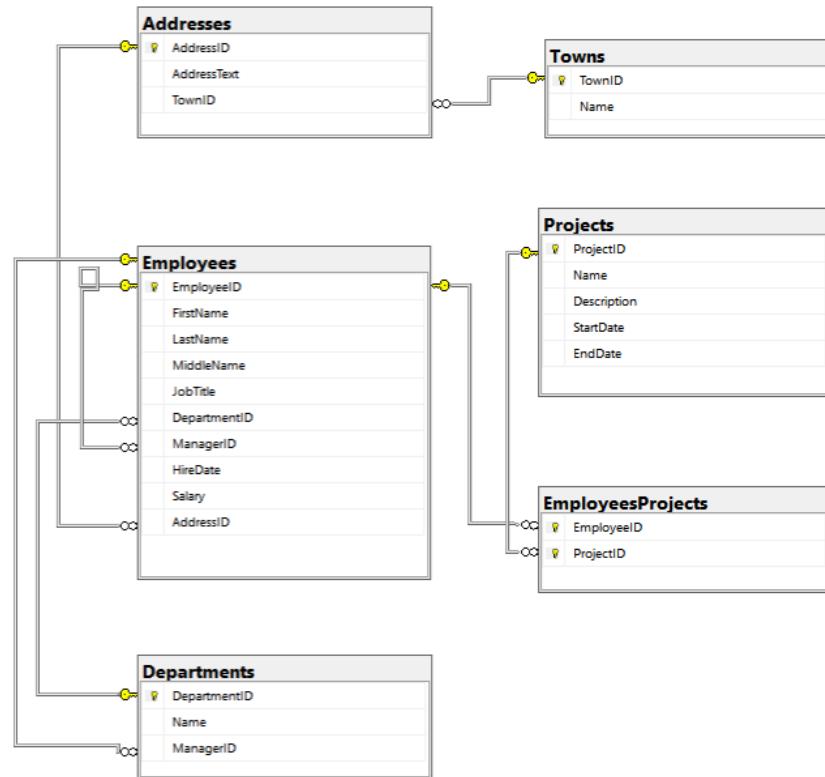
```
Install-Package Microsoft.EntityFrameworkCore
```

- EF Core is modular – any **data providers** must be installed too:

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

Database First Model

- Database First model models the **entity classes after the database**



Database First Model: Setup

- Scaffolding DbContext from DB with **Scaffold-DbContext** command in **Package Manager Console**:

Scaffold-DbContext

```
-Connection "Server=.;Database=...;Integrated Security=True"  
-Provider Microsoft.EntityFrameworkCore.SqlServer  
-OutputDir Data
```

- Scaffolding requires the following packages beforehand:

```
Install-Package Microsoft.EntityFrameworkCore.Tools
```

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer.Design
```

- The **DbContext** class
 - Holds the **database connection** and the **entity classes**
 - Provides **LINQ-based** data access
 - Provides **identity tracking**, **change tracking**, and an API for **CRUD** operations
- Entity classes
 - Hold **entities** (objects with their attributes and relations)
 - Each database **table** is typically mapped to a single **C# class**

- **Associations** (relationship mappings)
 - An association is a primary key / foreign key-based relationship between two entity classes
 - Allows navigation from one entity to another

```
var courses = student.Courses.Where(...);
```

- **Concurrency control**
 - Entity Framework uses **optimistic concurrency control**
 - No locking by default
 - Automatic concurrency conflict detection



Reading Data

Querying the DB using Entity Framework

The DbContext Class

- **DbContext** provides:
 - CRUD Operations
 - A way to **access entities**
 - Methods for **creating** new entities (**Add()** method)
 - Ability to **manipulate database data by modifying objects**
 - Easily navigate through **table relations**
 - Executing **LINQ queries** as native **SQL queries**
 - Managing database **creation/deletion/migration**

Using DbContext Class

- First create instance of the **DbContext**:

```
var context = new SoftUniDbContext();
```

- In the constructor you can pass a database connection string
- **DbContext** properties:
 - **Database** – **EnsureCreated/Deleted** methods, DB Connection
 - **ChangeTracker** – Holds info about the **automatic change tracker**
 - All entity classes (tables) are listed as properties
 - e.g. **DbSet<Employee> Employees { get; set; }**

Reading Data with LINQ Query

- Executing **LINQ-to-Entities** query over EF entity:

```
using (var context = new SoftUniEntities())
{
    var employees = context.Employees
        .Where(e => e.JobTitle == "Design Engineer")
        .ToArray();
}
```

EF translates this
to an SQL query

- **Employees** property in the **DbContext**:

```
public partial class SoftUniEntities : DbContext
{
    public DbSet<Employee> Employees { get; set; }
    public DbSet<Project> Projects { get; set; }
    public DbSet<Department> Departments { get; set; }
}
```

Reading Data with LINQ Query

- We can also use **extension methods** for constructing the query

```
using (var context = new SoftUniEntities())
{
    var employees = context.Employees
        .Where(c => c.JobTitle == "Design Engineering")
        .Select(c => c.FirstName)
        .ToList();
}
```

ToList() materializes the
query

- Find element by **ID**

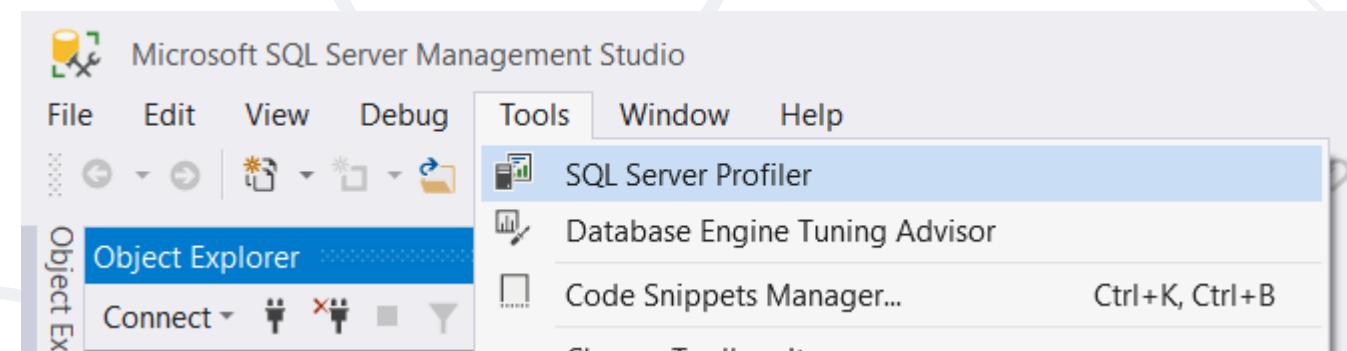
```
using (var context = new SoftUniEntities())
{
    var project = context.Projects.Find(2);
    Console.WriteLine(project.Name);
}
```

- **Where()**
 - Searches by given condition
- **First/Last() / FirstOrDefault/LastOrDefault()**
 - Gets the **first/last** element which matches the condition
 - Throws **InvalidOperationException** without **OrDefault**
- **Select()**
 - Projects (conversion) collection to another type
- **OrderBy() / ThenBy() / OrderByDescending()**
 - Orders a collection

- **Any()**
 - Checks if any element matches a condition
- **All()**
 - Checks if all elements match a condition
- **Distinct()**
 - Returns only unique elements
- **Skip() / Take()**
 - Skips or takes X number of elements

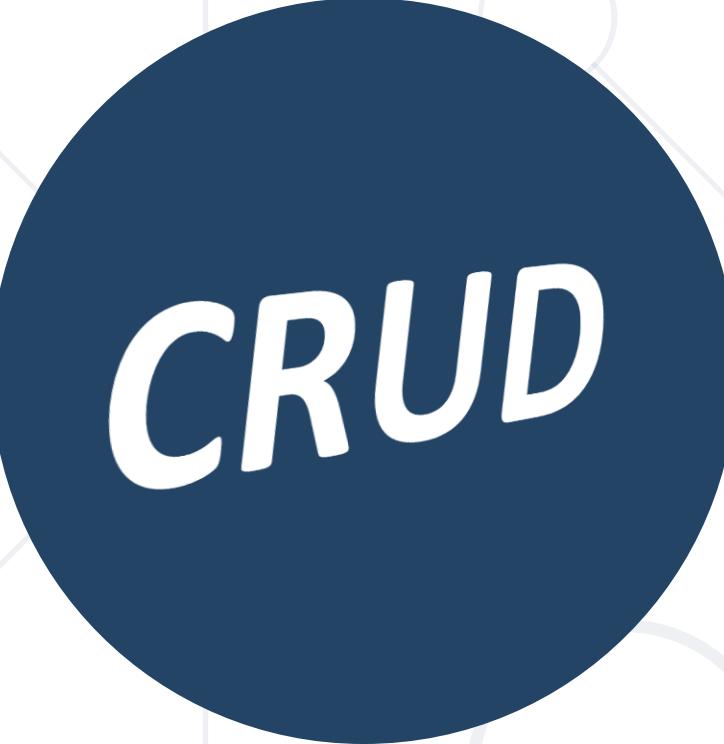
Logging the Native SQL Queries

- Queries sent to SQL Server can be monitored with SQL Server Profiler
 - Included in **SQL Server Management Studio**:



- Queries can also be monitored with Express Profiler

<https://expressprofiler.codeplex.com/>



CRUD

CRUD Operations

With Entity Framework

Creating New Data

- To create a new database row use the method **Add(...)** of the corresponding DbSet:

```
var project = new Project()
{
    Name = "Judge System",
    StartDate = new DateTime(2015, 4, 15),
};

context.Projects.Add(project);
context.SaveChanges();
```

Create a new
Project object

Add the object to the **DbSet**

Execute SQL statements

Cascading Inserts

- We can also add cascading entities to the database:

```
Employee employee = new Employee();
employee.FirstName = "Petya";
employee.LastName = "Grozgarska";
employee.Projects.Add(new Project { Name = "SoftUni Conf" } );
softUniEntities.Employees.Add(employee);
softUniEntities.SaveChanges();
```

- The **Project** will be added when the **Employee** entity (employee) is inserted to the database

Updating Existing Data

- **DbContext** allows modifying entity properties and persisting them in the database
 - Just load an entity, modify it and call **SaveChanges()**
- The **DbContext** automatically tracks all changes made on its entity objects

```
Employees employee =  
    softUniEntities.Employees.First();  
employee.FirstName = "Alex";  
context.SaveChanges();
```

SELECT the first
order

Execute an
SQL UPDATE

Deleting Existing Data

- Delete is done by **Remove()** on the specified entity collection
- **SaveChanges()** method performs the delete action in the database

```
Employees employee =  
softUniEntities.Employees.First();  
  
softUniEntities.Employees.Remove(employee);  
  
softUniEntities.SaveChanges();
```

Mark the entity for deleting
at the next save

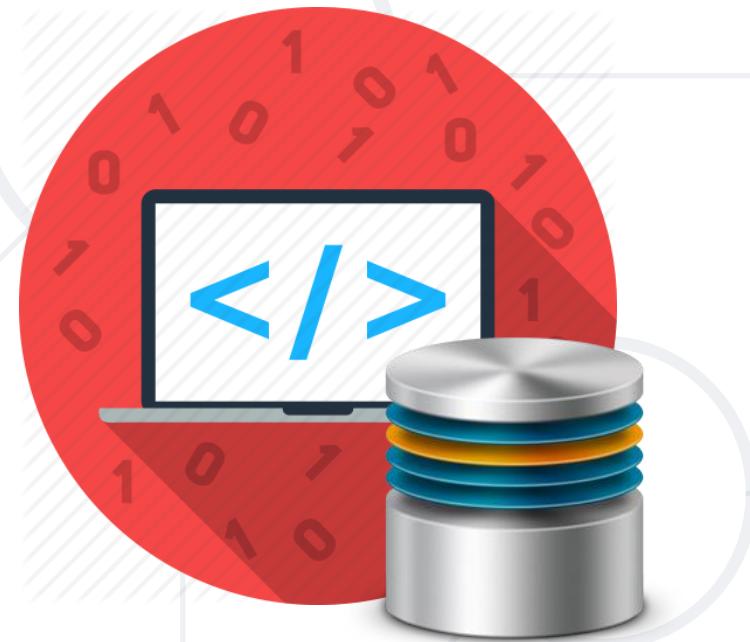
Execute the SQL DELETE
command

- ORM frameworks maps database schema to objects in a programming language
- Entity Framework Core is the standard .NET ORM
- LINQ can be used to query the DB through the DB context



EF Core Code First

Entity Framework DB From Code



SoftUni Team
Technical Trainers

Table of Contents

1. Code First model
2. EF Core Components
3. EF Core Configuration
4. Database Migrations





Code First Model

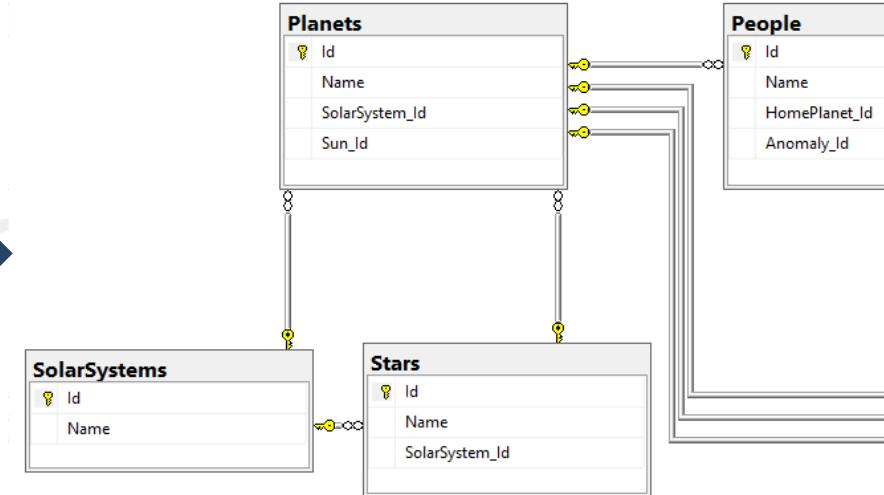
Motivation for Code First approach

What is the Code First Model?

- **Code First** means to write the .NET classes and let EF Core create the **database** from the **mappings**

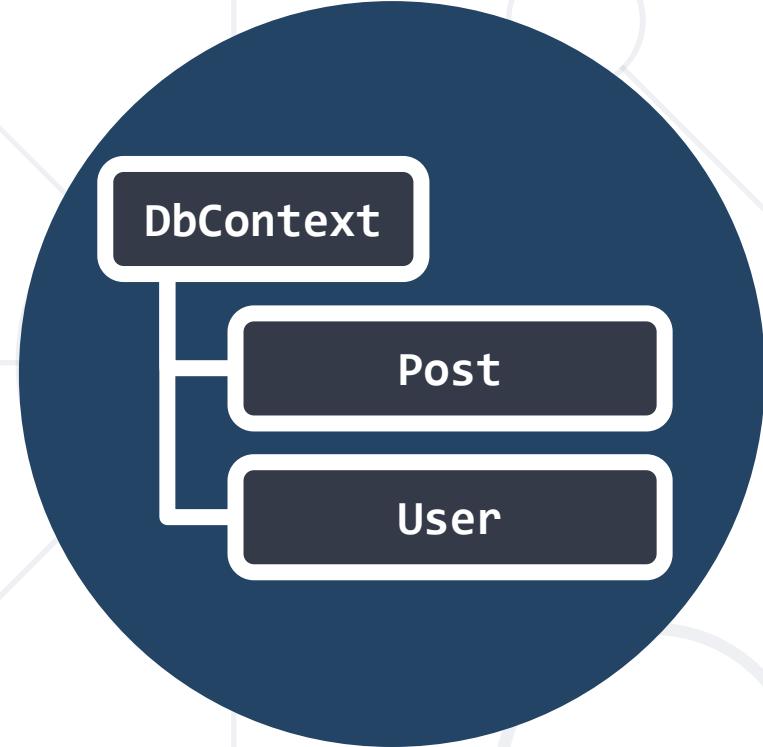


