EF Advanced Querying

Advanced Entity Framework Core

SoftUni Team

Technical Trainers

Software University

https://about.softuni.bg/

Table of Contents

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();
```

Enables to pass in a SQL command

- 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**

Native SQL Queries with Parameters

- Native SQL queries can also be parameterized:

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

Parameter placeholder

Parameter value

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

Executing a Stored Procedure

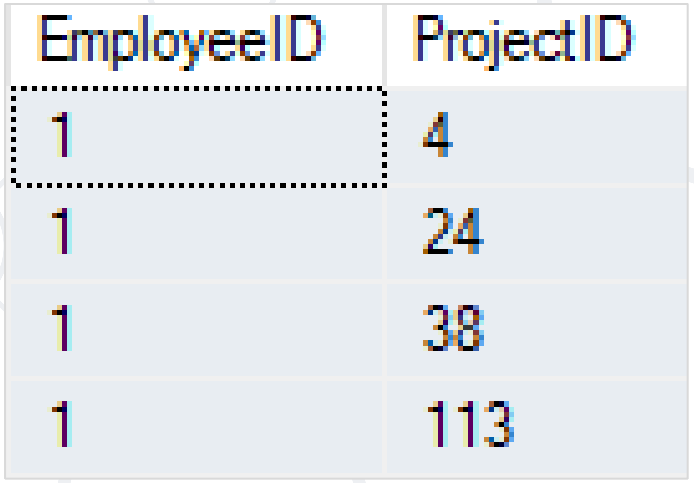
- Stored Procedures can be executed via SQL

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

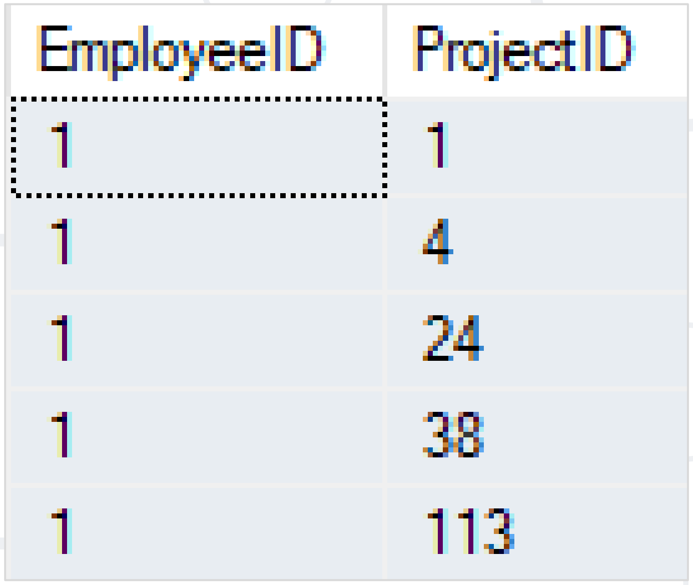
```csharp
var ageParameter = new SqlParameter("@age", 5);
var query = "EXEC UpdateAge @age";
context.Database.ExecuteSqlCommand(query, ageParameter);
```

Problem: Add Employee to Project

- Execute your own **SQL query** using the **stored procedure** to **add** a **project to** an **employee** in the SoftUni database



EmployeeID	ProjectID
1	4
1	24
1	38
1	113



EmployeeID	ProjectID
1	1
1	4
1	24
1	38
1	113

Solution: Add Employee to Project

- Create stored procedure

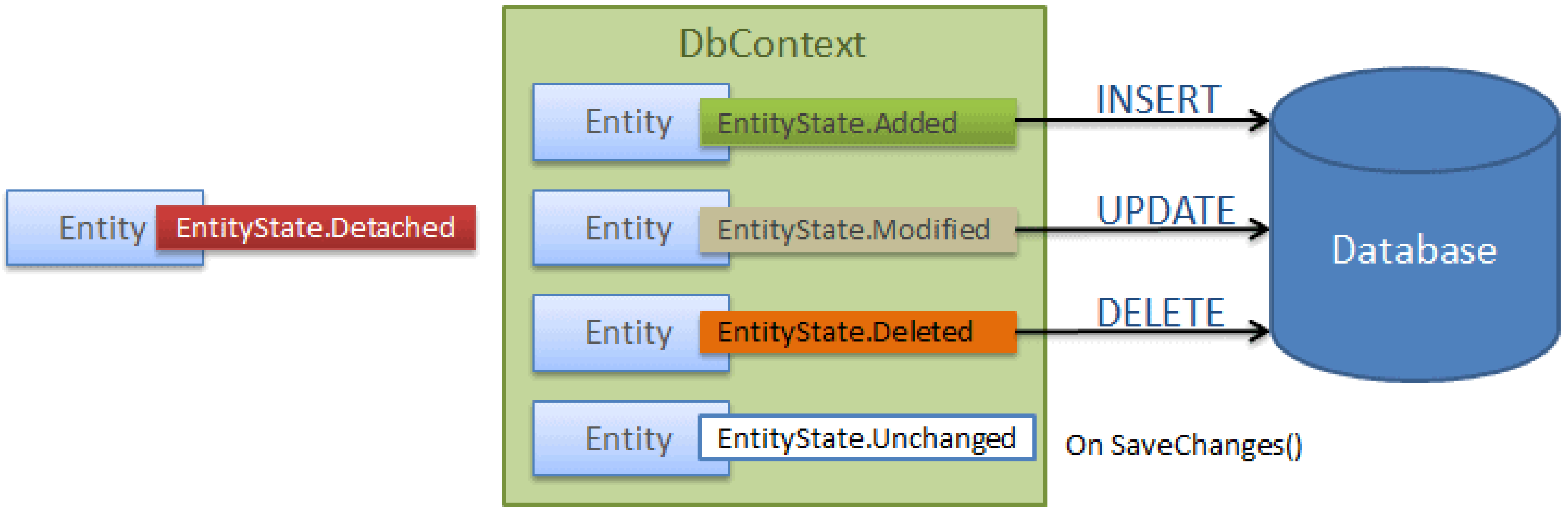
```
CREATE PROCEDURE sp_AddEmployeeToProjest
       @employeeId INT,
       @projectId INT
AS
BEGIN
       INSERT INTO EmployeesProjects
              (EmployeeID, ProjectID)
       VALUES
              (@employeeId, @projectId)
END
```

Solution: Add Employee to Project (2)

- Execute that procedure

