Data Access with Entity Framework

Filip Ekberg
Principal Consultant & CEO

@fekberg fekberg.com

Entity Framework Core Tooling

Package Manager Console:

```
Scaffold-DbContext "ConnectionString" Microsoft.EntityFrameworkCore.SqlServer
```

.NET Core CLI:

```
dotnet ef dbcontext scaffold "ConnectionString" Microsoft.EntityFrameworkCore.SqlServer
```

Reverse Engineer the Database

Automatically generate classes for the different tables (Entities).

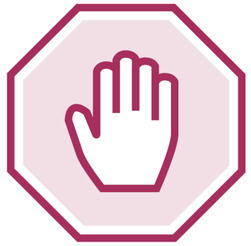
This includes relationships and a class to communicate with the database.

This process is known as
scaffolding

Reverse Engineering the Database

```
dotnet ef dbcontext scaffold --help

Usage: dotnet ef dbcontext scaffold [arguments] [options]

Arguments:
    <CONNECTION>   The connection string to the database.
    <PROVIDER>     The provider to use.
                   (E.g. Microsoft.EntityFrameworkCore.SqlServer)

Example Options:
    --context
    --namespace
    --output-dir
```

Keep the Connection String Secure



Never check it into source control

Consider a key vault

Inject using an environment variable

What does this scaffolding generate?

Does it reuse existing types that look similar? **No.**

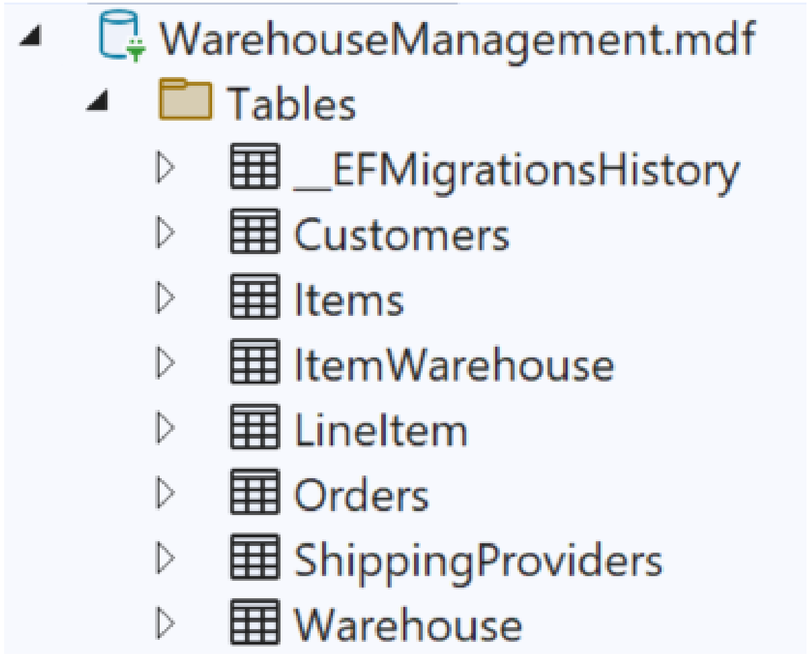
Navigation properties let you **load** the related **data when needed**