```
▶ C# Planet.cs
  ▶ Planet
    ⚡ Planet()
    ⚡ Id : int
    ⚡ Name : string
    ⚡ Sun : Star
    ⚡ SolarSystem : SolarSystem
    ⚡ OriginAnomalies : ICollection<Anomaly>
  ▶ C# SolarSystem.cs
  ▶ SolarSystem
    ⚡ Id : int
    ⚡ Name : string
  ▶ C# Star.cs
  ▶ Star
    ⚡ Id : int
    ⚡ Name : string
    ⚡ SolarSystem : SolarSystem
```



Why Use Code First?

- Write code **without** having to define **mappings** in XML or **create database tables**
- Define objects in **C# format**
- Enables database persistence with no configuration
- Changes to code can be **reflected** (migrated) in the schema
- **Data Annotations** or **Fluent API** describe properties
 - Key, Required, MinLength, etc.



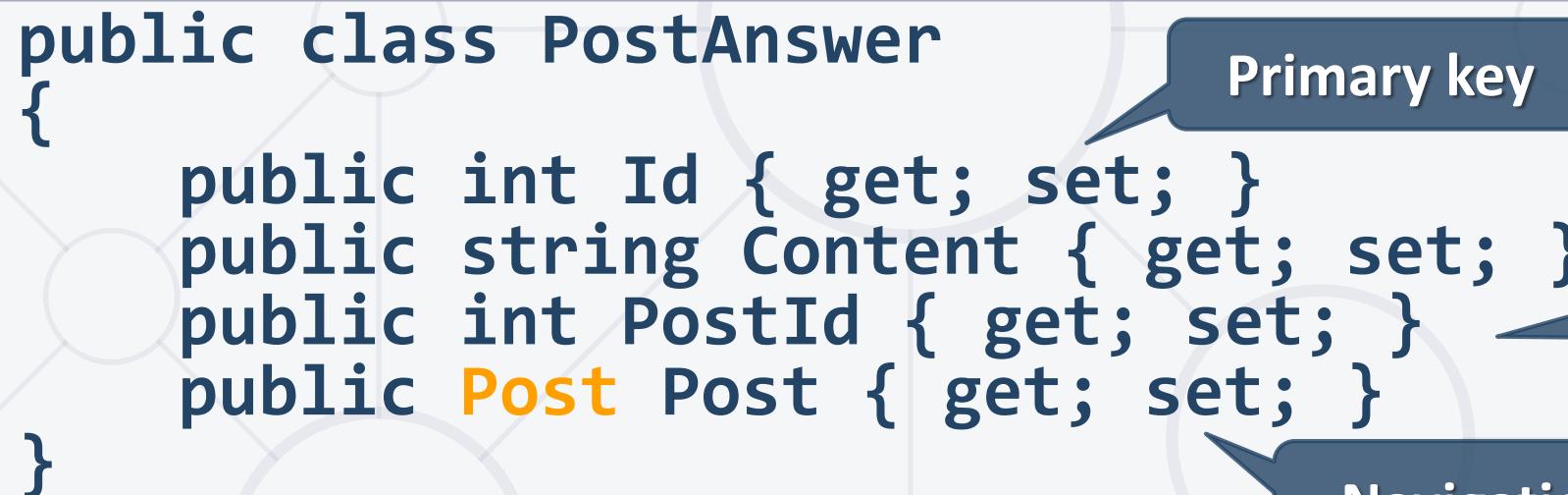
EF Core Components

Overview of system objects

Domain Classes (Models)

- Bunch of normal C# classes (POCO)
 - May contain **navigation properties** for **table relationships**

```
public class PostAnswer
{
    public int Id { get; set; }
    public string Content { get; set; }
    public int PostId { get; set; }
    public Post Post { get; set; }
}
```



The diagram shows three callout boxes pointing to specific parts of the code:

- A blue callout box points to the `Id` property with the text "Primary key".
- A blue callout box points to the `PostId` property with the text "Foreign key".
- A blue callout box points to the `Post` navigation property with the text "Navigation property".

- Recommended to be in a **separate class library**

Domain Classes (Models) (2)

- Another example of a domain class (model)

```
public class Post
{
    public int Id { get; set; }
    public string Content { get; set; }
    public int AuthorId { get; set; }
    public User Author { get; set; }

    public IList<Reply> Replies { get; set; }
}
```

Navigation
property

One-to-Many
relation

- Maps a **collection** of **entities** from a **table**
- Set operations: **Add**, **Attach**, **Remove**, **Find**
- DbContext** contains multiple **DbSet<T>** properties

```
public class DbSet<TEntity> :  
System.Data.Entity.Infrastructure.DbQuery<TEntity>  
    where TEntity : class  
Member of System.Data.Entity
```

```
public DbSet<Post> Posts { get; set; }
```

The DbContext Class

- Usually named after the database e.g. **BlogDbContext**, **ForumDbContext**
- Inherits from **DbContext**
- Manages model classes using **DbSet<T>** type
- Implements **identity tracking**, **change tracking**
- Provides **API** for **CRUD** operations and **LINQ-based** data access
- Recommended to be in a separate class library
 - Don't forget to reference the EF Core library + any providers
- Use several **DbContext** if you have too much models

Defining DbContext Class – Example

EF Reference

```
using Microsoft.EntityFrameworkCore;
using CodeFirst.Data.Models; Models Namespace
public class ForumDbContext : DbContext
{
    public DbSet<Category> Categories { get; set; }
    public DbSet<Post> Posts { get; set; }
    public DbSet<PostAnswer> PostAnswers { get; set; }
    public DbSet<User> Users { get; set; }
}
```

CRUD Operations with EF Code First

```
var db = new ForumDbContext();
var category = new Category { Name = "Database course" };
db.Categories.Add(category);

var post = new Post();
post.Title = "Homework Deadline";
post.Content = "Please extend the homework deadline";
post.Type = PostType.Normal;
post.Category = category;
post.Tags.Add(new Tag { Text = "homework" });
post.Tags.Add(new Tag { Text = "deadline" });

db.Posts.Add(post);

db.SaveChanges();
```



EF Core Configuration

NuGet Packages, Configuration

Code First with EF Core: Setup

- To add EF Core support to a project in Visual Studio:

- Install it from **Package Manager Console**

```
Install-Package Microsoft.EntityFrameworkCore
```

- EF Core is modular – any **data providers** must be installed too

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```



How to Connect to SQL Server?

- One way to connect is to create a **Configuration** class with your connection string:

```
public static class Configuration
{
    public const string ConnectionString = "Server=.;Database=...";
```

- Then add the connection string in the **OnConfiguring** method in the **DbContext** class

```
protected override void OnConfiguring(DbContextOptionsBuilder builder)
{
    if (!builder.IsConfigured)
        builder.UseSqlServer(Configuration.ConnectionString);
}
```

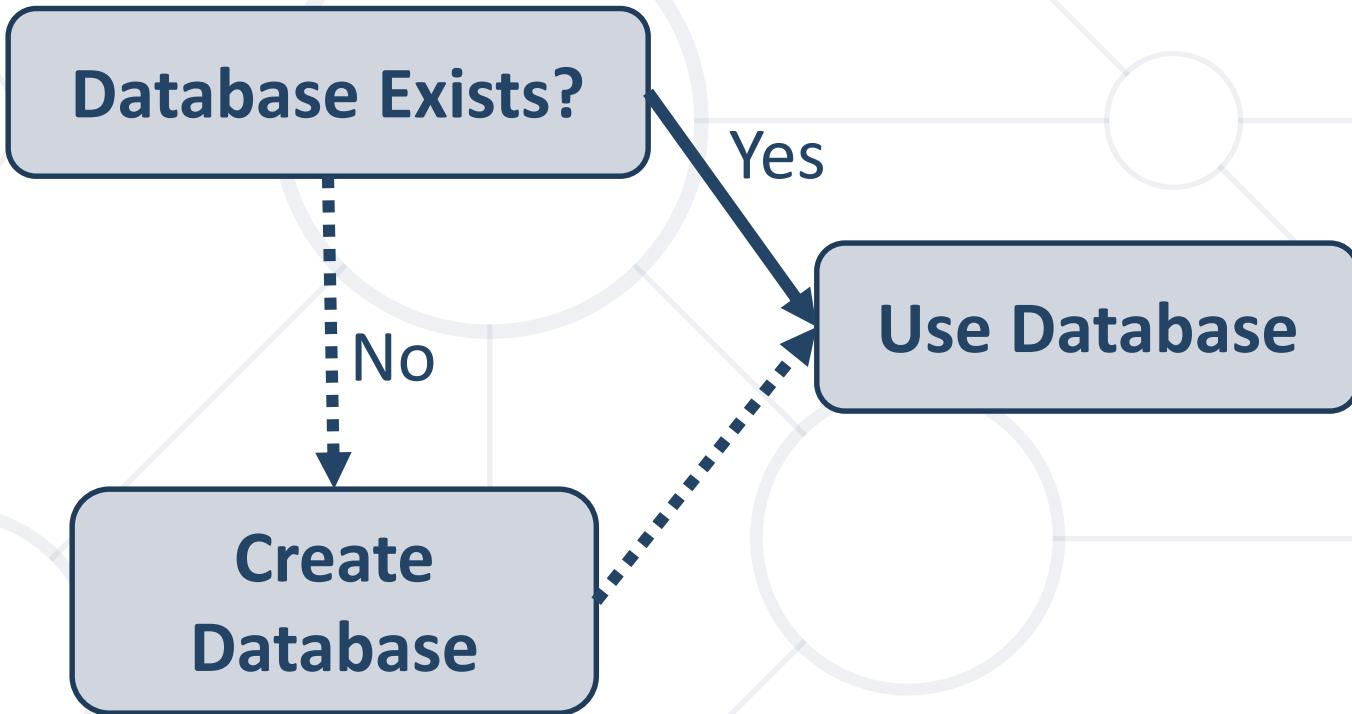
- The **OnModelCreating** Method let us use the Fluent API to describe our **table relations** to EF Core

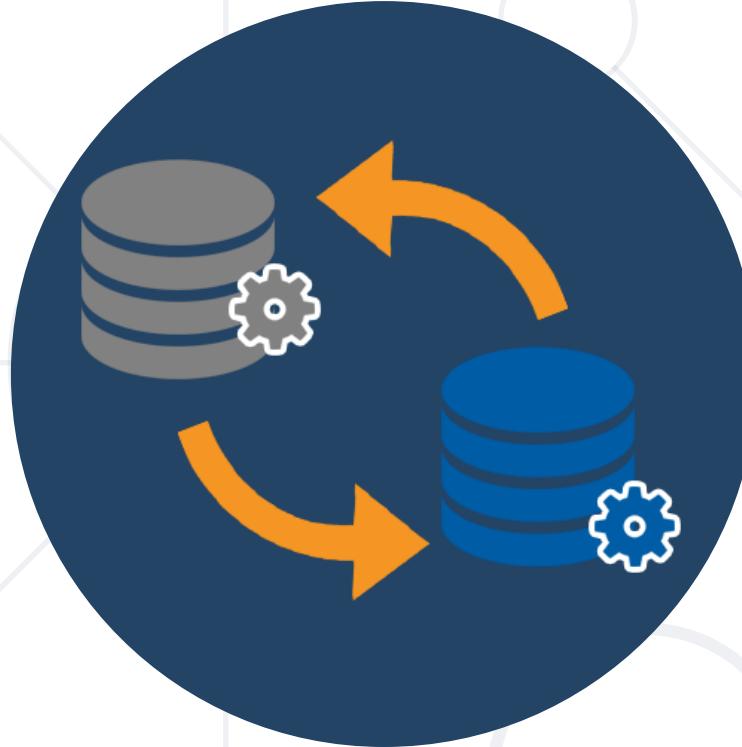
```
protected override void OnModelCreating(ModelBuilder builder)
{
    builder.Entity<Category>()
        .HasMany(c => c.Posts)
        .WithOne(p => p.Category);

    builder.Entity<Post>()
        .HasMany(p => p.Replies)
        .WithOne(r => r.Post);

    builder.Entity<User>()
        .HasMany(u => u.Posts)
        .WithOne(p => p.Author);
}
```

Database Connection Workflow

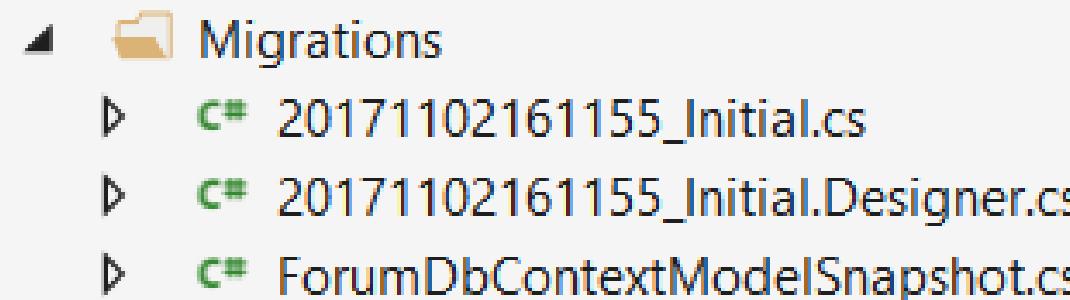




Database Migrations

What are Database Migrations?

- Updating database schema **without losing data**
 - Adding/dropping tables, columns, etc.
- Migrations in EF Core keep their **history**
 - Entity Classes, DB Context versions are all **preserved**
- **Automatically generated**



Migrations in EF Core

- To use migrations in EF Core, we use the Add-Migration command from the **Package Manager Console**