```
static void Main()
{
  var context = new SoftUniContext();
  var employeeId = 1;
  var projectId = 1;

  context.Database.ExecuteSqlInterpolated($
    "EXEC sp_AddEmployeeToProjest {employeeId}, {projectId}");
}
```



Tracking the State of Entities

Attaching Objects

- Objects can be **Attached** to the context (tracked object) by calling the **Add** method on **DbSet**
- **Attached** objects are tracked and managed by the **DbContext**
- Object in **Attached** state will be **inserted into** the **database** the next time when **SaveChange()** is called

```
using (var context = new BloggingContext())
{
  var blog = new Blog { Name = "ADO.NET Blog" };
  context.Blogs.Add(blog);
  context.SaveChanges();
}
```

Puts the object into the Attached state

Detaching Objects

- Objects can be **Detached** from an object context (untracked object)
- **Detached** objects are not referenced by the **DbContext**
  - Behave like a normal objects, which are not related to EF
  - We can get detached objects using **AsNoTracking()**

```
var blogs = context.Blogs
    .AsNoTracking()
    .ToList();
```

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 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**

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.Modified;
      employee.FirstName = newName;
      softUniDbContext.SaveChanges();
   }
}
```

The context is disposed

BULK

Bulk Operations

Multiple Update and Delete in Single Query

Bulk Operations

- Bulk operations are **actions** that are performed **on** a **large scale**

Operations	1,000 Entities	2,000 Entities	5,000 Entities
SaveChanges	1,000 ms	2,000 ms	5,000 ms
BulkInsert	6 ms	10 ms	15 ms
BulkUpdate	50 ms	55 ms	65 ms
BulkDelete	45 ms	50 ms	60 ms
BulkMerge	65 ms	80 ms	110 ms

