Kostenloses eBook

LERNEN

Entity Framework

Free unaffiliated eBook created from
Stack Overflow contributors.

#entity-framework

Inhaltsverzeichnis

Über

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: entity-framework

It is an unofficial and free Entity Framework ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Entity Framework.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit Entity Framework

Bemerkungen

Entity Framework (EF) ist ein objektrelationaler Mapper (ORM), mit dem .NET-Entwickler mit relationalen Daten unter Verwendung domänenspezifischer Objekte arbeiten können. Dadurch entfällt der größte Teil des Datenzugriffscodes, den Entwickler normalerweise schreiben müssen.

Mit Entity Framework können Sie ein Modell erstellen, indem Sie Code schreiben oder Boxen und Linien in EF Designer verwenden. Beide Ansätze können verwendet werden, um auf eine vorhandene Datenbank zu zielen oder eine neue Datenbank zu erstellen.

Entity Framework ist der Haupt-ORM, den Microsoft für .NET Framework und die von Microsoft empfohlene Datenzugriffstechnologie bereitstellt.

Versionen

Ausführung	Veröffentlichungsdatum
1,0	2008-08-11
4,0	2010-04-12
4.1	2011-04-12
4.1 Update 1	2011-07-25
4.3.1	2012-02-29
5,0	2012-08-11
6,0	2013-10-17
6.1	2014-03-17
Kern 1.0	2016-06-27

Versionshinweise: https://msdn.microsoft.com/de/ca/data/jj574253.aspx

Examples

Verwenden von Entity Framework aus C # (Code First)

Mit dem Code können Sie zunächst Ihre Entitäten (Klassen) erstellen, ohne einen GUI-Designer oder eine .edmx-Datei zu verwenden. Er wird zunächst als *Code bezeichnet* , da Sie Ihre Modelle

zuerst erstellen können und das *Entity-Framework* die Datenbank automatisch entsprechend den Zuordnungen erstellt. Sie können diesen Ansatz auch mit der vorhandenen Datenbank verwenden, die als *Code zuerst mit der vorhandenen Datenbank bezeichnet* wird. Wenn Sie beispielsweise möchten, dass eine Tabelle eine Liste von Planeten enthält:

```
public class Planet
{
    public string Name { get; set; }
    public decimal AverageDistanceFromSun { get; set; }
}
```

Erstellen Sie nun Ihren Kontext, der die Brücke zwischen Ihren Entitätsklassen und der Datenbank darstellt. `DbSet<>` Sie eine oder mehrere Eigenschaften von `DbSet<>` :

```
using System.Data.Entity;

public class PlanetContext : DbContext
{
    public DbSet<Planet> Planets { get; set; }
}
```

Wir können dies verwenden, indem Sie Folgendes tun:

```
using(var context = new PlanetContext())
{
    var jupiter = new Planet
    {
        Name = "Jupiter",
        AverageDistanceFromSun = 778.5
    };

    context.Planets.Add(jupiter);
    context.SaveChanges();
}
```

In diesem Beispiel erstellen wir einen neuen `Planet` mit der `Name` Eigenschaft mit dem Wert `"Jupiter"` und der `AverageDistanceFromSun` Eigenschaft mit dem Wert `778.5`

Wir können diesen `Planet` mit der `Add()` Methode von `DbSet` dem Kontext hinzufügen und unsere Änderungen mit der `SaveChanges()` Methode in der Datenbank `SaveChanges()` .

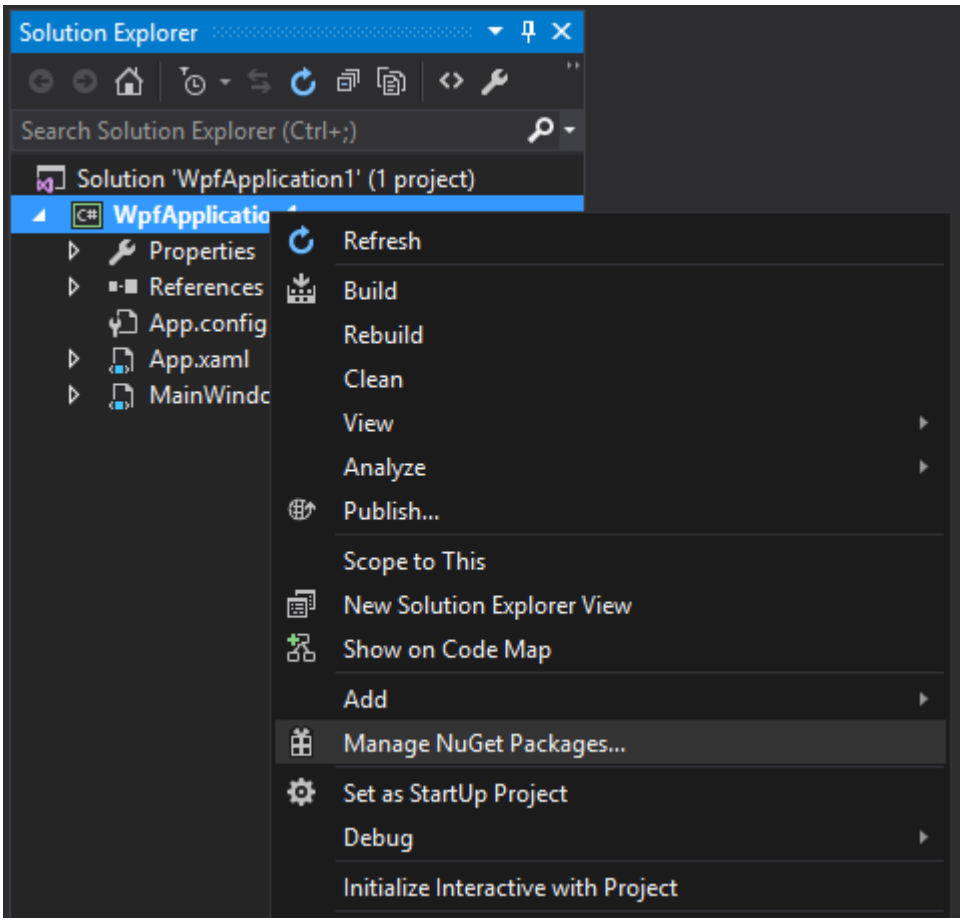
Oder wir können Zeilen aus der Datenbank abrufen:

```
using(var context = new PlanetContext())
{
    var jupiter = context.Planets.Single(p => p.Name == "Jupiter");
    Console.WriteLine($"Jupiter is {jupiter.AverageDistanceFromSun} million km from the
sun.");
}
```

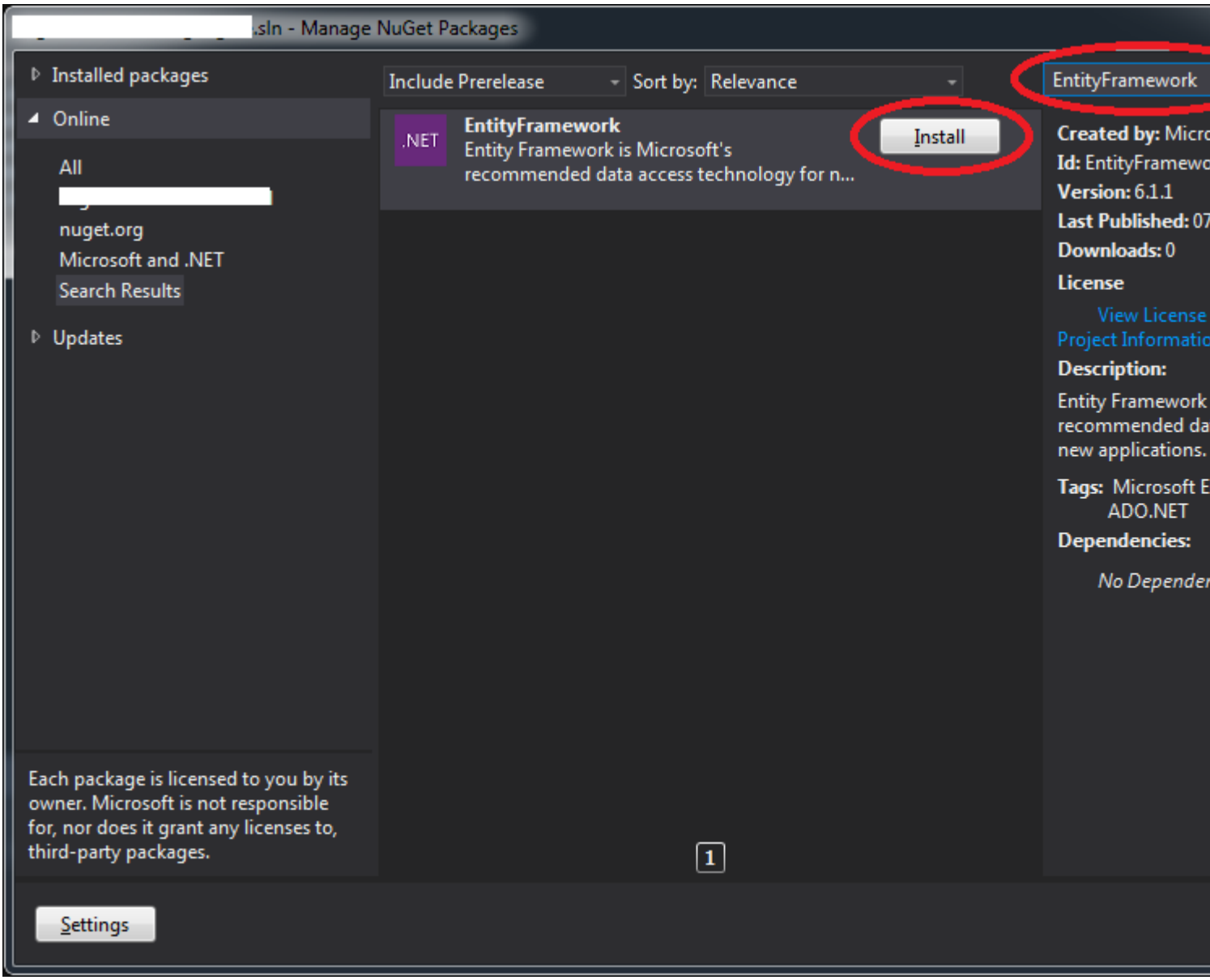
Das Entity Framework NuGet Package installieren

Öffnen Sie in Visual Studio das Projektmappen- **Explorerfenster** , `klicken` Sie mit der `rechten`

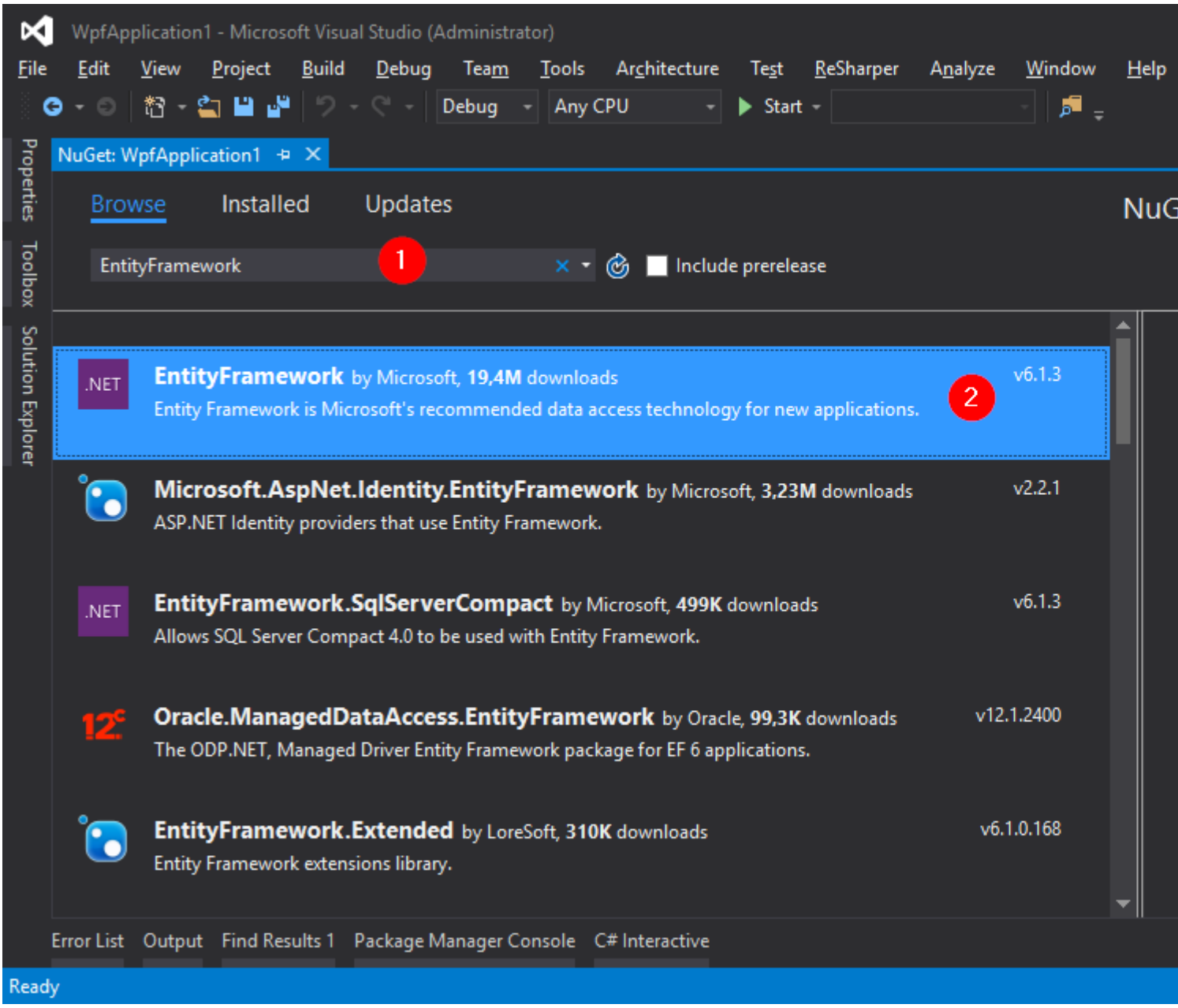
`Maustaste` auf Ihr Projekt, und wählen Sie im Menü die *Option NuGet-Pakete verwalten* :



In dem sich öffnenden Fenster geben Sie `EntityFramework` in das Suchfeld oben rechts ein.



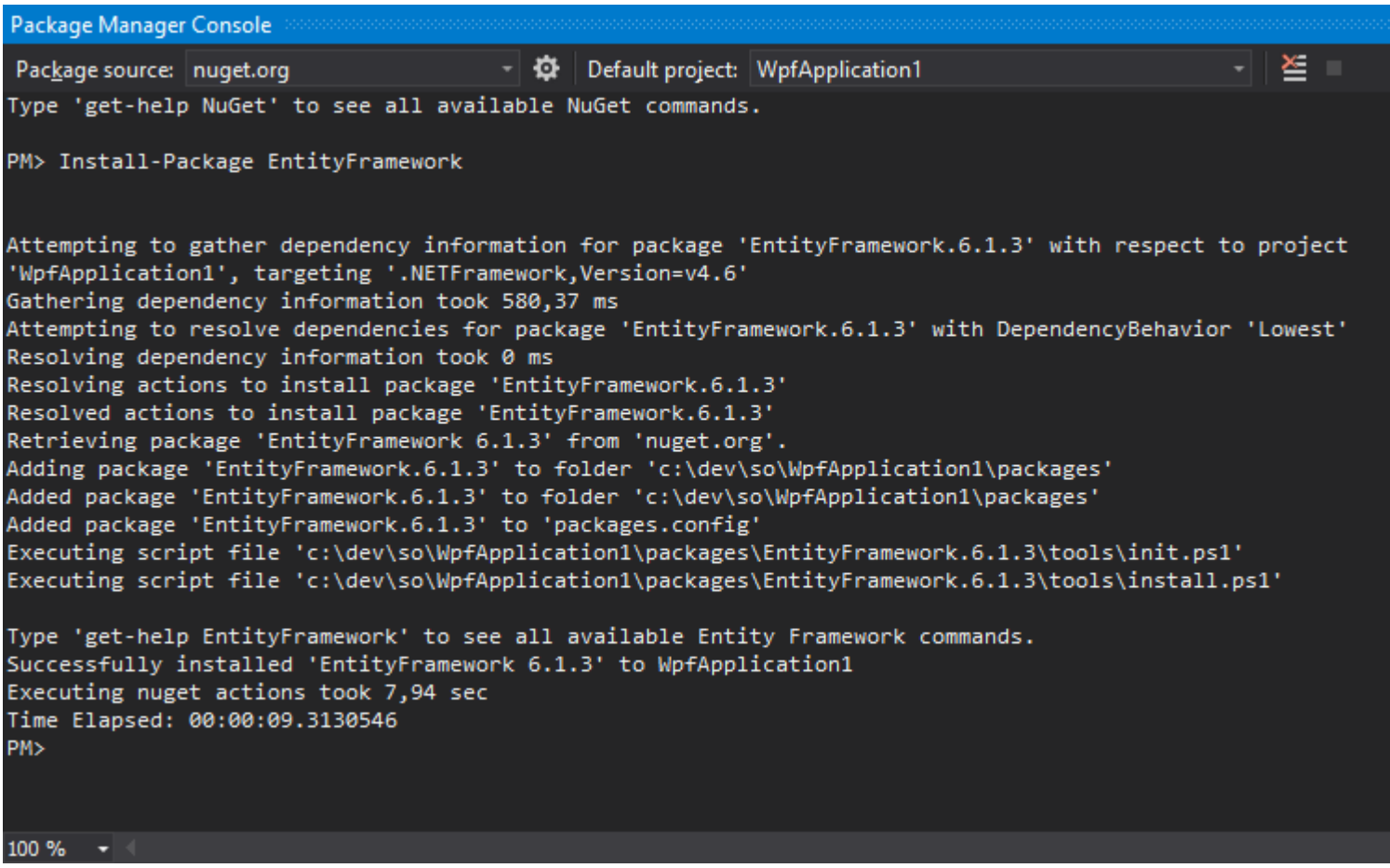
Wenn Sie Visual Studio 2015 verwenden, sehen Sie Folgendes:



Klicken Sie dann auf Installieren.

Wir können das Entity-Framework auch mithilfe der Paket-Manager-Konsole installieren. Dazu müssen Sie es zuerst über das *Menü Extras -> NuGet Package Manager -> Package Manager Console* öffnen und dann *Folgendes* eingeben:

```
Install-Package EntityFramework
```



Dadurch wird Entity Framework installiert und automatisch ein Verweis auf die Assembly in Ihrem Projekt hinzugefügt.

Was ist Entity Framework?

Das Schreiben und Verwalten von ADO.Net-Code für den Datenzugriff ist eine langwierige und monotone Arbeit. Microsoft hat ein **O / RM-Framework namens "Entity Framework"** bereitgestellt, um datenbankbezogene Aktivitäten für Ihre Anwendung zu automatisieren.

Das Entity Framework ist ein Object / Relational Mapping (O / RM) -Framework. Es ist eine Erweiterung von ADO.NET, die Entwicklern einen automatisierten Mechanismus für den Zugriff auf und das Speichern der Daten in der Datenbank bietet.

Was ist O / RM?

ORM ist ein Tool zum automatisierten Speichern von Daten aus Domänenobjekten in der relationalen Datenbank wie MS SQL Server, ohne viel Programmierung. O / RM besteht aus drei Hauptteilen:

1. Domänenklassenobjekte
2. Relationale Datenbankobjekte
3. Zuordnen von Informationen zur Zuordnung von Domänenobjekten zu relationalen Datenbankobjekten (z. **B.** Tabellen, Ansichten und gespeicherte Prozeduren)

Mit ORM können wir unser Datenbankdesign von unserem Design für Domänenklassen getrennt

halten. Dies macht die Anwendung wartungsfähig und erweiterbar. Es automatisiert außerdem den Standard-CRUD-Vorgang (Erstellen, Lesen, Aktualisieren und Löschen), sodass der Entwickler sie nicht manuell schreiben muss.

Erste Schritte mit Entity Framework online lesen: https://riptutorial.com/de/entity-framework/topic/815/erste-schritte-mit-entity-framework

Kapitel 2: .t4-Vorlagen im Entity-Framework

Examples

Schnittstellen dynamisch zum Modell hinzufügen

Wenn Sie mit einem vorhandenen Modell arbeiten, das recht groß ist und in Fällen, in denen eine Abstraktion erforderlich ist, häufig regeneriert wird, ist es möglicherweise kostspielig, das Modell mit Schnittstellen manuell zu überarbeiten. In solchen Fällen möchten Sie möglicherweise zur Modellgenerierung dynamisches Verhalten hinzufügen.

Das folgende Beispiel zeigt, wie automatisch Schnittstellen zu Klassen hinzugefügt werden, die bestimmte Spaltennamen haben:

In Ihrem Modell gehen `.tt` Datei ändern Sie die `EntityClassOpening` Methode in folgenden Weise wird dieser Add `IPolicyNumber` Schnittstelle auf Unternehmen , die haben `POLICY_NO` Spalt und `IUniqueId` auf `UNIQUE_ID`

```
public string EntityClassOpening(EntityType entity)
{
    var stringsToMatch = new Dictionary<string,string> { { "POLICY_NO", "IPolicyNumber" }, {
"UNIQUE_ID", "IUniqueId" } };
    return string.Format(
        CultureInfo.InvariantCulture,
        "{0} {1}partial class {2}{3}{4}",
        Accessibility.ForType(entity),
        _code.SpaceAfter(_code.AbstractOption(entity)),
        _code.Escape(entity),
        _code.StringBefore(" : ", _typeMapper.GetTypeName(entity.BaseType)),
        stringsToMatch.Any(o => entity.Properties.Any(n => n.Name == o.Key)) ? " : " +
string.Join(", ", stringsToMatch.Join(entity.Properties, l => l.Key, r => r.Name, (l,r) =>
l.Value)) : string.Empty);
}
```

Dies ist ein spezieller Fall, aber es zeigt die Möglichkeit, `.tt` Vorlagen ändern zu können.

XML-Dokumentation zu Entitätsklassen hinzufügen

In jeder generierten Modellklasse werden standardmäßig keine Dokumentationskommentare hinzugefügt. Wenn Sie XML-Dokumentationskommentare für alle generierten Entitätsklassen verwenden möchten, finden Sie diesen Teil in *[modelname]* .tt (*modelname* ist der aktuelle EDMX-Dateiname):

```
foreach (var entity in typeMapper.GetItemsToGenerate<EntityType>(itemCollection))
{
    fileManager.StartNewFile(entity.Name + ".cs");
    BeginNamespace(code); // used to write model namespace
#>
<#=codeStringGenerator.UsingDirectives(inHeader: false)#>
```

Sie können die XML-Dokumentationskommentare vor der Zeile `UsingDirectives` , wie im folgenden Beispiel gezeigt:

```
foreach (var entity in typeMapper.GetItemsToGenerate<EntityType>(itemCollection))
{
    fileManager.StartNewFile(entity.Name + ".cs");
    BeginNamespace(code);
#>
/// <summary>
/// <#=entity.Name#> model entity class.
/// </summary>
<#=codeStringGenerator.UsingDirectives(inHeader: false)#>
```

Der generierte Dokumentationskommentar sollte wie nachstehend angegeben den Namen der Entität enthalten.

```
/// <summary>
/// Example model entity class.
/// </summary>
public partial class Example
{
    // model contents
}
```

.t4-Vorlagen im Entity-Framework online lesen: https://riptutorial.com/de/entity-framework/topic/3964/-t4-vorlagen-im-entity-framework

Kapitel 3: Best Practices für das Entity Framework (einfach und professionell)

Einführung

Dieser Artikel soll eine einfache und professionelle Praxis für die Verwendung von Entity Framework einführen.

Einfach: weil es nur eine Klasse benötigt (mit einer Schnittstelle)

Professionell: weil es die Prinzipien der SOLID-Architektur anwendet

Ich möchte nicht mehr reden ... lasst es uns genießen!