```
Add-Migration {MigrationName}
```

- To undo a migration, we use **Remove-Migration**

```
Remove-Migration
```

- Commit changes to the database, using **Update-Database**

```
Update-Database
```

- Code First **increases productivity** by centralizing maintenance
- **Classes** represent real world objects with their **properties** and **behaviour**
- Entity Framework Core uses **data classes** (POCOs) to represent DB objects
- We can use **Database Migrations** to update our database without losing our data



Entity Relations

Customizing Entity Models



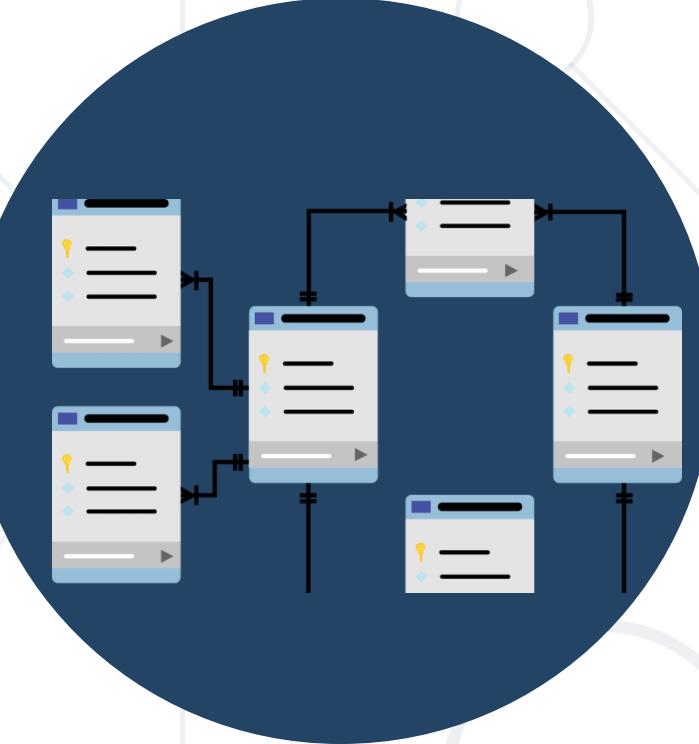
SoftUni Team
Technical Trainers



Table of Contents

1. Object Composition
2. Navigation Properties
3. Fluent API
4. Table Relationships



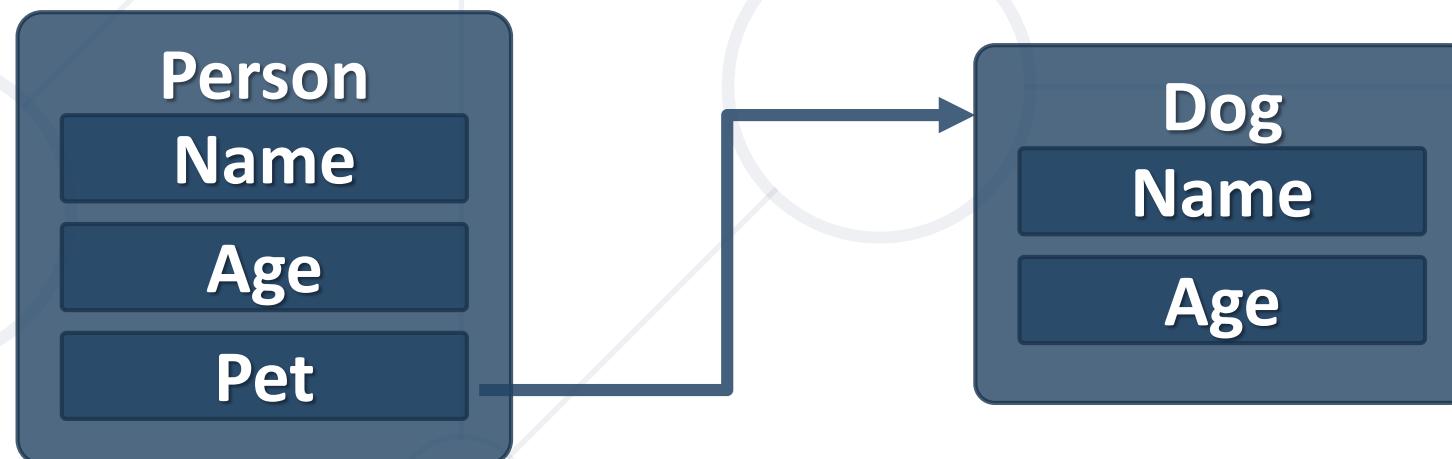


Object Composition

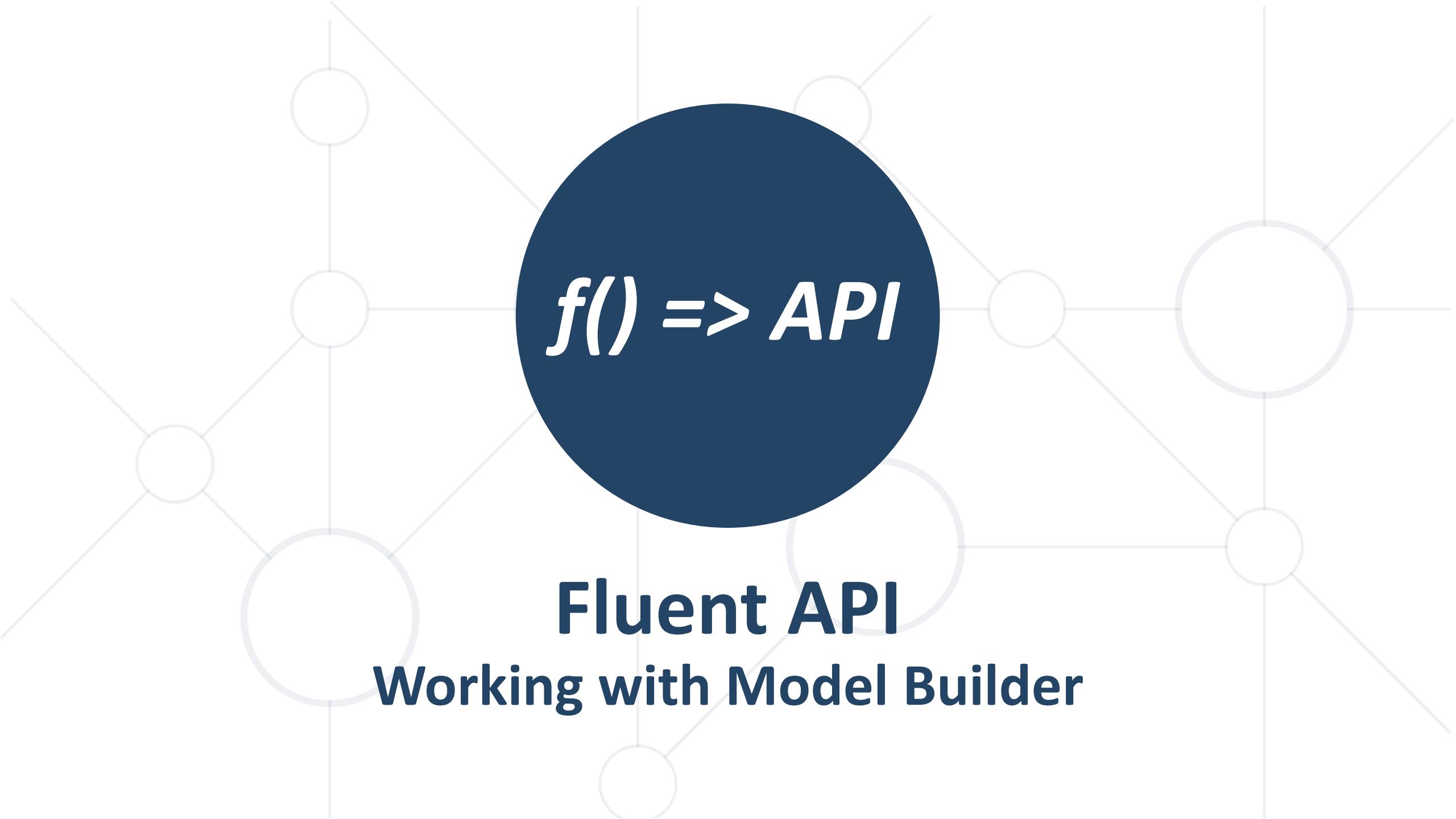
Describing Database Relationships

Object Composition

- Object composition denotes a "has-a" relationship
 - E.g. the car has an engine
- Defined in C# by one object having a property that is a reference to another



- Navigation properties create a **relationship** between entities
- Is either an **Entity Reference** (one to one or zero) or an **ICollection** (one to many or many to many)
- They provide **fast querying** of related records
- Can be **modified** by **directly** setting the reference



f() => API

Fluent API
Working with Model Builder

- **Code First** maps your POCO classes to tables using a **set of conventions**
 - E.g. property named "**Id**" maps to the **Primary Key**
- Can be customized using **annotations** and the **Fluent API**
- Fluent API (Model Builder) allows **full control** over DB mappings
 - Custom names of objects (columns, tables, etc.) in the DB
 - Validation and data types
 - Define complicated entity relationships

Working with Fluent API

- Custom mappings are placed inside the **OnModelCreating** method of the DB context class

```
protected override void OnModelCreating(DbModelBuilder builder)
{
    builder.Entity<Student>().HasKey(s => s.StudentKey);
}
```

Fluent API: Renaming DB Objects

- Specifying Custom Table name

```
modelBuilder.Entity<Order>()
    .ToTable("OrderRef", "Admin");
```

- Custom Column name/DB Type

```
modelBuilder.Entity<Student>()
    .Property(s => s.Name)
    .HasColumnName("StudentName")
    .HasColumnType("varchar");
```

Optional schema
name



Fluent API: Column Attributes

- Explicitly set Primary Key

```
modelBuilder  
    .Entity<Student>().HasKey("StudentKey");
```

- Other column attributes

```
modelBuilder.Entity<Person>()  
    .Property(p => p.FirstName)  
    .IsRequired()  
    .HasMaxLength(50)
```

```
modelBuilder.Entity<Post>()  
    .Property(p => p.LastUpdated)  
    .ValueGeneratedOnAddOrUpdate()
```

Fluent API: Miscellaneous Config

- Do not include property in DB (e.g. business logic properties)

```
modelBuilder  
    .Entity<Department>().Ignore(d => d.Budget);
```

- Disabling cascade delete

- If a FK property is non-nullable, cascade delete is **on by default**

```
modelBuilder.Entity<Course>()  
    .HasRequired(t => t.Department)  
    .WithMany(t => t.Courses)  
    .HasForeignKey(d => d.DepartmentID)  
    .OnDelete(DeleteBehavior.Restrict);
```

Throws exception on delete

Specialized Configuration Classes

- Mappings can be placed in entity-specific classes

```
public class StudentConfiguration  
    : IEntityConfiguration<Student>  
{  
    public void Configure(EntityTypeBuilder<Student> builder)  
    {  
        builder.HasKey(c => c.StudentKey);  
    }  
}
```

Specify target model

- Include in **OnModelCreating**:

```
builder.ApplyConfiguration(new StudentConfiguration());
```



Table Relationships

Expressed as Properties and Attributes

One-to-Zero-or-One

- Expressed in SQL Server as a shared primary key
- Relationship direction must be explicitly specified with a **ForeignKey** attribute
- ForeignKey is placed above the key property and contains the name of the navigation property and vice versa



One-to-Zero-or-One: Implementation

- Using the **ForeignKey** Attribute

```
public class Student
{
    [Key] Attributes
    public int Id { get; set; }
    public string Name { get; set; }

    public int AddressId { get; set; }
    [ForeignKey("Address")]
    public Address Address { get; set; }
}
```

One-to-Zero-or-One: Implementation (2)

- Using the **ForeignKey** Attribute

```
public class Address
{
    public int Id { get; set; }
    public string Text { get; set; }
    public int StudentId { get; set; }
    [ForeignKey(nameof(Student))]
    public Student Student { get; set; }
}
```

One-to-Zero-or-One: Fluent API

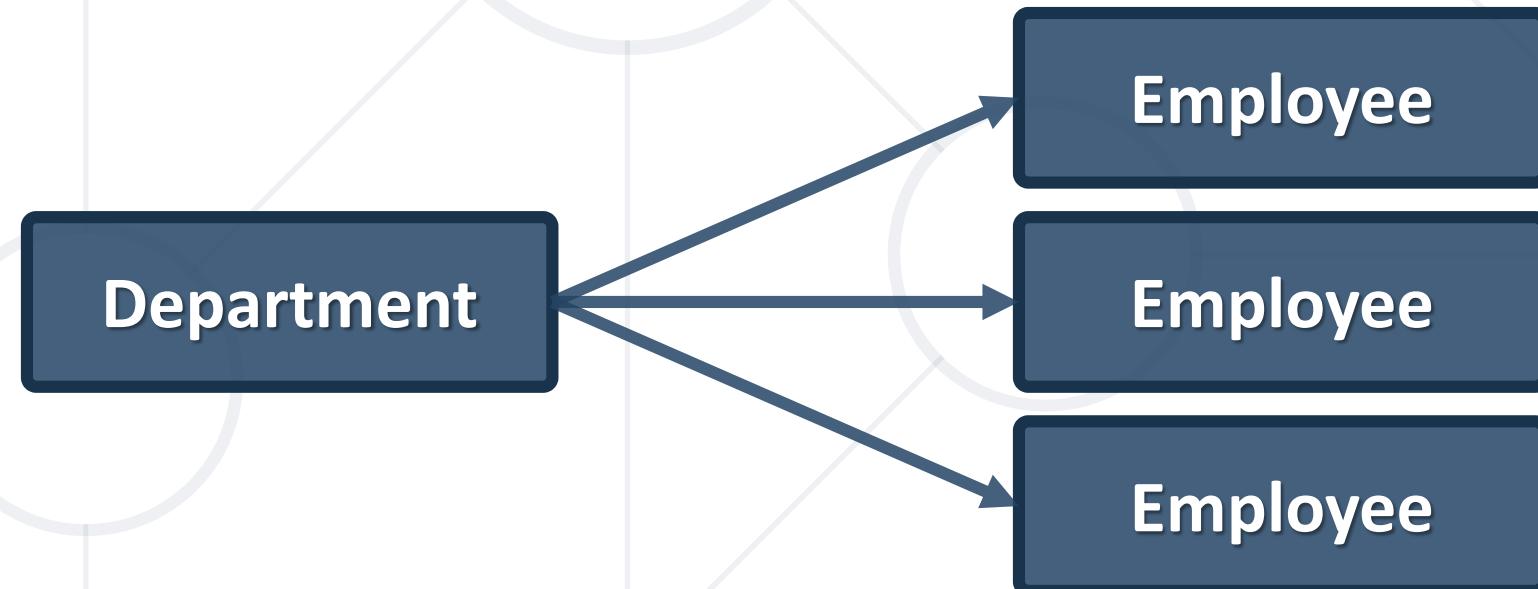
- **HasOne → WithOne**

