





using System.Data.Entity;

using System.Linq;

using System.Windows;

namespace WPFwithEFSample

{

public partial class MainWindow : Window

{

private ProductContext \_context = new ProductContext();

public MainWindow()

{

InitializeComponent();

}

private void Window\_Loaded(object sender, RoutedEventArgs e)

{

System.Windows.Data.CollectionViewSource categoryViewSource =

((System.Windows.Data.CollectionViewSource)(this.FindResource("categoryViewSource")));

// Load is an extension method on IQueryable,

// defined in the System.Data.Entity namespace.

// This method enumerates the results of the query,

// similar to ToList but without creating a list.

// When used with Linq to Entities this method

// creates entity objects and adds them to the context.

\_context.Categories.Load();

// After the data is loaded call the DbSet<T>.Local property

// to use the DbSet<T> as a binding source.

categoryViewSource.Source = \_context.Categories.Local;

}

private void SaveButton\_Click(object sender, RoutedEventArgs e)

{

// When you delete an object from the related entities collection

// (in this case Products), the Entity Framework doesn’t mark

// these child entities as deleted.

// Instead, it removes the relationship between the parent and the child

// by setting the parent reference to null.

// So we manually have to delete the products

// that have a Category reference set to null.

// The following code uses LINQ to Objects

// against the Local collection of Products.

// The ToList call is required because otherwise the collection will be modified

// by the Remove call while it is being enumerated.

// In most other situations you can use LINQ to Objects directly

// against the Local property without using ToList first.

foreach (var product in \_context.Products.Local.ToList())

{

if (product.Category == null)

{

\_context.Products.Remove(product);

}

}

\_context.SaveChanges();

// Refresh the grids so the database generated values show up.

this.categoryDataGrid.Items.Refresh();

this.productsDataGrid1.Items.Refresh();

}

protected override void OnClosing(System.ComponentModel.CancelEventArgs e)

{

base.OnClosing(e);

this.\_context.Dispose();

}

}

}

***The below code without syntax highlighting is from ProductModel1.tt, which only had the minor changes of the first instance of, and the first two instances of `HashSet`, and `ICollection` being replaced with `ObservableCollection`. To use the ObservableCollections, the `System.Collections.Generic` declaration was replaced with `System.Collections.ObjectModel`.***

<#@ template language="C#" debug="false" hostspecific="true"#>

<#@ include file="EF6.Utility.CS.ttinclude"#><#@

output extension=".cs"#><#

const string inputFile = @"ProductModel1.edmx";

var textTransform = DynamicTextTransformation.Create(this);

var code = new CodeGenerationTools(this);

var ef = new MetadataTools(this);

var typeMapper = new TypeMapper(code, ef, textTransform.Errors);

var fileManager = EntityFrameworkTemplateFileManager.Create(this);

var itemCollection = new EdmMetadataLoader(textTransform.Host, textTransform.Errors).CreateEdmItemCollection(inputFile);

var codeStringGenerator = new CodeStringGenerator(code, typeMapper, ef);

if (!typeMapper.VerifyCaseInsensitiveTypeUniqueness(typeMapper.GetAllGlobalItems(itemCollection), inputFile))

{

return string.Empty;

}

WriteHeader(codeStringGenerator, fileManager);

foreach (var entity in typeMapper.GetItemsToGenerate<EntityType>(itemCollection))

{

fileManager.StartNewFile(entity.Name + ".cs");

BeginNamespace(code);

#>

<#=codeStringGenerator.UsingDirectives(inHeader: false)#>

<#=codeStringGenerator.EntityClassOpening(entity)#>

{

<#

var propertiesWithDefaultValues = typeMapper.GetPropertiesWithDefaultValues(entity);

var collectionNavigationProperties = typeMapper.GetCollectionNavigationProperties(entity);

var complexProperties = typeMapper.GetComplexProperties(entity);

if (propertiesWithDefaultValues.Any() || collectionNavigationProperties.Any() || complexProperties.Any())

{

#>

[System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage", "CA2214:DoNotCallOverridableMethodsInConstructors")]

public <#=code.Escape(entity)#>()

{

<#

foreach (var edmProperty in propertiesWithDefaultValues)

{

#>

this.<#=code.Escape(edmProperty)#> = <#=typeMapper.CreateLiteral(edmProperty.DefaultValue)#>;

<#

}

foreach (var navigationProperty in collectionNavigationProperties)