Examples

1- Entity Framework @ Data Layer (Grundlagen)

In diesem Artikel verwenden wir eine einfache Datenbank namens "Company" mit zwei Tabellen:

[dbo]. [Kategorien] ([CategoryID], [CategoryName])

[dbo]. [Produkte] ([ProductID], [CategoryID], [ProductName])

1-1 Entity Framework-Code generieren

In dieser Ebene generieren wir den Entity Framework-Code (in der Projektbibliothek) (in diesem Artikel erfahren Sie, wie Sie dies tun können). Dann haben Sie die folgenden Klassen

```
public partial class CompanyContext : DbContext
public partial class Product
public partial class Category
```

1-2 Erstellen Sie eine grundlegende Schnittstelle

Wir werden eine Schnittstelle für unsere Basisfunktionen erstellen

```
public interface IDbRepository : IDisposable
{
    #region Tables and Views functions

    IQueryable<TResult> GetAll<TResult>(bool noTracking = true) where TResult : class;
    TEntity Add<TEntity>(TEntity entity) where TEntity : class;
    TEntity Delete<TEntity>(TEntity entity) where TEntity : class;
    TEntity Attach<TEntity>(TEntity entity) where TEntity : class;
    TEntity AttachIfNot<TEntity>(TEntity entity) where TEntity : class;

    #endregion Tables and Views functions
```

```
    #region Transactions Functions

    int Commit();
    Task<int> CommitAsync(CancellationToken cancellationToken = default(CancellationToken));

    #endregion Transactions Functions

    #region Database Procedures and Functions

    TResult Execute<TResult>(string functionName, params object[] parameters);

    #endregion Database Procedures and Functions
}
```

1-3 Basisschnittstelle implementieren

```
/// <summary>
/// Implementing basic tables, views, procedures, functions, and transaction functions
/// Select (GetAll), Insert (Add), Delete, and Attach
/// No Edit (Modify) function (can modify attached entity without function call)
/// Executes database procedures or functions (Execute)
/// Transaction functions (Commit)
/// More functions can be added if needed
/// </summary>
/// <typeparam name="TEntity">Entity Framework table or view</typeparam>
public class DbRepository : IDbRepository
{
    #region Protected Members

    protected DbContext _dbContext;

    #endregion Protected Members

    #region Constractors

    /// <summary>
    /// Repository constructor
    /// </summary>
    /// <param name="dbContext">Entity framework databse context</param>
    public DbRepository(DbContext dbContext)
    {
        _dbContext = dbContext;

        ConfigureContext();
    }

    #endregion Constractors

    #region IRepository Implementation

    #region Tables and Views functions

    /// <summary>
    /// Query all
    /// Set noTracking to true for selecting only (read-only queries)
    /// Set noTracking to false for insert, update, or delete after select
    /// </summary>
    public virtual IQueryable<TResult> GetAll<TResult>(bool noTracking = true) where TResult :
class
```

```
    {
        var entityDbSet = GetDbSet<TResult>();

        if (noTracking)
            return entityDbSet.AsNoTracking();

        return entityDbSet;
    }

    public virtual TEntity Add<TEntity>(TEntity entity) where TEntity : class
    {
        return GetDbSet<TEntity>().Add(entity);
    }

    /// <summary>
    /// Delete loaded (attached) or unloaded (Detached) entitiy
    /// No need to load object to delete it
    /// Create new object of TEntity and set the id then call Delete function
    /// </summary>
    /// <param name="entity">TEntity</param>
    /// <returns></returns>
    public virtual TEntity Delete<TEntity>(TEntity entity) where TEntity : class
    {
        if (_dbContext.Entry(entity).State == EntityState.Detached)
        {
            _dbContext.Entry(entity).State = EntityState.Deleted;
            return entity;
        }
        else
            return GetDbSet<TEntity>().Remove(entity);
    }

    public virtual TEntity Attach<TEntity>(TEntity entity) where TEntity : class
    {
        return GetDbSet<TEntity>().Attach(entity);
    }

    public virtual TEntity AttachIfNot<TEntity>(TEntity entity) where TEntity : class
    {
        if (_dbContext.Entry(entity).State == EntityState.Detached)
            return Attach(entity);

        return entity;
    }

    #endregion Tables and Views functions

    #region Transactions Functions

    /// <summary>
    /// Saves all changes made in this context to the underlying database.
    /// </summary>
    /// <returns>The number of objects written to the underlying database.</returns>
    public virtual int Commit()
    {
        return _dbContext.SaveChanges();
    }

    /// <summary>
    /// Asynchronously saves all changes made in this context to the underlying database.
    /// </summary>
```

```
    /// <param name="cancellationToken">A System.Threading.CancellationToken to observe while
waiting for the task to complete.</param>
    /// <returns>A task that represents the asynchronous save operation.  The task result
contains the number of objects written to the underlying database.</returns>
    public virtual Task<int> CommitAsync(CancellationToken cancellationToken =
default(CancellationToken))
    {
        return _dbContext.SaveChangesAsync(cancellationToken);
    }

    #endregion Transactions Functions

    #region Database Procedures and Functions

    /// <summary>
    /// Executes any function in the context
    /// use to call database procesdures and functions
    /// </summary>>
    /// <typeparam name="TResult">return function type</typeparam>
    /// <param name="functionName">context function name</param>
    /// <param name="parameters">context function parameters in same order</param>
    public virtual TResult Execute<TResult>(string functionName, params object[] parameters)
    {
        MethodInfo method = _dbContext.GetType().GetMethod(functionName);

        return (TResult)method.Invoke(_dbContext, parameters);
    }

    #endregion Database Procedures and Functions

    #endregion IRepository Implementation

    #region IDisposable Implementation

    public void Dispose()
    {
        _dbContext.Dispose();
    }

    #endregion IDisposable Implementation

    #region Protected Functions

    /// <summary>
    /// Set Context Configuration
    /// </summary>
    protected virtual void ConfigureContext()
    {
        // set your recommended Context Configuration
        _dbContext.Configuration.LazyLoadingEnabled = false;
    }

    #endregion Protected Functions

    #region Private Functions

    private DbSet<TEntity> GetDbSet<TEntity>() where TEntity : class
    {
        return _dbContext.Set<TEntity>();
    }
```

```
    #endregion Private Functions

}
```

2- Entity Framework @ Business-Schicht

In dieser Schicht schreiben wir das Anwendungsgeschäft.

Es wird empfohlen, für jeden Präsentationsbildschirm die Geschäftsschnittstelle und die Implementierungsklasse zu erstellen, die alle für den Bildschirm erforderlichen Funktionen enthalten.

Nachfolgend schreiben wir das Geschäft für den Produktbildschirm als Beispiel

```
/// <summary>
/// Contains Product Business functions
/// </summary>
public interface IProductBusiness
{
    Product SelectById(int productId, bool noTracking = true);
    Task<IEnumerable<dynamic>> SelectByCategoryAsync(int CategoryId);
    Task<Product> InsertAsync(string productName, int categoryId);
    Product InsertForNewCategory(string productName, string categoryName);
    Product Update(int productId, string productName, int categoryId);
    Product Update2(int productId, string productName, int categoryId);
    int DeleteWithoutLoad(int productId);
    int DeleteLoadedProduct(Product product);
    IEnumerable<GetProductsCategory_Result> GetProductsCategory(int categoryId);
}



/// <summary>
/// Implementing Product Business functions
/// </summary>
public class ProductBusiness : IProductBusiness
{
    #region Private Members

    private IDbRepository _dbRepository;

    #endregion Private Members

    #region Constructors

    /// <summary>
    /// Product Business Constructor
    /// </summary>
    /// <param name="dbRepository"></param>
    public ProductBusiness(IDbRepository dbRepository)
    {
        _dbRepository = dbRepository;
    }

    #endregion Constructors

    #region IProductBusiness Function
```

```
    /// <summary>
    /// Selects Product By Id
    /// </summary>
    public Product SelectById(int productId, bool noTracking = true)
    {
        var products = _dbRepository.GetAll<Product>(noTracking);

        return products.FirstOrDefault(pro => pro.ProductID == productId);
    }

    /// <summary>
    /// Selects Products By Category Id Async
    /// To have async method, add reference to EntityFramework 6 dll or higher
    /// also you need to have the namespace "System.Data.Entity"
    /// </summary>
    /// <param name="CategoryId">CategoryId</param>
    /// <returns>Return what ever the object that you want to return</returns>
    public async Task<IEnumerable<dynamic>> SelectByCategoryAsync(int CategoryId)
    {
        var products = _dbRepository.GetAll<Product>();
        var categories = _dbRepository.GetAll<Category>();

        var result = (from pro in products
                      join cat in categories
                      on pro.CategoryID equals cat.CategoryID
                      where pro.CategoryID == CategoryId
                      select new
                      {
                          ProductId = pro.ProductID,
                          ProductName = pro.ProductName,
                          CategoryName = cat.CategoryName
                      }
                              );

        return await result.ToListAsync();
    }

    /// <summary>
    /// Insert Async new product for given category
    /// </summary>
    public async Task<Product> InsertAsync(string productName, int categoryId)
    {
        var newProduct = _dbRepository.Add(new Product() { ProductName = productName,
CategoryID = categoryId });

        await _dbRepository.CommitAsync();

        return newProduct;
    }

    /// <summary>
    /// Insert new product and new category
    /// Do many database actions in one transaction
    /// each _dbRepository.Commit(); will commit one transaction
    /// </summary>
    public Product InsertForNewCategory(string productName, string categoryName)
    {
        var newCategory = _dbRepository.Add(new Category() { CategoryName = categoryName });
        var newProduct = _dbRepository.Add(new Product() { ProductName = productName, Category
= newCategory });
```

```
        _dbRepository.Commit();

        return newProduct;
    }

    /// <summary>
    /// Update given product with tracking
    /// </summary>
    public Product Update(int productId, string productName, int categoryId)
    {
        var product = SelectById(productId,false);
        product.CategoryID = categoryId;
        product.ProductName = productName;

        _dbRepository.Commit();

        return product;
    }

    /// <summary>
    /// Update given product with no tracking and attach function
    /// </summary>
    public Product Update2(int productId, string productName, int categoryId)
    {
        var product = SelectById(productId);
        _dbRepository.Attach(product);

        product.CategoryID = categoryId;
        product.ProductName = productName;

        _dbRepository.Commit();

        return product;
    }

    /// <summary>
    /// Deletes product without loading it
    /// </summary>
    public int DeleteWithoutLoad(int productId)
    {
        _dbRepository.Delete(new Product() { ProductID = productId });

        return _dbRepository.Commit();
    }

    /// <summary>
    /// Deletes product after loading it
    /// </summary>
    public int DeleteLoadedProduct(Product product)
    {
        _dbRepository.Delete(product);

        return _dbRepository.Commit();
    }

    /// <summary>
    /// Assuming we have the following procedure in database
    /// PROCEDURE [dbo].[GetProductsCategory] @CategoryID INT, @OrderBy VARCHAR(50)
    /// </summary>
    public IEnumerable<GetProductsCategory_Result> GetProductsCategory(int categoryId)
    {
```

```
        return
_dbRepository.Execute<IEnumerable<GetProductsCategory_Result>>("GetProductsCategory",
categoryId, "ProductName DESC");
    }


    #endregion IProductBusiness Function
}
```

3- Verwenden der Business-Schicht @ Präsentationsschicht (MVC)

In diesem Beispiel verwenden wir die Business-Schicht in der Präsentationsschicht. Wir verwenden MVC als Beispiel für die Präsentationsschicht (Sie können jedoch auch jede andere Präsentationsschicht verwenden).

Wir müssen zuerst die IoC registrieren (wir werden Unity verwenden, aber Sie können jede IoC verwenden) und schreiben dann unsere Präsentationsebene

3-1 Registrieren Sie Unity-Typen in MVC

3-1-1 Fügen Sie den NuGet-Backage „Unity-Bootstrapper für ASP.NET MVC“ hinzu

3-1-2 UnityWebActivator.Start () hinzufügen; in der Datei "Global.asax.cs" (Funktion Application_Start ())

3-1-3 Ändern Sie UnityConfig.RegisterTypes wie folgt

```
    public static void RegisterTypes(IUnityContainer container)
    {
        // Data Access Layer
        container.RegisterType<DbContext, CompanyContext>(new PerThreadLifetimeManager());
        container.RegisterType(typeof(IDbRepository), typeof(DbRepository), new
PerThreadLifetimeManager());

        // Business Layer
        container.RegisterType<IProductBusiness, ProductBusiness>(new
PerThreadLifetimeManager());

    }
```

3-2 Verwenden der Business-Schicht @ Präsentationsschicht (MVC)

```
public class ProductController : Controller
{
    #region Private Members

    IProductBusiness _productBusiness;

    #endregion Private Members

    #region Constractors

    public ProductController(IProductBusiness productBusiness)
    {
        _productBusiness = productBusiness;
```

```
    }

    #endregion Constractors

    #region Action Functions

    [HttpPost]
    public ActionResult InsertForNewCategory(string productName, string categoryName)
    {
        try
        {
            // you can use any of IProductBusiness functions
            var newProduct = _productBusiness.InsertForNewCategory(productName, categoryName);

            return Json(new { success = true, data = newProduct });
        }
        catch (Exception ex)
        {   /* log ex*/
            return Json(new { success = false, errorMessage = ex.Message});
        }
    }

    [HttpDelete]
    public ActionResult SmartDeleteWithoutLoad(int productId)
    {
        try
        {
            // deletes product without load
            var deletedProduct = _productBusiness.DeleteWithoutLoad(productId);

            return Json(new { success = true, data = deletedProduct });
        }
        catch (Exception ex)
        {   /* log ex*/
            return Json(new { success = false, errorMessage = ex.Message });
        }
    }

    public async Task<ActionResult> SelectByCategoryAsync(int CategoryId)
    {
        try
        {
            var results = await _productBusiness.SelectByCategoryAsync(CategoryId);

            return Json(new { success = true, data = results },JsonRequestBehavior.AllowGet);
        }
        catch (Exception ex)
        {   /* log ex*/
            return Json(new { success = false, errorMessage = ex.Message
},JsonRequestBehavior.AllowGet);
        }
    }
    #endregion Action Functions
}
```

4- Entity Framework @ Unit Test Layer

In der Einheitentestebene testen wir normalerweise die Business Layer-Funktionen. Dazu entfernen wir die Abhängigkeiten von Data Layer (Entity Framework).

Nun stellt sich die Frage: Wie kann ich die Entity Framework-Abhängigkeiten entfernen, um die Business Layer-Funktionen zu testen?

Und die Antwort ist einfach: Wir werden eine gefälschte Implementierung für die IDbRepository-Schnittstelle durchführen und dann unseren Komponententest durchführen

4-1 Implementieren einer Basisschnittstelle (gefälschte Implementierung)

```
class FakeDbRepository : IDbRepository
{
    #region Protected Members

    protected Hashtable _dbContext;
    protected int _numberOfRowsAffected;
    protected Hashtable _contextFunctionsResults;

    #endregion Protected Members

    #region Constractors

    public FakeDbRepository(Hashtable contextFunctionsResults = null)
    {
        _dbContext = new Hashtable();
        _numberOfRowsAffected = 0;
        _contextFunctionsResults = contextFunctionsResults;
    }

    #endregion Constractors

    #region IRepository Implementation

    #region Tables and Views functions

    public IQueryable<TResult> GetAll<TResult>(bool noTracking = true) where TResult : class
    {
        return GetDbSet<TResult>().AsQueryable();
    }

    public TEntity Add<TEntity>(TEntity entity) where TEntity : class
    {
        GetDbSet<TEntity>().Add(entity);
        ++_numberOfRowsAffected;
        return entity;
    }

    public TEntity Delete<TEntity>(TEntity entity) where TEntity : class
    {
        GetDbSet<TEntity>().Remove(entity);
        ++_numberOfRowsAffected;
        return entity;
    }

    public TEntity Attach<TEntity>(TEntity entity) where TEntity : class
    {
        return Add(entity);
    }

    public TEntity AttachIfNot<TEntity>(TEntity entity) where TEntity : class
    {
```

```
        if (!GetDbSet<TEntity>().Contains(entity))
            return Attach(entity);

        return entity;
    }

    #endregion Tables and Views functions

    #region Transactions Functions


    public virtual int Commit()
    {
        var numberOfRowsAffected = _numberOfRowsAffected;
        _numberOfRowsAffected = 0;
        return numberOfRowsAffected;
    }

    public virtual Task<int> CommitAsync(CancellationToken cancellationToken =
default(CancellationToken))
    {
        var numberOfRowsAffected = _numberOfRowsAffected;
        _numberOfRowsAffected = 0;
        return new Task<int>(() => numberOfRowsAffected);
    }

    #endregion Transactions Functions

    #region Database Procedures and Functions

    public virtual TResult Execute<TResult>(string functionName, params object[] parameters)
    {
        if (_contextFunctionsResults != null &&
_contextFunctionsResults.Contains(functionName))
            return (TResult)_contextFunctionsResults[functionName];

        throw new NotImplementedException();
    }

    #endregion Database Procedures and Functions

    #endregion IRepository Implementation

    #region IDisposable Implementation

    public void Dispose()
    {

    }

    #endregion IDisposable Implementation

    #region Private Functions

    private List<TEntity> GetDbSet<TEntity>() where TEntity : class
    {
        if (!_dbContext.Contains(typeof(TEntity)))
            _dbContext.Add(typeof(TEntity), new List<TEntity>());

        return (List<TEntity>)_dbContext[typeof(TEntity)];
    }
```

```
    #endregion Private Functions
}
```

4-2 Führen Sie den Gerätetest durch

```
[TestClass]
public class ProductUnitTest
{
    [TestMethod]
    public void TestInsertForNewCategory()
    {
        // Initialize repositories
        FakeDbRepository _dbRepository = new FakeDbRepository();

        // Initialize Business object
        IProductBusiness productBusiness = new ProductBusiness(_dbRepository);

        // Process test method
        productBusiness.InsertForNewCategory("Test Product", "Test Category");

        int _productCount = _dbRepository.GetAll<Product>().Count();
        int _categoryCount = _dbRepository.GetAll<Category>().Count();

        Assert.AreEqual<int>(1, _productCount);
        Assert.AreEqual<int>(1, _categoryCount);
    }

    [TestMethod]
    public void TestProceduresFunctionsCall()
    {
        // Initialize Procedures / Functions result
        Hashtable _contextFunctionsResults = new Hashtable();
        _contextFunctionsResults.Add("GetProductsCategory", new
List<GetProductsCategory_Result> {
            new GetProductsCategory_Result() { ProductName = "Product 1", ProductID = 1,
CategoryName = "Category 1" },
            new GetProductsCategory_Result() { ProductName = "Product 2", ProductID = 2,
CategoryName = "Category 1" },
            new GetProductsCategory_Result() { ProductName = "Product 3", ProductID = 3,
CategoryName = "Category 1" }});

        // Initialize repositories
        FakeDbRepository _dbRepository = new FakeDbRepository(_contextFunctionsResults);

        // Initialize Business object
        IProductBusiness productBusiness = new ProductBusiness(_dbRepository);

        // Process test method
        var results = productBusiness.GetProductsCategory(1);

        Assert.AreEqual<int>(3, results.Count());
    }
}
```

Best Practices für das Entity Framework (einfach und professionell) online lesen: https://riptutorial.com/de/entity-framework/topic/8879/best-practices-fur-das-entity-framework--einfach-und-professionell-

Kapitel 4: Code First - Fließende API

Bemerkungen

Es gibt zwei generelle Möglichkeiten, anzugeben, wie Entity Framework POCO-Klassen Datenbanktabellen, Spalten usw. zuordnet: **Datenanmerkungen** und **Fluent-API** .

Während Datenanmerkungen einfach zu lesen und zu verstehen sind, fehlen ihnen bestimmte Funktionen, beispielsweise das Festlegen des Verhaltens "Kaskadieren beim Löschen" für eine Entität. Die Fluent-API ist dagegen etwas komplexer zu bedienen, bietet jedoch weitaus mehr Funktionen.

Examples

Modelle zuordnen

Mit der EntityFramewok Fluent-API können Sie Ihre **Code-First-** Domänenmodelle auf eine zugrunde liegende Datenbank übertragen. Dies kann auch mit *Code-First mit der vorhandenen Datenbank verwendet werden* . Bei der Verwendung der *Fluent-API* haben Sie zwei Möglichkeiten: Sie können Ihre Modelle direkt der *OnModelCreating*- Methode *zuordnen* oder Sie können Mapper-Klassen erstellen, die von *EntityTypeConfiguration erben,* und diese Modelle *dann* in die *modelBuilder*- Methode *OnModelCreating einfügen* . *Die zweite* Option ist die, die ich vorziehen möchte, und werde ein Beispiel davon zeigen.

Schritt eins: Modell erstellen.

```
public class Employee
{
    public int Id { get; set; }
    public string Surname { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public short Age { get; set; }
    public decimal MonthlySalary { get; set; }

    public string FullName
    {
        get
        {
            return $"{Surname} {FirstName} {LastName}";
        }
    }
}
```

Schritt 2: Erstellen Sie eine Mapper-Klasse

```
public class EmployeeMap
```

```
    : EntityTypeConfiguration<Employee>
{
    public EmployeeMap()
    {
        // Primary key
        this.HasKey(m => m.Id);

        this.Property(m => m.Id)
            .HasColumnType("int")
            .HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity);

        // Properties
        this.Property(m => m.Surname)
            .HasMaxLength(50);

        this.Property(m => m.FirstName)
            .IsRequired()
            .HasMaxLength(50);

        this.Property(m => m.LastName)
            .HasMaxLength(50);

        this.Property(m => m.Age)
            .HasColumnType("smallint");

        this.Property(m => m.MonthlySalary)
            .HasColumnType("number")
            .HasPrecision(14, 5);

        this.Ignore(m => m.FullName);

        // Table & column mappings
        this.ToTable("TABLE_NAME", "SCHEMA_NAME");
        this.Property(m => m.Id).HasColumnName("ID");
        this.Property(m => m.Surname).HasColumnName("SURNAME");
        this.Property(m => m.FirstName).HasColumnName("FIRST_NAME");
        this.Property(m => m.LastName).HasColumnName("LAST_NAME");
        this.Property(m => m.Age).HasColumnName("AGE");
        this.Property(m => m.MonthlySalary).HasColumnName("MONTHLY_SALARY");
    }
}
```

Lassen Sie uns Mappings erklären:

- **HasKey** - **Definiert** den Primärschlüssel. *Zusammengesetzte Primärschlüssel* können ebenfalls verwendet werden. Zum Beispiel: *this.HasKey (m => new {m.DepartmentId, m.PositionId})* .
- **Property** - ermöglicht das Konfigurieren von Modelleigenschaften.
- **HasColumnType** - Angabe des **Spaltentyps auf** Datenbankebene. Bitte beachten Sie, dass es für verschiedene Datenbanken wie Oracle und MS SQL unterschiedlich sein kann.
- **HasDatabaseGeneratedOption** - **Gibt** an, ob die Eigenschaft auf Datenbankebene berechnet wird. Numerische PKs sind standardmäßig *DatabaseGeneratedOption.Identity* . Sie sollten *DatabaseGeneratedOption.None* angeben, wenn dies nicht der Fall sein soll.
- **HasMaxLength** - begrenzt die Länge der Zeichenfolge.
- **IsRequired** - kennzeichnet die Eigenschaft als erforderlich.
- **HasPrecision** - lässt uns die Genauigkeit für Dezimalzahlen angeben.

- **Ignorieren** - Ignoriert die Eigenschaft vollständig und ordnet sie nicht der Datenbank zu. Wir haben FullName ignoriert, da wir diese Spalte nicht in unserer Tabelle haben möchten.
- **ToTable** - **Geben Sie den** Tabellennamen und den Schemanamen (optional) für das Modell an.
- **HasColumnName** - beziehen sich auf den Spaltennamen. Dies ist nicht erforderlich, wenn Eigenschaftsnamen und Spaltennamen identisch sind.

Schritt 3: Mapping-Klasse zu Konfigurationen hinzufügen.

Wir müssen EntityFramework anweisen, unsere Mapper-Klasse zu verwenden. Dazu müssen wir es zu *modelBuilder.Configurations* auf der *OnModelCreating*- Methode *hinzufügen* :

```
public class DbContext()
    : base("Name=DbContext")
{
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Configurations.Add(new EmployeeMap());
    }
}
```

Und das ist alles. Wir sind alle bereit zu gehen.

Primärschlüssel

Mit der Methode .HasKey () kann eine Eigenschaft explizit als Primärschlüssel der Entität konfiguriert werden.

```
using System.Data.Entity;
// ..

public class PersonContext : DbContext
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..

        modelBuilder.Entity<Person>().HasKey(p => p.PersonKey);
    }
}
```

Zusammengesetzter Primärschlüssel

Mit der .HasKey () - Methode kann ein Satz von Eigenschaften explizit als zusammengesetzter Primärschlüssel der Entität konfiguriert werden.

```
using System.Data.Entity;
// ..
```

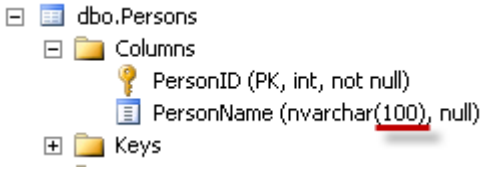
```
public class PersonContext : DbContext
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..

        modelBuilder.Entity<Person>().HasKey(p => new { p.FirstName, p.LastName });
    }
}
```

Maximale Länge

Mit der .HasMaxLength () -Methode kann die maximale Zeichenanzahl für eine Eigenschaft konfiguriert werden.

```
using System.Data.Entity;
// ..

public class PersonContext : DbContext
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..

        modelBuilder.Entity<Person>()
                    .Property(t => t.Name)
                    .HasMaxLength(100);
    }
}
```

Die resultierende Spalte mit der angegebenen Spaltenlänge:



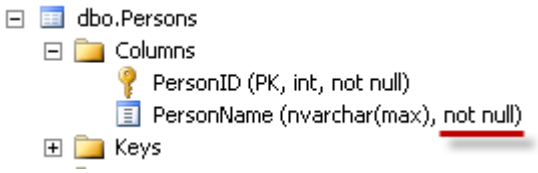
Erforderliche Eigenschaften (NOT NULL)

Mithilfe der .IsRequired () - Methode können Eigenschaften als obligatorisch angegeben werden. Dies bedeutet, dass die Spalte eine NOT NULL-Einschränkung aufweist.

```
using System.Data.Entity;
// ..

public class PersonContext : DbContext
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
```

```
        // ..

        modelBuilder.Entity<Person>()
                    .Property(t => t.Name)
                    .IsRequired();
    }
}
```

Die resultierende Spalte mit der NOT NULL-Einschränkung:



Explizite Fremdschlüsselbenennung

Wenn eine Navigationseigenschaft in einem Modell vorhanden ist, erstellt Entity Framework automatisch eine Fremdschlüsselspalte. Wenn ein bestimmter Fremdschlüsselname gewünscht wird, aber nicht als Eigenschaft im Modell enthalten ist, kann er explizit mithilfe der Fluent-API festgelegt werden. Durch Verwendung der `Map` Methode beim Herstellen der Fremdschlüsselbeziehung kann für Fremdschlüssel ein eindeutiger Name verwendet werden.

```
public class Company
{
    public int Id { get; set; }
}

public class Employee
{
    property int Id { get; set; }
    property Company Employer { get; set; }
}

public class EmployeeContext : DbContext
{
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Employee>()
                    .HasRequired(x => x.Employer)
                    .WithRequiredDependent()
                    .Map(m => m.MapKey("EmployerId"));
    }
}
```

Nach dem Festlegen der Beziehung ermöglicht die `Map` Methode, dass der Name des `MapKey` durch Ausführen von `MapKey` explizit festgelegt wird. In diesem Beispiel lautet die EmployerId jetzt, was zu einem Spaltennamen von Employer_Id geführt hätte.