```
modelBuilder.Entity<Address>()
    .HasOne(a => a.Student)
    .WithOne(s => s.Address)
    .HasForeignKey(a => a.StudentId);
```

Address contains
FK to Student

- If **StudentId** property is **nullable (int?)**, relation becomes **One-To-Zero-Or-One**

- Most common type of relationship
- Implemented with a **collection** inside the **parent entity**
 - The collection should be **initialized** in the **constructor!**



One-to-Many: Implementation

- Department has many employees

```
public class Department
{
    public int Id { get; set; }
    public string Name { get; set; }

    public ICollection<Employee> Employees { get; set; }
}
```

One-to-Many: Implementation (2)

- Employees have one **department**

```
public class Employee
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public int DepartmentId { get; set; }
    public Department Department { get; set; }
}
```

One-to-Many: Fluent API

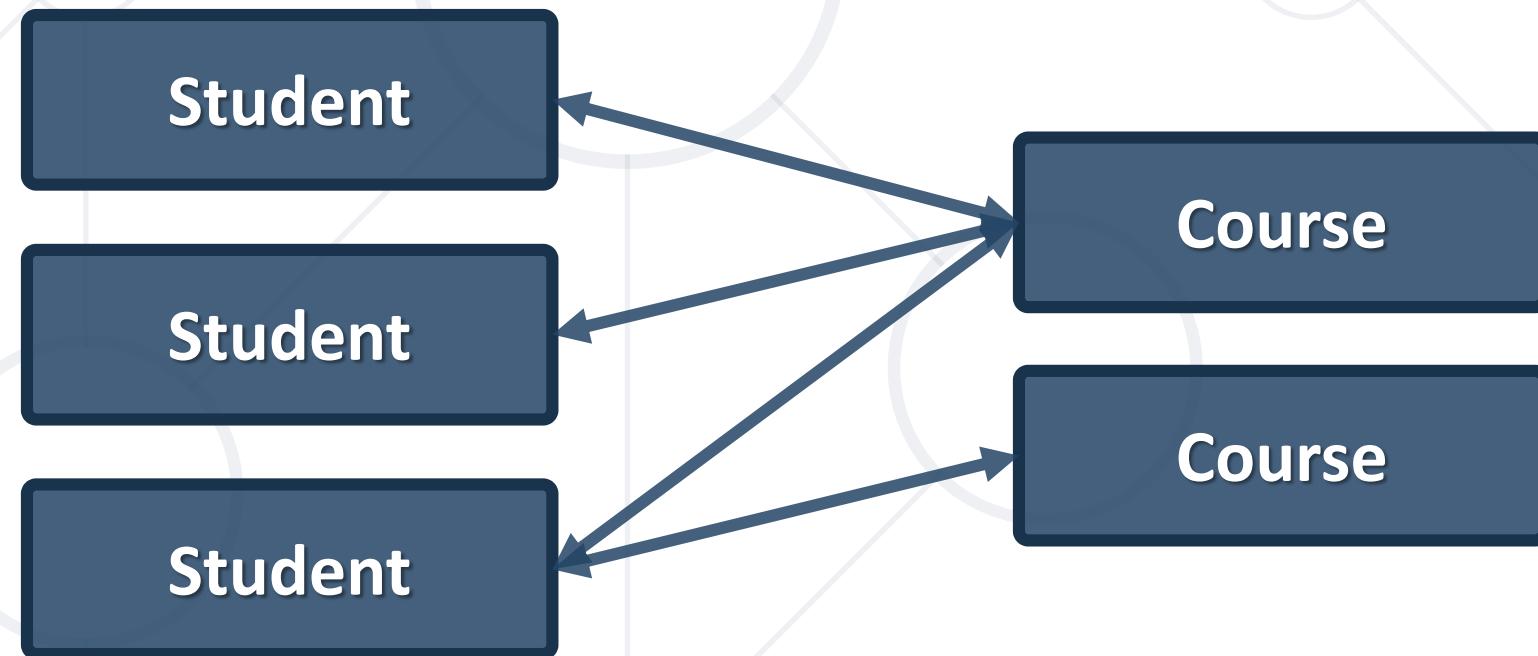
- **HasMany** → **WithOne**

```
modelBuilder.Entity<Post>()
    .HasMany(p => p.Comments)
    .WithOne(c => c.Post)
    .HasForeignKey(c => c.PostId);
```

```
modelBuilder.Entity<Employee>()
    .HasMany(e => e.Addresses)
    .WithOne(a => a.Employee)
    .HasForeignKey(a => a.EmployeeId);
```

Many-to-Many

- Requires a **join entity (separate class)** in EF Core
- Implemented with collections in each entity, referring the other



Many-to-Many Implementation (1)

```
public class Course
{
    public string Name { get; set; }

    public ICollection<StudentCourse> StudentsCourses { get; set; }
}
```

```
public class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public ICollection<StudentCourse> StudentsCourses { get; set; }
}
```

Many-to-Many Implementation (2)

- EF Core requires a **Join Entity**

```
public class StudentCourse
{
    public int StudentId { get; set; }
    public Student Student { get; set; }

    public int CourseId { get; set; }
    public Course Course { get; set; }
}
```

Many-to-Many: Fluent API

- Mapping **both sides** of relationship

```
modelBuilder.Entity<StudentCourse>()
    .HasKey(sc => new { sc.StudentId, sc.CourseId }));
```

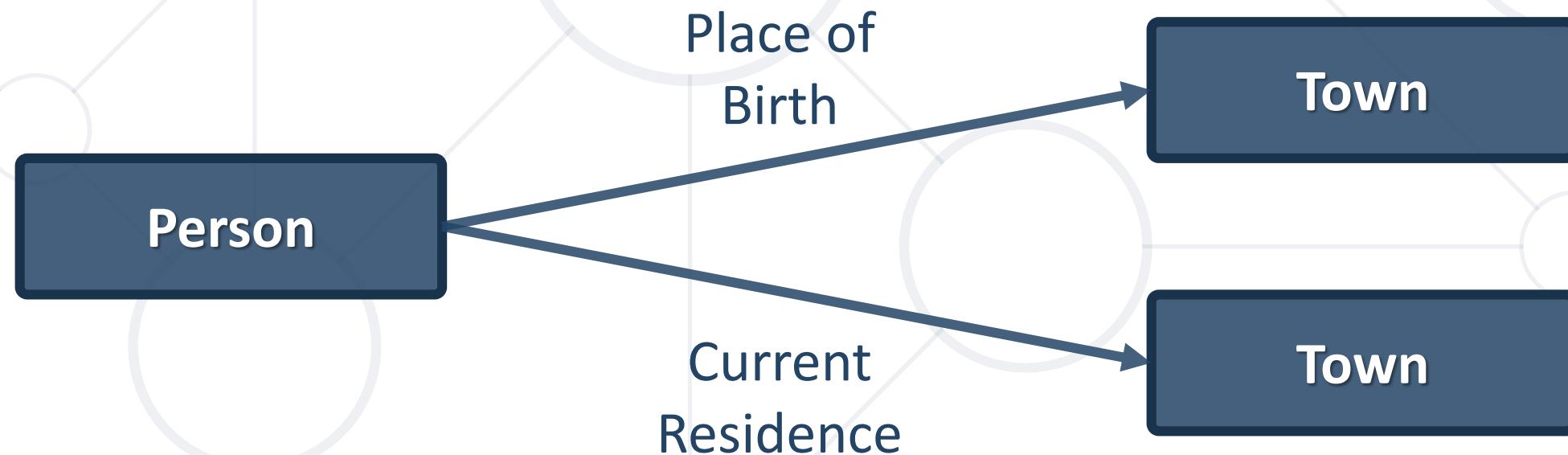
Composite
Primary Key

```
builder.Entity<StudentCourse>()
    .HasOne(sc => sc.Student)
    .WithMany(s => s.StudentCourses)
    .HasForeignKey(sc => sc.StudentId);
```

```
builder.Entity<StudentCourse>()
    .HasOne(sc => sc.Course)
    .WithMany(s => s.StudentCourses)
    .HasForeignKey(sc => sc.CourseId);
```

Multiple Relations

- When two entities are related by more than one key
- Entity Framework needs help from **Inverse Properties**



Multiple Relations Implementation

- **Person** Domain Model – defined as usual

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }

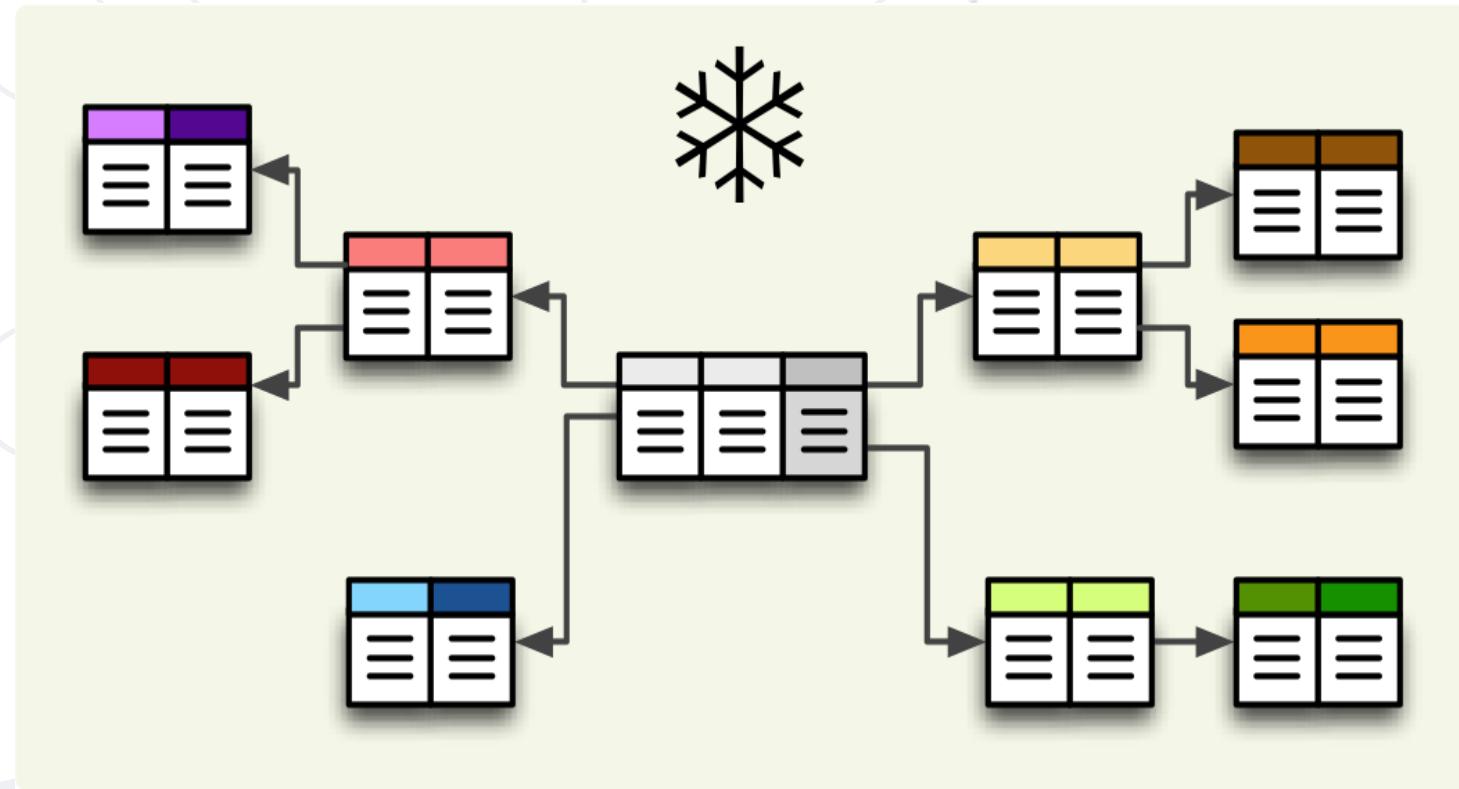
    public Town PlaceOfBirth { get; set; }
    public Town CurrentResidence { get; set; }
}
```

Multiple Relations Implementation (2)

- **Town** Domain Model

```
public class Town
{
    public int Id { get; set; }
    public string Name { get; set; }
    [InverseProperty("PlaceOfBirth")]
    public ICollection<Person> Natives { get; set; }
    [InverseProperty("CurrentResidence")]
    public ICollection<Person> Residents { get; set; }
}
```

Point towards related property



Filtering and Aggregating Tables

Select, Join and Group Data Using LINQ

Good Reasons to Use Select

- Limit network traffic by reducing the queried columns
- Syntax:

```
var employeesWithTown = context
    .Employees
    .Select(employee => new
{
    EmployeeName = employee.FirstName,
    TownName = employee.Address.Town.Name
});
```

- SQL Server Profiler

```
SELECT [employee].[FirstName] AS [EmployeeName], [employee.Address.Town].[Name] AS [TownName]
FROM [Employees] AS [employee]
LEFT JOIN [Addresses] AS [employee.Address] ON [employee].[AddressID] = [employee.Address].[AddressID]
LEFT JOIN [Towns] AS [employee.Address.Town] ON [employee.Address].[TownID] =
[employee.Address.Town].[TownID]
```

Good Reasons not to Use Select

- Data that is selected is **not** of the **initial entity type**

- **Anonymous type**, generated at runtime

```
[•] (local variable) System.Collections.Generic.List<'a> employeesWithTown
```

Anonymous Types:

```
'a is new { string EmployeeName, string TownName }
```

Local variable 'employeesWithTown' is never used

- **Data cannot be modified** (updated, deleted)
 - Entity is of a **different type**
 - Not associated with the **context** anymore

Joining Tables in EF: Using Join()

- Join tables in EF with **LINQ / extension methods** on **IEnumerable<T>** (like when joining collections)

```
var employees =  
    softUniEntities.Employees.Join(  
        softUniEntities.Departments,  
        (e => e.DepartmentID),  
        (d => d.DepartmentID),  
        (e, d) => new {  
            Employee = e.FirstName,  
            JobTitle = e.JobTitle,  
            Department = d.Name  
        }  
    );
```