Many-to-many Relationship

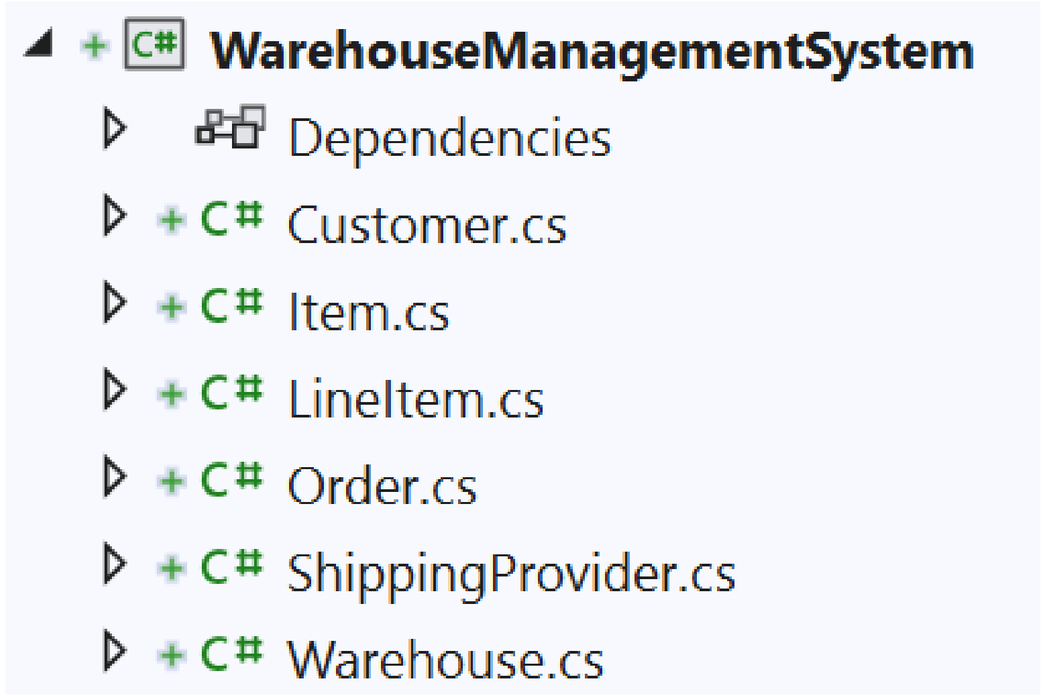
```
class Item
{
    public virtual ICollection<Warehouse> Warehouses { get; set; }
}

class Warehouse
{
    public virtual ICollection<Item> Items { get; set; }
}
```

Scaffolded Classes

Database schema



Entities represented in C#



You may **not** want to **expose**
the **entities** / database models
directly **through** an **API**

Requires a **mapping between domain** and **data models.**

This provides total control!

Query for Data Using **LINQ**

```
using var context = new WarehouseContext();
```

Query for Data Using **LINQ**

```
using var context = new WarehouseContext();

var customers =    context
                   .Customers
                   .Where(customer => customer.Name == "Filip Ekberg")
                   .ToList();
```

Query for Data Using **LINQ**

```
using var context = new WarehouseContext();

var customers =    context
                   .Customers
                   .Where(customer => customer.Name == "Filip Ekberg")
                   .ToList();
```

```
SELECT * FROM [Customers] WHERE Name = 'Filip Ekberg';
```

Avoid expensive queries!

It is always a good idea to **profile** your **queries** using SQL profiling tools

The **consumer** of the context
doesn't have to **care** about
what **database** is used!

The **connection** to the database is **handled** by the **dbcontext**

Learn More About IDisposable

C# Advanced Language Features

Filip Ekberg

Execute the Query

```
using var context = new WarehouseContext();

IQueryable<Customer>  query = context
                              .Customers
                              .Where(c => c.Name == "Filip");
```

Execute the Query

```
using var context = new WarehouseContext();

IQueryable<Customer>  query = context
                              .Customers
                              .Where(c => c.Name == "Filip");
```

No query hast been executed against the database yet

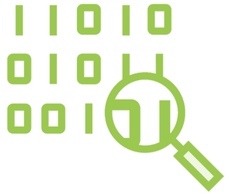
Execute the Query

```
using var context = new WarehouseContext();

IQueryable<Customer>  query = context
                              .Customers
                              .Where(c => c.Name == "Filip");


Customer first = query.First();

List<Customer> all = query.ToList();
```

What We've Achieved so Far

A pre-existing database
Could have been SQL Server, SQLite, MySQL, Oracle, ...

Reverse engineered the schema
Classes representing entities were scaffolded and a dbcontext created



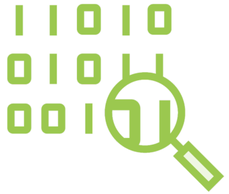
Consuming the data
Using the context and the exposed DbSets

No knowledge of SQL necessary
SQL generated by Entity Framework Core

What We've Achieved so Far

A pre-existing database
Could have been SQL Server, SQLite, MySQL, Oracle, ...