Code First - Fließende API online lesen: https://riptutorial.com/de/entity-framework/topic/4530/code-first---flie-ende-api

Kapitel 5: Datenbankinitialisierer

Examples

CreateDatabaseIfNotExists

Implementierung von `IDatabaseInitializer` , die standardmäßig in EntityFramework verwendet wird. Wie der Name schon sagt, erstellt er die Datenbank, wenn keine vorhanden ist. Wenn Sie jedoch das Modell ändern, wird eine Ausnahme ausgelöst.

Verwendungszweck:

```
public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(new CreateDatabaseIfNotExists<MyContext>());
    }
}
```

DropCreateDatabaseIfModelChanges

Diese Implementierung von `IDatabaseInitializer` die Datenbank und erstellt sie neu, wenn das Modell automatisch geändert wird.

Verwendungszweck:

```
public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(new DropCreateDatabaseIfModelChanges<MyContext>());
    }
}
```

DropCreateDatabaseAlways

Bei dieser Implementierung von `IDatabaseInitializer` die Datenbank jedes Mal `IDatabaseInitializer` und neu erstellt, wenn Ihr Kontext in der Anwendungs-App-Domäne verwendet wird. Beachten Sie den Datenverlust aufgrund der Tatsache, dass die Datenbank neu erstellt wird.

Verwendungszweck:

```
public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(new DropCreateDatabaseAlways<MyContext>());
    }
}
```

Benutzerdefinierter Datenbankinitialisierer

Sie können Ihre eigene Implementierung von `IDatabaseInitializer` .

Beispielimplementierung eines Initialisierers, der die Datenbank auf 0 migriert und dann bis zur neuesten Migration (z. B. beim Ausführen von Integrationstests) vollständig migriert Dazu benötigen Sie auch einen `DbMigrationsConfiguration` Typ.

```
public class RecreateFromScratch<TContext, TMigrationsConfiguration> :
IDatabaseInitializer<TContext>
where TContext : DbContext
where TMigrationsConfiguration : DbMigrationsConfiguration<TContext>, new()
{
    private readonly DbMigrationsConfiguration<TContext> _configuration;

    public RecreateFromScratch()
    {
        _configuration = new TMigrationsConfiguration();
    }

    public void InitializeDatabase(TContext context)
    {
        var migrator = new DbMigrator(_configuration);
        migrator.Update("0");
        migrator.Update();
    }
}
```

MigrateDatabaseToLatestVersion

Eine Implementierung von `IDatabaseInitializer` , die Code First Migrations zum Aktualisieren der Datenbank auf die neueste Version verwendet. Um diesen Initialisierer verwenden zu können, müssen Sie auch den Typ `DbMigrationsConfiguration` verwenden.

Verwendungszweck:

```
public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(
            new MigrateDatabaseToLatestVersion<MyContext, Configuration>());
    }
}
```

Datenbankinitialisierer online lesen: https://riptutorial.com/de/entity-framework/topic/5526/datenbankinitialisierer

Kapitel 6: Entitätsstatus verwalten

Bemerkungen

Entitäten in Entity Framework können verschiedene Status haben, die in der System.Data.Entity.EntityState- Enumeration aufgelistet sind. Diese Zustände sind:

```
Added
Deleted
Detached
Modified
Unchanged
```

Entity Framework arbeitet mit POCOs. Dies bedeutet, dass Entitäten einfache Klassen sind, die keine Eigenschaften und Methoden zum Verwalten ihres eigenen Status haben. Der Entitätsstatus wird im `ObjectStateManager` von einem Kontext selbst verwaltet.

In diesem Thema werden verschiedene Möglichkeiten zum Festlegen des Entitätsstatus beschrieben.

Examples

Einstellungsstatus Hinzugefügt von einer einzelnen Entität

`EntityState.Added` kann auf zwei völlig gleichwertige Arten festgelegt werden:

1. Durch Festlegen des Status des Eintrags im Kontext:

   ```
   context.Entry(entity).State = EntityState.Added;
   ```

2. Durch Hinzufügen zu einem `DbSet` des Kontextes:

   ```
   context.Entities.Add(entity);
   ```

Beim Aufruf von `SaveChanges` wird die Entität in die Datenbank eingefügt. Wenn es eine Identitätsspalte (einen automatisch festgelegten, automatisch inkrementierenden Primärschlüssel) `SaveChanges` , enthält die Primärschlüsseleigenschaft der Entität nach `SaveChanges` den neu generierten Wert, *auch wenn diese Eigenschaft bereits einen Wert hatte* .

Einstellungsstatus Hinzugefügt eines Objektdiagramms

Das Festlegen des Zustands eines *Objektdiagramms* (einer Sammlung verwandter Entitäten) auf `Added` unterscheidet sich vom Festlegen einer einzelnen Entität als `Added` (siehe dieses Beispiel).

Im Beispiel speichern wir Planeten und ihre Monde:

Klassenmodell

```
public class Planet
{
    public Planet()
    {
        Moons = new HashSet<Moon>();
    }
    public int ID { get; set; }
    public string Name { get; set; }
    public ICollection<Moon> Moons { get; set; }
}

public class Moon
{
    public int ID { get; set; }
    public int PlanetID { get; set; }
    public string Name { get; set; }
}
```

Kontext

```
public class PlanetDb : DbContext
{
    public property DbSet<Planet> Planets { get; set; }
}
```

Wir verwenden eine Instanz dieses Kontexts, um Planeten und ihre Monde hinzuzufügen:

Beispiel

```
var mars = new Planet { Name = "Mars" };
mars.Moons.Add(new Moon { Name = "Phobos" });
mars.Moons.Add(new Moon { Name = "Deimos" });

context.Planets.Add(mars);

Console.WriteLine(context.Entry(mars).State);
Console.WriteLine(context.Entry(mars.Moons.First()).State);
```

Ausgabe:

```
Added
Added
```

Was wir hier sehen, ist, dass das Hinzufügen eines `Planet` auch den Status eines Mondes auf `Added` .

Wenn ein Unternehmen des Staates zu Einstellung `Added` , alle Einheiten in ihren Navigationseigenschaften (Eigenschaften , dass „navigieren“ zu anderen Einrichtungen, wie `Planet.Moons`) werden auch als markiert `Added` , *soweit sie* nicht *bereits mit dem Kontext verbunden sind.*

Entitätsstatus verwalten online lesen: https://riptutorial.com/de/entity-framework/topic/5256/entitatsstatus-verwalten

Kapitel 7: Entity Framework mit SQLite

Einführung

SQLite ist eine eigenständige, serverlose, transaktionale SQL-Datenbank. Es kann in einer .NET-Anwendung verwendet werden, indem sowohl eine frei verfügbare .NET SQLite-Bibliothek als auch ein Entity Framework SQLite-Anbieter verwendet wird. Dieses Thema befasst sich mit der Einrichtung und Verwendung des Entity Framework SQLite-Anbieters.

Examples

Einrichten eines Projekts zur Verwendung von Entity Framework mit einem SQLite-Anbieter

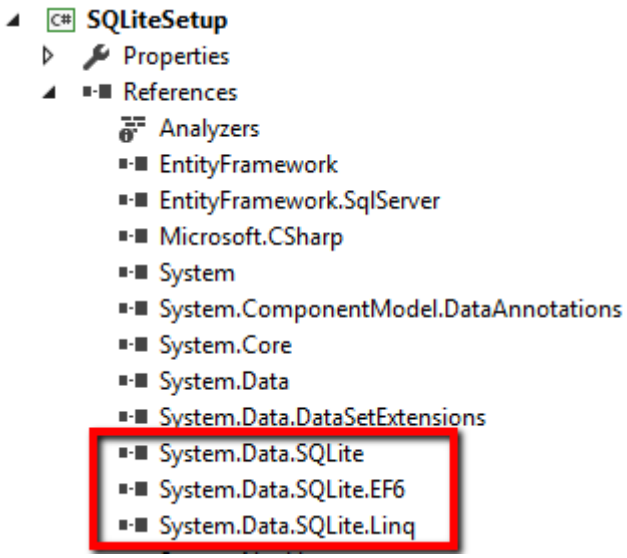
Die Entity Framework-Bibliothek wird nur mit einem SQL Server-Anbieter geliefert. Für die Verwendung von SQLite sind zusätzliche Abhängigkeiten und Konfigurationen erforderlich. Alle erforderlichen Abhängigkeiten sind in NuGet verfügbar.

Installieren Sie verwaltete SQLite-Bibliotheken

Alle manipulierten Versionen können mithilfe der NuGet Package Manager Console installiert werden. Führen Sie den Befehl `Install-Package System.Data.SQLite` .

```
PM> Install-Package System.Data.SQLite
Attempting to gather dependency information for package 'System.Data.SQLite.1.0.104' with respect to proje
Attempting to resolve dependencies for package 'System.Data.SQLite.1.0.104' with DependencyBehavior 'Lowes
Resolving actions to install package 'System.Data.SQLite.1.0.104'
Resolved actions to install package 'System.Data.SQLite.1.0.104'
Adding package 'EntityFramework.6.0.0' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Projec
Added package 'EntityFramework.6.0.0' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Project
Added package 'EntityFramework.6.0.0' to 'packages.config'
Executing script file 'c:\users\jason.tyler\documents\visual studio 2015\Projects\SQLiteSetup\packages\Ent
Executing script file 'c:\users\jason.tyler\documents\visual studio 2015\Projects\SQLiteSetup\packages\Ent

Type 'get-help EntityFramework' to see all available Entity Framework commands.
Successfully installed 'EntityFramework 6.0.0' to SQLiteSetup
Adding package 'System.Data.SQLite.Core.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2
Added package 'System.Data.SQLite.Core.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 20
Added package 'System.Data.SQLite.Core.1.0.104' to 'packages.config'
Successfully installed 'System.Data.SQLite.Core 1.0.104' to SQLiteSetup
Adding package 'System.Data.SQLite.EF6.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 20
Added package 'System.Data.SQLite.EF6.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 201
Added package 'System.Data.SQLite.EF6.1.0.104' to 'packages.config'
Executing script file 'c:\users\jason.tyler\documents\visual studio 2015\Projects\SQLiteSetup\packages\Sys
Successfully installed 'System.Data.SQLite.EF6 1.0.104' to SQLiteSetup
Adding package 'System.Data.SQLite.Linq.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2
Added package 'System.Data.SQLite.Linq.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 20
Added package 'System.Data.SQLite.Linq.1.0.104' to 'packages.config'
Successfully installed 'System.Data.SQLite.Linq 1.0.104' to SQLiteSetup
Adding package 'System.Data.SQLite.1.0.104', which only has dependencies, to project 'SQLiteSetup'.
Adding package 'System.Data.SQLite.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015\P
Added package 'System.Data.SQLite.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Pr
Added package 'System.Data.SQLite.1.0.104' to 'packages.config'
Successfully installed 'System.Data.SQLite 1.0.104' to SQLiteSetup
```

Wie oben gezeigt, werden bei der Installation von `System.Data.SQLite` alle verwalteten verwalteten Bibliotheken zusammen mit diesem installiert. Dies umfasst `System.Data.SQLite.EF6` , den EF-Anbieter für SQLite. Das Projekt verweist jetzt auch auf die Assemblys, die zur Verwendung des SQLite-Providers erforderlich sind.



Einschließlich der nicht verwalteten Bibliothek

Die von SQLite verwalteten Bibliotheken sind von einer nicht verwalteten Assembly mit dem Namen `SQLite.Interop.dll` . Es ist in den Paketassemblys enthalten, die mit dem SQLite-Paket heruntergeladen wurden. Diese werden beim Erstellen des Projekts automatisch in Ihr Build-Verzeichnis kopiert. Da es jedoch nicht verwaltet wird, wird es nicht in Ihre Referenzliste aufgenommen. Beachten Sie jedoch, dass diese Assembly meistens mit der Anwendung verteilt wird, damit die SQLite-Assemblys funktionieren.

Hinweis: Diese Assembly ist bitabhängig, dh Sie müssen für jede zu unterstützende Bitness (x86 / x64) eine spezifische Assembly einfügen.

Bearbeiten der App.config des Projekts

Die Datei `app.config` erfordert einige Änderungen, bevor SQLite als Entity Framework-Anbieter verwendet werden kann.

Erforderliche Korrekturen

Bei der Installation des Pakets wird die Datei `app.config` automatisch aktualisiert, um die erforderlichen Einträge für SQLite und SQLite EF aufzunehmen. Leider enthalten diese Einträge einige Fehler. Sie müssen modifiziert werden, bevor es korrekt funktioniert.

`DbProviderFactories` das `DbProviderFactories` Element in der Konfigurationsdatei. Es befindet sich im Element `system.data` und enthält Folgendes

```
<DbProviderFactories>
  <remove invariant="System.Data.SQLite.EF6" />
  <add name="SQLite Data Provider (Entity Framework 6)" invariant="System.Data.SQLite.EF6"
description=".NET Framework Data Provider for SQLite (Entity Framework 6)"
type="System.Data.SQLite.EF6.SQLiteProviderFactory, System.Data.SQLite.EF6" />
  <remove invariant="System.Data.SQLite" /><add name="SQLite Data Provider"
invariant="System.Data.SQLite" description=".NET Framework Data Provider for SQLite"
type="System.Data.SQLite.SQLiteFactory, System.Data.SQLite" />
</DbProviderFactories>
```

Dies kann vereinfacht werden, um einen einzelnen Eintrag zu enthalten

```
<DbProviderFactories>
    <add name="SQLite Data Provider" invariant="System.Data.SQLite.EF6" description=".NET
Framework Data Provider for SQLite" type="System.Data.SQLite.SQLiteFactory,
System.Data.SQLite" />
</DbProviderFactories>
```

Damit haben wir festgelegt, dass die EF6-SQLite-Anbieter die SQLite-Factory verwenden sollen.

Fügen Sie die SQLite-Verbindungszeichenfolge hinzu

Verbindungszeichenfolgen können der Konfigurationsdatei innerhalb des Root-Elements hinzugefügt werden. Fügen Sie eine Verbindungszeichenfolge für den Zugriff auf eine SQLite-

Datenbank hinzu.

```
<connectionStrings>
    <add name="TestContext" connectionString="data source=testdb.sqlite;initial
catalog=Test;App=EntityFramework;" providerName="System.Data.SQLite.EF6"/>
</connectionStrings>
```

Wichtig hierbei ist der `provider` . Es wurde auf `System.Data.SQLite.EF6` . Dies teilt EF mit, dass bei Verwendung dieser Verbindungszeichenfolge SQLite verwendet werden soll. Die angegebene `data source` ist nur ein Beispiel und hängt vom Speicherort und Namen Ihrer SQLite-Datenbank ab.

Ihr erster SQLite DbContext

`DbContext` die Installation und Konfiguration abgeschlossen ist, können Sie jetzt einen `DbContext` , der für Ihre SQLite-Datenbank verwendet werden kann.

```
public class TestContext : DbContext
{
    public TestContext()
        : base("name=TestContext") { }
}
```

Durch die Angabe von `name=TestContext` ich `app.config` , dass die TestContext-Verbindungszeichenfolge in der Datei `app.config` zum Erstellen des Kontexts verwendet werden soll. Diese Verbindungszeichenfolge wurde für die Verwendung von SQLite konfiguriert. In diesem Kontext wird eine SQLite-Datenbank verwendet.

Entity Framework mit SQLite online lesen: https://riptutorial.com/de/entity-framework/topic/9280/entity-framework-mit-sqlite

Kapitel 8: Entity-Framework mit Postgresql

Examples

Erforderliche Vorschritte, um Entity Framework 6.1.3 mit PostgresSql mit Npgsqlddexprovider verwenden zu können

1) Die Sicherung von Machine.config wurde von den Speicherorten C: \ Windows \ Microsoft.NET \ Framework \ v4.0.30319 \ Config und C: \ Windows \ Microsoft.NET \ Framework64 \ v4.0.30319 \ Config übernommen

2) Kopieren Sie sie an einen anderen Ort und bearbeiten Sie sie als

a) `<system.data> <DbProviderFactories>` und hinzufügen unter `<system.data> <DbProviderFactories>`

```
            <add name="Npgsql Data Provider" invariant="Npgsql" support="FF"
            description=".Net Framework Data Provider for Postgresql Server"
            type="Npgsql.NpgsqlFactory, Npgsql, Version=2.2.5.0, Culture=neutral,
PublicKeyToken=5d8b90d52f46fda7" />
```

b) Wenn der Eintrag bereits vorhanden ist, überprüfen Sie die Version und aktualisieren Sie sie.

3. Ersetzen Sie Originaldateien durch geänderte.
4. Führen Sie die Entwickler-Eingabeaufforderung für VS2013 als Administrator aus.
5. Wenn Npgsql bereits installiert ist, verwenden Sie den Befehl "gacutil -u Npgsql" zum Deinstallieren. Installieren Sie dann die neue Version von Npgsql 2.5.0 mit dem Befehl "gacutil -i [Pfad der DLL]".
6. Machen Sie oben für Mono.Security 4.0.0.0
7. Laden Sie NpgsqlDdexProvider-2.2.0-VS2013.zip herunter und führen Sie NpgsqlDdexProvider.vsix aus. (Schließen Sie alle Instanzen von Visual Studio.)
8. EFTools6.1.3-beta1ForVS2013.msi gefunden und ausgeführt.
9. Nachdem Sie ein neues Projekt erstellt haben, installieren Sie die Version von EntityFramework (6.1.3), NpgSql (2.5.0) und NpgSql.EntityFramework (2.5.0) von Manage Nuget Packages.10) Ihr MVc-Projekt

Entity-Framework mit Postgresql online lesen: https://riptutorial.com/de/entity-framework/topic/7647/entity-framework-mit-postgresql

Kapitel 9: Entity-Framework-Code zuerst

Examples

Stellen Sie eine Verbindung zu einer vorhandenen Datenbank her

Um die einfachste Aufgabe in Entity Framework zu erreichen: Um eine Verbindung zu einer vorhandenen Datenbank `ExampleDatabase` in Ihrer lokalen Instanz von MSSQL `ExampleDatabase` , müssen Sie nur zwei Klassen implementieren.

Zuerst wird die Entitätsklasse angegeben, die unserer Datenbanktabelle `dbo.People` .

```
class Person
{
    public int PersonId { get; set; }
    public string FirstName { get; set; }
}
```

Die Klasse verwendet die Konventionen von Entity Framework und `dbo.People` der Tabelle `dbo.People` deren Primärschlüssel `PersonId` und die varchar (max) `FirstName` .

Die zweite Klasse ist die von `System.Data.Entity.DbContext` Kontextklasse, die die Entitätsobjekte zur Laufzeit verwaltet, aus der Datenbank kopiert, die Parallelität handhabt und sie in der Datenbank speichert.

```
class Context : DbContext
{
    public Context(string connectionString) : base(connectionString)
    {
        Database.SetInitializer<Context>(null);
    }

    public DbSet<Person> People { get; set; }
}
```

Bitte beachten Sie, dass wir im Konstruktor unseres Kontexts den Datenbankinitialisierer auf null setzen müssen. Wir möchten nicht, dass Entity Framework die Datenbank erstellt, sondern lediglich auf die Datenbank.

Jetzt können Sie Daten aus dieser Tabelle `FirstName` , z. B. den `FirstName` der ersten Person in der Datenbank von einer Konsolenanwendung wie `FirstName` ändern:

```
class Program
{
    static void Main(string[] args)
    {
        using (var ctx = new Context("DbConnectionString"))
        {
            var firstPerson = ctx.People.FirstOrDefault();
            if (firstPerson != null) {
```

```
                    firstPerson.FirstName = "John";
                    ctx.SaveChanges();
                }
            }
        }
    }
```

Im obigen Code haben wir eine Instanz von Context mit dem Argument "DbConnectionString" erstellt. Dies muss in unserer `app.config` Datei wie `app.config` angegeben werden:

```
<connectionStrings>
  <add  name="DbConnectionString"
  connectionString="Data Source=.;Initial Catalog=ExampleDatabase;Integrated Security=True"
  providerName="System.Data.SqlClient"/>
</connectionStrings>
```

Entity-Framework-Code zuerst online lesen: https://riptutorial.com/de/entity-framework/topic/5337/entity-framework-code-zuerst

Kapitel 10: Erste Datenanmerkungen kodieren

Bemerkungen

Entity Framework Code-First stellt eine Reihe von DataAnnotation-Attributen bereit, die Sie auf Ihre Domänenklassen und -eigenschaften anwenden können. DataAnnotation-Attribute überschreiben die Standard-Code-First-Konventionen.

1. **System.ComponentModel.DataAnnotations** enthält Attribute, die sich auf die Nullfähigkeit oder die Größe der Spalte auswirken.
2. **Der** Namespace **System.ComponentModel.DataAnnotations.Schema** enthält Attribute, die sich auf das Schema der Datenbank auswirken.

Hinweis: Mit DataAnnotations erhalten Sie nur einen Teil der Konfigurationsoptionen. Fluent API bietet einen vollständigen Satz von Konfigurationsoptionen, die in Code-First verfügbar sind.

Examples

[Schlüssel] -Attribut

Schlüssel ist ein Feld in einer Tabelle, das jede Zeile / jeden Datensatz in einer Datenbanktabelle eindeutig identifiziert.

Verwenden Sie dieses Attribut, **um die Standard-Code-First-Konvention** zu **überschreiben** . Wenn sie auf eine Eigenschaft angewendet werden, wird sie als **Primärschlüsselspalte** für diese Klasse verwendet.

```
using System.ComponentModel.DataAnnotations;

public class Person
{
    [Key]
    public int PersonKey { get; set; }        // <- will be used as primary key

    public string PersonName { get; set; }
}
```

Wenn ein zusammengesetzter Primärschlüssel erforderlich ist, kann das Attribut [Schlüssel] auch mehreren Eigenschaften hinzugefügt werden. Die Reihenfolge der Spalten innerhalb des zusammengesetzten Schlüssels muss in der Form [**Schlüssel, Spalte (Reihenfolge = x)] angegeben werden** .

```
using System.ComponentModel.DataAnnotations;

public class Person
```

```
{
    [Key, Column(Order = 0)]
    public int PersonKey1 { get; set; }    // <- will be used as part of the primary key

    [Key, Column(Order = 1)]
    public int PersonKey2 { get; set; }    // <- will be used as part of the primary key

    public string PersonName { get; set; }
}
```

Ohne das Attribut [Key] greift EntityFramework auf die Standardkonvention zurück, bei der die Eigenschaft der Klasse als Primärschlüssel mit dem Namen "Id" oder "{ClassName} Id" verwendet wird.

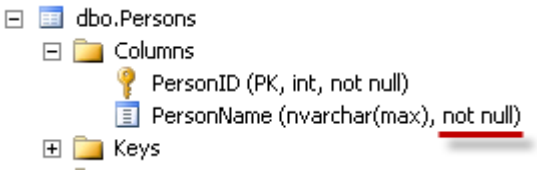
```
public class Person
{
    public int PersonID { get; set; }        // <- will be used as primary key

    public string PersonName { get; set; }
}
```

[Erforderlich] Attribut

Bei der Anwendung auf eine Eigenschaft einer Domänenklasse erstellt die Datenbank eine NOT NULL-Spalte.

```
using System.ComponentModel.DataAnnotations;

public class Person
{
    public int PersonID { get; set; }

    [Required]
    public string PersonName { get; set; }
}
```

Die resultierende Spalte mit der NOT NULL-Einschränkung:



Hinweis: Es kann auch mit asp.net-mvc als Validierungsattribut verwendet werden.