EntityFramework-Plus

- 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

Bulk Delete

- 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 == "Niki")
  .Update(u => new Employee { Name = "Stoyan" });
```

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

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

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

Problem: Delete Records with ProjectId

- **Delete** the **records** in the **EmployeesProjects** table, where **ProjectId is less than 3**

```
static void Main()
{
  var context = new SoftUniContext();

  context.EmployeesProjects
    .Where(x => x.ProjectId < 3).Delete();
}
```

- We **can't delete** in tables which don't have a primary key
- But Z.EntityFramework.Plus.EFCore and the **using Z.EntityFramework.Plus makes that possible**

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

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

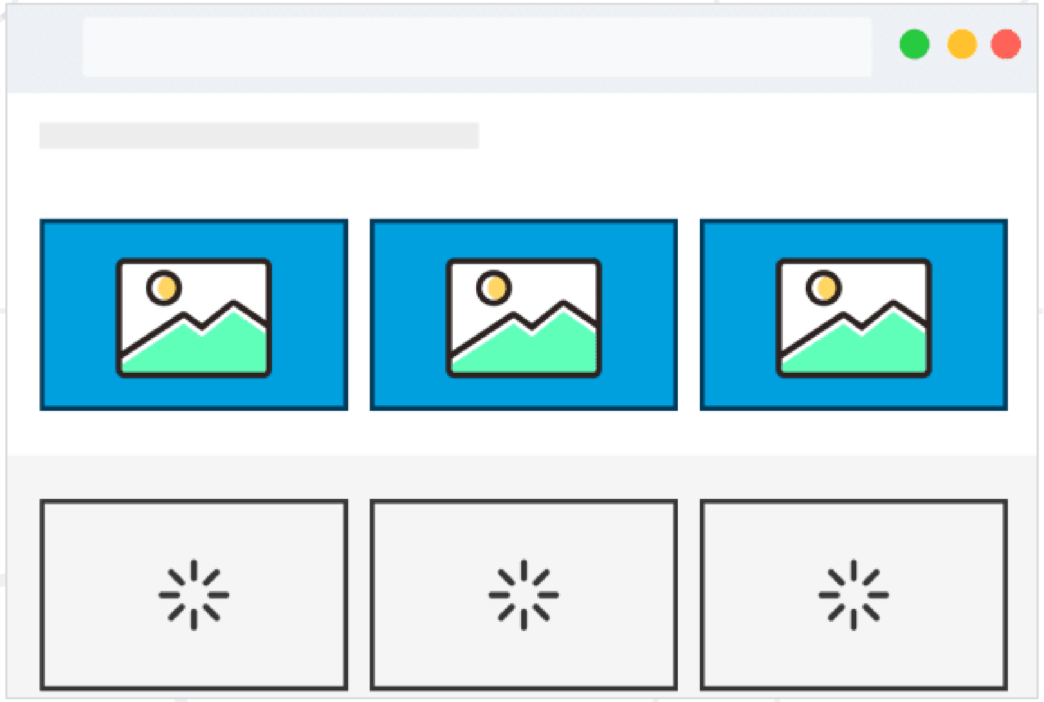
```
context.Towns.Include("Employees");
```

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

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

Lazy Loading

- Lazy Loading **delays** loading of data **until it is used**
- EF Core enables lazy-loading for any navigation property that can be **overridden** (**virtual**)