Reverse engineered the schema
Classes representing entities were scaffolded and a dbcontext created



Consuming the data
Using the context and the exposed DbSets

No knowledge of SQL necessary
SQL generated by Entity Framework Core

Navigation properties marked
as **virtual** are **lazy loaded!**

Lazy Loading

Reduces memory footprint

Less data transferred

Include Referenced Data

```
context.Orders.Include(order => order.Customer)
              .ThenInclude(customer => customer.Invoices);
```

Used LINQ before?

This is exactly the same!

Always **perform** a **lookup** on an **indexed column**!

Query **Considerations**

Profile queries

Add an index where necessary and avoid row scans

Assume larger data sets

Problems may not occur locally because you are working with a small data set

Cascade Delete

“Cascading deletes are needed when a dependent/**child entity** can **no longer** be **associated** with its current principal/**parent**.

This can happen because the principal/**parent** is **deleted**, or it can happen when the principal/parent still exists but the dependent/child is no longer associated with it.”

docs.microsoft.com/en-us/ef/core/saving/cascade-delete

Example: Mapping Domain Model to Entity

```
foreach(var orderDomainModel in LocalData.Load())
{
    var orderEntity = new Order
    {
        Id                = orderDomainModel.Id, // Shouldn't be re-created..
        Customer          = ...                  // Does it already exist?
        ShippingProvider  = ...                  // Does it already exist?
    };

    foreach(var item in orderDomainModel.LineItems)
    {
        // Does it already exist? Attach..
    }

    context.Orders.Add(orderEntity);
}

context.SaveChanges();
```

We'd like to **introduce** a **library**
that many **projects** can **share**
to **access data**

This makes it easier to experiment with queries and profile them

Inspecting the Interaction while Debugging

```
protected override void OnConfiguring(DbContextOptionsBuilder
                                      optionsBuilder)
{
    optionsBuilder.UseLoggerFactory(
        new LoggerFactory(new[] {
            new DebugLoggerProvider()
        })
    );
}
```

Already Have **Another Database**?

Install the correct provider

This lets Entity Framework Core know exactly how to communicate with that database

Scaffold the entities

Reverse engineer the schema into correct models together with a dbcontext that you use to interact with the db

It’s **not common** that you **change** the **database** and provider mid-project

When Do You **Reverse Engineer** the Schema?

At the start of the project

When there's a schema change

Scaffolding Multiple Databases

```
dotnet ef dbcontext scaffold
    "Data source=...“
    Microsoft.EntityFrameworkCore.Sqlite
    --context WarehouseSQLiteContext
    --output-dir "SQLite"
    --namespace "Warehouse.Data.SQLite“

dotnet ef dbcontext scaffold
    "Data source=..."
    Microsoft.EntityFrameworkCore.SqlServer
    --context WarehouseSqlServerContext
    --output-dir "SqlServer"
    --namespace "Warehouse.Data.SqlServer“
```

Scaffolding SQLite

```
dotnet ef dbcontext scaffold
    "Data source=...“
    Microsoft.EntityFrameworkCore.Sqlite
    --context WarehouseSQLiteContext
    --output-dir "SQLite"
    --namespace "Warehouse.Data.SQLite“

dotnet ef dbcontext scaffold
    "Data source=..."
    Microsoft.EntityFrameworkCore.SqlServer
    --context WarehouseSqlServerContext
    --output-dir "SqlServer"
    --namespace "Warehouse.Data.SqlServer“
```

Scaffolding SQLite

```
dotnet ef dbcontext scaffold
    "Data source=...“
    Microsoft.EntityFrameworkCore.Sqlite
    --context WarehouseSQLiteContext
    --output-dir "SQLite"
    --namespace "Warehouse.Data.SQLite“

dotnet ef dbcontext scaffold
    "Data source=..."
    Microsoft.EntityFrameworkCore.SqlServer
    --context WarehouseSqlServerContext
    --output-dir "SqlServer"
    --namespace "Warehouse.Data.SqlServer“
```