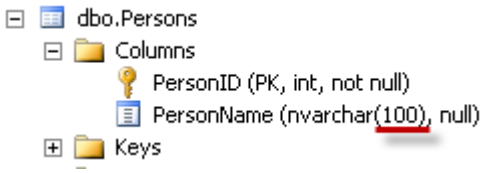
Attribute [MaxLength] und [MinLength]

Das Attribut **[MaxLength (int)]** kann auf eine **String-** oder Array-Typ-Eigenschaft einer Domänenklasse angewendet werden. Entity Framework setzt die Größe einer Spalte auf den angegebenen Wert.

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

```
public class Person
{
    public int PersonID { get; set; }

    [MinLength(3), MaxLength(100)]
    public string PersonName { get; set; }
}
```

Die resultierende Spalte mit der angegebenen Spaltenlänge:



Das Attribut **[MinLength (int)]** ist ein Validierungsattribut. Es hat keinen Einfluss auf die Datenbankstruktur. Wenn wir versuchen, eine Person mit PersonName mit einer Länge von weniger als 3 Zeichen einzufügen / zu aktualisieren, schlägt diese Festschreibung fehl. Wir erhalten eine `DbUpdateConcurrencyException` , die wir behandeln müssen.

```
using (var db = new ApplicationDbContext())
{
     db.Staff.Add(new Person() { PersonName = "ng" });
     try
     {
          db.SaveChanges();
     }
     catch (DbEntityValidationException ex)
     {
          //ErrorMessage = "The field PersonName must be a string or array type with a minimum
length of '3'."
     }
}
```

Beide Attribute **[MaxLength]** und **[MinLength]** können auch mit asp.net-mvc als Validierungsattribut verwendet werden.

Attribut [Bereich (min, max)]

Gibt einen numerischen Minimal- und Maximalbereich für eine Eigenschaft an

```
using System.ComponentModel.DataAnnotations;

public partial class Enrollment
{
    public int EnrollmentID { get; set; }

    [Range(0, 4)]
    public Nullable<decimal> Grade { get; set; }
}
```

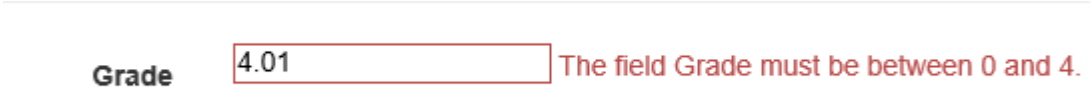
Wenn wir versuchen, eine Note mit einem Wert außerhalb des Bereichs einzufügen / zu aktualisieren, schlägt diese Festschreibung fehl. Wir erhalten eine `DbUpdateConcurrencyException` ,

die wir behandeln müssen.

```
using (var db = new ApplicationDbContext())
{
    db.Enrollments.Add(new Enrollment() { Grade = 1000 });

    try
    {
        db.SaveChanges();
    }
    catch (DbEntityValidationException ex)
    {
        // Validation failed for one or more entities
    }
}
```

Es kann auch mit asp.net-mvc als Validierungsattribut verwendet werden.

Ergebnis:



[DatabaseGenerated] -Attribut

Gibt an, wie die Datenbank Werte für die Eigenschaft generiert. Es gibt drei mögliche Werte:

1. `None` gibt an, dass die Werte nicht von der Datenbank generiert werden.
2. `Identity` gibt an, dass die Spalte eine Identitätsspalte ist , die normalerweise für Ganzzahl-Primärschlüssel verwendet wird.
3. `Computed` gibt an, dass die Datenbank den Wert für die Spalte generiert.

Wenn der Wert nicht " `None` , werden die an der Eigenschaft vorgenommenen Änderungen nicht in die Datenbank zurückgegeben.

Standardmäßig (basierend auf `StoreGeneratedIdentityKeyConvention`) wird eine Integer-Schlüsseleigenschaft als Identitätsspalte behandelt. Um diese Konvention zu überschreiben und zu erzwingen, dass sie als Nicht-Identitätsspalte behandelt wird, können Sie das `DatabaseGenerated` Attribut mit dem Wert `None` .

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

public class Foo
{
    [Key]
    public int Id { get; set; } // identity (auto-increment) column
}

public class Bar
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int Id { get; set; } // non-identity column
```

```
}
```

Die folgende SQL erstellt eine Tabelle mit einer berechneten Spalte:

```
CREATE TABLE [Person] (
    Name varchar(100) PRIMARY KEY,
    DateOfBirth Date NOT NULL,
    Age AS DATEDIFF(year, DateOfBirth, GETDATE())
)
GO
```

Um eine Entität für die Darstellung der Datensätze in der obigen Tabelle zu erstellen, müssen Sie das `DatabaseGenerated` Attribut mit dem Wert `Computed` .

```
[Table("Person")]
public class Person
{
    [Key, StringLength(100)]
    public string Name { get; set; }
    public DateTime DateOfBirth { get; set; }
    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
    public int Age { get; set; }
}
```

[NotMapped] -Attribut

Nach der Code-First-Konvention erstellt Entity Framework eine Spalte für jede öffentliche Eigenschaft, die einen unterstützten Datentyp hat und sowohl einen Getter als auch einen Setter enthält. **Die** Annotation **[NotMapped]** muss auf alle Eigenschaften angewendet werden, für die **KEINE** Spalte in einer Datenbanktabelle gewünscht wird.

Ein Beispiel für eine Eigenschaft, die wir möglicherweise nicht in der Datenbank speichern möchten, ist der vollständige Name eines Schülers, der auf seinem Vor- und Nachnamen basiert. Das kann man sofort berechnen und muss nicht in der Datenbank gespeichert werden.

```
public string FullName => string.Format("{0} {1}", FirstName, LastName);
```

Die Eigenschaft "FullName" hat nur einen Getter und keinen Setter. Daher erstellt Entity Framework standardmäßig **KEINE** Spalte dafür.

Ein anderes Beispiel für eine Eigenschaft, die wir möglicherweise nicht in der Datenbank speichern möchten, ist das "AverageGrade" eines Schülers. Wir möchten den AverageGrade nicht auf Abruf erhalten. Stattdessen könnten wir an anderer Stelle eine Routine haben, die sie berechnet.

```
[NotMapped]
public float AverageGrade { set; get; }
```

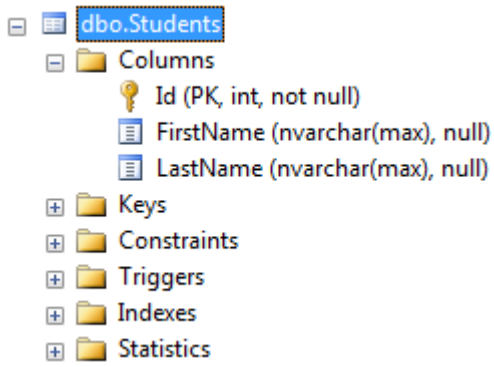
"AverageGrade" muss mit **[NotMapped] gekennzeichnet sein** , andernfalls erstellt Entity Framework eine Spalte dafür.

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

public class Student
{
    public int Id { set; get; }

    public string FirstName { set; get; }

    public string LastName { set; get; }

    public string FullName => string.Format("{0} {1}", FirstName, LastName);

    [NotMapped]
    public float AverageGrade { set; get; }
}
```

Für die obige Entität sehen wir `DbMigration.cs`

```
CreateTable(
    "dbo.Students",
     c => new
         {
             Id = c.Int(nullable: false, identity: true),
             FirstName = c.String(),
             LastName = c.String(),
         })
     .PrimaryKey(t => t.Id);
```

und in SQL Server Management Studio



[Tabelle] -Attribut

```
[Table("People")]
public class Person
{
    public int PersonID { get; set; }
    public string PersonName { get; set; }
}
```

Weist Entity Framework an, einen bestimmten Tabellennamen zu verwenden, anstatt einen zu generieren (zB `Person` oder `Persons`)

Wir können auch ein Schema für die Tabelle mit dem Attribut [Tabelle] angeben

```
[Table("People", Schema = "domain")]
```

[Spalte] -Attribut

```
public class Person
{
    public int PersonID { get; set; }

    [Column("NameOfPerson")]
    public string PersonName { get; set; }
}
```

Weist Entity Framework an, einen bestimmten Spaltennamen zu verwenden, anstatt den Namen der Eigenschaft zu verwenden. Sie können auch den Datenbankdatentyp und die Reihenfolge der Spalte in der Tabelle angeben:

```
[Column("NameOfPerson", TypeName = "varchar", Order = 1)]
public string PersonName { get; set; }
```

[Index] -Attribut

```
public class Person
{
    public int PersonID { get; set; }
    public string PersonName { get; set; }

    [Index]
    public int Age { get; set; }
}
```

Erstellt einen Datenbankindex für eine Spalte oder einen Satz von Spalten.

```
[Index("IX_Person_Age")]
public int Age { get; set; }
```

Dadurch wird ein Index mit einem bestimmten Namen erstellt.

```
[Index(IsUnique = true)]
public int Age { get; set; }
```

Dadurch wird ein eindeutiger Index erstellt.

```
[Index("IX_Person_NameAndAge", 1)]
public int Age { get; set; }

[Index("IX_Person_NameAndAge", 2)]
public string PersonName { get; set; }
```

Dadurch wird ein zusammengesetzter Index mit zwei Spalten erstellt. Dazu müssen Sie denselben Indexnamen angeben und eine Spaltenreihenfolge angeben.

Hinweis : Das Index-Attribut wurde in Entity Framework 6.1 eingeführt. Wenn Sie eine frühere Version verwenden, gelten die Informationen in diesem Abschnitt nicht.

[ForeignKey (Zeichenfolge)] Attribut

Gibt den benutzerdefinierten Fremdschlüsselnamen an, wenn ein Fremdschlüssel gewünscht wird, der nicht der Konvention von EF entspricht.

```
public class Person
{
    public int IdAddress { get; set; }

    [ForeignKey(nameof(IdAddress))]
    public virtual Address HomeAddress { get; set; }
}
```

Dies kann auch verwendet werden, wenn Sie mehrere Beziehungen zu demselben Entitätstyp haben.

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

public class Customer
{
    ...

    public int MailingAddressID { get; set; }
    public int BillingAddressID { get; set; }

    [ForeignKey("MailingAddressID")]
    public virtual Address MailingAddress { get; set; }
    [ForeignKey("BillingAddressID")]
    public virtual Address BillingAddress { get; set; }
}
```

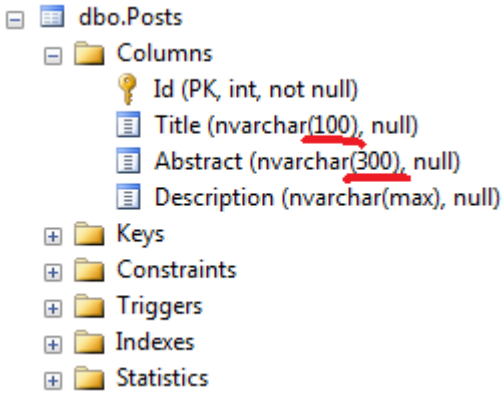
Ohne die `ForeignKey` Attribute könnte EF sie `BillingAddressID` beim Abrufen der `MailingAddress` den Wert von `BillingAddressID` `MailingAddress` , oder es wird nur ein anderer Name für die Spalte `MailingAddress` , der auf seinen eigenen Namenskonventionen (wie `Address_MailingAddress_Id`) basiert Stattdessen (was zu einem Fehler führen würde, wenn Sie dies mit einer vorhandenen Datenbank verwenden).

[StringLength (int)] Attribut

```
using System.ComponentModel.DataAnnotations;

public class Post
{
    public int Id { get; set; }

    [StringLength(100)]
    public string Title { get; set;}

    [StringLength(300)]
    public string Abstract { get; set; }
```

```
    public string Description { get; set; }
}
```

Definiert eine maximale Länge für ein Zeichenfolgefeld.

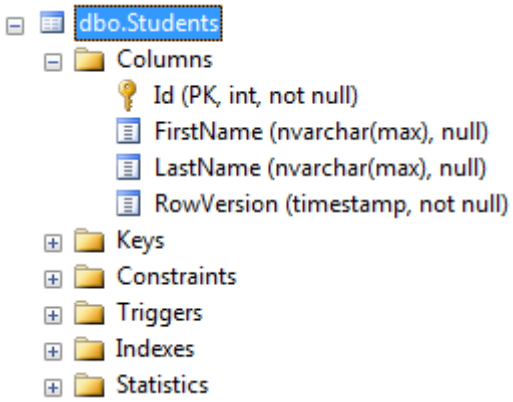


Hinweis : Es kann auch mit asp.net-mvc als Validierungsattribut verwendet werden.

[Zeitstempel] -Attribut

Das Attribut **[TimeStamp]** kann nur auf eine Byte-Array-Eigenschaft in einer bestimmten Entitätsklasse angewendet werden. Entity Framework erstellt für diese Eigenschaft eine nicht-nullfähige Zeitstempelpalte in der Datenbanktabelle. Entity Framework verwendet diese TimeStamp-Spalte automatisch für die Parallelitätsprüfung.

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

public class Student
{
    public int Id { set; get; }

    public string FirstName { set; get; }

    public string LastName { set; get; }

    [Timestamp]
    public byte[] RowVersion { get; set; }
}
```



[ConcurrencyCheck] Attribut

Dieses Attribut wird auf die Klasseneigenschaft angewendet. Sie können das ConcurrencyCheck-Attribut verwenden, wenn Sie vorhandene Spalten für die Parallelitätsprüfung und keine separate Zeitstempelpalte für die Parallelität verwenden möchten.

```
using System.ComponentModel.DataAnnotations;

public class Author
{
    public int AuthorId { get; set; }

    [ConcurrencyCheck]
    public string AuthorName { get; set; }
}
```

Im obigen Beispiel wird das ConcurrencyCheck-Attribut auf die AuthorName-Eigenschaft der Author-Klasse angewendet. Daher enthält Code-First die Spalte AuthorName im Aktualisierungsbefehl (where-Klausel), um die optimistische Parallelität zu überprüfen.

Attribut [InverseProperty (Zeichenfolge)]

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

public class Department
{
    ...

    public virtual ICollection<Employee> PrimaryEmployees { get; set; }
    public virtual ICollection<Employee> SecondaryEmployees { get; set; }
}

public class Employee
{
    ...

    [InverseProperty("PrimaryEmployees")]
    public virtual Department PrimaryDepartment { get; set; }

    [InverseProperty("SecondaryEmployees")]
    public virtual Department SecondaryDepartment { get; set; }
}
```

InverseProperty kann verwendet werden, um *Zweiwege*- Beziehungen zu identifizieren , wenn zwischen zwei Entitäten **mehrere** *Zweiwege*- Beziehungen bestehen.

Sie teilt Entity Framework mit, welche Navigationseigenschaften mit den Eigenschaften auf der anderen Seite übereinstimmen sollten.

Entity Framework weiß nicht, welche Navigationseigenschaftskarte mit welchen Eigenschaften auf der anderen Seite verbunden ist, wenn zwischen zwei Entitäten mehrere bidirektionale Beziehungen bestehen.

Es benötigt den Namen der entsprechenden Navigationseigenschaft in der zugehörigen Klasse als Parameter.

Dies kann auch für Entitäten verwendet werden, die eine Beziehung zu anderen Entitäten desselben Typs haben und eine rekursive Beziehung bilden.

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

public class TreeNode
{
    [Key]
    public int ID { get; set; }
    public int ParentID { get; set; }

    ...

    [ForeignKey("ParentID")]
    public TreeNode ParentNode { get; set; }
    [InverseProperty("ParentNode")]
    public virtual ICollection<TreeNode> ChildNodes { get; set; }
}
```

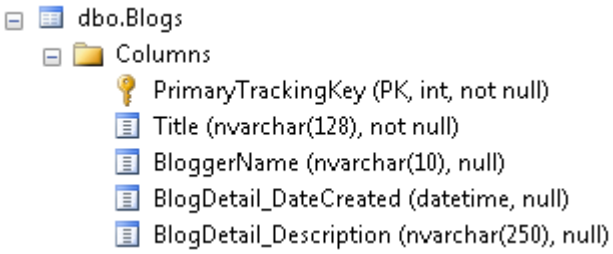
Beachten Sie auch die Verwendung des `ForeignKey` Attributs, um die Spalte anzugeben, die für den Fremdschlüssel in der Tabelle verwendet wird. Im ersten Beispiel wurde `ForeignKey` für die beiden Eigenschaften der `Employee` Klasse das `ForeignKey` Attribut angewendet, um die Spaltennamen zu definieren.

[ComplexType] -Attribut

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

[ComplexType]
public class BlogDetails
{
    public DateTime? DateCreated { get; set; }

    [MaxLength(250)]
    public string Description { get; set; }
}

public class Blog
{
    ...

    public BlogDetails BlogDetail { get; set; }
}
```

Markieren Sie die Klasse als komplexen Typ in Entity Framework.

Komplexe Typen (oder *Wertobjekte* im domänengesteuerten Design) können nicht alleine verfolgt werden, sie werden jedoch als Teil einer Entität verfolgt. Aus diesem Grund verfügt BlogDetails im Beispiel über keine Schlüsseleigenschaft.



Sie können nützlich sein, wenn Sie Domänenentitäten über mehrere Klassen hinweg beschreiben und diese Klassen zu einer vollständigen Entität zusammenfassen.

Erste Datenanmerkungen kodieren online lesen: https://riptutorial.com/de/entity-framework/topic/4161/erste-datenanmerkungen-kodieren

Kapitel 11: Erste Konventionen für den Code

Bemerkungen

Konvention ist ein Satz von Standardregeln zum automatischen Konfigurieren eines konzeptionellen Modells basierend auf Domänenklassendefinitionen bei der Arbeit mit Code-First. Code-First-Konventionen sind im Namespace *System.Data.Entity.ModelConfiguration.Conventions* definiert (EF 5 & EF 6).

Examples

Primärschlüsselkonvention

Eine Eigenschaft ist standardmäßig ein Primärschlüssel, wenn eine Eigenschaft in einer Klasse "ID" (nicht Groß- und Kleinschreibung beachten) oder der Klassenname gefolgt von "ID" lautet. Wenn der Typ der Primärschlüsseleigenschaft numerisch oder GUID ist, wird er als Identitätsspalte konfiguriert. Einfaches Beispiel:

```
public class Room
{
    // Primary key
    public int RoomId{ get; set; }
    ...
}
```

Konventionen entfernen

Sie können alle im Namensraum "System.Data.Entity.ModelConfiguration.Conventions" definierten Konventionen entfernen, indem Sie die `OnModelCreating` Methode überschreiben.

Im folgenden Beispiel wird PluralizingTableNameConvention entfernt.

```
public class EshopContext : DbContext
{
    public DbSet<Product> Products { set; get; }
    . . .

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    }
}
```

Standardmäßig erstellt EF eine DB-Tabelle mit dem Entitätsklassennamen mit dem Zusatz 's'. In diesem Beispiel wird der Code zunächst konfiguriert PluralizingTableName Konvention so, statt zu ignorieren `dbo.Products` Tabelle `dbo.Product` Tabelle erstellt wird.

Geben Sie Discovery ein

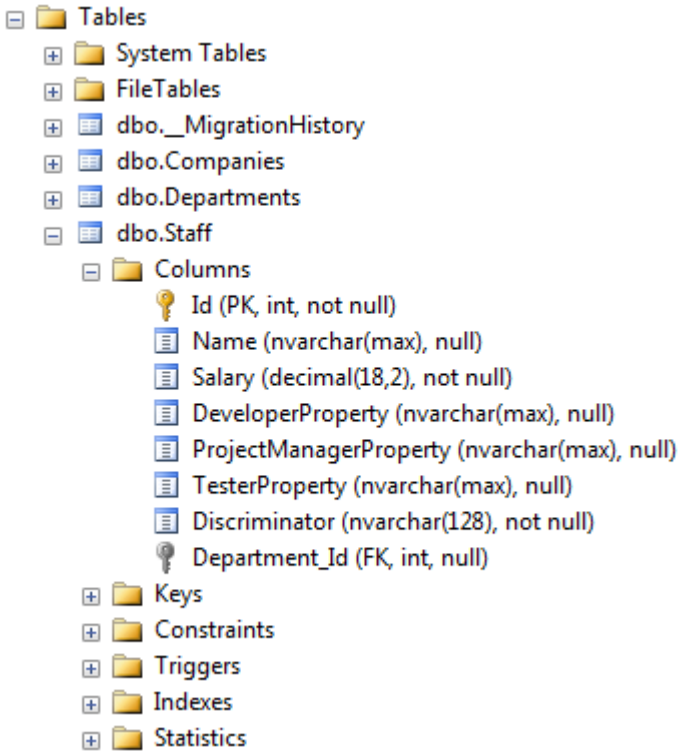
Standardmäßig enthält Code First das Modell

1. Typen, die als DbSet-Eigenschaft in der Kontextklasse definiert sind.
2. Referenztypen, die in Entitätstypen enthalten sind, auch wenn sie in einer anderen Assembly definiert sind.
3. Abgeleitete Klassen, auch wenn nur die Basisklasse als DbSet-Eigenschaft definiert ist

Hier ein Beispiel, dass wir in unserer `DbSet<Company>` nur `Company` als `DbSet<Company>` hinzufügen:

```
public class Company
{
    public int Id { set; get; }
    public string Name { set; get; }
    public virtual ICollection<Department> Departments { set; get; }
}

public class Department
{
    public int Id { set; get; }
    public string Name { set; get; }
    public virtual ICollection<Person> Staff { set; get; }
}

[Table("Staff")]
public class Person
{
    public int Id { set; get; }
    public string Name { set; get; }
    public decimal Salary { set; get; }
}

public class ProjectManager : Person
{
   public string ProjectManagerProperty { set; get; }
}

public class Developer : Person
{
    public string DeveloperProperty { set; get; }
}

public class Tester : Person
{
    public string TesterProperty { set; get; }
}

public class ApplicationDbContext : DbContext
{
    public DbSet<Company> Companies { set; get; }
}
```

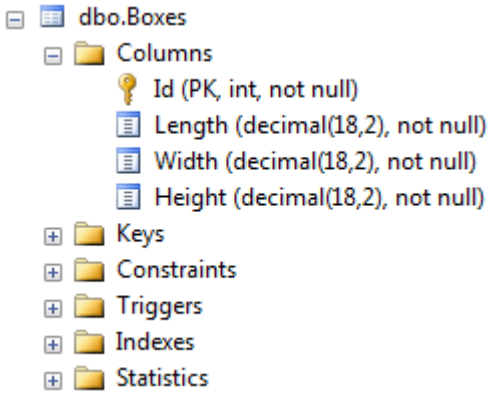
Wir können sehen, dass alle Klassen im Modell enthalten sind



DecimalPropertyConvention

Standardmäßig ordnet Entity Framework dezimale Eigenschaften in dezimalen (18,2) Spalten in Datenbanktabellen zu.

```
public class Box
{
    public int Id { set; get; }
    public decimal Length { set; get; }
    public decimal Width { set; get; }
    public decimal Height { set; get; }
}
```

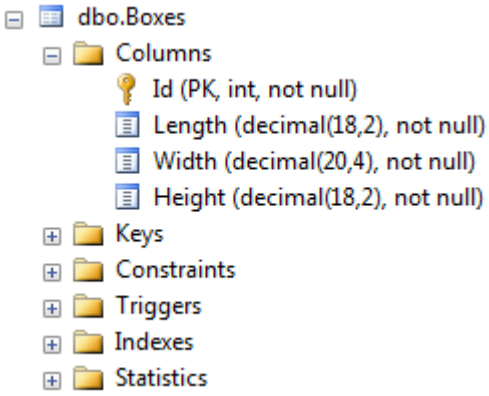


Wir können die Genauigkeit der Dezimaleigenschaften ändern:

1. Verwenden Sie Fluent API:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Box>().Property(b => b.Width).HasPrecision(20, 4);
```

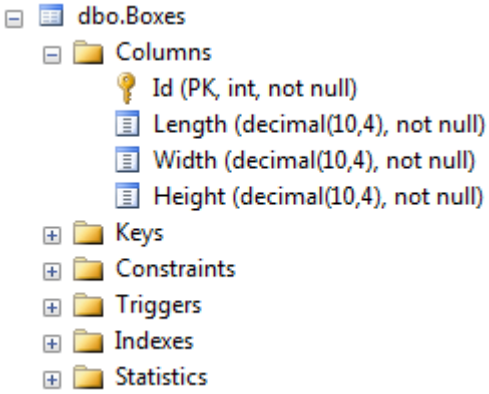
```
}
```



Nur die Eigenschaft "Width" wird dezimal (20, 4) zugeordnet.