{

#>

this.<#=code.Escape(navigationProperty)#> = new ObservableCollection<<#=typeMapper.GetTypeName(navigationProperty.ToEndMember.GetEntityType())#>>();

<#

}

foreach (var complexProperty in complexProperties)

{

#>

this.<#=code.Escape(complexProperty)#> = new <#=typeMapper.GetTypeName(complexProperty.TypeUsage)#>();

<#

}

#>

}

<#

}

var simpleProperties = typeMapper.GetSimpleProperties(entity);

if (simpleProperties.Any())

{

foreach (var edmProperty in simpleProperties)

{

#>

<#=codeStringGenerator.Property(edmProperty)#>

<#

}

}

if (complexProperties.Any())

{

#>

<#

foreach(var complexProperty in complexProperties)

{

#>

<#=codeStringGenerator.Property(complexProperty)#>

<#

}

}

var navigationProperties = typeMapper.GetNavigationProperties(entity);

if (navigationProperties.Any())

{

#>

<#

foreach (var navigationProperty in navigationProperties)

{

if (navigationProperty.ToEndMember.RelationshipMultiplicity == RelationshipMultiplicity.Many)

{

#>

[System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage", "CA2227:CollectionPropertiesShouldBeReadOnly")]

<#

}

#>

<#=codeStringGenerator.NavigationProperty(navigationProperty)#>

<#

}

}

#>

}

<#

EndNamespace(code);

}

foreach (var complex in typeMapper.GetItemsToGenerate<ComplexType>(itemCollection))

{

fileManager.StartNewFile(complex.Name + ".cs");

BeginNamespace(code);

#>

<#=codeStringGenerator.UsingDirectives(inHeader: false, includeCollections: false)#>

<#=Accessibility.ForType(complex)#> partial class <#=code.Escape(complex)#>

{

<#

var complexProperties = typeMapper.GetComplexProperties(complex);

var propertiesWithDefaultValues = typeMapper.GetPropertiesWithDefaultValues(complex);

if (propertiesWithDefaultValues.Any() || complexProperties.Any())

{

#>

public <#=code.Escape(complex)#>()

{

<#

foreach (var edmProperty in propertiesWithDefaultValues)

{

#>

this.<#=code.Escape(edmProperty)#> = <#=typeMapper.CreateLiteral(edmProperty.DefaultValue)#>;

<#

}

foreach (var complexProperty in complexProperties)

{

#>

this.<#=code.Escape(complexProperty)#> = new <#=typeMapper.GetTypeName(complexProperty.TypeUsage)#>();

<#

}

#>

}

<#

}

var simpleProperties = typeMapper.GetSimpleProperties(complex);

if (simpleProperties.Any())

{

foreach(var edmProperty in simpleProperties)

{

#>

<#=codeStringGenerator.Property(edmProperty)#>

<#

}

}

if (complexProperties.Any())

{

#>

<#

foreach(var edmProperty in complexProperties)

{

#>

<#=codeStringGenerator.Property(edmProperty)#>

<#

}

}

#>

}

<#

EndNamespace(code);

}

foreach (var enumType in typeMapper.GetEnumItemsToGenerate(itemCollection))

{

fileManager.StartNewFile(enumType.Name + ".cs");

BeginNamespace(code);

#>

<#=codeStringGenerator.UsingDirectives(inHeader: false, includeCollections: false)#>

<#

if (typeMapper.EnumIsFlags(enumType))

{

#>

[Flags]

<#

}

#>

<#=codeStringGenerator.EnumOpening(enumType)#>

{

<#

var foundOne = false;

foreach (MetadataItem member in typeMapper.GetEnumMembers(enumType))

{

foundOne = true;

#>

<#=code.Escape(typeMapper.GetEnumMemberName(member))#> = <#=typeMapper.GetEnumMemberValue(member)#>,

<#

}

if (foundOne)

{

this.GenerationEnvironment.Remove(this.GenerationEnvironment.Length - 3, 1);

}

#>

}

<#

EndNamespace(code);

}

fileManager.Process();

#>

<#+

public void WriteHeader(CodeStringGenerator codeStringGenerator, EntityFrameworkTemplateFileManager fileManager)

{

fileManager.StartHeader();

#>

//------------------------------------------------------------------------------

// <auto-generated>

// <#=CodeGenerationTools.GetResourceString("Template\_GeneratedCodeCommentLine1")#>

//

// <#=CodeGenerationTools.GetResourceString("Template\_GeneratedCodeCommentLine2")#>

// <#=CodeGenerationTools.GetResourceString("Template\_GeneratedCodeCommentLine3")#>