- Offers better performance in certain cases
  - Less RAM usage
  - Smaller result sets returned
- Each loading of navigational property is an addition query (N+1)



Enable Lazy Loading Proxies

- Install Lazy Loading Proxies

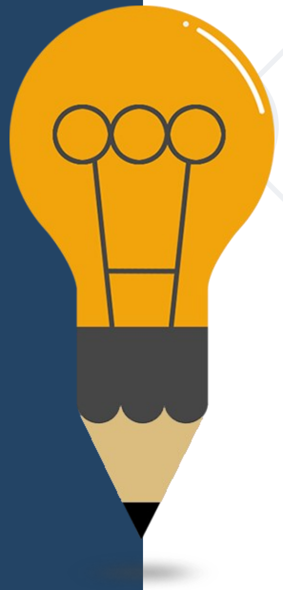
```
Install-Package Microsoft.EntityFrameworkCore.Proxies
```

- Enable the package

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

N+1 Problem

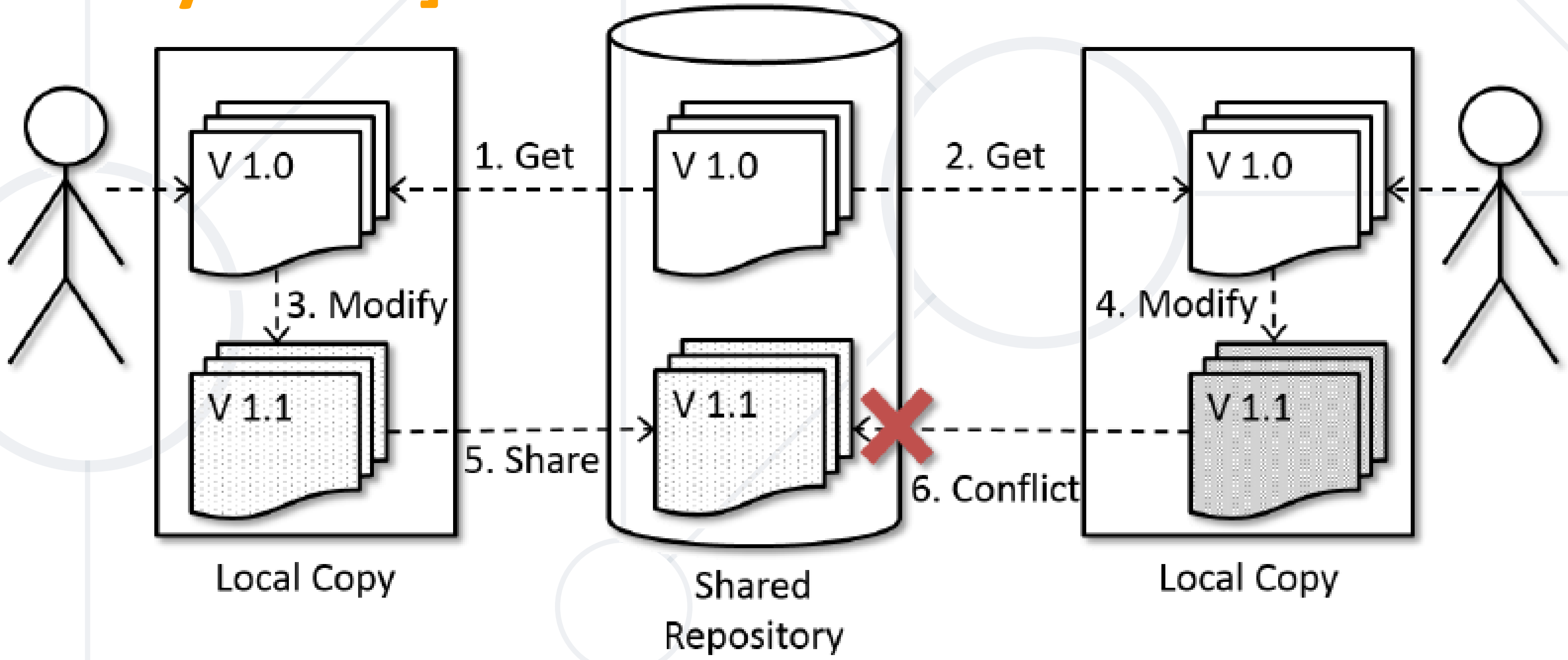
- Refreshing the article list page, sends 11 queries to the database
  - The **first query** finds the first 10 articles
  - The subsequent **10 queries**, find each article's comments
  - Total of 11 queries (N + 1)



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();
contextSecondUser.SaveChanges();
```

[ConcurrencyCheck]

Changes get saved

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 (1)

- 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);
```

Summary

- 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
- EF supports **lazy**, **eager** and **explicit loading**
- With multiple users, **concurrency** of operations must be observed
- **Cascade delete** is on by default

Questions?

Software University

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

License