2. Ersetzen Sie die Konvention:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Conventions.Remove<DecimalPropertyConvention>();
    modelBuilder.Conventions.Add(new DecimalPropertyConvention(10, 4));
}
```



Jede dezimale Eigenschaft wird dezimalen (10,4) Spalten zugeordnet.

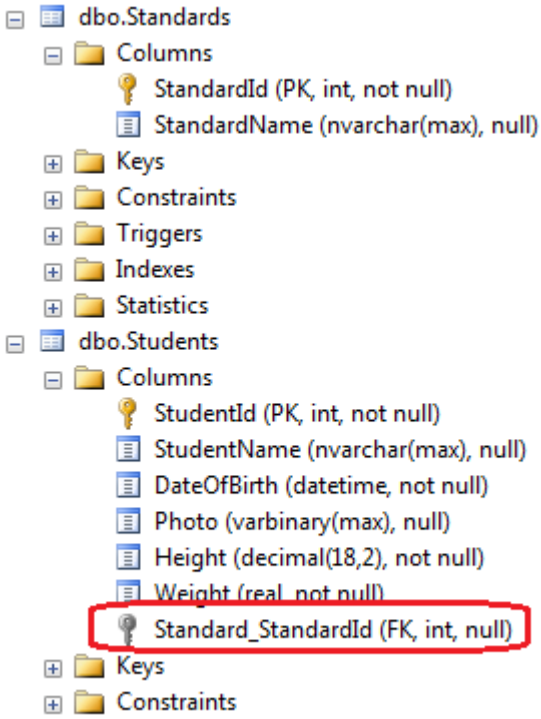
Beziehungskonvention

Code Ermitteln Sie zunächst die Beziehung zwischen den beiden Entitäten mithilfe der Navigationseigenschaft. Diese Navigationseigenschaft kann ein einfacher Referenztyp oder ein Auflistungstyp sein. Beispielsweise haben wir die Standardnavigationseigenschaft in der Student-Klasse und die ICollection-Navigationseigenschaft in der Standardklasse definiert. So erstellte Code First automatisch eine Eins-zu-Viele-Beziehung zwischen der Tabelle "Standards" und "Students DB", indem die Fremdschlüsselspalte Standard_StandardId in die Tabelle Students eingefügt wurde.

```
public class Student
{

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

```
    public string StudentName { get; set; }
    public DateTime DateOfBirth { get; set; }

    //Navigation property
    public Standard Standard { get; set; }
}

public class Standard
{

    public int StandardId { get; set; }
    public string StandardName { get; set; }

    //Collection navigation property
    public IList<Student> Students { get; set; }

}
```

Die obigen Entitäten erstellten die folgende Beziehung mit dem Fremdschlüssel Standard_StandardId.



Fremdschlüsselübereinkommen

Wenn die Klasse A mit der Klasse B in Beziehung steht und Klasse B eine Eigenschaft mit demselben Namen und Typ wie der Primärschlüssel von A hat, nimmt EF automatisch an, dass die Eigenschaft ein Fremdschlüssel ist.

```
public class Department
{
    public int DepartmentId { set; get; }
    public string Name { set; get; }
    public virtual ICollection<Person> Staff { set; get; }
}
```

```
public class Person
{
    public int Id { set; get; }
    public string Name { set; get; }
    public decimal Salary { set; get; }
    public int DepartmentId { set; get; }
    public virtual Department Department { set; get; }
}
```

In diesem Fall ist DepartmentId ein Fremdschlüssel ohne explizite Angabe.

Erste Konventionen für den Code online lesen: https://riptutorial.com/de/entity-framework/topic/2447/erste-konventionen-fur-den-code

Kapitel 12: Erste Migrationen für Entity-Framework-Code

Examples

Migrationen aktivieren

Verwenden Sie den Befehl, um Code First Migrations im Entity Framework zu aktivieren

```
Enable-Migrations
```

in der *Package Manager Console* .

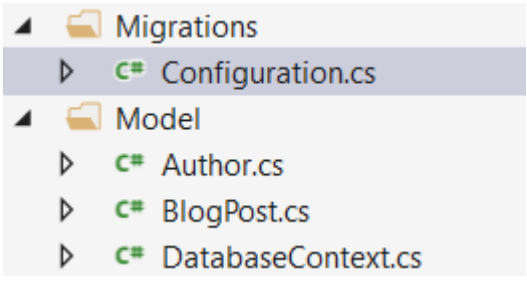
Sie benötigen eine gültige `DbContext` Implementierung, die Ihre von EF verwalteten Datenbankobjekte enthält. In diesem Beispiel enthält der Datenbankkontext die Objekte `BlogPost` und `Author` :

```
internal class DatabaseContext: DbContext
{
    public DbSet<Author> Authors { get; set; }

    public DbSet<BlogPost> BlogPosts { get; set; }
}
```

Nach der Ausführung des Befehls sollte die folgende Ausgabe erscheinen:

```
PM> Enable-Migrations
Checking if the context targets an existing database...
Code First Migrations enabled for project <YourProjectName>.
PM>
```

Darüber hinaus sollte ein neuer Ordner `Migrations` mit einer einzigen Datei `Configuration.cs` angezeigt werden:



Der nächste Schritt wäre das Erstellen des ersten Datenbankmigrationsskripts, das die ursprüngliche Datenbank erstellt (siehe nächstes Beispiel).

Fügen Sie Ihre erste Migration hinzu

Nachdem Sie Migrationen aktiviert haben (siehe dieses Beispiel), können Sie jetzt Ihre erste

Migration mit einer ersten Erstellung aller Datenbanktabellen, Indizes und Verbindungen erstellen.

Mit dem Befehl kann eine Migration erstellt werden

```
Add-Migration <migration-name>
```

Dieser Befehl erstellt eine neue Klasse mit zwei Methoden `Up` und `Down` , die zum Anwenden und Entfernen der Migration verwendet werden.

Wenden Sie nun den Befehl gemäß dem obigen Beispiel an, um eine Migration mit dem Namen *Initial* zu erstellen:

```
PM> Add-Migration Initial
Scaffolding migration 'Initial'.
The Designer Code for this migration file includes a snapshot of your current Code
First model. This snapshot is used to calculate the changes to your model when you
scaffold the next migration. If you make additional changes to your model that you
want to include in this migration, then you can re-scaffold it by running
'Add-Migration Initial' again.
```

Eine neue Datei *timestamp* _Initial.cs wird erstellt (hier werden nur die wichtigsten Elemente angezeigt):

```
public override void Up()
{
   CreateTable(
       "dbo.Authors",
        c => new
           {
               AuthorId = c.Int(nullable: false, identity: true),
               Name = c.String(maxLength: 128),
           })
        .PrimaryKey(t => t.AuthorId);

    CreateTable(
       "dbo.BlogPosts",
       c => new
           {
                Id = c.Int(nullable: false, identity: true),
                Title = c.String(nullable: false, maxLength: 128),
                Message = c.String(),
                Author_AuthorId = c.Int(),
            })
        .PrimaryKey(t => t.Id)
        .ForeignKey("dbo.Authors", t => t.Author_AuthorId)
        .Index(t => t.Author_AuthorId);
}

public override void Down()
{
    DropForeignKey("dbo.BlogPosts", "Author_AuthorId", "dbo.Authors");
    DropIndex("dbo.BlogPosts", new[] { "Author_AuthorId" });
    DropTable("dbo.BlogPosts");
    DropTable("dbo.Authors");
}
```

Wie Sie sehen, werden in der Methode `Up()` zwei Tabellen `Authors` und `BlogPosts` erstellt, und die Felder werden entsprechend erstellt. Außerdem wird die Beziehung zwischen den beiden Tabellen erstellt, indem das Feld `Author_AuthorId` . Auf der anderen Seite versucht die Methode `Down()` , die Migrationsaktivitäten umzukehren.

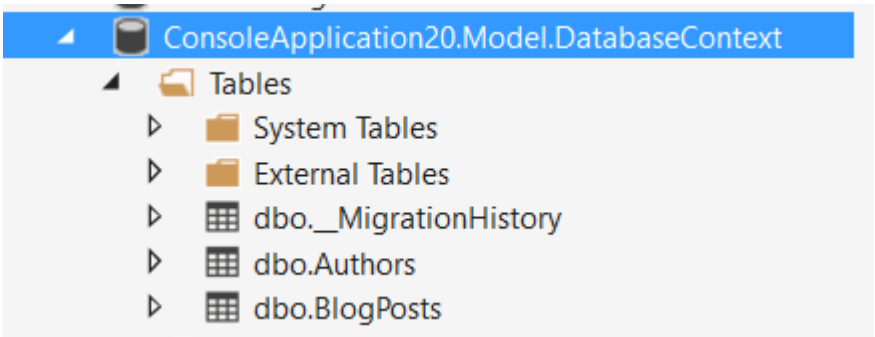
Wenn Sie mit Ihrer Migration vertraut sind, können Sie die Migration mit dem Befehl auf die Datenbank anwenden:

```
Update-Database
```

Alle anstehenden Migrationen (in diesem Fall der *Anfangs* -Migration) werden in die Datenbank übernommen und danach die Samen Methode angewendet wird (das entsprechende Beispiel)

```
PM> update-database
Specify the '-Verbose' flag to view the SQL statements being applied to the target
database.
Applying explicit migrations: [201609302203541_Initial].
Applying explicit migration: 201609302203541_Initial.
Running Seed method.
```

Sie können die Ergebnisse der Aktivitäten im SQL-Explorer sehen:



Für die Befehle `Add-Migration` und `Update-Database` verschiedene Optionen zur Verfügung, mit denen die Aktivitäten angepasst werden können. Um alle Optionen anzuzeigen, verwenden Sie bitte

```
get-help Add-Migration
```

und

```
get-help Update-Database
```

Daten während Migrationen suchen

Nach dem Aktivieren und Erstellen von Migrationen müssen Sie möglicherweise zunächst Daten in Ihrer Datenbank füllen oder migrieren. Es gibt mehrere Möglichkeiten, aber für einfache Migrationen können Sie die Methode 'Seed ()' in der Datei Configuration verwenden, die nach dem Aufruf von `enable-migrations` .

Die `Seed()` Funktion ruft einen Datenbankkontext als einzigen Parameter ab und Sie können EF-

Operationen innerhalb dieser Funktion ausführen:

```
protected override void Seed(Model.DatabaseContext context);
```

Sie können alle Arten von Aktivitäten in `Seed()` ausführen. Im Falle eines Fehlers wird die vollständige Transaktion (auch die angewendeten Patches) zurückgesetzt.

Eine Beispielfunktion, die nur Daten erstellt, wenn eine Tabelle leer ist, könnte folgendermaßen aussehen:

```
protected override void Seed(Model.DatabaseContext context)
{
    if (!context.Customers.Any()) {
        Customer c = new Customer{ Id = 1, Name = "Demo" };
        context.Customers.Add(c);
        context.SaveData();
    }
}
```

Eine nette Funktion der EF-Entwickler ist die Erweiterungsmethode `AddOrUpdate()` . Diese Methode ermöglicht das Aktualisieren von Daten basierend auf dem Primärschlüssel oder das Einfügen von Daten, falls diese noch nicht vorhanden sind (das Beispiel stammt aus dem generierten Quellcode von Configuration.cs):

```
protected override void Seed(Model.DatabaseContext context)
{
    context.People.AddOrUpdate(
      p => p.FullName,
      new Person { FullName = "Andrew Peters" },
      new Person { FullName = "Brice Lambson" },
      new Person { FullName = "Rowan Miller" }
    );
}
```

> Bitte beachten Sie, dass `Seed()` aufgerufen wird, nachdem der **letzte** Patch angewendet wurde. Wenn Sie während der Patches Daten migrieren oder überarbeiten müssen, müssen andere Ansätze verwendet werden.

Verwenden von SQL () während Migrationen

Beispiel: Sie migrieren eine vorhandene Spalte von nicht erforderlich zu erforderlich. In diesem Fall müssen Sie möglicherweise einige Standardwerte in der Migration für Zeilen eingeben, bei denen die geänderten Felder tatsächlich `NULL` . `defaultSql` der Standardwert einfach ist (z. B. "0"), können Sie in Ihrer `defaultSql` eine `default` oder `defaultSql` Eigenschaft verwenden. Falls es nicht so einfach ist, können Sie die `Sql()` Funktion in den `Up()` oder `Down() Sql()` Ihrer Migrationen verwenden.

Hier ist ein Beispiel. Annahme einer Klasse *Autor,* der eine E-Mail-Adresse als Teil des Datensatzes enthält. Jetzt entscheiden wir uns, die E-Mail-Adresse als Pflichtfeld zu verwenden. Um vorhandene Spalten zu migrieren, hat das Unternehmen die *intelligente* Idee, Dummy-E-Mail-Adressen zu `fullname@example.com` , z. B. `fullname@example.com` , wobei der vollständige Name der

vollständige Name des `fullname@example.com` ohne Leerzeichen ist. Durch das Hinzufügen des Attributs `[Required]` zum Feld `Email` die folgende Migration erstellt:

```
public partial class AuthorsEmailRequired : DbMigration
{
    public override void Up()
    {
        AlterColumn("dbo.Authors", "Email", c => c.String(nullable: false, maxLength: 512));
    }

    public override void Down()
    {
        AlterColumn("dbo.Authors", "Email", c => c.String(maxLength: 512));
    }
}
```

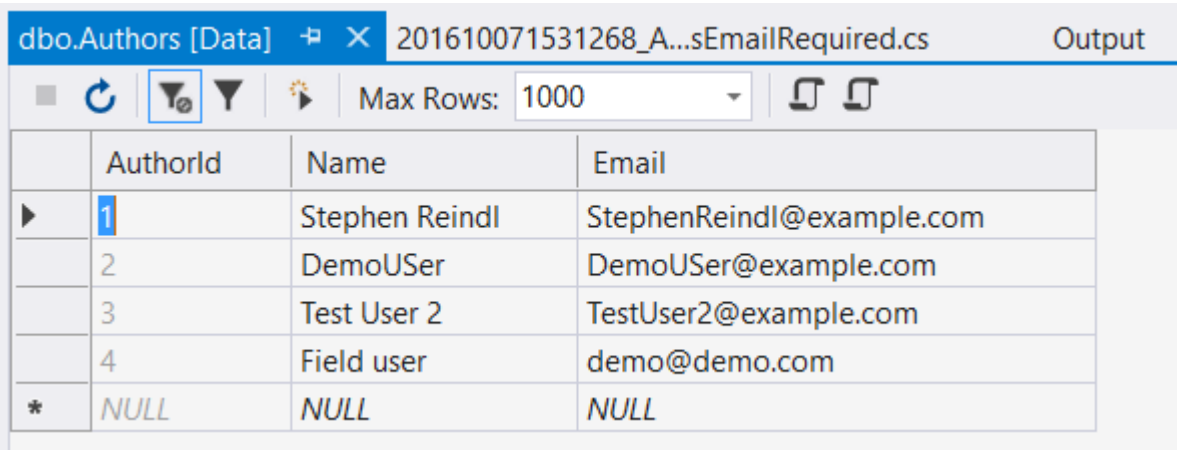
Dies würde fehlschlagen, wenn sich einige NULL-Felder in der Datenbank befinden:

> Der Wert NULL kann nicht in die Spalte 'Email' eingefügt werden, Tabelle 'App.Model.DatabaseContext.dbo.Authors'; Spalte erlaubt keine Nullen. UPDATE schlägt fehl

Das Hinzufügen des folgenden Befehls **vor** dem Befehl `AlterColumn` hilft,

```
Sql(@"Update dbo.Authors
    set Email = REPLACE(name, ' ', '') + N'@example.com'
    where Email is null");
```

Der Aufruf der `update-database` erfolgreich und die Tabelle sieht folgendermaßen aus (Beispieldaten werden angezeigt):



Andere Verwendung

Sie können die `Sql()` Funktion für alle Arten von DML- und DDL-Aktivitäten in Ihrer Datenbank verwenden. Es wird als Teil der Migrationstransaktion ausgeführt. Wenn die SQL fehlschlägt, schlägt die vollständige Migration fehl und ein Rollback wird durchgeführt.

"Update-Datenbank" in Ihrem Code ausführen

Anwendungen, die nicht in Entwicklungsumgebungen ausgeführt werden, erfordern häufig Datenbankaktualisierungen. Nachdem Sie den Befehl `Add-Migration` zum Erstellen Ihrer Datenbank-Patches verwendet haben, müssen Sie die Aktualisierungen in anderen Umgebungen und anschließend in der Testumgebung ausführen.

Die häufigsten Herausforderungen sind:

- Kein Visual Studio in Produktionsumgebungen installiert und
- Keine Verbindungen zu Verbindungs- / Kundenumgebungen in der Praxis erlaubt.

Eine Problemumgehung ist die folgende Codefolge, die prüft, ob Aktualisierungen ausgeführt werden, und führt sie der Reihe nach aus. Bitte sorgen Sie für korrekte Transaktionen und Ausnahmebehandlung, um sicherzustellen, dass im Fehlerfall keine Daten verloren gehen.

```
void UpdateDatabase(MyDbConfiguration configuration) {
   DbMigrator dbMigrator = new DbMigrator( configuration);
   if ( dbMigrator.GetPendingMigrations().Any() )
   {
      // there are pending migrations run the migration job
      dbMigrator.Update();
   }
}
```

Wo `MyDbConfiguration` Ihre Migrationseinrichtung irgendwo in Ihren Quellen ist:

```
public class MyDbConfiguration : DbMigrationsConfiguration<ApplicationDbContext>
```

Ursprünglicher Entity Framework Code Erste Migration Schritt für Schritt

1. Erstellen Sie eine Konsolenanwendung.
2. Installieren Sie das EntityFramework-Nuget-Paket, indem Sie `Install-Package EntityFramework` in der "Package Manager Console" `Install-Package EntityFramework`
3. Fügen Sie Ihre Verbindungszeichenfolge in der Datei app.config hinzu. Es ist wichtig, `providerName="System.Data.SqlClient"` in Ihre Verbindung aufzunehmen.
4. Erstellen Sie eine öffentliche Klasse nach Ihren Wünschen, z. B. " `Blog` ".
5. Erstellen Sie Ihre ContextClass, die von DbContext erben, wie " `BlogContext` ".
6. Definieren Sie eine Eigenschaft in Ihrem Kontext des DbSet-Typs. Etwas wie folgt:

```
public class Blog
{
    public int Id { get; set; }

    public string Name { get; set; }
}
public class BlogContext: DbContext
{
    public BlogContext(): base("name=Your_Connection_Name")
    {
    }
```

```
    public virtual DbSet<Blog> Blogs{ get; set; }
}
```

7. Es ist wichtig, den Verbindungsnamen im Konstruktor zu übergeben (hier Your_Connection_Name).
8. Führen Sie in der Package Manager Console den Befehl zum `Enable-Migration` . Dadurch wird ein Migrationsordner in Ihrem Projekt erstellt
9. Führen `Add-Migration Your_Arbitrary_Migraiton_Name` Befehl `Add-Migration Your_Arbitrary_Migraiton_Name` . Dies erstellt eine Migrationsklasse im Migrationsordner mit der Methode Up () und Down ()
10. Führen Sie den Befehl `Update-Database` , um eine Datenbank mit einer Blogtabelle zu erstellen

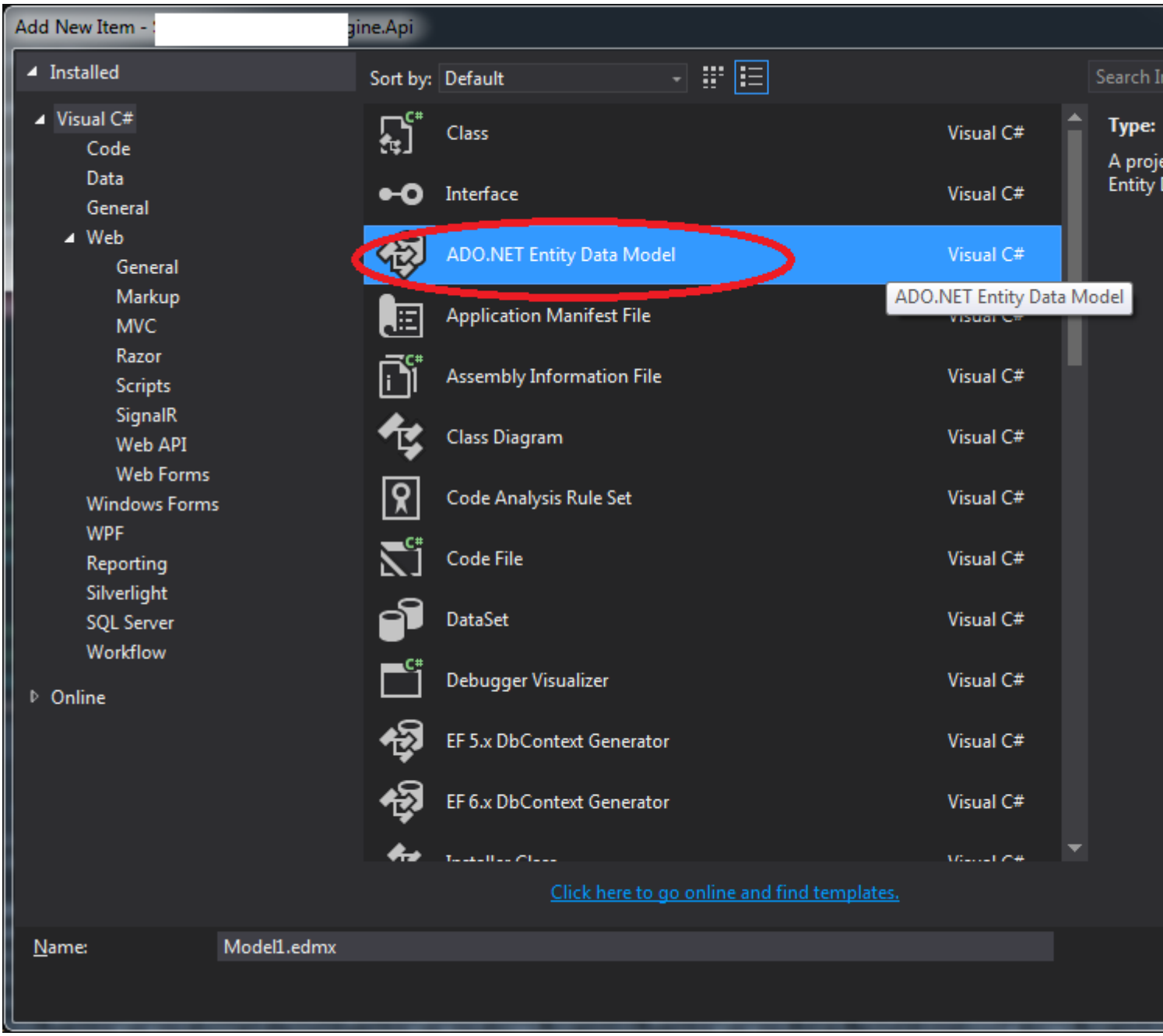
Erste Migrationen für Entity-Framework-Code online lesen: https://riptutorial.com/de/entity-framework/topic/7157/erste-migrationen-fur-entity-framework-code

Kapitel 13: Erste Modellgeneration der Datenbank

Examples

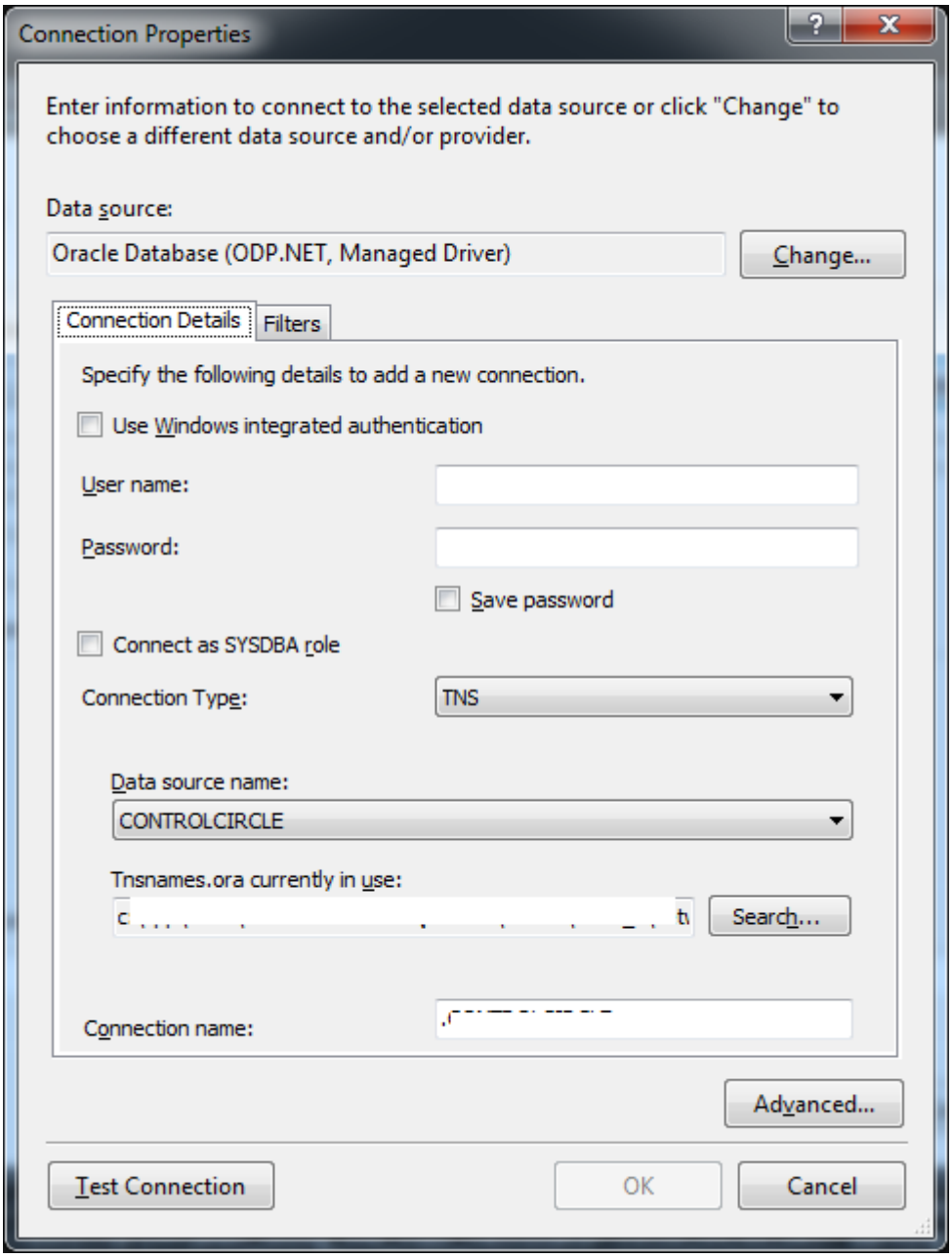
Modell aus Datenbank generieren

Gehen Sie in `Visual Studio` zu Ihrem `Project Solution Explorer` und klicken Sie auf `Project` , um das Modell mit der `rechten Maustaste` hinzuzufügen. Wählen Sie `ADO.NET Entity Data Model`



Wählen `Generate from database` und klicken Sie im nächsten Fenster auf `Next` , klicken Sie auf `New Connection...` und zeigen Sie auf die Datenbank, aus der Sie das Modell generieren möchten

(möglicherweise `MSSQL` , `MySQL` oder `Oracle`).