Grouping Tables in EF

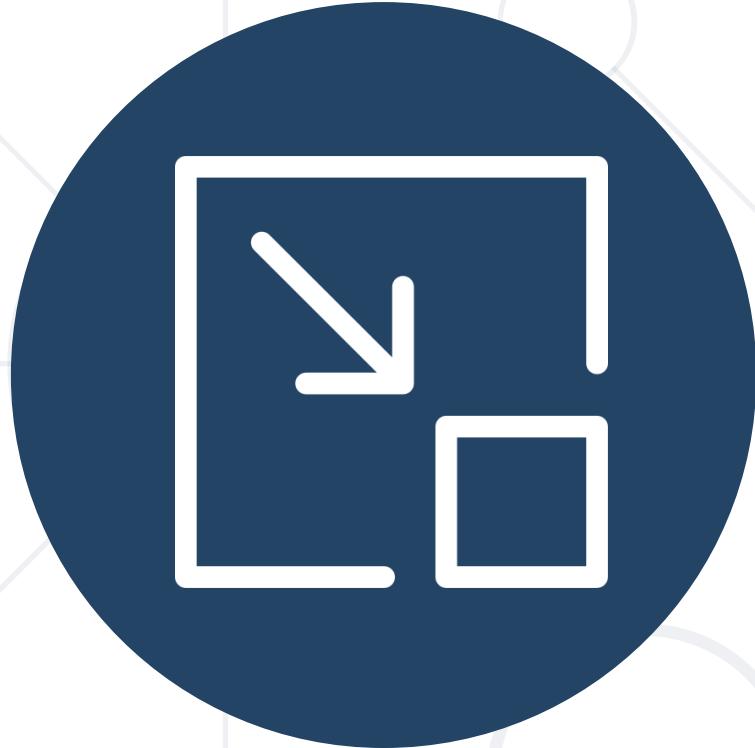
- Grouping also can be done by LINQ
 - The same way as with collections in LINQ

- Grouping with LINQ:

```
var groupedEmployees =  
    from employee in softUniEntities.Employees  
    group employee by employee.JobTitle;
```

- Grouping with extension methods:

```
var groupedCustomers = softUniEntities.Employees  
    .GroupBy(employee => employee.JobTitle);
```



Result Models

Simplifying Models

Result Models

- **Select()**, **GroupBy()** can work with **custom classes**
 - Allows you to **pass them** to methods and use them as a return type
 - Requires some **extra code** (class definition)
- Sample Result Model:

```
public class UserResultModel
{
    public string FullName { get; set; }
    public string Age { get; set; }
}
```



Result Models (2)

- Assign the fields as you would with an anonymous object:

```
var currentUser = context.Users
    .Where(u => u.Id == 8)
    .Select(u => new UserResultModel
    {
        FullName = u.FirstName + " " + u.LastName,
        Age = u.Age
    })
    .SingleOrDefault();
```

- The new type can be used in a method signature:

```
public UserResultModel GetUserInfo(int Id) { ... }
```



Attributes

Custom Entity Framework Behavior

- EF Code First provides a set of **DataAnnotation attributes**
 - You can override default Entity Framework behavior
- To access nullability and size of fields:

```
using System.ComponentModel.DataAnnotations;
```
- To access schema customizations:

```
using System.ComponentModel.DataAnnotations.Schema;
```
- For a full set of configuration options you need the **Fluent API**

- **[Key]** – explicitly specify **primary key**
 - When your PK column doesn't have an "Id" suffix

[Key]

```
public int StudentKey { get; set; }
```

- **Composite key** is only defined using **Fluent API** for now

```
builder.Entity<EmployeesProjects>()
    .HasKey(k => new { k.EmployeeId, k.ProjectId });
```

Key Attributes (2)

- **ForeignKey** – explicitly **link** navigation property and foreign key property within the same class
- Works in **either direction** (FK to navigation property or navigation property to FK)

```
public class Client
{
    ...
    [ForeignKey("Order")]
    public int OrderRefId { get; set; }
    public Order Order { get; set; }
}
```

Renaming Objects

- **Table** – manually specify the name of the table in the DB

```
[Table("StudentMaster")]
public class Student
{
    ...
}
```

```
[Table("StudentMaster", Schema = "Admin")]
public class Student
{
    ...
}
```

Renaming Objects (2)

- **Column** – manually specify the name of the column in the DB
 - You can also specify order and explicit data type

```
public class Student
{
    ...
    [Column("StudentName", Order = 2, TypeName="varchar(50)")]
    public string Name { get; set; }
}
```

Optional parameters

- **Required** – mark a nullable property as **NOT NULL** in the DB
 - Will throw an exception if not set to a value
 - Non-nullable types (e.g. **int**) will **not throw** an exception (will be set to language-specific default value)
- **MinLength** – specifies min length of a string (client validation)
- **MaxLength / StringLength** – specifies max length of a string (both client and DB validation)
- **Range** – set lower and/or upper limits of numeric property (client validation)

Other Attributes

- **Index** – create index for column
 - Primary key will always have an index

```
builder.Entity<Car>()
    .HasIndex(u => u.RegistrationNumber)
    .IsUnique();
```

- **NotMapped** – property will not be mapped to a column
 - For business logic properties

Validation Method

- Add using:

```
using System.ComponentModel.DataAnnotations;
```

- Create following method to validate entities:

```
private bool IsValid(object obj)
{
    var validationContext = new ValidationContext(obj);
    var validationResults = new List<ValidationResult>();

    return Validator.TryValidateObject(obj, validationContext,
        validationResults, true);
}
```



Shadow Properties

Shadow Properties

- Shadow properties are **not defined** in your .NET entity class
 - They are **useful** when there is data in the database that **should not be exposed** on the mapped entity types.
- **Configure** shadow property:

```
builder.Entity<Project>()
    .Property<DateTime>("LastUpdated");
```

- **Change** value of shadow property:

```
context.Entry(Project).Property("LastUpdated")
    .CurrentValue = DateTime.Now;
```

- Objects can be composed from other objects to represent complex relationships
- Navigation properties speed up the traversal of related entities



- The **Fluent API** gives us full control over Entity Framework object mappings
- Information overhead can be limited by **selecting** only the needed properties
- **ResultModels** can be used to move aggregated data between methods
- **Attributes** can be used to express special table relationships and to customize entity behaviour



EF Advanced Querying

Performance Optimizations



SoftUni Team
Technical Trainers

Table of Contents

1. Native SQL Queries
2. Object State
3. Batch Operations
4. Stored Procedures
5. Concurrency
6. Cascade Operations





Executing Native SQL Queries

Parameterless and Parameterized

Executing Native SQL Queries

- Executing a **native SQL query** in EF Core directly:

```
var query = "SELECT * FROM Employees";
var employees = db.Employees
    .FromSqlRaw(query)
    .ToArray();
```

- Limitations:
 - **JOIN** statements **don't** get mapped to the entity class
 - **Required columns** must **always** be selected
 - **Target table** must be the same as the **DbSet**
 - **SQL statements** other than **SELECT** are recognized automatically as non-composable. As a consequence, the full results of stored procedures are always returned to the client and any **LINQ** operators applied after **FromSqlRaw** are evaluated in-memory.

Native SQL Queries with Parameters

- Native SQL queries can also be parameterized:

```
var context = new SoftUniDbContext();
string nativeSQLQuery =
    "SELECT FirstName, LastName, JobTitle" + Parameter placeholder
    "FROM dbo.Employees WHERE JobTitle = {0}";
var employees = context.Employees.FromSqlRaw(
    nativeSQLQuery, "Marketing Specialist");
foreach (var employee in employees) Parameter value
{
    Console.WriteLine(employee);
}
```

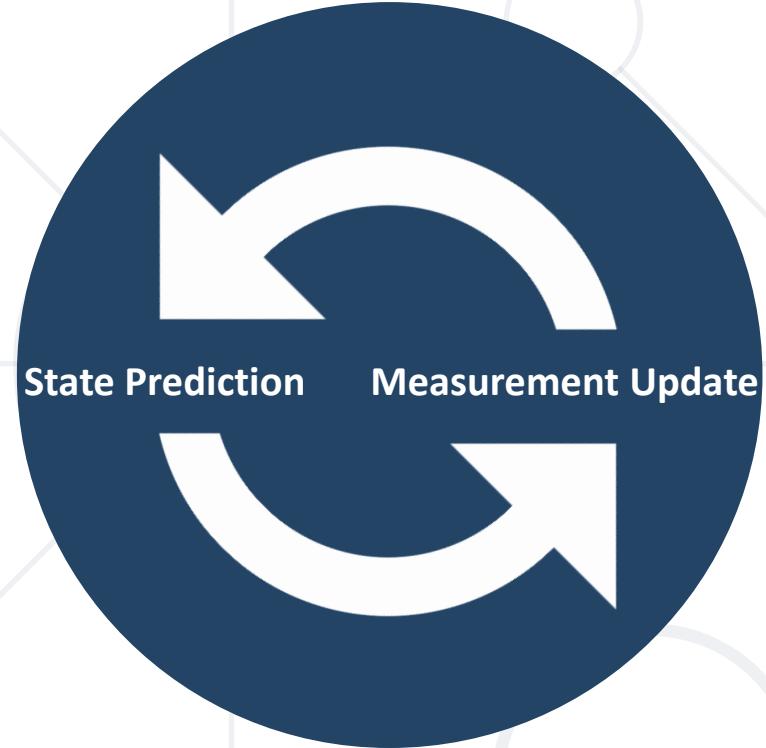
Interpolation in SQL Queries

- FromSqlInterpolated allows string interpolation syntax

```
var context = new SoftUniDbContext();
string jobTitle = "Marketing Specialist";
string nativeSQLQuery =
    "SELECT FirstName, LastName, JobTitle" +
    "FROM dbo.Employees WHERE JobTitle = {jobTitle}";
var employees = context.Employees.FromSqlInterpolated(
    nativeSQLQuery)
foreach (var employee in employees)
{
    Console.WriteLine(employee);
}
```



Interpolated parameter



Object State Tracking

Attaching and Detaching Objects

- In Entity Framework, objects can be:
 - **Attached** to the object context (tracked object)
 - **Detached** from an object context (untracked object)
- Attached objects are tracked and managed by the **DbContext**
 - **SaveChanges()** persists all changes in DB
- Detached objects are not referenced by the **DbContext**
 - Behave like a normal objects, which are not related to EF

Attaching Detached Objects

- When a query is executed inside a **DbContext**, the returned objects are **automatically attached** to it
- When a context is destroyed, all objects in it are automatically detached
 - E.g. in **Web applications** between requests
- You might later on **attach** objects that have been previously **detached** to a **new context**

Detaching Objects

- When is an object detached?
 - When we get the object from a **DbContext** and then **Dispose** it
 - Manually: by setting the **EntryState** to **Detached**

```
Employee GetEmployeeById(int id)
{
    using (var SoftUniDbContext = new SoftUniDbContext())
    {
        return SoftUniDbContext.Employees
            .First(p => p.EmployeeID == id);
    }
}
```

Returned employee
is detached

Attaching Objects

- When we want to update a detached object we need to **reattach it** and then update it: change to **Attached** state

```
void UpdateName(Employee employee, string newName)
{
    using (var SoftUniDbContext = new SoftUniDbContext())
    {
        var entry = SoftUniDbContext.Entry(employee);
        entry.State = EntityState.Added;
        employee.FirstName = newName;
        SoftUniDbContext.SaveChanges();
    }
}
```



Stored Procedures

Executing a Stored Procedure

- Stored Procedures can be executed via SQL

```
CREATE PROCEDURE UpdateAge @param int
AS
UPDATE Employees SET Age = Age + @param;
```

```
var ageParameter = new SqlParameter("@age", 5);
var query = "EXEC UpdateAge @age";
context.Database.ExecuteSqlCommand(query, ageParameter);
```



BULK

Bulk Operations
Multiple Update and Delete in Single Query

- Entity Framework **does not** support bulk operations
- **Z.EntityFramework.Plus** gives you the ability to perform **bulk update/delete** of entities
- Install **Z.EntityFramework.Plus.EFCore** as a NuGet package

Install-Package Z.EntityFramework.Plus.EFCore

- Read more: <https://entityframework-plus.net>

- Delete all users where **FirstName** matches given string

```
context.Employees  
.Where(u => u.FirstName == "Pesho")  
.Delete();
```



```
DELETE [dbo].[Employees]  
FROM [dbo].[Employees] AS j0 INNER JOIN (  
SELECT  
[Extent1].[Id] AS [Id]  
FROM [dbo].[Employees] AS [Extent1].[Name]  
WHERE N'Pesho' = [Extent1].[Name]  
) AS j1 ON (j0.[Id] = j1.[Id])
```

Bulk Update: Syntax

- Update all Employees with name “Nasko” to “Plamen”

```
context.Employees
    .Where(t => t.Name == "Nasko")
    .Update(u => new Employee() {Name = "Plamen"});
```

- Update all Employees' age to 99 who have the name “Plamen”

```
IQueryable<Employee> employees = context.Employees
    .Where(employee => employee.Name == "Plamen");

employees.Update(employee => new Employee() { Age = 99 });
```



Types of Loading

Lazy, Eager and Explicit Loading

Explicit Loading

- **Explicit loading** loads all records when they're needed
- Performed with the **Collection().Load()** method

```
var employee = context.Employees.First();
```

```
context.Entry(employee)
    .Reference(e => e.Department)
    .Load();
```

```
context.Entry(employee)
    .Collection(e => e.EmployeeProjects)
    .Load();
```

- Eager loading loads all related records of an entity at once
- Performed with the **Include** method

```
context.Towns.Include("Employees");
```

```
context.Towns.Include(town => town.Employees);
```

```
context.Employees
    .Include(employee => employee.Address)
    .ThenInclude(address => address.Town)
```

- Lazy Loading **delays** loading of data until it is used
- EF Core enables lazy-loading for any navigation property that can be **overridden**
- Offers better performance in certain cases
 - Less RAM usage
 - Smaller result sets returned

- Install Lazy Loading Proxies

```
Install-Package Microsoft.EntityFrameworkCore.Proxies
```

- Enable the package

```
void OnConfiguring (DbContextOptionsBuilder options)
{
    options
        .UseLazyLoadingProxies()
        .UseSqlServer(myConnectionString);
}
```



Concurrency Checks

Optimistic Concurrency Control in EF

- EF Core runs in **optimistic concurrency** mode (no locking)
 - By default the conflict resolution strategy in EF is "**last one wins**"
 - The last change overwrites all previous concurrent changes
- Enabling "**first wins**" strategy for certain property in EF:
 - **[ConcurrencyCheck]**