Exercise: Change All Ids to Guids on Your Own

```
// Before
public string Id { get; set; }
public string CustomerId { get; set; }
public string ShippingProviderId { get; set; }

// After
public Guid Id { get; set; }
public Guid CustomerId { get; set; }
public Guid ShippingProviderId { get; set; }
```

Code First

Start with the models

Add properties and reference other models

Create a simple dbcontext

Define which tables to create in the database

Example: Code First

```
class Customer
{
    public Guid Id { get; set; }
    public string Name { get; set; }
}
class WarehouseContext : DbContext
{
    public DbSet<Customer> Customers { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder
                                          optionsBuilder)
    {
        optionsBuilder.UseSqlite(@"Data source=warehouse.db");
    }
}
```

Migrations

Built into Entity Framework

Generate code to update the database to reflect the current models and relationships

Lazy Loading

This is an optimization we can keep as we know we are using Entity Framework Core!

We need to create a migration!

Entity Framework Core **Migrations**

Generate code to create tables, columns & relationships

Schema to match your entities

Supports update/rollback

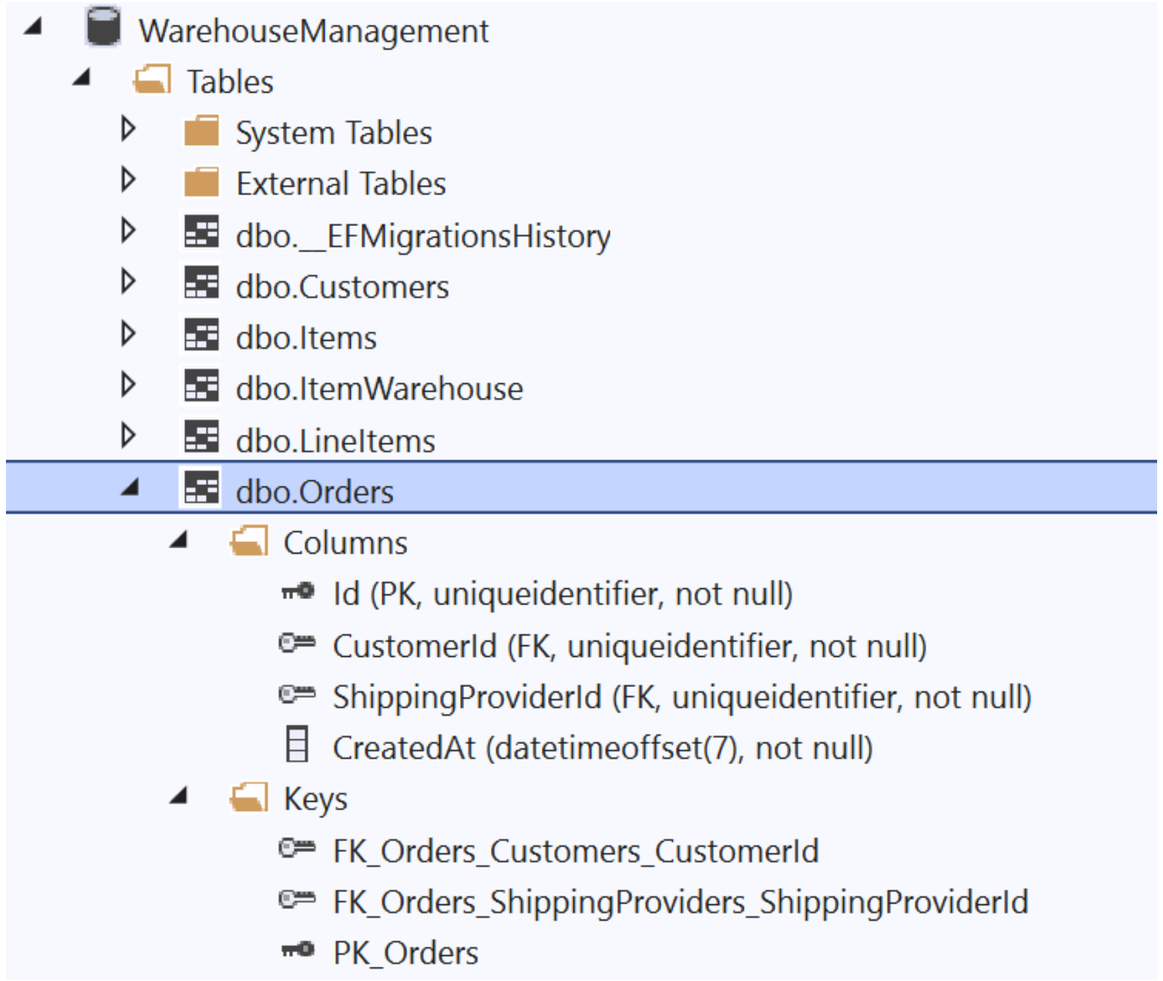
The **migrations** are **NOT automatically applied**.