// </auto-generated>

//------------------------------------------------------------------------------

<#=codeStringGenerator.UsingDirectives(inHeader: true)#>

<#+

fileManager.EndBlock();

}

public void BeginNamespace(CodeGenerationTools code)

{

var codeNamespace = code.VsNamespaceSuggestion();

if (!String.IsNullOrEmpty(codeNamespace))

{

#>

namespace <#=code.EscapeNamespace(codeNamespace)#>

{

<#+

PushIndent(" ");

}

}

public void EndNamespace(CodeGenerationTools code)

{

if (!String.IsNullOrEmpty(code.VsNamespaceSuggestion()))

{

PopIndent();

#>

}

<#+

}

}

public const string TemplateId = "CSharp\_DbContext\_Types\_EF6";

public class CodeStringGenerator

{

private readonly CodeGenerationTools \_code;

private readonly TypeMapper \_typeMapper;

private readonly MetadataTools \_ef;

public CodeStringGenerator(CodeGenerationTools code, TypeMapper typeMapper, MetadataTools ef)

{

ArgumentNotNull(code, "code");

ArgumentNotNull(typeMapper, "typeMapper");

ArgumentNotNull(ef, "ef");

\_code = code;

\_typeMapper = typeMapper;

\_ef = ef;

}

public string Property(EdmProperty edmProperty)

{

return string.Format(

CultureInfo.InvariantCulture,

"{0} {1} {2} {{ {3}get; {4}set; }}",

Accessibility.ForProperty(edmProperty),

\_typeMapper.GetTypeName(edmProperty.TypeUsage),

\_code.Escape(edmProperty),

\_code.SpaceAfter(Accessibility.ForGetter(edmProperty)),

\_code.SpaceAfter(Accessibility.ForSetter(edmProperty)));

}

public string NavigationProperty(NavigationProperty navProp)

{

var endType = \_typeMapper.GetTypeName(navProp.ToEndMember.GetEntityType());

return string.Format(

CultureInfo.InvariantCulture,

"{0} {1} {2} {{ {3}get; {4}set; }}",

AccessibilityAndVirtual(Accessibility.ForNavigationProperty(navProp)),

navProp.ToEndMember.RelationshipMultiplicity == RelationshipMultiplicity.Many ? ("ObservableCollection<" + endType + ">") : endType,

\_code.Escape(navProp),

\_code.SpaceAfter(Accessibility.ForGetter(navProp)),

\_code.SpaceAfter(Accessibility.ForSetter(navProp)));

}

public string AccessibilityAndVirtual(string accessibility)

{

return accessibility + (accessibility != "private" ? " virtual" : "");

}

public string EntityClassOpening(EntityType entity)

{

return string.Format(

CultureInfo.InvariantCulture,

"{0} {1}partial class {2}{3}",

Accessibility.ForType(entity),

\_code.SpaceAfter(\_code.AbstractOption(entity)),

\_code.Escape(entity),

\_code.StringBefore(" : ", \_typeMapper.GetTypeName(entity.BaseType)));

}

public string EnumOpening(SimpleType enumType)

{

return string.Format(

CultureInfo.InvariantCulture,

"{0} enum {1} : {2}",

Accessibility.ForType(enumType),

\_code.Escape(enumType),

\_code.Escape(\_typeMapper.UnderlyingClrType(enumType)));

}

public void WriteFunctionParameters(EdmFunction edmFunction, Action<string, string, string, string> writeParameter)

{

var parameters = FunctionImportParameter.Create(edmFunction.Parameters, \_code, \_ef);

foreach (var parameter in parameters.Where(p => p.NeedsLocalVariable))

{

var isNotNull = parameter.IsNullableOfT ? parameter.FunctionParameterName + ".HasValue" : parameter.FunctionParameterName + " != null";

var notNullInit = "new ObjectParameter(\"" + parameter.EsqlParameterName + "\", " + parameter.FunctionParameterName + ")";

var nullInit = "new ObjectParameter(\"" + parameter.EsqlParameterName + "\", typeof(" + TypeMapper.FixNamespaces(parameter.RawClrTypeName) + "))";

writeParameter(parameter.LocalVariableName, isNotNull, notNullInit, nullInit);

}

}

public string ComposableFunctionMethod(EdmFunction edmFunction, string modelNamespace)