Last One Wins – Example

```
var contextFirst = new SoftUniDbContext();
var lastProjectFirstUser = contextFirst.Projects.First();
lastProjectFirstUser.Name = "Changed by the First User";

// The second user changes the same record
var contextSecondUser = new SoftUniDbContext();
var lastProjectSecond = contextSecondUser.Projects.First();
lastProjectSecond.Name = "Changed by the Second User";

// Conflicting changes: Last wins
contextFirst.SaveChanges();
contextSecondUser.SaveChanges();
```

Second user wins

First One Wins – Example

```
var context = new SoftUniDbContext();
var lastTownFirstUser = contextFirst.Towns.First();
lastTownFirstUser.Name = "First User";

var contextSecondUser = new SoftUniDbContext();
var lastTownSecondUser = contextSecondUser.Towns.First();
lastTownSecondUser.Name = "Second User";

context.SaveChanges();    Changes get saved
contextSecondUser.SaveChanges();
```

DbUpdateConcurrencyException



cascade

Cascade Operations

Deleting Related Entities

Cascade Delete Scenarios

- Required FK with **cascade delete** set to **true**, deletes everything related to the deleted property
- Required FK with **cascade delete** set to **false**, throws exception (it cannot leave the navigational property with no value)
- Optional FK with **cascade delete** set to **true**, deletes everything related to the deleted property.
- Optional FK with **cascade delete** set to **false**, sets the value of the FK to **NULL**

Cascade Delete with Fluent API

- Using **OnDelete** with **DeleteBehavior** Enumeration:
 - **DeleteBehavior.Cascade**
 - Deletes related entities (default for required FK)
 - **DeleteBehavior.Restrict**
 - Throws exception on delete
 - **DeleteBehavior.ClientSetNull**
 - Default behavior for optional FK (does not affect database)
 - **DeleteBehavior.SetNull**
 - Sets the property to null (affects database)

Cascade Delete with Fluent API (2)

- Cascade delete syntax:

```
modelBuilder.Entity<User>()
    .HasMany(u => u.Replies)
    .WithOne(a => a.Author)
    .OnDelete(DeleteBehavior.Restrict);
```

```
modelBuilder.Entity<User>()
    .HasMany(u => u.Replies)
    .WithOne(a => a.Author)
    .OnDelete(DeleteBehavior.Cascade);
```

- Databases can be accessed directly with **SQL queries** from C# code
- EF keeps track of the **model state**
- **Entity Framework-Plus** lets you bundle **update** and **delete** operations
- With multiple users, **concurrency** of operations must be observed
- **Cascade delete** is on by default



C# Auto Mapping Objects

Manual Mapping
and AutoMapper Library



SoftUni Team
Technical Trainers

Table of Contents

1. DTO Definition
2. Manual Mapping
3. AutoMapper





Data Transfer Objects

Definition and Usage

What is a Data Transfer Object?

- A **DTO** is an object that **carries data** between processes
 - Used to **aggregate** only the **needed information** in a single call
 - Example: In web applications, between the **server** and **client**
- Doesn't contain any logic – only **stores values**

```
public class ProductDTO
{
    public string Name { get; set; }
    public int StockQty { get; set; }
}
```

- **Remove** circular references
- **Hide** particular properties that **clients** are not supposed to view
- **Omit** some properties in order to **reduce** payload **size**
- **Flatten** object graphs that contain nested objects to make them more convenient for clients
- **Decouple** your service layer from your database layer

Manual Mapping

- Relationship Diagram



Manual Mapping (2)

- Get product name and stock quantity in a new DTO object

```
var product =  
    context.Products.FirstOrDefault();  
var productDto = new ProductDTO  
{  
    Name = product.Name,  
    StockQty = product.ProductStocks  
        .Sum(ps => ps.Quantity)  
};
```

Aggregate information from
mapping table

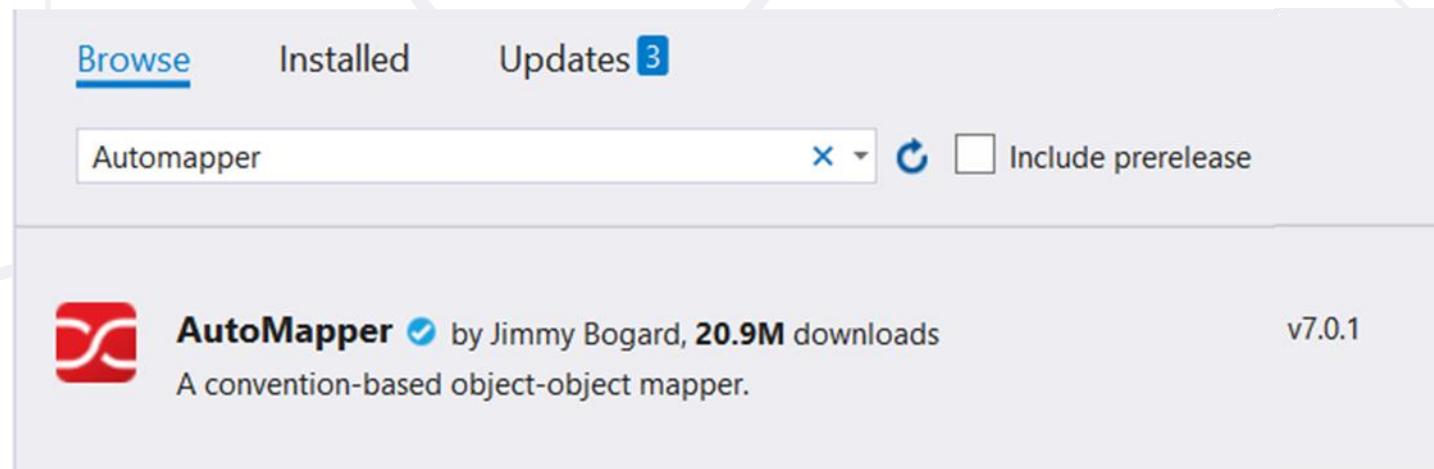


autoxmapper

AutoMapper Library
Automatic Translation of Domain Objects

What is AutoMapper?

- Library to eliminate **manual mapping** code
- Available as a **NuGet** Package
- Official [**GitHub Page**](#)



Install-Package AutoMapper

Initialization and Configuration

- AutoMapper offers a **static service** for use and configuration

- Add mappings between objects and DTOs

```
Mapper.Initialize(cfg => cfg.CreateMap<Product,  
ProductDTO>());
```

Target

Source

- Properties will be mapped **by name**

```
var product = context.Products.FirstOrDefault();  
ProductDTO dto = Mapper.Map<ProductDTO>(product);
```

Multiple Mappings

- You can configure all mapping configurations at once

```
Mapper.Initialize(cfg =>
{
    cfg.CreateMap<Product, ProductDTO>();
    cfg.CreateMap<Order, OrderDTO>();
    cfg.CreateMap<Client, ClientDTO>();
    cfg.CreateMap<SupportTicket, TicketDTO>();
});
```

Custom Member Mapping

- Map properties that don't match naming convention

```
Mapper.Initialize(cfg =>
    cfg.CreateMap<Product, ProductDTO>()
        .ForMember(dto => dto.StockQty,
                    opt => opt.MapFrom(src =>
                        src.ProductStocks.Sum(p =>
                            p.Quantity))));
);
```



The diagram illustrates the mapping process. It shows a blue arrow pointing from the 'src' variable (labeled 'Source') to the 'dto.StockQty' property (labeled 'Destination property'). Another blue arrow points from the 'src' variable to the expression 'src.ProductStocks.Sum(p => p.Quantity)', which is part of the 'MapFrom' lambda.

Flattening Complex Properties

- **AutoMapper** can also be used to flatten complex properties

```
Mapper.Initialize(cfg =>
    cfg.CreateMap<Event, CalendarEventViewModel>()
        .ForMember(dest => dest.Date,
                   opt => opt.MapFrom(src => src.Date.Date))
        .ForMember(dest => dest.Hour,
                   opt => opt.MapFrom(src => src.Date.Hour))
        .ForMember(dest => dest.Minute,
                   opt => opt.MapFrom(src => src.Date.Minute)));
```

Flattening Complex Objects

- **Flattening** of related objects is automatically supported

```
public class OrderDTO
{
    public string ClientName { get; set; }
    public decimal Total { get; set; }
}
```

- **AutoMapper** understands ClientName is the **Name** of a **Client**

```
Mapper.Initialize(cfg => cfg.CreateMap<Order, OrderDTO>());
OrderDTO dto = Mapper.Map<Order, OrderDTO>(order);
```

Unflattening Complex Objects

- **Unflattening** of related objects is automatically supported

```
public class OrderDTO
{
    public string ClientName { get; set; }
    public decimal Total { get; set; }
}
```

- **AutoMapper** understands ClientName is the **Name** of a **Client**,
but to unflatten it, it needs **ReverseMap**

```
Mapper
    .Initialize(cfg => cfg.CreateMap<Order, OrderDTO>()
    .ReverseMap());
```

Mapping Collections

- EF Core uses **IQueryable<T>** for all DB operations
 - AutoMapper can work with **IQueryable<T>** to map classes
- Using AutoMapper to map an entire DB collection:

```
var posts = context.Posts
    .Where(p => p.Author.Username == "gosho")
    .ProjectTo<PostDto>()
    .ToArray();
```

IQueryable<PostDto>

IQueryable<Post>

- Works like an automatic **.Select()**
 - EF Core generates **optimized SQL** (like with an **anonymous object**)

AutoMapper.Collection

- Adds ability to map collections to existing collections **without re-creating** the collection object
- Will **Add/Update/Delete** items from a preexisting collection object based on user defined equivalency between the collection's generic item type from the source collection and the destination collection

Mapper

```
.Initialize(cfg => cfg.AddCollectionMappers());
```

- **Automapper.Collection.EntityFrameworkCore** will help you mapping of EntityFrameowrk Core DbContext-object

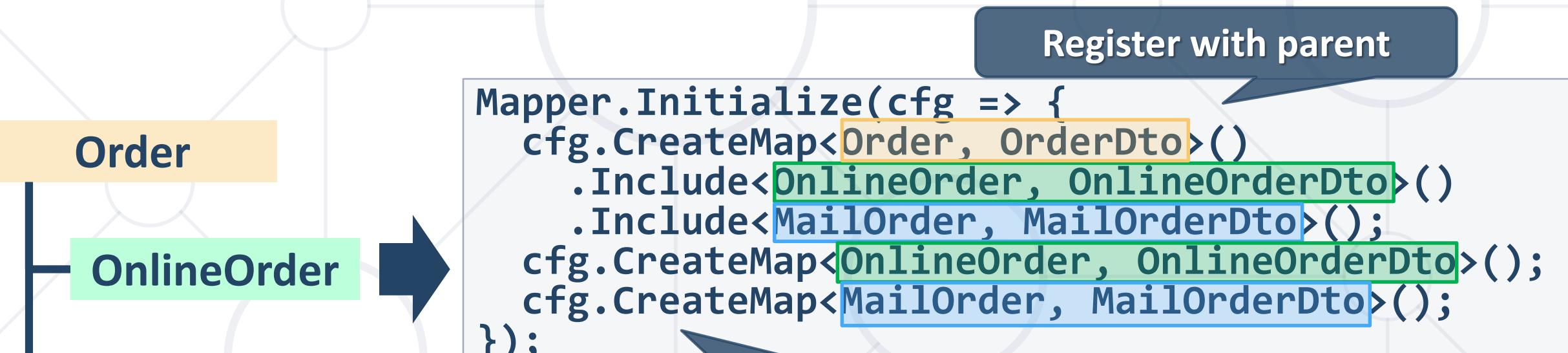
```
Mapper.Initialize(cfg =>
{
    cfg.AddCollectionMappers();
    cfg.SetGeneratePropertyMaps
        <GenerateEntityFrameworkCorePrimaryKeyPropertyMaps<Context>>();
    // Configuration code
});
```

- Comparing to a single existing Entity for updating

```
dbContext.Orders.Persist().InsertOrUpdate<OrderDTO>(newOrderDto);
dbContext.Orders.Persist().InsertOrUpdate<OrderDTO>(existingOrderDto);
dbContext.Orders.Persist().Remove<OrderDTO>(deletedOrderDto);
dbContext.SubmitChanges();
```

Inheritance Mapping

- Inheritance chains are defined via `Include()`
- AutoMapper chooses the most appropriate child class



Mapping Profiles

- We can extract our configuration to a class (called a **profile**)

```
public class ForumProfile : Profile
{
    public ForumProfile()
    {
        CreateMap<Post, PostDto>();
        CreateMap<Category, CategoryDto>();
    }
}
```

using AutoMapper;

- Using our configuration class:

```
Mapper.Initialize(cfg => cfg.AddProfile<ForumProfile>());
```

- To reduce round-trip latency and payload size, data is transformed into a **DTO**
- **AutoMapper** is a library that automates this process and reduces boilerplate code
- Complex objects can be **flattened** to fractions of their sizes



XML Processing

Parsing XML

XDocument and LINQ



SoftUni Team
Technical Trainers



SoftUni
Foundation



Technical Trainers
Software University

Table of Contents

1. XML Format
2. Processing XML
3. XML in Entity Framework
4. XML Attributes





What is XML?