Nachdem Sie dies getan haben, klicken Sie auf Verbindung `Test Connection` zu sehen, ob Sie die Verbindung ordnungsgemäß konfiguriert haben (fahren Sie nicht weiter fort, falls die Verbindung hier fehlschlägt).

Klicken Sie auf `Next` und wählen Sie dann die gewünschten Optionen aus (z. B. Stil zum Generieren von Entitätsnamen oder zum Hinzufügen von Fremdschlüsseln).

Klicken Sie erneut auf `Next` . An diesem Punkt sollte das Modell aus der Datenbank generiert werden.

Hinzufügen von Datenanmerkungen zum generierten Modell

In der T4-Codegenerierungsstrategie, die von Entity Framework 5 und höher verwendet wird, werden die Attribute für die Anmerkungsdaten standardmäßig nicht berücksichtigt. Um bei jeder

Modellregenerierung Datenanmerkungen zu bestimmten Eigenschaften hinzuzufügen, öffnen Sie die in EDMX enthaltene Vorlagendatei (mit der Erweiterung `.tt`) und fügen Sie eine `using` Anweisung unter der `UsingDirectives` Methode wie `UsingDirectives` :

```
foreach (var entity in typeMapper.GetItemsToGenerate<EntityType>
(itemCollection))
{
    fileManager.StartNewFile(entity.Name + ".cs");
    BeginNamespace(code);
#>
<#=codeStringGenerator.UsingDirectives(inHeader: false)#>
using System.ComponentModel.DataAnnotations;  // --> add this line
```

Angenommen, die Vorlage sollte `KeyAttribute` das eine Primärschlüsseleigenschaft angibt. Um `KeyAttribute` automatisch während der Modellregenerierung einzufügen, suchen Sie einen Teil des Codes, der `codeStringGenerator.Property` enthält, wie `codeStringGenerator.Property` :

```
var simpleProperties = typeMapper.GetSimpleProperties(entity);
    if (simpleProperties.Any())
    {
        foreach (var edmProperty in simpleProperties)
        {
#>
    <#=codeStringGenerator.Property(edmProperty)#>
<#
        }
    }
```

Fügen Sie dann eine if-Bedingung ein, um die Schlüsseleigenschaft wie folgt zu überprüfen:

```
var simpleProperties = typeMapper.GetSimpleProperties(entity);
    if (simpleProperties.Any())
    {
        foreach (var edmProperty in simpleProperties)
        {
             if (ef.IsKey(edmProperty)) {
#>    [Key]
<#      }
#>
    <#=codeStringGenerator.Property(edmProperty)#>
<#
        }
    }
```

Wenn Sie die obigen Änderungen anwenden, wird für alle generierten Modellklassen nach der Aktualisierung des Modells aus der Datenbank `KeyAttribute` in der Primärschlüsseleigenschaft `KeyAttribute` .

Vor

```
using System;

public class Example
{
```

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

Nach dem

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

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

Erste Modellgeneration der Datenbank online lesen: https://riptutorial.com/de/entity-framework/topic/4414/erste-modellgeneration-der-datenbank

Kapitel 14: Erweiterte Mapping-Szenarien: Entitätsaufteilung, Aufteilung der Tabelle

Einführung

So konfigurieren Sie Ihr EF-Modell zur Unterstützung der Entitätsaufteilung oder der Aufteilung von Tabellen.

Examples

Aufteilung der Entitäten

Nehmen wir an, Sie haben eine Entitätsklasse wie diese:

```
public class Person
{
   public int PersonId { get; set; }
   public string Name { get; set; }
   public string ZipCode { get; set; }
   public string City { get; set; }
   public string AddressLine { get; set; }
}
```

Nehmen wir an, Sie möchten diese Entität "Person" in zwei Tabellen abbilden - eine mit der Personen-ID und dem Namen und eine andere mit den Adressdetails. Natürlich benötigen Sie auch hier die PersonId, um zu ermitteln, zu welcher Person die Adresse gehört. Im Grunde möchten Sie die Entität in zwei (oder sogar mehr) Teile aufteilen. Daher der Name Entity Splitting. Sie können dies tun, indem Sie jede Eigenschaft einer anderen Tabelle zuordnen:

```
public class MyDemoContext : DbContext
{
  public DbSet<Person> Products { get; set; }

  protected override void OnModelCreating(DbModelBuilder modelBuilder)
  {
      modelBuilder.Entity<Person>().Map(m =>
      {
         m.Properties(t => new { t.PersonId, t.Name });
         m.ToTable("People");
      }).Map(m =>
      {
        m.Properties(t => new { t.PersonId, t.AddressLine, t.City, t.ZipCode });
        m.ToTable("PersonDetails");
      });
  }
}
```

Dadurch werden zwei Tabellen erstellt: Personen und Personendetails. Person hat zwei Felder, PersonId und Name, PersonDetails hat vier Spalten: PersonId, Adresszeile, Ort und Postleitzahl.

In People ist PersonId der Primärschlüssel. In PersonDetails ist der Primärschlüssel auch PersonId, es ist jedoch auch ein Fremdschlüssel, der auf PersonId in der Personentabelle verweist.

Wenn Sie das People-DbSet abfragen, führt EF einen Join für die PersonIds aus, um die Daten aus beiden Tabellen zum Abrufen der Entitäten zu erhalten.

Sie können auch den Namen der Spalten ändern:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
  modelBuilder.Entity<Person>().Map(m =>
  {
    m.Properties(t => new { t.PersonId });
    m.Property(t => t.Name).HasColumnName("PersonName");
    m.ToTable("People");
  }).Map(m =>
  {
    m.Property(t => t.PersonId).HasColumnName("ProprietorId");
    m.Properties(t => new { t.AddressLine, t.City, t.ZipCode });
    m.ToTable("PersonDetails");
  });
}
```

Dadurch wird dieselbe Tabellenstruktur erstellt. In der People-Tabelle wird jedoch eine PersonName-Spalte anstelle der Name-Spalte und in der PersonDetails-Tabelle eine ProprietorId anstelle der PersonId-Spalte angezeigt.

Tischaufteilung

Angenommen, Sie möchten das Gegenteil der Entitätsaufteilung tun: Anstatt eine Entität in zwei Tabellen abzubilden, möchten Sie eine Tabelle in zwei Entitäten abbilden. Dies wird als Tabellenaufteilung bezeichnet. Angenommen, Sie haben eine Tabelle mit fünf Spalten: PersonId, Name, Adresszeile, Ort, Postleitzahl, wobei PersonId der Primärschlüssel ist. Und dann möchten Sie ein EF-Modell wie folgt erstellen:

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

public class Address
{
  public string ZipCode { get; set; }
  public string City { get; set; }
  public string AddressLine { get; set; }
  public int PersonId { get; set; }
  public Person Person { get; set; }
}
```

Eines springt sofort heraus: In Address gibt es keine AddressId. Das liegt daran, dass die beiden Entitäten derselben Tabelle zugeordnet sind und daher auch denselben Primärschlüssel haben

müssen. Wenn Sie Tisch teilen, müssen Sie sich nur damit beschäftigen. Neben der Aufteilung von Tabellen müssen Sie also auch die Entität "Address" konfigurieren und den Primärschlüssel angeben. Und so geht's:

```
public class MyDemoContext : DbContext
{
  public DbSet<Person> Products { get; set; }
  public DbSet<Address> Addresses { get; set; }

  protected override void OnModelCreating(DbModelBuilder modelBuilder)
  {
     modelBuilder.Entity<Address>().HasKey(t => t.PersonId);
     modelBuilder.Entity<Person>().HasRequired(t => t.Address)
                                  .WithRequiredPrincipal(t => t.Person);

     modelBuilder.Entity<Person>().Map(m => m.ToTable("People"));
     modelBuilder.Entity<Address>().Map(m => m.ToTable("People"));
  }
}
```

Erweiterte Mapping-Szenarien: Entitätsaufteilung, Aufteilung der Tabelle online lesen: https://riptutorial.com/de/entity-framework/topic/9362/erweiterte-mapping-szenarien--entitatsaufteilung--aufteilung-der-tabelle

Kapitel 15: Komplexe Typen

Examples

Erste komplexe Typen des Codes

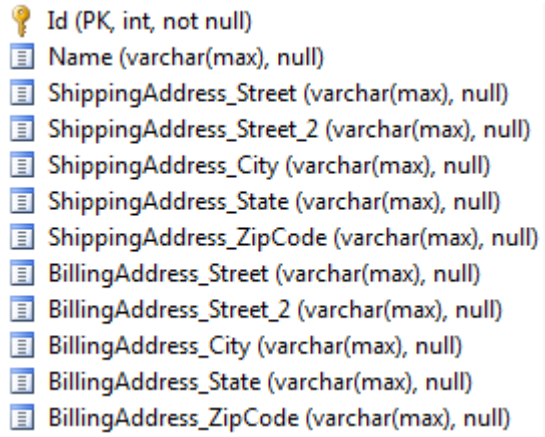
Mit einem komplexen Typ können Sie ausgewählte Felder einer Datenbanktabelle einem einzigen Typ zuordnen, der dem Haupttyp untergeordnet ist.

```
[ComplexType]
public class Address
{
    public string Street { get; set; }
    public string Street_2 { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string ZipCode { get; set; }
}
```

Dieser komplexe Typ kann dann in mehreren Entitätstypen verwendet werden. Es kann sogar mehrmals in demselben Entitätstyp verwendet werden.

```
public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
    ...
    public Address ShippingAddress { get; set; }
    public Address BillingAddress { get; set; }
}
```

Dieser Entitätstyp würde dann in einer Tabelle in der Datenbank gespeichert, die ungefähr so aussieht.



In diesem Fall wäre natürlich eine 1: n-Zuordnung (Kundenadresse) das bevorzugte Modell, aber das Beispiel zeigt, wie komplexe Typen verwendet werden können.

Komplexe Typen online lesen: https://riptutorial.com/de/entity-framework/topic/5527/komplexe-

typen

Kapitel 16: Modellfesseln

Examples

Eins-zu-Viele-Beziehungen

UserType gehört zu vielen Benutzern <-> Benutzer haben einen UserType

Einwegnavigationseigenschaft mit erforderlich

```
public class UserType
{
    public int UserTypeId {get; set;}
}
public class User
{
    public int UserId {get; set;}
    public int UserTypeId {get; set;}
    public virtual UserType UserType {get; set;}
}

Entity<User>().HasRequired(u => u.UserType).WithMany().HasForeignKey(u => u.UserTypeId);
```

`Nullable` mit optionalem (Fremdschlüssel muss `Nullable` Typ " `Nullable` " sein)

```
public class UserType
{
    public int UserTypeId {get; set;}
}
public class User
{
    public int UserId {get; set;}
    public int? UserTypeId {get; set;}
    public virtual UserType UserType {get; set;}
}

Entity<User>().HasOptional(u => u.UserType).WithMany().HasForeignKey(u => u.UserTypeId);
```

Zweiwege-Navigationseigenschaft mit (Erforderliche / optionale Änderung der Fremdschlüsseleigenschaft nach Bedarf)

```
public class UserType
{
    public int UserTypeId {get; set;}
    public virtual ICollection<User> Users {get; set;}
}
public class User
{
    public int UserId {get; set;}
    public int UserTypeId {get; set;}
    public virtual UserType UserType {get; set;}
}
```

Erforderlich

```
Entity<User>().HasRequired(u => u.UserType).WithMany(ut => ut.Users).HasForeignKey(u =>
u.UserTypeId);
```

Wahlweise

```
Entity<User>().HasOptional(u => u.UserType).WithMany(ut => ut.Users).HasForeignKey(u =>
u.UserTypeId);
```

Modellfesseln online lesen: https://riptutorial.com/de/entity-framework/topic/4528/modellfesseln

Kapitel 17: Optimierungstechniken in EF

Examples

AsNoTracking verwenden

Schlechtes Beispiel:

```
var location =  dbContext.Location
                     .Where(l => l.Location.ID == location_ID)
                     .SingleOrDefault();

return location;
```

Da der obige Code lediglich eine Entität zurückgibt, ohne sie zu ändern oder hinzuzufügen, können wir die Kosten für die Nachverfolgung vermeiden.

Gutes Beispiel:

```
var location =  dbContext.Location.AsNoTracking()
                     .Where(l => l.Location.ID == location_ID)
                     .SingleOrDefault();

return location;
```

Wenn wir die Funktion `AsNoTracking()` verwenden, `AsNoTracking()` wir Entity Framework explizit mit, dass die Entitäten nicht durch den Kontext verfolgt werden. Dies kann besonders nützlich sein, wenn große Datenmengen aus Ihrem Datenspeicher abgerufen werden. Wenn Sie jedoch Änderungen an nicht nachverfolgten Entitäten vornehmen möchten, müssen Sie diese vor dem Aufruf von `SaveChanges` .

Nur benötigte Daten werden geladen

Ein häufig beim Code auftretendes Problem ist das Laden aller Daten. Dadurch wird die Belastung des Servers erheblich erhöht.

Angenommen, ich habe ein Modell mit dem Namen "location", das 10 Felder enthält, aber nicht alle Felder werden gleichzeitig benötigt. Angenommen, ich möchte nur den Parameter 'LocationName' dieses Modells.

Schlechtes Beispiel

```
var location =  dbContext.Location.AsNoTracking()
                        .Where(l => l.Location.ID == location_ID)
                        .SingleOrDefault();

return location.Name;
```

Gutes Beispiel

```
var location =  dbContext.Location
                          .Where(l => l.Location.ID == location_ID)
                          .Select(l => l.LocationName);
                          .SingleOrDefault();

return location;
```

Der Code im "guten Beispiel" wird nur "LocationName" und nichts anderes abrufen.

Beachten Sie, dass `AsNoTracking()` nicht erforderlich ist, da in diesem Beispiel keine Entität `AsNoTracking()` wird. Es gibt sowieso nichts zu verfolgen.

Weitere Felder mit anonymen Typen abrufen

```
var location = dbContext.Location
                    .Where(l => l.Location.ID == location_ID)
                    .Select(l => new { Name = l.LocationName, Area = l.LocationArea })
                    .SingleOrDefault();

return location.Name + " has an area of " + location.Area;
```

Wie im vorherigen Beispiel werden nur die Felder 'LocationName' und 'LocationArea' aus der Datenbank abgerufen. Der anonyme Typ kann so viele Werte enthalten, wie Sie möchten.

Führen Sie, wenn möglich, Abfragen in der Datenbank aus, nicht im Speicher.

Angenommen, wir wollen zählen, wie viele Grafschaften es in Texas gibt:

```
var counties = dbContext.States.Single(s => s.Code == "tx").Counties.Count();
```

Die Abfrage ist korrekt, aber ineffizient. `States.Single(…)` lädt einen Status aus der Datenbank. Als Nächstes lädt `Counties` alle 254 Counties mit allen Feldern in einer zweiten Abfrage. `.Count()` wird dann *in* der geladenen `Counties` Sammlung *im Speicher ausgeführt* .
Wir haben viele Daten geladen, die wir nicht benötigen, und wir können es besser machen:

```
var counties = dbContext.Counties.Count(c => c.State.Code == "tx");
```

Hier führen wir nur eine Abfrage aus, die in SQL in eine Anzahl und einen Join übersetzt wird. Wir geben nur die Anzahl aus der Datenbank zurück - wir haben zurückgegebene Zeilen, Felder und die Erstellung von Objekten gespeichert.

Der Abfragetyp lässt sich anhand des Auflistungstyps leicht erkennen: `IQueryable<T>` vs. `IEnumerable<T>` .

Führen Sie mehrere Abfragen asynchron und parallel aus

Wenn Sie asynchrone Abfragen verwenden, können Sie mehrere Abfragen gleichzeitig ausführen, jedoch nicht in demselben Kontext. Wenn die Ausführungszeit einer Abfrage 10 Sekunden beträgt,

beträgt die Zeit für das fehlerhafte Beispiel 20 Sekunden, während die Zeit für das gute Beispiel 10 Sekunden beträgt.

Schlechtes Beispiel

```
IEnumerable<TResult1> result1;
IEnumerable<TResult2> result2;

using(var context = new Context())
{
    result1 = await context.Set<TResult1>().ToListAsync().ConfigureAwait(false);
    result2 = await context.Set<TResult1>().ToListAsync().ConfigureAwait(false);
}
```

Gutes Beispiel

```
public async Task<IEnumerable<TResult>> GetResult<TResult>()
{
    using(var context = new Context())
    {
        return await context.Set<TResult1>().ToListAsync().ConfigureAwait(false);
    }
}



IEnumerable<TResult1> result1;
IEnumerable<TResult2> result2;

var result1Task = GetResult<TResult1>();
var result2Task = GetResult<TResult2>();

await Task.WhenAll(result1Task, result2Task).ConfigureAwait(false);

var result1 = result1Task.Result;
var result2 = result2Task.Result;
```

Deaktivieren Sie die Änderungsnachverfolgung und die Proxy-Generierung

Wenn Sie nur Daten abrufen möchten, aber nichts ändern möchten, können Sie die Änderungsnachverfolgung und die Proxy-Erstellung deaktivieren. Dies verbessert Ihre Leistung und verhindert auch ein verzögertes Laden.

Schlechtes Beispiel:

```
using(var context = new Context())
{
    return await context.Set<MyEntity>().ToListAsync().ConfigureAwait(false);
}
```

Gutes Beispiel:

```
using(var context = new Context())
{
    context.Configuration.AutoDetectChangesEnabled = false;
    context.Configuration.ProxyCreationEnabled = false;

    return await context.Set<MyEntity>().ToListAsync().ConfigureAwait(false);
}
```

Es ist besonders üblich, diese Einstellungen im Konstruktor Ihres Kontexts zu deaktivieren, insbesondere wenn Sie möchten, dass diese in Ihrer Lösung angezeigt werden:

```
public class MyContext : DbContext
{
    public MyContext()
        : base("MyContext")
    {
        Configuration.AutoDetectChangesEnabled = false;
        Configuration.ProxyCreationEnabled = false;
    }

    //snip
}
```

Mit Stub-Entitäten arbeiten

Nehmen wir an, wir haben `Product` und `Category` in einer Viele-zu-Viele-Beziehung:

```
public class Product
{
    public Product()
    {
        Categories = new HashSet<Category>();
    }
    public int ProductId { get; set; }
    public string ProductName { get; set; }
    public virtual ICollection<Category> Categories { get; private set; }
}

public class Category
{
    public Category()
    {
        Products = new HashSet<Product>();
    }
    public int CategoryId { get; set; }
    public string CategoryName { get; set; }
    public virtual ICollection<Product> Products { get; set; }
}
```

Wenn Sie einem `Product` eine `Category` hinzufügen möchten, müssen Sie das Produkt laden und die Kategorie zu seinen `Categories` hinzufügen. Beispiel:

Schlechtes Beispiel:

```
var product = db.Products.Find(1);
```

```
var category = db.Categories.Find(2);
product.Categories.Add(category);
db.SaveChanges();
```

(wobei `db` eine `DbContext` Unterklasse ist).

Dadurch wird ein Datensatz in der Junction-Tabelle zwischen `Product` und `Category` . Diese Tabelle enthält jedoch nur zwei `Id` Werte. Es ist eine Verschwendung von Ressourcen, zwei vollständige Entitäten zu laden, um einen kleinen Datensatz zu erstellen.

Eine effizientere Methode ist die Verwendung von *Stub-Entities* , dh Entity-Objekten, die im Speicher erstellt werden und nur das absolute Minimum an Daten enthalten, normalerweise nur einen `Id` Wert. So sieht es aus:

Gutes Beispiel:

```
// Create two stub entities
var product = new Product { ProductId = 1 };
var category = new Category { CategoryId = 2 };

// Attach the stub entities to the context
db.Entry(product).State = System.Data.Entity.EntityState.Unchanged;
db.Entry(category).State = System.Data.Entity.EntityState.Unchanged;

product.Categories.Add(category);
db.SaveChanges();
```

Das Endergebnis ist dasselbe, aber es werden zwei Roundtrips zur Datenbank vermieden.

Duplikate vermeiden

Wenn Sie prüfen möchten, ob die Verknüpfung bereits vorhanden ist, genügt eine billige Abfrage. Zum Beispiel:

```
var exists = db.Categories.Any(c => c.Id == 1 && c.Products.Any(p => p.Id == 14));
```

Auch hier werden keine vollständigen Elemente in den Speicher geladen. Es fragt effektiv die Junction-Tabelle ab und gibt nur einen Boolean zurück.

Optimierungstechniken in EF online lesen: https://riptutorial.com/de/entity-framework/topic/2714/optimierungstechniken-in-ef

Kapitel 18: Tracking vs. No-Tracking

Bemerkungen

Mit dem Verfolgungsverhalten wird gesteuert, ob Entity Framework Informationen zu einer Entitätsinstanz in seinem Änderungs-Tracker speichert. Wenn eine Entität verfolgt wird, werden alle in der Entität festgestellten Änderungen während `SaveChanges()` in der Datenbank `SaveChanges()` .

Examples

Abfragen verfolgen

- Abfragen, die Entitätstypen zurückgeben, werden standardmäßig **verfolgt**
- Das bedeutet, dass Sie Änderungen an diesen Entitätsinstanzen vornehmen können und diese Änderungen durch `SaveChanges()`

Beispiel:

- Die Änderung des `book` Bewertung wird nachgewiesen werden , und beharrte auf die Datenbank während `SaveChanges()` .

```
using (var context = new BookContext())
{
  var book = context.Books.FirstOrDefault(b => b.BookId == 1);
  book.Rating = 5;
  context.SaveChanges();
}
```

No-Tracking-Abfragen

- Keine Verfolgungsabfragen sind nützlich, wenn die Ergebnisse in einem `read-only` Szenario verwendet werden
- Sie sind `quicker to execute` da keine Änderungsnachverfolgungsinformationen eingerichtet werden müssen

Beispiel:

```
using (var context = new BookContext())
{
    var books = context.Books.AsNoTracking().ToList();
}
```

Mit EF Core 1.0 können Sie auch das Standardverfolgungsverhalten auf `context instance` ändern.

Beispiel:

```
using (var context = new BookContext())
{
    context.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;

    var books = context.Books.ToList();
}
```

Tracking und Projektionen

- Auch wenn der Ergebnistyp der Abfrage kein Entitätstyp ist, wird das Ergebnis, wenn `contains entity tracked by default` weiterhin `tracked by default`

Beispiel:

- In der folgenden Abfrage, die einen `anonymous type` zurückgibt, werden die Instanzen von `Book` in der Ergebnismenge `will be tracked`

```
using (var context = new BookContext())
{
   var book = context.Books.Select(b => new { Book = b, Authors = b.Authors.Count() });
}
```

- Wenn die Ergebnismenge `does not` keine enthalten `entity` - Typen, dann `no tracking` durchgeführt

Beispiel:

- In der folgenden Abfrage, die einen `anonymous type` mit einigen Werten der Entität zurückgibt (jedoch `no instances` des tatsächlichen `entity`), wird **keine Nachverfolgung** durchgeführt.

```
using (var context = new BookContext())
{
   var book = context.Books.Select(b => new { Id = b.BookId, PublishedDate = b.Date });
}
```

Tracking vs. No-Tracking online lesen: https://riptutorial.com/de/entity-framework/topic/6836/tracking-vs--no-tracking

Kapitel 19: Transaktionen

Examples

Database.BeginTransaction ()

Mehrere Vorgänge können für eine einzelne Transaktion ausgeführt werden, sodass Änderungen rückgängig gemacht werden können, wenn einer der Vorgänge fehlschlägt.

```
using (var context = new PlanetContext())
{
    using (var transaction = context.Database.BeginTransaction())
    {
        try
        {
            //Lets assume this works
            var jupiter = new Planet { Name = "Jupiter" };
            context.Planets.Add(jupiter);
            context.SaveChanges();

            //And then this will throw an exception
            var neptune = new Planet { Name = "Neptune" };
            context.Planets.Add(neptune);
            context.SaveChanges();

            //Without this line, no changes will get applied to the database
            transaction.Commit();
        }
        catch (Exception ex)
        {
            //There is no need to call transaction.Rollback() here as the transaction object
            //will go out of scope and disposing will roll back automatically
        }
    }
}
```

Beachten Sie, dass es eine Konvention von Entwicklern sein kann, `transaction.Rollback()` explizit aufzurufen, da der Code dadurch selbsterklärender wird. Außerdem gibt es *möglicherweise* (weniger bekannte) Abfrageanbieter für Entity Framework, die `Dipsose` korrekt implementieren. `Dipsose` würde auch einen expliziten `transaction.Rollback()` erfordern.