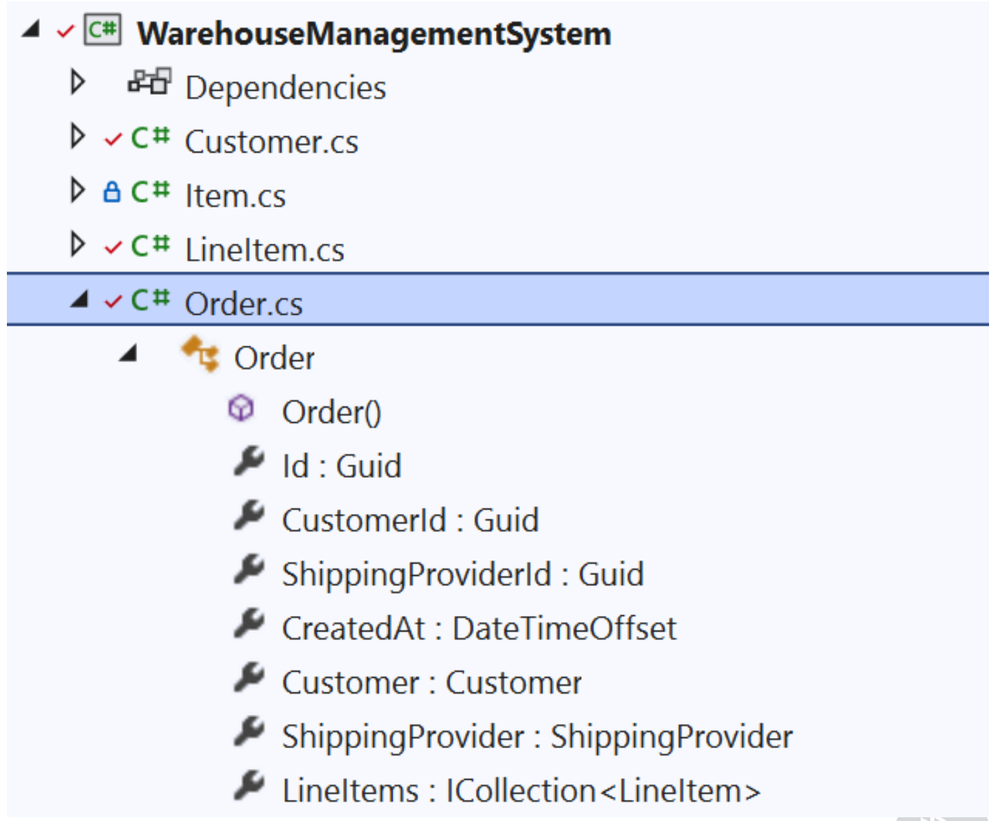
Unless explicitly setup.