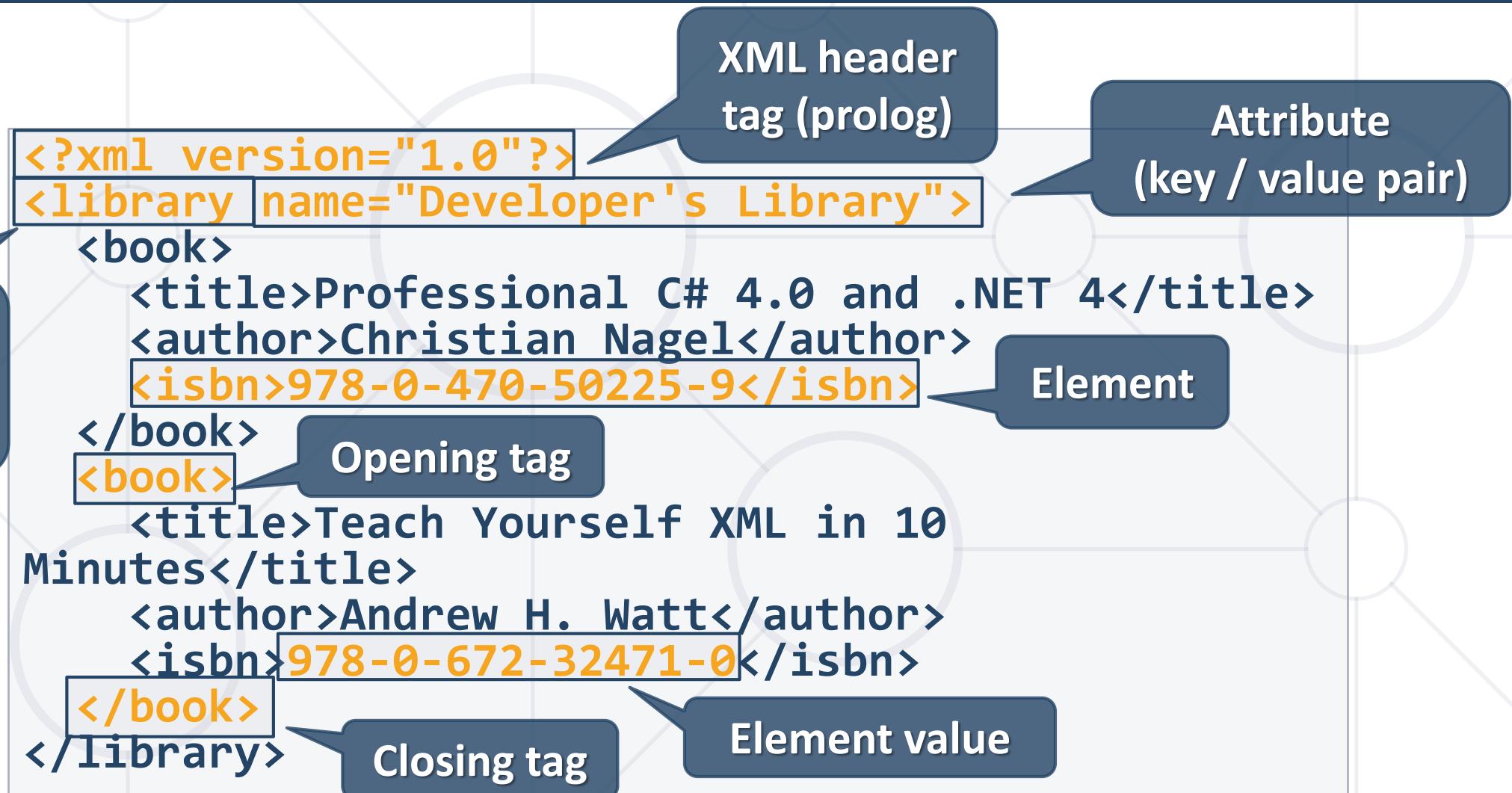
Format Description and Application

What is XML?

- EXtensible Markup Language
 - Universal notation (data format / language) for describing structured data using text with tags
 - Designed to store and transport data
 - The data is stored together with the meta-data about it



XML – Example



- Header – defines a **version** and character **encoding**

```
<?xml version="1.0" encoding="UTF-8"?>
```

- **Elements** – define the structure
- **Attributes** – element metadata
- **Values** – actual data, that can also be nested elements

Element name

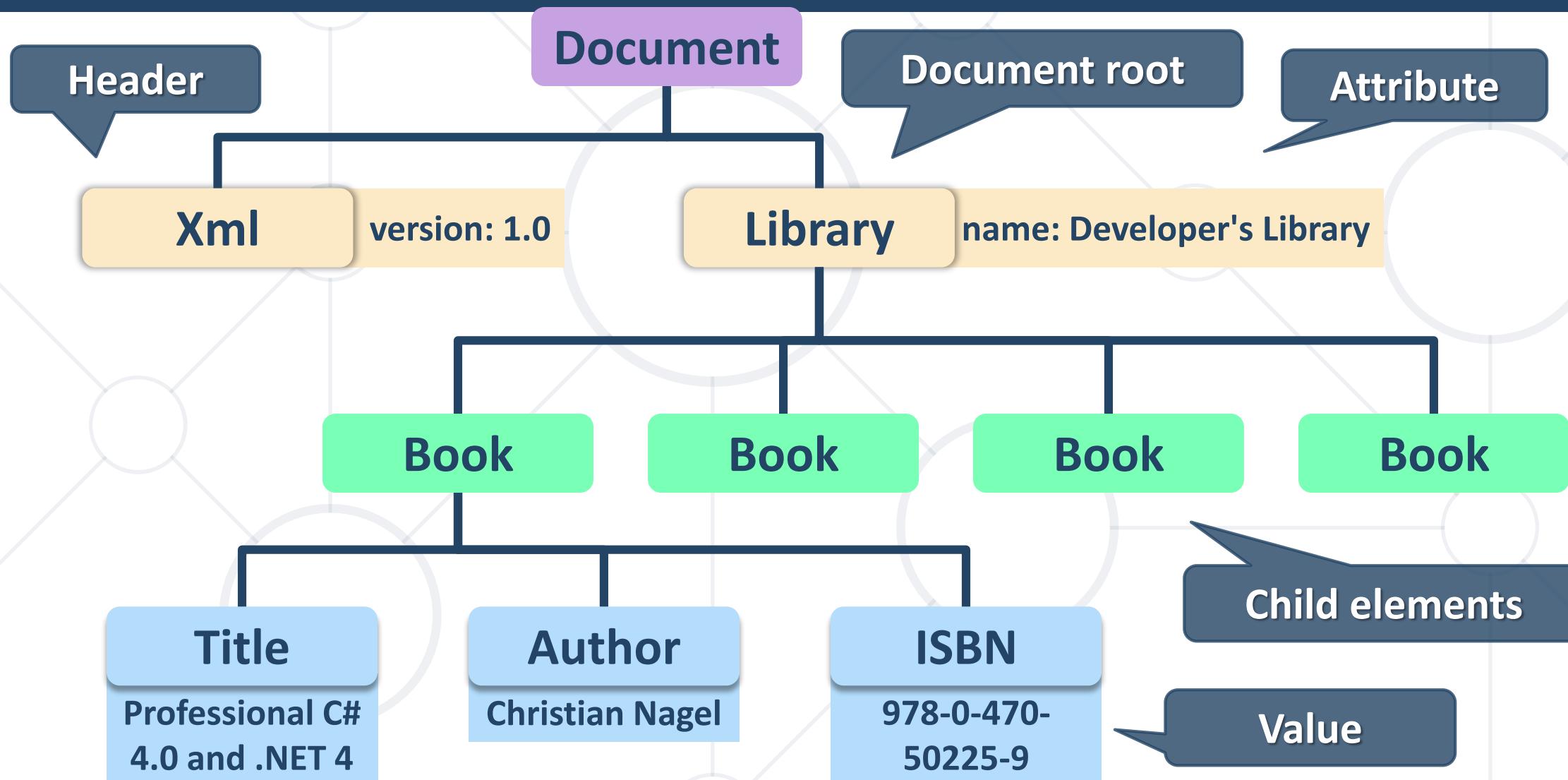
Attribute

Value

```
<title lang="en">Professional C# 4.0 and .NET 4</title>
```

- Root element – required to **only** have **one**

XML - Structure



- Similarities between XML and HTML
 - Both are **text based** notations
 - Both use **tags** and **attributes**
- Differences between XML and HTML
 - HTML describes documents, XML is a syntax for describing other languages (**meta-language**)
 - HTML describes the **layout** and the structure of information
 - XML requires the documents to be **well-formatted**



XML: Advantages

- Advantages of XML:
 - XML is **human-readable** (unlike binary formats)
 - Stores any kind of **structured data**
 - Data comes with self-describing **meta-data**
 - Full Unicode support
 - Custom XML-based languages can be designed for certain apps
 - **Parsers** available for virtually all languages and platforms



XML: Disadvantages

- Disadvantages of XML:
 - XML data is **bigger** (takes more space) than binary or JSON
 - More memory consumption, more network traffic, more hard-disk space, more resources, etc.
 - **Decreased performance**
 - CPU consumption: need of parsing / constructing the XML tags
 - XML is **not** suitable for **all** kinds of **data**
 - E.g. binary data: graphics, images, videos, etc.



Parsing XML

Using XDocument and LINQ

LINQ to XML

- LINQ to XML
 - Use the power of **LINQ** to process XML data
 - Easily read, search, write, modify XML documents
- LINQ to XML classes:
 - **XDocument** – represents a LINQ-enabled XML document (containing prolog, root element, ...)
 - **XElement** – main component holding information



- To process an XML string:

```
string str =  
@"<?xml version=""1.0""?>  
<!-- comment at the root level --&gt;<br/><Root>  
    <Child>Content</Child>  
</Root>";  
XDocument doc = XDocument.Parse(str);
```

- Loading XML directly from file:

```
XDocument xmlDoc = XDocument.Load("../books.xml");
```

Working with XDocument

```
var cars = xmlDoc.Root.Elements();  
foreach (var car in cars)  
{  
    string make = car.Element("make").Value;  
    string model = car.Element("model").Value;  
    Console.WriteLine($"{make} {model}");  
}
```

Access root element

Get collection of children

Access element by name

Get value

Working with XDocument (2)

- Set an element value by name
 - If it doesn't exist, it will be **added**
 - If it is set to **null**, it will be **removed**

```
customer.SetElementValue("birth-date", "1990-10-04T00:00:00");
```

- Remove an element from its parent

```
var youngDriver = customer.Element("is-young-driver");
youngDriver.Remove();
```

Working with XDocument (3)

- Get or set an element attribute by name

```
customer.Attribute("name").Value
```

- Get a list of all attributes for an element

```
var attrs = customer.Attributes;
```

- Set an attribute value by name

- If it doesn't exist, it will be **added**

- If it is set to **null**, it will be **removed**

```
customer.SetAttributeValue("age", "21");
```

LINQ to XML – Searching with LINQ

- Searching in XML with LINQ is like searching with LINQ in array

```
XDocument xmlDoc = XDocument.Load("cars.xml");
var cars = xmlDoc.Root.Elements()
    .Where(e => e.Element("make").Value == "Opel" &&
        (long)e.Element("travelled-distance") >= 300000)
    .Select(c => new
    {
        Model = c.Element("model").Value,
        Traveled = c.Element("travelled-distance").Value
    })
    .ToList();
foreach (var car in cars)
    Console.WriteLine(car.Model + " " + car.Traveled);
```

Creating XML with XElement

- XDocuments can be composed from XElements and XAttributes

```
<books>
  <book>
    <author>Don Box</author>
    <title lang="en">Essential .NET</title>
  </book>
</book>
```

```
XDocument xmlDoc = new XDocument();
xmlDoc.Add(
  new XElement("books",
    new XElement("book",
      new XElement("author", "Don Box"),
      new XElement("title", "ASP.NET", new XAttribute("lang",
    "en")))
  ));
```

Add as root

Added with value

Optional attribute

Serializing XML to File

- To flush a XDocument to file with default settings:

```
xmlDoc.Save("myBooks.xml");
```

- To disable automatic indentation:

```
xmlDoc.Save("myBooks.xml", SaveOptions.DisableFormatting);
```

- To serialize **any object** to file:

```
var serializer = new XmlSerializer(typeof(ProductDTO));  
using (var writer = new StreamWriter("myProduct.xml"))  
{  
    serializer.Serialize(writer, product);  
}
```

Deserialize XML from String XML

- To **deserialize** an object from a XML string

```
var serializer = new XmlSerializer(typeof(OrderDto[]), new  
XmlRootAttribute("Orders"));  
  
var deserializedOrders =  
(OrderDto[])serializer.Deserialize(new StringReader(xmlString));
```

- Specifying **root attribute** name

```
var attr = new XmlRootAttribute("Orders");  
var serializer = new XmlSerializer(typeof(OrderDto[]), attr);  
  
var deserializedOrders =  
(OrderDto[])serializer.Deserialize(new  
StringReader(xmlString));
```



XML Attributes

Using `xml attributes`

- We can use several attributes to control serialization to XML
 - **[XmlAttribute("Name")]** – Specifies the type's **name** in XML
 - **[XmlAttribute("name")]** – Serializes as **XML Attribute**
 - **[XmlElement]** – Serialize as **XML Element**
 - **[XmlIgnore]** – **Do not** serialize
 - **[XmlArray]** – Serialize as an **array** of XML elements
 - **[XmlRoot]** – Specifies the **root** element name
 - **[XmlText]** – Serialize **multiple xml elements on one line**

XML Attributes: Example

- We can use several XML attributes to control serialization

```
[XmlAttribute("Book")]
public class BookDto
{
    [XmlAttribute("name")]
    public string Name { get; }

    [XmlElement("Author")]
    public string Author { get; }

    [XmlIgnore]
    public decimal Price { get; }
}
```

XML Type name

Not serialized



```
<Book name="It">
    <Author>Stephen King</Author>
</Book>
<Book name="Frankenstein">
    <Author>Mary Shelley</Author>
</Book>
<Book name="Queen Lucia">
    <Author>E.F. Benson</Author>
</Book>
<Book name="Paper Towns">
    <Author>John Green</Author>
</Book>
```



XML Serialization

Live Demo

Summary

- **XDocument** is a system object for working with XML in .NET, which supports LINQ
- XML can be read and saved **directly to file**
- **XML Attributes** are easy way to describe the **XML file**