Transaktionen online lesen: https://riptutorial.com/de/entity-framework/topic/4944/transaktionen

Kapitel 20: Vererbung mit EntityFramework (Code First)

Examples

Tabelle pro Hierarchie

Dieser Ansatz generiert eine Tabelle in der Datenbank, die die gesamte Vererbungsstruktur darstellt.

Beispiel:

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

public class Employee : Person
{
    public DateTime AdmissionDate { get; set; }
    public string JobDescription { get; set; }
}

public class Customer : Person
{
    public DateTime LastPurchaseDate { get; set; }
    public int TotalVisits { get; set; }
}

// On DbContext
public DbSet<Person> People { get; set; }
public DbSet<Employee> Employees { get; set; }
public DbSet<Customer> Customers { get; set; }
```

Die generierte Tabelle wird sein:

Tabelle: Personen Felder: ID Name Geburtsdatum Discrimitator ZulassungDatum JobDescription LastPurchaseDate Gesamtbesichtigungen

Wo 'Discriminator' den Namen der Unterklasse für die Vererbung enthält und 'AdmissionDate', 'JobDescription', 'LastPurchaseDate', 'TotalVisits' sind nullbar.

Vorteile

- Bessere Leistung, da keine Joins erforderlich sind, obwohl die Datenbank für viele Spalten viele Paging-Vorgänge erfordern kann.
- Einfach zu bedienen und zu erstellen
- Einfach weitere Unterklassen und Felder hinzufügen

Nachteile

- Verstößt gegen die 3. Normalform Wikipedia: Dritte Normalform
- Erzeugt viele nullfähige Felder

Tabelle pro Typ

Dieser Ansatz generiert (n + 1) Tabellen in der Datenbank, um die gesamte Vererbungsstruktur darzustellen, wobei n die Anzahl der Unterklassen ist.

Wie man:

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

[Table("Employees")]
public class Employee : Person
{
    public DateTime AdmissionDate { get; set; }
    public string JobDescription { get; set; }
}

[Table("Customers")]
public class Customer : Person
{
    public DateTime LastPurchaseDate { get; set; }
    public int TotalVisits { get; set; }
}

// On DbContext
public DbSet<Person> People { get; set; }
public DbSet<Employee> Employees { get; set; }
public DbSet<Customer> Customers { get; set; }
```

Die generierte Tabelle wird sein:

Tabelle: Personenfelder: Id-Name Geburtsdatum

Tabelle: Mitarbeiterfelder: PersonId AdmissionDate JobDescription

Tabelle: Kunden: Felder: PersonId LastPurchaseDate Gesamtbesichtigungen

Wobei 'PersonId' in allen Tabellen ein Primärschlüssel und eine Einschränkung für People.Id ist

Vorteile

- Normalisierte Tabellen
- Spalten und Unterklassen einfach hinzuzufügen
- Keine nullfähigen Spalten

Nachteile

- Join ist erforderlich, um die Daten abzurufen
- Unterklassenschluss ist teurer

Vererbung mit EntityFramework (Code First) online lesen: https://riptutorial.com/de/entity-framework/topic/7715/vererbung-mit-entityframework--code-first-

Kapitel 21: Verwandte Entitäten laden

Bemerkungen

Wenn Modelle richtig miteinander verbunden sind, können Sie mit EntityFramework problemlos verwandte Daten laden. Sie haben drei Optionen zur Auswahl: *faules Laden* , *eifriges Laden* und *explizites Laden* .

In Beispielen verwendete Modelle:

```
public class Company
{
    public int Id { get; set; }
    public string FullName { get; set; }
    public string ShortName { get; set; }

    // Navigation properties
    public virtual Person Founder { get; set; }
    public virtual ICollection<Address> Addresses { get; set; }
}

public class Address
{
    public int Id { get; set; }
    public int CompanyId { get; set; }
    public int CountryId { get; set; }
    public int CityId { get; set; }
    public string Street { get; set; }

    // Navigation properties
    public virtual Company Company { get; set; }
    public virtual Country Country { get; set; }
    public virtual City City { get; set; }
}
```

Examples

Faules Laden

Lazy Loading ist standardmäßig aktiviert. Lazy Loading wird durch das Erstellen abgeleiteter Proxy-Klassen und das Überschreiben von virtuellen Navigationsrechten erreicht. Beim erstmaligen Zugriff auf die Eigenschaft wird faul geladen.

```
int companyId = ...;
Company company = context.Companies
    .First(m => m.Id == companyId);
Person founder = company.Founder; // Founder is loaded
foreach (Address address in company.Addresses)
{
    // Address details are loaded one by one.
}
```

Um das Lazy-Laden für bestimmte Navigationseigenschaften zu deaktivieren, entfernen Sie einfach das virtuelle Schlüsselwort aus der Deklaration der Eigenschaften:

```
public Person Founder { get; set; } // "virtual" keyword has been removed
```

Wenn Sie das Lazy-Laden vollständig deaktivieren möchten, müssen Sie die Konfiguration beispielsweise im *Kontext-Konstruktor* ändern:

```
public class MyContext : DbContext
{
    public MyContext(): base("Name=ConnectionString")
    {
        this.Configuration.LazyLoadingEnabled = false;
    }
}
```

Hinweis: Denken Sie daran, Lazy Loading *auszuschalten,* wenn Sie die Serialisierung verwenden. Da die Serialisierer auf jede Eigenschaft zugreifen, laden Sie sie alle aus der Datenbank. Darüber hinaus können Sie eine Schleife zwischen den Navigationseigenschaften ausführen.

Eifriges Laden

Durch das eifrige Laden können Sie alle benötigten Objekte gleichzeitig laden. Wenn Sie es vorziehen, alle Ihre Entitäten mit einem einzigen Datenbankaufruf zu bearbeiten, ist das *Laden mit Eager der richtige* Weg. Sie können auch mehrere Ebenen laden.

Sie haben *zwei Möglichkeiten* , verwandte Entitäten zu laden. Sie können entweder die *stark typisierte* oder die Überladung der *Zeichenfolge* der *Include*- Methode wählen.

Stark getippt

```
// Load one company with founder and address details
int companyId = ...;
Company company = context.Companies
    .Include(m => m.Founder)
    .Include(m => m.Addresses)
    .SingleOrDefault(m => m.Id == companyId);

// Load 5 companies with address details, also retrieve country and city
// information of addresses
List<Company> companies = context.Companies
    .Include(m => m.Addresses.Select(a => a.Country));
    .Include(m => m.Addresses.Select(a => a.City))
    .Take(5).ToList();
```

Diese Methode ist seit Entity Framework 4.1 verfügbar. Stellen Sie sicher, dass Sie über die Referenz verfügen, die `using System.Data.Entity;` einstellen.

String überladen.

```
// Load one company with founder and address details
int companyId = ...;
Company company = context.Companies
    .Include("Founder")
    .Include("Addresses")
    .SingleOrDefault(m => m.Id == companyId);

// Load 5 companies with address details, also retrieve country and city
// information of addresses
List<Company> companies = context.Companies
    .Include("Addresses.Country");
    .Include("Addresses.City"))
    .Take(5).ToList();
```

Explizites Laden

Nach dem Deaktivieren des *Lazy-Ladens* können Sie Entities faul laden, indem Sie die *Load*-Methode für Einträge explizit aufrufen. *Referenz* wird zum Laden einzelner Navigationseigenschaften verwendet, während *Collection* zum Abrufen von Sammlungen verwendet wird.

```
Company company = context.Companies.FirstOrDefault();
// Load founder
context.Entry(company).Reference(m => m.Founder).Load();
// Load addresses
context.Entry(company).Collection(m => m.Addresses).Load();
```

Wie beim *Eager Loading können* Sie Überladungen der obigen Methoden verwenden, um Berechtigungen nach ihren Namen zu laden:

```
Company company = context.Companies.FirstOrDefault();
// Load founder
context.Entry(company).Reference("Founder").Load();
// Load addresses
context.Entry(company).Collection("Addresses").Load();
```

Verwandte Entitäten filtern

Mit der *Query*- Methode können wir geladene Entitäten filtern:

```
Company company = context.Companies.FirstOrDefault();
// Load addresses which are in Baku
context.Entry(company)
    .Collection(m => m.Addresses)
    .Query()
    .Where(a => a.City.Name == "Baku")
    .Load();
```

Projektionsabfragen

Benötigt man verwandte Daten in einem denormalisierten Typ oder zB nur eine Teilmenge von Spalten, kann man Projektionsabfragen verwenden. Wenn es keinen Grund gibt, einen

zusätzlichen Typ zu verwenden, besteht die Möglichkeit, die Werte mit einem anonymen Typ zu verknüpfen .

```
var dbContext = new MyDbContext();
var denormalizedType = from company in dbContext.Company
                       where company.Name == "MyFavoriteCompany"
                       join founder in dbContext.Founder
                       on company.FounderId equals founder.Id
                       select new
                       {
                           CompanyName = company.Name,
                           CompanyId = company.Id,
                           FounderName = founder.Name,
                           FounderId = founder.Id
                       };
```

Oder mit der Abfragesyntax:

```
var dbContext = new MyDbContext();
var denormalizedType = dbContext.Company
                       .Join(dbContext.Founder,
                                   c => c.FounderId,
                                   f => f.Id ,
                                   (c, f) => new
                                   {
                                       CompanyName = c.Name,
                                       CompanyId = c.Id,
                                       FounderName = f.Name,
                                       FounderId = f.Id
                                   })
                       .Select(cf => cf);
```

Verwandte Entitäten laden online lesen: https://riptutorial.com/de/entity-framework/topic/4678/verwandte-entitaten-laden

Kapitel 22: Zuordnungsbeziehung mit Entity Framework Code First: One-to-Many und Many-to-Many

Einführung

In diesem Thema wird erläutert, wie Sie mithilfe von Entity Framework Code First Eins-zu-Viele- und Viele-zu-Viele-Beziehungen zuordnen können.

Examples

Eins-zu-Viele-Mapping

Nehmen wir also an, Sie haben zwei verschiedene Entitäten:

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

public class Car
{
   public int CarId { get; set; }
   public string LicensePlate { get; set; }
}

public class MyDemoContext : DbContext
{
   public DbSet<Person> People { get; set; }
   public DbSet<Car> Cars { get; set; }
}
```

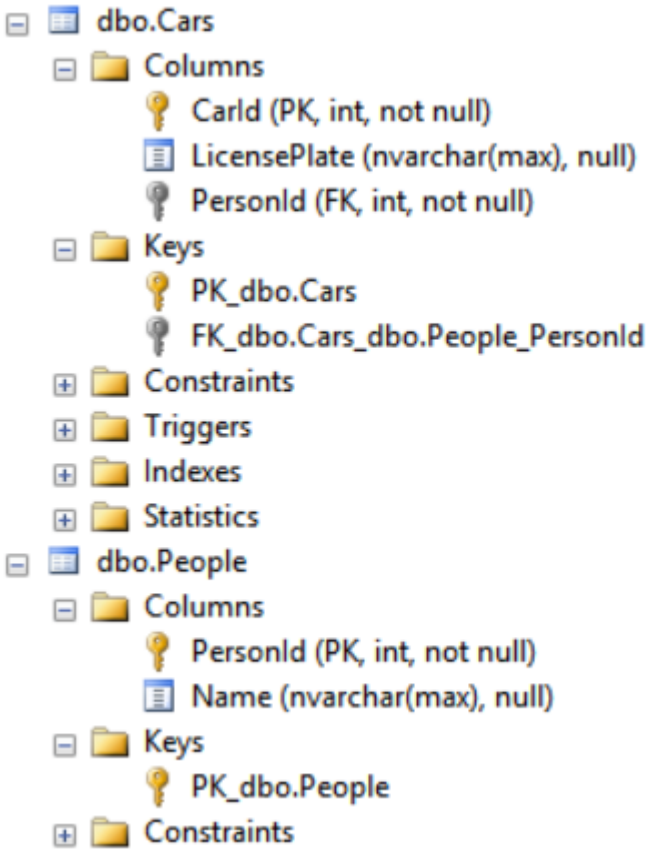
Und Sie möchten eine Eins-zu-Viele-Beziehung zwischen ihnen aufbauen, dh eine Person kann null, ein oder mehrere Autos haben und ein Auto gehört genau zu einer Person. Jede Beziehung ist bidirektional. Wenn eine Person ein Auto hat, gehört das Auto zu dieser Person.

Dazu modifizieren Sie einfach Ihre Modellklassen:

```
public class Person
{
   public int PersonId { get; set; }
   public string Name { get; set; }
   public virtual ICollection<Car> Cars { get; set; } // don't forget to initialize (use
HashSet)
}

public class Car
```

```
{
   public int CarId { get; set; }
   public string LicensePlate { get; set; }
   public int PersonId { get; set; }
   public virtual Person Person { get; set; }
}
```

Und das ist es :) Du hast bereits eine Beziehung aufgebaut. In der Datenbank wird dies natürlich mit Fremdschlüsseln dargestellt.



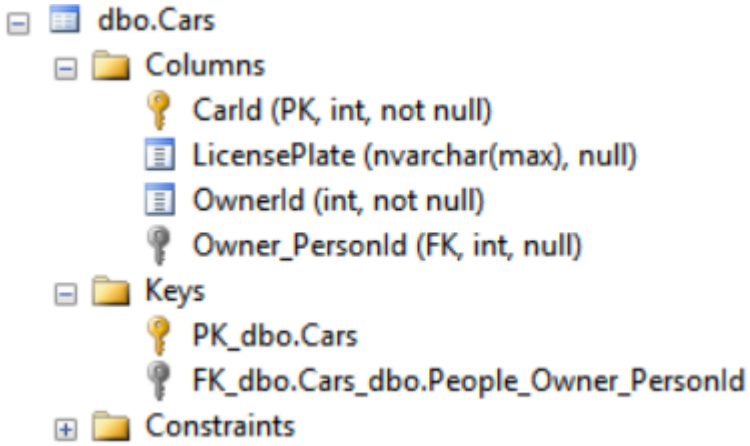
Eins-zu-Viele-Karten: gegen die Konvention

Im letzten Beispiel können Sie sehen, dass EF herausfindet, in welcher Spalte sich der Fremdschlüssel befindet und wohin er verweisen soll. Wie? Mit Konventionen. Wenn Sie eine Eigenschaft des Typs `Person` dem Namen `Person` mit einer `PersonId` Eigenschaft haben, kommt EF zu dem Schluss, dass `PersonId` ein Fremdschlüssel ist, und verweist auf den Primärschlüssel der durch den Typ `Person` dargestellten Tabelle.

Aber was , wenn Sie waren *PersonId* zu *ownerID* und *Person* an *Vermieter* im **Autotyp** ändern?

```
public class Car
{
   public int CarId { get; set; }
   public string LicensePlate { get; set; }
   public int OwnerId { get; set; }
   public virtual Person Owner { get; set; }
}
```

Nun, leider reichen die Konventionen in diesem Fall nicht aus, um das richtige DB-Schema zu

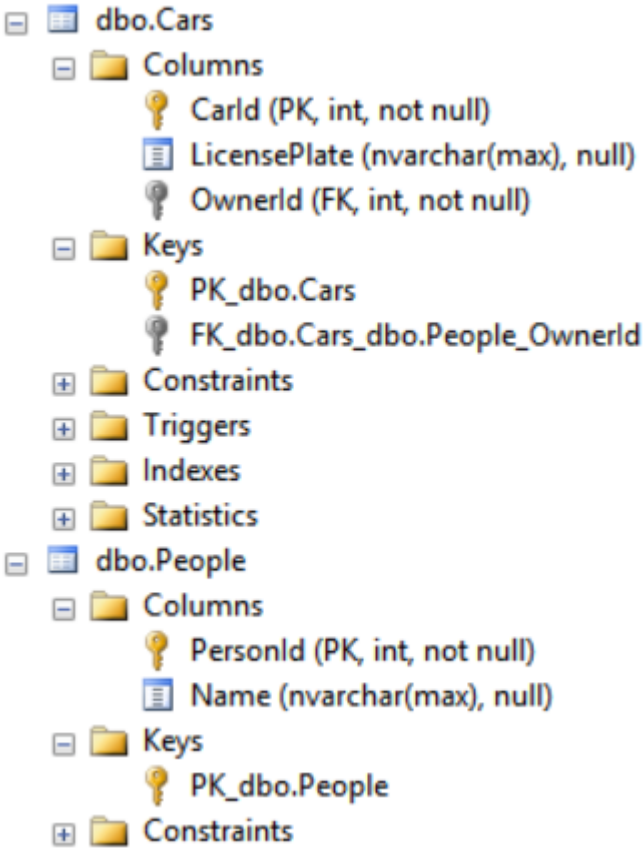


erzeugen:

Keine Bange; Sie können EF mit einigen Hinweisen zu Ihren Beziehungen und Schlüsseln im Modell helfen. Einfach Ihren konfigurieren `Car` - Typ die verwenden `OwnerId` Eigenschaft als FK. Erstellen Sie eine Entitätstypkonfiguration und wenden Sie sie in `OnModelCreating()` :

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
     this.HasRequired(c => c.Owner).WithMany(p => p.Cars).HasForeignKey(c => c.OwnerId);
  }
}
```

Dies besagt im Wesentlichen, dass `Car` eine erforderliche Eigenschaft besitzt, `Owner` (*HasRequired ()*), und in der Art des `Owner` wird die `Cars` Eigenschaft verwendet, um auf die *Autoentitäten (WithMany ()*) zurückzugreifen. Schließlich wird die Eigenschaft angegeben, die den Fremdschlüssel darstellt (*HasForeignKey ()*). Dies gibt uns das Schema, das wir wollen:



Sie können die Beziehung auch von der `Person` aus konfigurieren:

```
public class PersonEntityTypeConfiguration : EntityTypeConfiguration<Person>
{
  public PersonEntityTypeConfiguration()
  {
    this.HasMany(p => p.Cars).WithRequired(c => c.Owner).HasForeignKey(c => c.OwnerId);
  }
}
```

Die Idee ist die gleiche, nur die Seiten sind unterschiedlich (beachten Sie, wie Sie das Ganze lesen können: "Diese Person hat viele Autos, jedes Auto mit einem erforderlichen Besitzer"). Es spielt keine Rolle, ob Sie die Beziehung von der `Person` oder der `Car` aus konfigurieren. Sie können sogar beide einschließen, aber in diesem Fall müssen Sie auf beiden Seiten dieselbe Beziehung angeben!

Zuordnung von Null oder Eins-zu-Viele

In den vorherigen Beispielen kann ein Auto ohne eine Person nicht existieren. Was wäre, wenn Sie wollten, dass die Person von der Autoseite optional ist? Nun, es ist ziemlich einfach zu wissen, wie man eins zu viele macht. Ändern `PersonId` einfach die `PersonId` in `Car` , dass sie nullwertfähig ist:

```
public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public int? PersonId { get; set; }
```

```
    public virtual Person Person { get; set; }
}
```

Und dann verwenden Sie *HasOptional ()* (oder *WithOptional ()* , abhängig von welcher Seite Sie die Konfiguration durchführen):

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
     this.HasOptional(c => c.Owner).WithMany(p => p.Cars).HasForeignKey(c => c.OwnerId);
  }
}
```

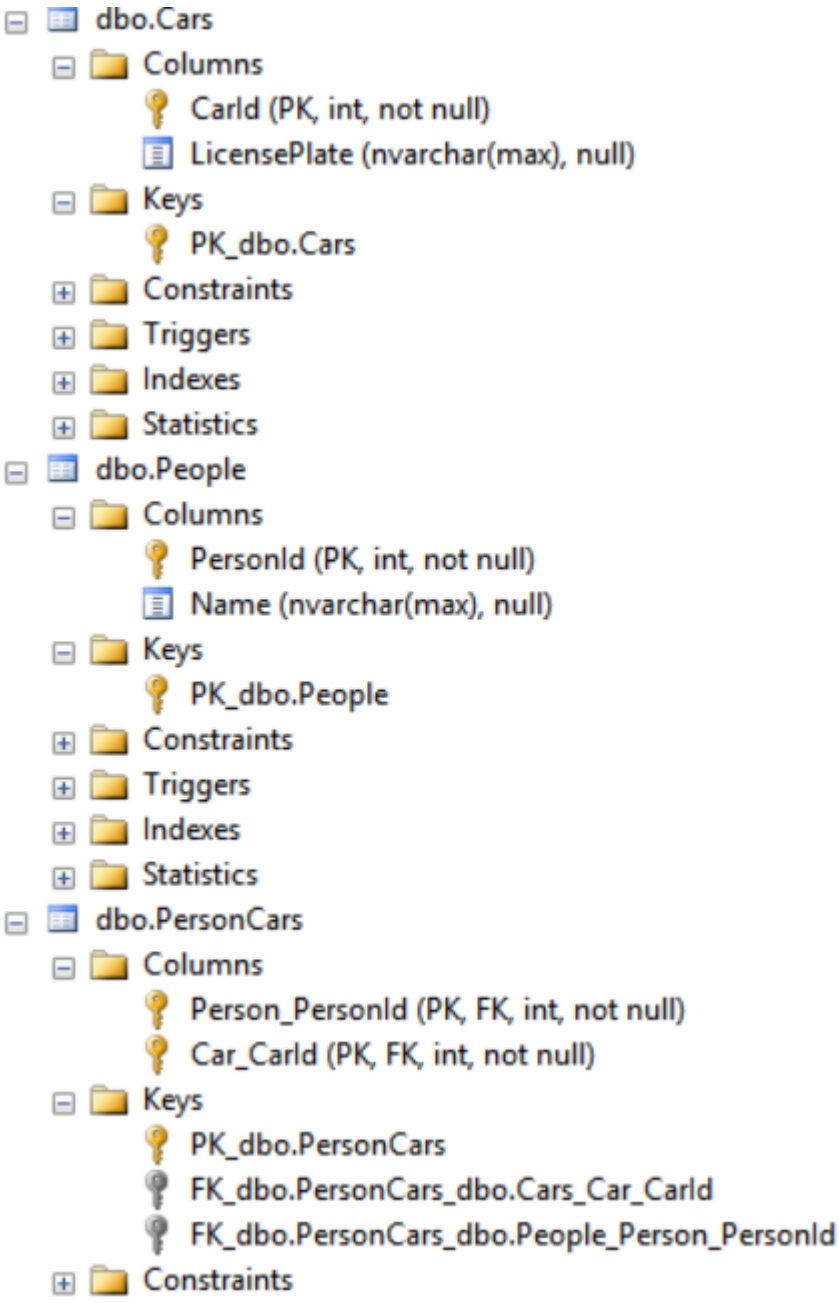
Viel zu viel

Lassen Sie uns zu dem anderen Szenario übergehen, in dem jede Person mehrere Autos haben kann und jedes Auto mehrere Besitzer haben kann. Dies ist eine viele-zu-viele-Beziehung. Der einfachste Weg ist, EF seine Zauberei mit Konventionen zu überlassen.

Ändern Sie einfach das Modell wie folgt:

```
 public class Person
{
   public int PersonId { get; set; }
   public string Name { get; set; }
   public virtual ICollection<Car> Cars { get; set; }
}

public class Car
{
   public int CarId { get; set; }
   public string LicensePlate { get; set; }
   public virtual ICollection<Person> Owners { get; set; }
}
```

Und das Schema:



Fast perfekt. Wie Sie sehen, hat EF die Notwendigkeit eines Join-Tisches erkannt, an dem Sie die Personen-Auto-Paarungen verfolgen können.

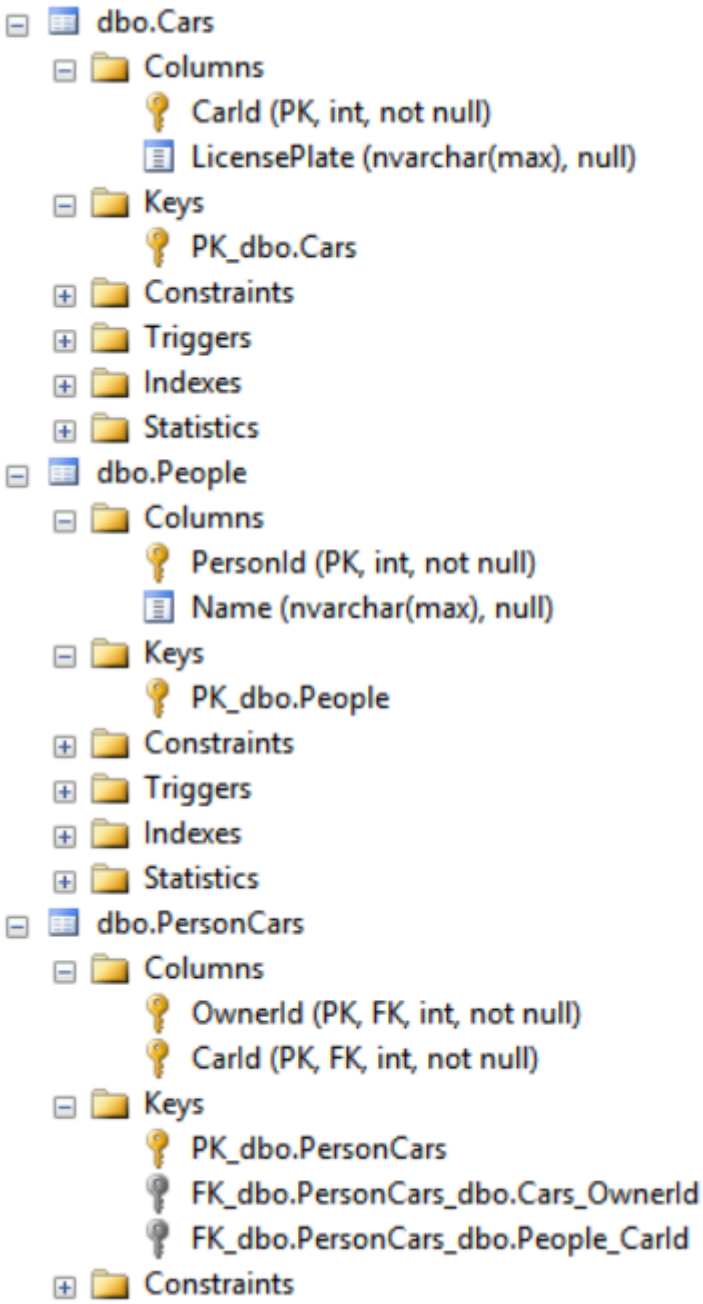
Many-to-many: Anpassen der Join-Tabelle

Möglicherweise möchten Sie die Felder in der Join-Tabelle umbenennen, um ein wenig freundlicher zu sein. Sie können dies tun, indem Sie die üblichen Konfigurationsmethoden verwenden (wiederum spielt es keine Rolle, von welcher Seite Sie die Konfiguration vornehmen):

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
   public CarEntityTypeConfiguration()
   {
      this.HasMany(c => c.Owners).WithMany(p => p.Cars)
          .Map(m =>
              {
                 m.MapLeftKey("OwnerId");
                 m.MapRightKey("CarId");
```

```
                m.ToTable("PersonCars");
            }
        );
    }
}
```

Ganz einfach sogar zu lesen: Dieses Auto hat viele Besitzer (*HasMany ()*), wobei jeder Besitzer viele Autos hat (*WithMany ()*). Ordnen Sie dies so zu, dass Sie den linken Schlüssel OwnerId (*MapLeftKey ()*), den rechten Schlüssel CarId (*MapRightKey ()*) und das Ganze der Tabelle PersonCars (*ToTable ()*) *zuordnen* . Und das gibt Ihnen genau dieses Schema:



Many-to-many: Benutzerdefinierte Join-Entität

Ich muss zugeben, ich bin nicht wirklich ein Fan davon, EF die Join-Tabelle ohne eine Join-Entität entnehmen zu lassen. Sie können keine zusätzlichen Informationen zu einer Personen-Auto-Vereinigung verfolgen (beispielsweise das Datum, ab dem sie gültig sind), da Sie die Tabelle nicht

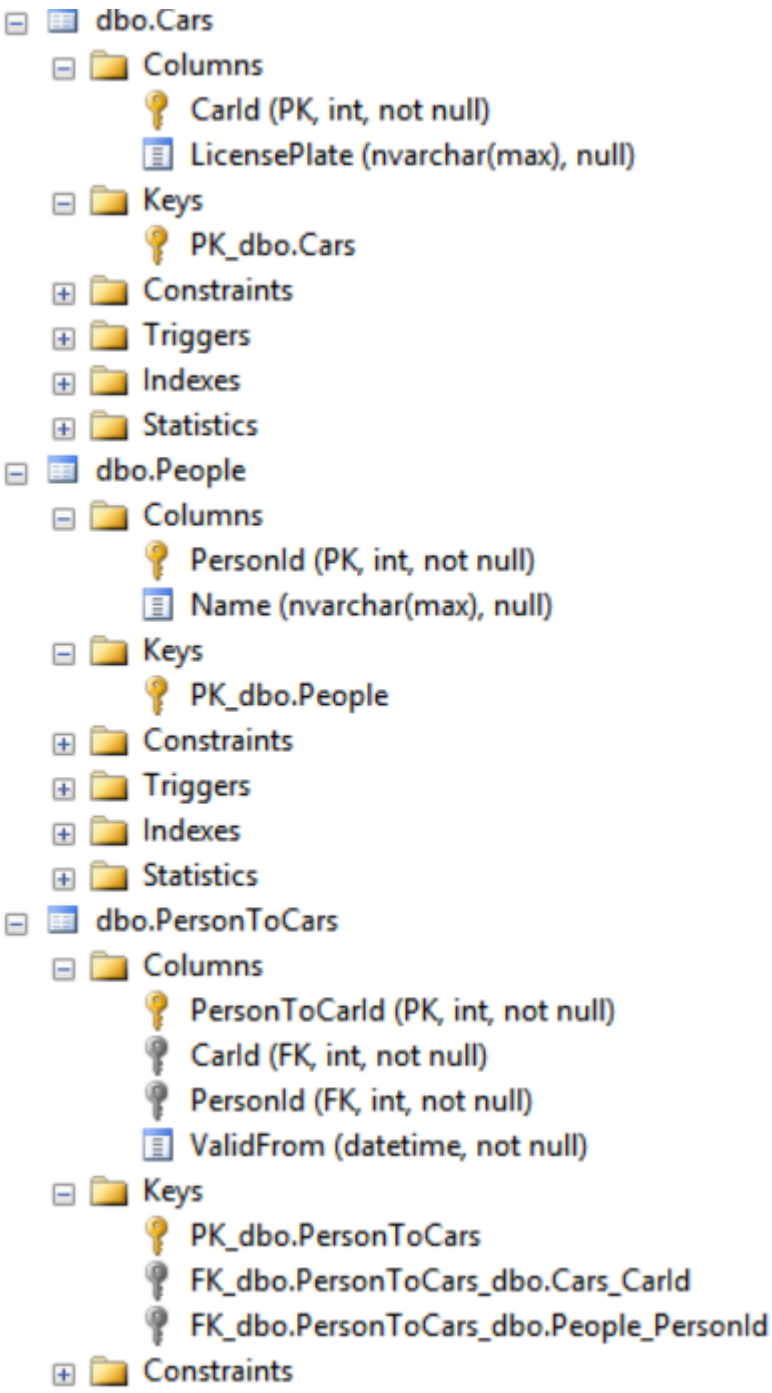
ändern können.

Außerdem ist die `CarId` in der Join-Tabelle Teil des Primärschlüssels. Wenn die Familie ein neues Auto kauft, müssen Sie zuerst die alten Zuordnungen löschen und neue hinzufügen. EF verbirgt dies vor Ihnen, aber dies bedeutet, dass Sie diese beiden Vorgänge anstelle eines einfachen Updates ausführen müssen (ganz zu schweigen davon, dass häufiges Einfügen / Löschen zu Indexfragmentierung führen kann - gut, es gibt eine einfache Lösung).

In diesem Fall können Sie eine Join-Entität erstellen, die sowohl auf ein bestimmtes Fahrzeug als auch auf eine bestimmte Person verweist. Grundsätzlich betrachten Sie Ihre Many-to-Many-Assoziation als Kombination zweier One-to-Many-Assoziationen:

```
public class PersonToCar
{
   public int PersonToCarId { get; set; }
   public int CarId { get; set; }
   public virtual Car Car { get; set; }
   public int PersonId { get; set; }
   public virtual Person Person { get; set; }
   public DateTime ValidFrom { get; set; }
}

public class Person
{
  public int PersonId { get; set; }
  public string Name { get; set; }
  public virtual ICollection<PersonToCar> CarOwnerShips { get; set; }
}

public class Car
{
  public int CarId { get; set; }
  public string LicensePlate { get; set; }
  public virtual ICollection<PersonToCar> Ownerships { get; set; }
}

public class MyDemoContext : DbContext
{
  public DbSet<Person> People { get; set; }
  public DbSet<Car> Cars { get; set; }
  public DbSet<PersonToCar> PersonToCars { get; set; }
}
```

Dies gibt mir viel mehr Kontrolle und ist viel flexibler. Ich kann jetzt benutzerdefinierte Daten zur Assoziation hinzufügen, und jede Assoziation verfügt über einen eigenen Primärschlüssel, sodass ich das Auto oder die Eigentümerreferenz darin aktualisieren kann.



Beachten Sie, dass dies wirklich nur eine Kombination aus zwei Eins-zu-Viele-Beziehungen ist. Sie können also alle Konfigurationsoptionen verwenden, die in den vorherigen Beispielen beschrieben wurden.

Zuordnungsbeziehung mit Entity Framework Code First: One-to-Many und Many-to-Many online lesen: https://riptutorial.com/de/entity-framework/topic/9413/zuordnungsbeziehung-mit-entity-framework-code-first--one-to-many-und-many-to-many

Kapitel 23: Zuordnungsbeziehung mit Entity Framework Code First: One-to-One und Variationen

Einführung

In diesem Thema wird beschrieben, wie One-to-One-Beziehungen mithilfe von Entity Framework abgebildet werden.

Examples

Zuordnung von Eins zu Null oder Eins

Sagen wir noch einmal, dass Sie folgendes Modell haben:

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

public class Car
{
  public int CarId { get; set; }
  public string LicensePlate { get; set; }
}

public class MyDemoContext : DbContext
{
  public DbSet<Person> People { get; set; }
  public DbSet<Car> Cars { get; set; }
}
```

Und jetzt möchten Sie es so einrichten, dass Sie die folgende Spezifikation ausdrücken können: Eine Person kann ein oder null Auto haben, und jedes Auto gehört genau zu einer Person (Beziehungen sind bidirektional, wenn CarA zu PersonA gehört, dann gehört PersonA dazu.) 'CarA).

Lassen Sie uns das Modell ein wenig ändern: Fügen Sie die Navigationseigenschaften und die Fremdschlüsseleigenschaften hinzu:

```
public class Person
{
  public int PersonId { get; set; }
  public string Name { get; set; }
  public int CarId { get; set; }
  public virtual Car Car { get; set; }
}
```

```
public class Car
{
  public int CarId { get; set; }
  public string LicensePlate { get; set; }
  public int PersonId { get; set; }
  public virtual Person Person { get; set; }
}
```

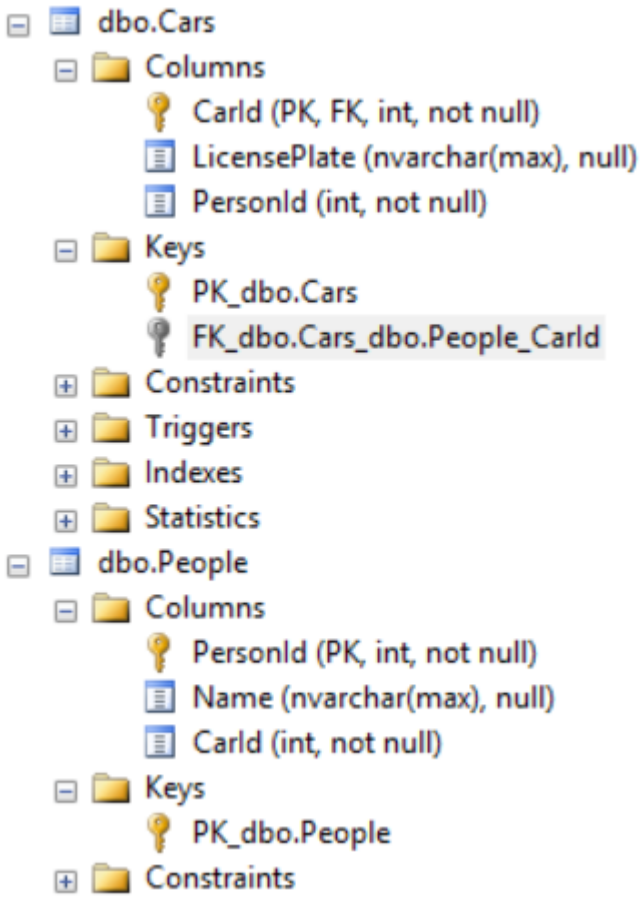
Und die Konfiguration:

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
     this.HasRequired(c => c.Person).WithOptional(p => p.Car);
  }
}
```

Zu diesem Zeitpunkt sollte dies selbsterklärend sein. Das Auto hat eine erforderliche Person (*HasRequired ()*), wobei die Person ein optionales Auto hat (*WithOptional ()*). Auch hier spielt es keine Rolle, von welcher Seite aus Sie diese Beziehung konfigurieren. Seien Sie vorsichtig, wenn Sie die richtige Kombination aus Has / With und Required / Optional verwenden. Auf der `Person` würde es so aussehen:

```
public class PersonEntityTypeConfiguration : EntityTypeConfiguration<Person>
{
  public PersonEntityTypeConfiguration()
  {
     this.HasOptional(p => p.Car).WithOptional(c => c.Person);
  }
}
```

Jetzt schauen wir uns das Datenbankschema an:



Schauen Sie genau hin: Sie können sehen, dass es in `People` keinen FK gibt, der sich auf `Car` bezieht. Außerdem ist der FK in `Car` nicht die `PersonId` , sondern die `CarId` . Hier ist das eigentliche Skript für den FK:

```
ALTER TABLE [dbo].[Cars]  WITH CHECK ADD  CONSTRAINT [FK_dbo.Cars_dbo.People_CarId] FOREIGN
KEY([CarId])
REFERENCES [dbo].[People] ([PersonId])
```

Dies bedeutet, dass die Schlüsseleigenschaften `CarId` und `PersonId` foregn, die wir im Modell haben, grundsätzlich ignoriert werden. Sie befinden sich in der Datenbank, sind jedoch, wie erwartet, keine Fremdschlüssel. Das liegt daran, dass One-to-One-Mappings das Hinzufügen von FK in Ihrem EF-Modell nicht unterstützen. One-to-One-Mappings sind in einer relationalen Datenbank ziemlich problematisch.

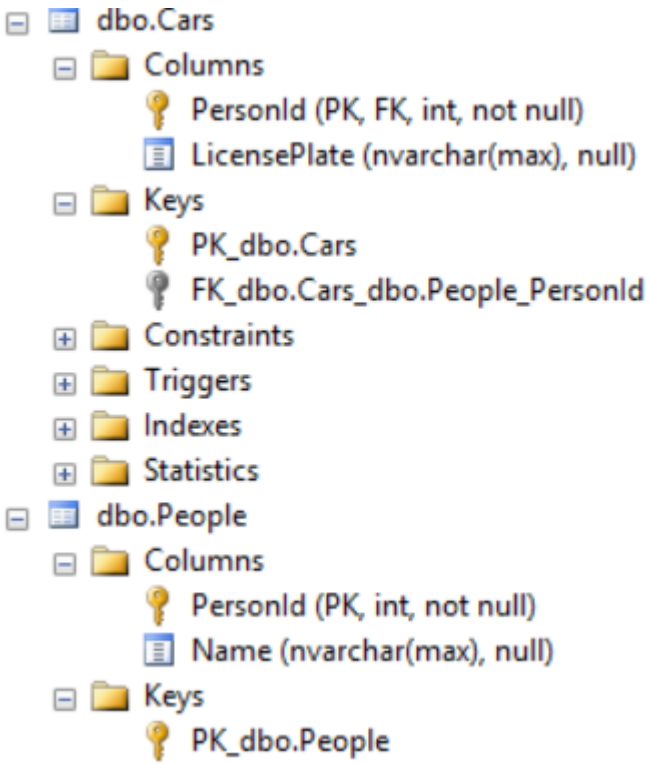
Die Idee ist, dass jede Person genau ein Auto haben kann, und dass Auto nur dieser Person gehören kann. Oder es gibt Personendatensätze, denen keine Autos zugeordnet sind.

Wie konnte dies mit Fremdschlüsseln dargestellt werden? Offensichtlich könnte es eine `PersonId` in `Car` und eine `CarId` in `People` . Um zu erzwingen, dass jede Person nur ein Auto haben kann, muss `PersonId` in `Car` eindeutig sein. Wenn jedoch `PersonId` in `People` eindeutig ist, wie können Sie dann zwei oder mehr Datensätze hinzufügen, bei denen `PersonId` `NULL` (mehr als ein Auto ohne Besitzer)? Antwort: Das können Sie nicht (eigentlich können Sie in SQL Server 2008 und neuer einen gefilterten eindeutigen Index erstellen, aber vergessen Sie diese technischen Aspekte für einen Moment; ganz zu schweigen von anderen RDBMS). Ganz zu schweigen von dem Fall, in dem Sie beide Enden der Beziehung angeben ...

Die einzige Möglichkeit, diese Regel zu erzwingen, wenn die Tabellen `People` und `Car` denselben Primärschlüssel haben (gleiche Werte in den verbundenen Datensätzen). Und um dies zu erreichen, muss `CarId` in `Car` sowohl PK als auch FK für die PK of People sein. Und das macht das ganze Schema zu einem Chaos. Wenn ich dies verwende, `PersonId` ich den PK / FK lieber in `Car` `PersonId` und konfiguriere ihn entsprechend:

```
public class Person
{
  public int PersonId { get; set; }
  public string Name { get; set; }
  public virtual Car Car { get; set; }
}

public class Car
{
  public string LicensePlate { get; set; }
  public int PersonId { get; set; }
  public virtual Person Person { get; set; }
}

public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
     this.HasRequired(c => c.Person).WithOptional(p => p.Car);
     this.HasKey(c => c.PersonId);
  }
}
```

Nicht ideal, aber vielleicht ein bisschen besser. Bei der Verwendung dieser Lösung müssen Sie jedoch wachsam sein, da dies gegen die üblichen Namenskonventionen verstößt, die Sie in die Irre führen können. Hier ist das aus diesem Modell generierte Schema:



Daher wird diese Beziehung nicht vom Datenbankschema erzwungen, sondern von Entity Framework selbst. Deshalb müssen Sie sehr vorsichtig sein, wenn Sie dies verwenden, um niemanden direkt mit der Datenbank abkühlen zu lassen.

Eins-zu-eins-Mapping

Eins-zu-Eins-Mapping (wenn beide Seiten erforderlich sind) ist auch eine schwierige Sache.

Stellen wir uns vor, wie dies mit Fremdschlüsseln dargestellt werden kann. Wieder eine `CarId` in `People` , die sich auf `CarId` in `Car` bezieht, und eine `PersonId` in Car, die sich auf die `PersonId` in `People` bezieht.

Was passiert nun, wenn Sie einen Fahrzeugdatensatz einfügen möchten? Damit dies gelingt, muss in diesem Fahrzeugdatensatz eine `PersonId` angegeben sein, da dies erforderlich ist. Damit diese `PersonId` gültig ist, muss der entsprechende Datensatz in `People` vorhanden sein. OK, also lasst uns fortfahren und den Personendatensatz einfügen. Damit dies gelingt, muss eine gültige `CarId` im Personenrekord vorhanden sein - aber das Auto ist noch nicht eingefügt! Dies kann nicht der Fall sein, weil wir zuerst den Datensatz der betreffenden Person einfügen müssen. Wir können den referenzierten Personendatensatz jedoch nicht einfügen, da er auf den Fahrzeugdatensatz zurückgreift, so dass dieser zuerst eingefügt werden muss (Fremdschlüsselaufzeichnung :).

Dies kann auch nicht auf „logische“ Weise dargestellt werden. Wieder müssen Sie einen der Fremdschlüssel ablegen. Welches Sie fallen lassen, liegt bei Ihnen. Die mit einem Fremdschlüssel belegte Seite wird als "abhängig" bezeichnet, die Seite, die keinen Fremdschlüssel enthält, wird als "Principal" bezeichnet. Um die Eindeutigkeit der Abhängigen zu gewährleisten, muss die PK die FK sein. Das Hinzufügen einer FK-Spalte und das Importieren der Spalte in Ihr Modell wird nicht unterstützt.

Also hier ist die Konfiguration:

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
    this.HasRequired(c => c.Person).WithRequiredDependent(p => p.Car);
    this.HasKey(c => c.PersonId);
  }
}
```

Inzwischen sollten Sie wirklich die Logik dazu bekommen haben :) Denken Sie daran, dass Sie auch die andere Seite wählen können. Seien Sie nur vorsichtig, wenn Sie die abhängigen / Hauptversionen von WithRequired verwenden (und Sie müssen die PK in Car noch konfigurieren).

```
public class PersonEntityTypeConfiguration : EntityTypeConfiguration<Person>
{
  public PersonEntityTypeConfiguration()
  {
    this.HasRequired(p => p.Car).WithRequiredPrincipal(c => c.Person);
  }
}
```

Wenn Sie das DB-Schema überprüfen, werden Sie feststellen, dass es genau dasselbe ist wie bei der Eins-zu-Eins-Lösung oder der Nulllösung. Das liegt daran, dass dies wiederum nicht vom Schema, sondern von EF selbst erzwungen wird. Also nochmal, sei vorsichtig :)

Zuordnung von Eins oder Null-zu-Eins oder Null

Zum Abschluss betrachten wir kurz den Fall, wenn beide Seiten optional sind.

Inzwischen sollten Sie sich mit diesen Beispielen wirklich langweilig fühlen :), also gehe ich nicht ins Detail und spiele mit der Idee, zwei FKs und mögliche Probleme zu haben, und warne Sie vor den Gefahren, wenn Sie diese Regeln nicht durchsetzen Schema aber nur in EF selbst.

Hier ist die Konfiguration, die Sie anwenden müssen:

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
    this.HasOptional(c => c.Person).WithOptionalPrincipal(p => p.Car);
    this.HasKey(c => c.PersonId);
  }
}
```

Sie können auch von der anderen Seite aus konfigurieren, achten Sie nur auf die richtigen Methoden :)

Zuordnungsbeziehung mit Entity Framework Code First: One-to-One und Variationen online lesen: https://riptutorial.com/de/entity-framework/topic/9412/zuordnungsbeziehung-mit-entity-framework-code-first--one-to-one-und-variationen

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit Entity Framework	Adil Mammadov, Community, DavidG, Eldho, Jacob Linney, Martin4ndersen, Matas Vaitkevicius, Nasreddine, NovaDev, Parth Patel, Stephen Reindl, tmg
2	.t4-Vorlagen im Entity-Framework	Matas Vaitkevicius, Tetsuya Yamamoto
3	Best Practices für das Entity Framework (einfach und professionell)	Braiam, Mina Matta
4	Code First - Fließende API	Adil Mammadov, Daniel Lemke, Jason Tyler, tmg
5	Datenbankinitialisierer	Jozef Lačný
6	Entitätsstatus verwalten	Gert Arnold
7	Entity Framework mit SQLite	Jason Tyler
8	Entity-Framework mit Postgresql	skj123
9	Entity-Framework-Code zuerst	Balázs Nagy, Jozef Lačný
10	Erste Datenanmerkungen kodieren	bubi, CptRobby, Daniel A. White, Daniel Lemke, DavidG, Diego, Gert Arnold, Jozef Lačný, Mark Shevchenko, Matas Vaitkevicius, Parth Patel, Piotrek, tmg, Tushar patel
11	Erste Konventionen für den Code	MacakM, Parth Patel, Sivanantham Padikkasu, Stephen Reindl, tmg
12	Erste Migrationen für Entity-Framework-Code	CGritton, hasan, Joshit, Mostafa, RamenChef, Stephen Reindl
13	Erste Modellgeneration der	Matas Vaitkevicius, Tetsuya Yamamoto

	Datenbank	
14	Erweiterte Mapping-Szenarien: Entitätsaufteilung, Aufteilung der Tabelle	Akos Nagy
15	Komplexe Typen	CptRobby, Gert Arnold
16	Modellfesseln	SOfanatic, Tushar patel
17	Optimierungstechniken in EF	Amit Shahani, Anshul Nigam, DavidG, Gert Arnold, Jacob Linney, Kobi, lucavgobbi, Stephen Reindl, tmg, wertzui
18	Tracking vs. No-Tracking	hasan, Sampath, Stephen Reindl, tmg
19	Transaktionen	CptRobby, DavidG, Gert Arnold
20	Vererbung mit EntityFramework (Code First)	lucavgobbi
21	Verwandte Entitäten laden	Adil Mammadov, Florian Haider, Gert Arnold, hasan, Joshit, Matas Vaitkevicius, tmg
22	Zuordnungsbeziehung mit Entity Framework Code First: One-to-Many und Many-to-Many	Akos Nagy
23	Zuordnungsbeziehung mit Entity Framework Code First: One-to-One und Variationen	Akos Nagy