Apply Migrations

```
using var context = new WarehouseContext();

context.Database.Migrate();
```

Using **Entity Framework Core**





Lazy Loaded Properties

```
class Order
{
    ...

    public virtual Customer Customer                 { get; set; }
    public virtual ShippingProvider ShippingProvider { get; set; }
    public virtual ICollection<LineItem> LineItems   { get; set; }
}
```

Lazy Loaded Properties

```
class Order
{
    ...

    public virtual Customer Customer                    { get; set; }
    public virtual ShippingProvider ShippingProvider    { get; set; }
    public virtual ICollection<LineItem> LineItems      { get; set; }
}
```

Mark the relational data as virtual and Entity Framework Core will let you lazy load this

Avoid eager loading!

Leads to unnecessary allocations and transferred data

The **provider** you **install** will know how to **translate** your queries into **SQL** for that database

Creating, Updating and Deleting

```
using var context = new WarehouseContext();

Order newOrder = new() { ... };

context.Orders.Add(newOrder);
```

Creating, Updating and Deleting

```
using var context = new WarehouseContext();

Order newOrder = new() { ... };

context.Orders.Add(newOrder);

context.Customers.Update(customerToUpdate);
```

Creating, Updating and Deleting

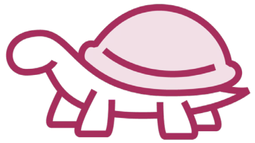
```
using var context = new WarehouseContext();

Order newOrder = new() { ... };

context.Orders.Add(newOrder);

context.Customers.Update(customerToUpdate);

context.Customers.Remove(customerToRemove);
```

Creating, Updating and Deleting

```
using var context = new WarehouseContext();

Order newOrder = new() { ... };

context.Orders.Add(newOrder);

context.Customers.Update(customerToUpdate);

context.Customers.Remove(customerToRemove);

context.SaveChanges();
```

Considerations

It helps to understand the internals
Inspect the generated SQL and the database interaction to understand how it performs



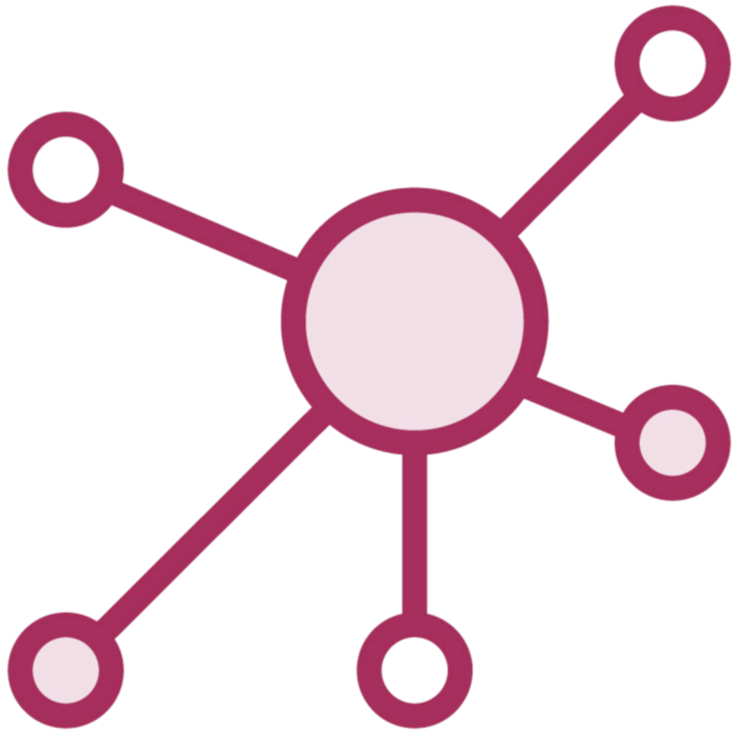
Entity Framework Core will generate SQL
Based on your expressions, even if you trust its optimization you may have produced a slow query

Index!
No matter what provider and database, you need good indices. Profile the database and queries to find bottlenecks

High load + No index = **BAD**

Can lead to database becoming unresponsive

Entity Framework: **A Very Powerful ORM**



Entity Framework Core makes us more productive!

Make sure to check out more of the courses in the library after completing this course.

Data Access through an API