External Format Processing

Parsing JSON, JSON.NET



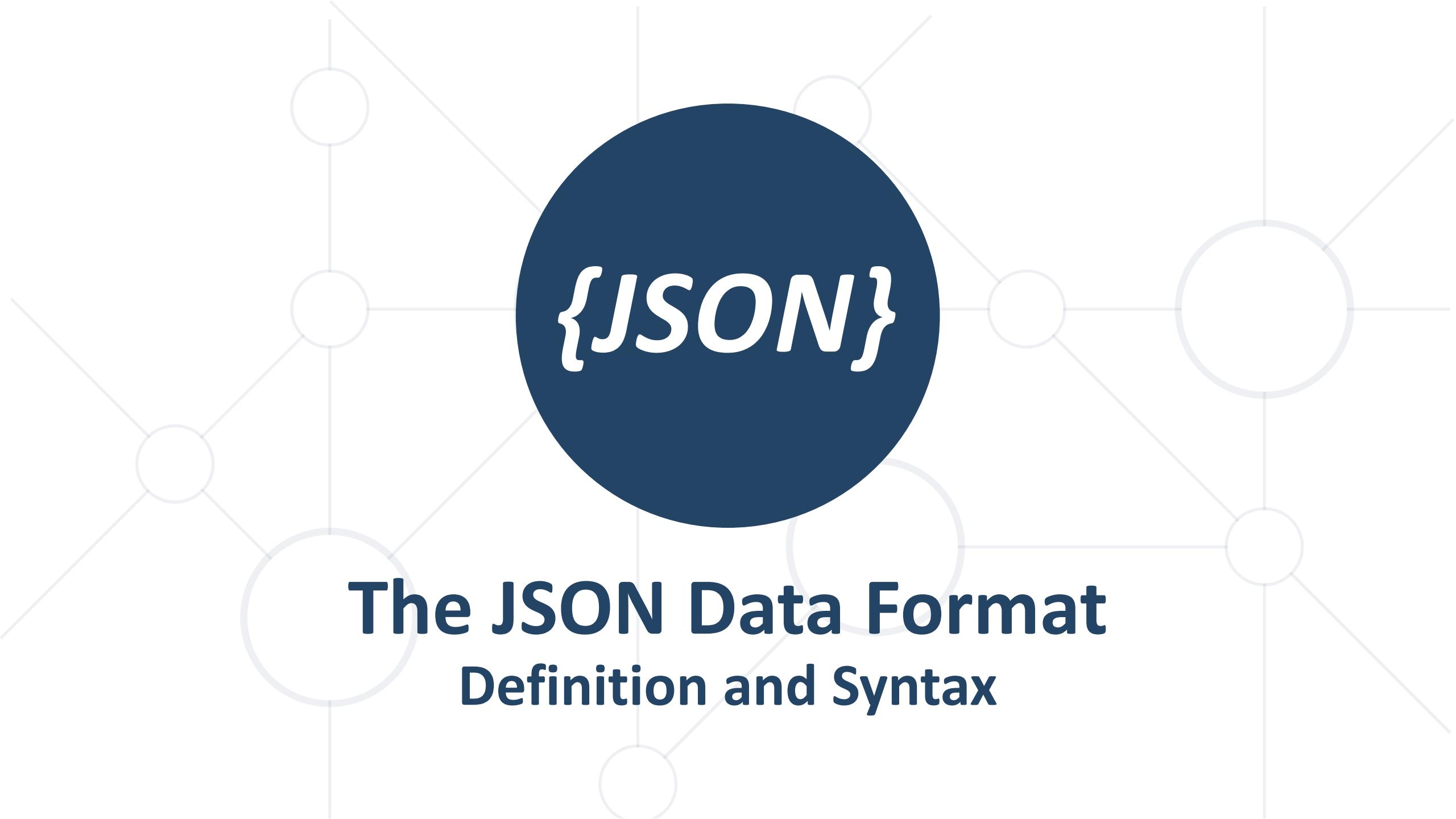
SoftUni Team

Technical Trainers

Table of Contents

1. JSON Data Format
2. Processing JSON
3. JSON.NET





{JSON}

The JSON Data Format

Definition and Syntax

- **JSON (JavaScript Object Notation)** is a lightweight data format
 - Human and machine-readable plain text
 - Based on **JavaScript** objects
 - Independent of development platforms and languages
 - JSON data consists of:
 - Values (strings, numbers, etc.)
 - Key-value pairs: { **key : value** }
 - Arrays: [**value1, value2, ...**]

JSON Data Format (2)

- The JSON data format follows the rules of object creation in JS

- Strings, numbers and Booleans** are valid JSON:

```
"this is a string and is valid JSON"
```

```
3.14
```

```
true
```

- Arrays** are valid JSON:

```
[5, "text", true]
```

- Objects** are valid JSON (key-value pairs):

```
{  
    "firstName": "Vladimir", "lastName": "Georgiev",  
    "jobTitle": "Technical Trainer", "age": 25  
}
```



Processing JSON

Parsing JSON in C# and .NET

Built-in JSON Support

- .NET has a built-in **DataContractJsonSerializer** class
- Contained in **System.Runtime.Serialization** assembly
- Supports **deserializing** (parsing) strings and **serializing** objects
- Including **DataContractJsonSerializer** into a project:

```
using System.Runtime.Serialization.Json;
```



Serializing JSON

- DataContractJsonSerializer can serialize an object:

```
static string SerializeJson<T>(T obj)
{
    var serializer = new DataContractJsonSerializer(obj.GetType());
    using (var stream = new MemoryStream())
    {
        serializer.WriteObject(stream, obj);
        var result = Encoding.UTF8.GetString(stream.ToArray());
        return result;
    }
}
```

```
var product = new Product();
product {Json.Product}
  Cost      25
  Description null
  Id        0
  Name     "Oil Pump"
```



```
{
  "Id":0,
  "Name":"Oil Pump",
  "Description":null,
  "Cost":25
}
```

Deserializing JSON

- DataContractJsonSerializer can deserialize a JSON string:

```
static T DeserializeJson<T>(string jsonString)
{
    var serializer = new DataContractJsonSerializer(typeof(T));
    var jsonStringBytes = Encoding.UTF8.GetBytes(jsonString);
    using (var stream = new MemoryStream(jsonStringBytes))
    {
        var result = (T)serializer.ReadObject(stream);
        return result;
    }
}
```

```
{
    "Id":0,
    "Name":"Oil Pump",
    "Description":null,
    "Cost":25
}
```



```
var product = new Product();
product {Json.Product}
  Cost      25
  Description null
  Id        0
  Name     "Oil Pump"
```



JSON.NET

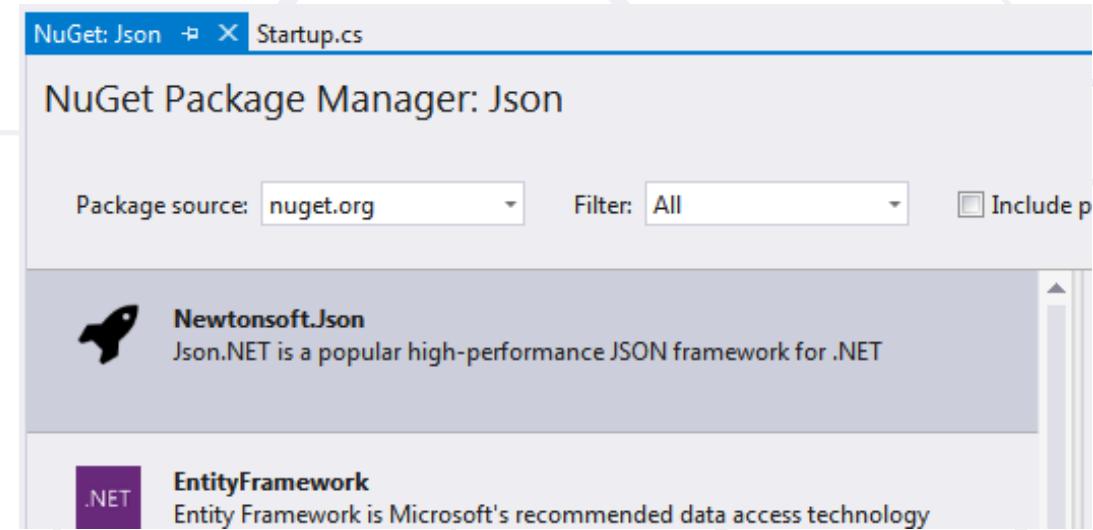
Better JSON Parsing for .NET Developers

What is JSON.NET?

- **JSON.NET** is a **JSON framework** for .NET
 - **More functionality** than built-in functionality
 - Supports **LINQ-to-JSON**
 - Out-of-the-box support for parsing between **JSON** and **XML**
 - Open-source project: <http://www.newtonsoft.com>
 - Json.NET vs .NET Serializers: <https://www.newtonsoft.com/json/help/html/JsonNetVsDotNetSerializers.htm>

Installing JSON.NET

- To install JSON.NET use the **NuGet Package Manager**:



- Or with a command in the Package Manager Console:

```
Install-Package Newtonsoft.Json
```

General Usage

- JSON.NET exposes a static service **JsonConvert**
- Used for parsing and configuration
 - To **Serialize** an object:

```
var jsonProduct = JsonConvert.SerializeObject(product);
```

- To **Deserialize** an object:

```
var objProduct =  
    JsonConvert.DeserializeObject<Product>(jsonProduct);
```

- JSON.NET can be configured to:
 - **Indent** the output JSON string
 - To convert JSON to **anonymous types**
 - To control the **casing** and **properties** to parse
 - To skip errors
- JSON.NET also supports:
 - LINQ-to-JSON
 - Direct parsing between XML and JSON

Configuring JSON.NET

- By default, the result is a **single line of text**
- To indent the output string use **Formatting.Indented**

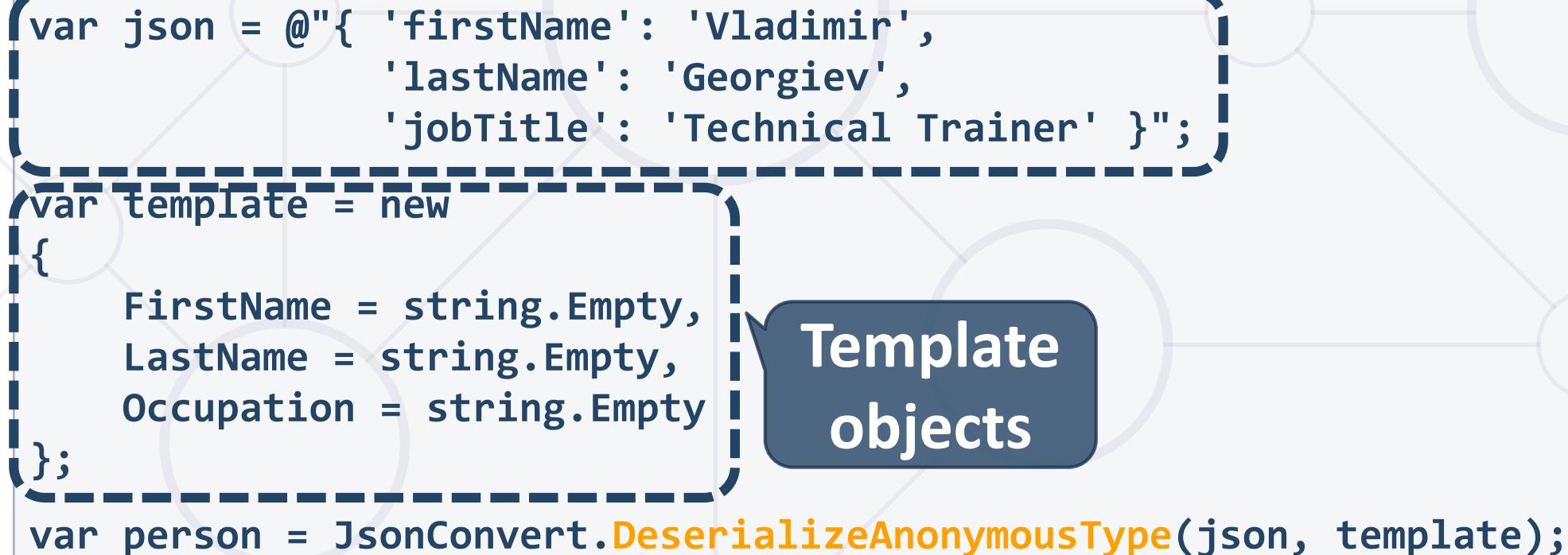
```
JsonConvert.SerializeObject(products, Formatting.Indented);
```

```
{  
    "pump": {  
        "Id": 0,  
        "Name": "Oil Pump",  
        "Description": null,  
        "Cost": 25.0  
    },  
    "filter": {  
        "Id": 0,  
        "Name": "Oil Filter",  
        "Description": null,  
        "Cost": 15.0  
    }  
}
```

Configuring JSON.NET

- Deserializing to **anonymous** types:

```
var json = @"{ 'firstName': 'Vladimir',
    'lastName': 'Georgiev',
    'jobTitle': 'Technical Trainer' }";  
  
var template = new  
{  
    FirstName = string.Empty,  
    LastName = string.Empty,  
    Occupation = string.Empty  
};  
  
var person = JsonConvert.DeserializeAnonymousType(json, template);
```



The diagram illustrates the process of deserializing JSON to an anonymous type. It features a central vertical rectangle containing the C# code. A dashed blue line labeled "Incoming JSON" originates from the top right of the code area and points to a blue speech bubble labeled "Template objects". Another dashed blue line originates from the bottom right of the code area and points to the final line of code, "JsonConvert.DeserializeAnonymousType(json, template);".

JSON.NET Parsing of Objects

- By default JSON.NET takes each property / field from the class and parses it
 - This can be controlled using attributes:

```
public class User
{
    [JsonProperty("user")]
    public string Username { get; set; }

    [JsonIgnore] Skip the property
    public string Password { get; set; }
}
```

Parse Username to user

JSON.NET Parsing of Objects

- By default JSON.NET takes each property / field from the class and parses it
 - This can be controlled using **ContractResolver**:

```
DefaultContractResolver contractResolver = new DefaultContractResolver()  
{  
    NamingStrategy = new SnakeCaseNamingStrategy()  
};  
  
var serialized = JsonConvert.SerializeObject(person, new JsonSerializerSettings()  
{  
    ContractResolver = contractResolver,  
    Formatting = Formatting.Indented  
});
```

- LINQ-to-JSON works with JObjects

- Create from JSON string:

```
JObject obj = JObject.Parse(jsonProduct);
```

- Reading from file:

```
var people = JObject.Parse(File.ReadAllText(@"c:\people.json"))
```

- Using **JObject**:

```
foreach (JToken person in people)
{
    Console.WriteLine(person["FirstName"]); // Ivan
    Console.WriteLine(person["LastName"]); // Petrov
}
```

- JObjects can be queried with LINQ

```
var json = JObject.Parse(@"{'products': [
    {'name': 'Fruits', 'products': ['apple', 'banana']},
    {'name': 'Vegetables', 'products': ['cucumber']}]}");

var products = json["products"].Select(t =>
    string.Format("{0} ({1})",
        t["name"],
        string.Join(", ", c["products"]))
)

// Fruits (apple, banana)
// Vegetables (cucumber)
```

XML-to-JSON

```
string xml = @"<?xml version='1.0' standalone='no'?>
<root>
    <person id='1'>
        <name>Alan</name>
        <url>www.google.com</url>
    </person>
    <person id='2'>
        <name>Louis</name>
        <url>www.yahoo.com</url>
    </person>
</root>";

 XmlDocument doc = new XmlDocument();
 doc.LoadXml(xml);
 string jsonText = JsonConvert.SerializeXmlNode(doc);
```

```
{
    "?xml": {
        "@version": "1.0",
        "@standalone": "no"
    },
    "root": [
        {
            "person": [
                {
                    "@id": "1",
                    "name": "Alan",
                    "url": "www.google.com"
                },
                {
                    "@id": "2",
                    "name": "Louis",
                    "url": "www.yahoo.com"
                }
            ]
        }
    ]
}
```

- JSON is cross platform data format
- **DataContractJsonSerializer** is the default JSON Parser in C#
- **JSON.NET** is a fast framework for working with JSON data