{

var parameters = \_typeMapper.GetParameters(edmFunction);

return string.Format(

CultureInfo.InvariantCulture,

"{0} IQueryable<{1}> {2}({3})",

AccessibilityAndVirtual(Accessibility.ForMethod(edmFunction)),

\_typeMapper.GetTypeName(\_typeMapper.GetReturnType(edmFunction), modelNamespace),

\_code.Escape(edmFunction),

string.Join(", ", parameters.Select(p => TypeMapper.FixNamespaces(p.FunctionParameterType) + " " + p.FunctionParameterName).ToArray()));

}

public string ComposableCreateQuery(EdmFunction edmFunction, string modelNamespace)

{

var parameters = \_typeMapper.GetParameters(edmFunction);

return string.Format(

CultureInfo.InvariantCulture,

"return ((IObjectContextAdapter)this).ObjectContext.CreateQuery<{0}>(\"[{1}].[{2}]({3})\"{4});",

\_typeMapper.GetTypeName(\_typeMapper.GetReturnType(edmFunction), modelNamespace),

edmFunction.NamespaceName,

edmFunction.Name,

string.Join(", ", parameters.Select(p => "@" + p.EsqlParameterName).ToArray()),

\_code.StringBefore(", ", string.Join(", ", parameters.Select(p => p.ExecuteParameterName).ToArray())));

}

public string FunctionMethod(EdmFunction edmFunction, string modelNamespace, bool includeMergeOption)

{

var parameters = \_typeMapper.GetParameters(edmFunction);

var returnType = \_typeMapper.GetReturnType(edmFunction);

var paramList = String.Join(", ", parameters.Select(p => TypeMapper.FixNamespaces(p.FunctionParameterType) + " " + p.FunctionParameterName).ToArray());

if (includeMergeOption)

{

paramList = \_code.StringAfter(paramList, ", ") + "MergeOption mergeOption";

}

return string.Format(

CultureInfo.InvariantCulture,

"{0} {1} {2}({3})",

AccessibilityAndVirtual(Accessibility.ForMethod(edmFunction)),

returnType == null ? "int" : "ObjectResult<" + \_typeMapper.GetTypeName(returnType, modelNamespace) + ">",

\_code.Escape(edmFunction),

paramList);

}

public string ExecuteFunction(EdmFunction edmFunction, string modelNamespace, bool includeMergeOption)

{

var parameters = \_typeMapper.GetParameters(edmFunction);

var returnType = \_typeMapper.GetReturnType(edmFunction);

var callParams = \_code.StringBefore(", ", String.Join(", ", parameters.Select(p => p.ExecuteParameterName).ToArray()));

if (includeMergeOption)

{

callParams = ", mergeOption" + callParams;

}

return string.Format(

CultureInfo.InvariantCulture,

"return ((IObjectContextAdapter)this).ObjectContext.ExecuteFunction{0}(\"{1}\"{2});",

returnType == null ? "" : "<" + \_typeMapper.GetTypeName(returnType, modelNamespace) + ">",

edmFunction.Name,

callParams);

}

public string DbSet(EntitySet entitySet)

{

return string.Format(

CultureInfo.InvariantCulture,

"{0} virtual DbSet<{1}> {2} {{ get; set; }}",

Accessibility.ForReadOnlyProperty(entitySet),

\_typeMapper.GetTypeName(entitySet.ElementType),

\_code.Escape(entitySet));

}

public string UsingDirectives(bool inHeader, bool includeCollections = true)

{

return inHeader == string.IsNullOrEmpty(\_code.VsNamespaceSuggestion())

? string.Format(

CultureInfo.InvariantCulture,

"{0}using System;{1}" +

"{2}",

inHeader ? Environment.NewLine : "",

includeCollections ? (Environment.NewLine + "using System.Collections.ObjectModel;") : "",

inHeader ? "" : Environment.NewLine)

: "";

}

}

public class TypeMapper

{

private const string ExternalTypeNameAttributeName = @"http://schemas.microsoft.com/ado/2006/04/codegeneration:ExternalTypeName";

private readonly System.Collections.IList \_errors;

private readonly CodeGenerationTools \_code;

private readonly MetadataTools \_ef;

public TypeMapper(CodeGenerationTools code, MetadataTools ef, System.Collections.IList errors)

{

ArgumentNotNull(code, "code");

ArgumentNotNull(ef, "ef");

ArgumentNotNull(errors, "errors");

\_code = code;

\_ef = ef;

\_errors = errors;

}

public static string FixNamespaces(string typeName)

{

return typeName.Replace("System.Data.Spatial.", "System.Data.Entity.Spatial.");

}

public string GetTypeName(TypeUsage typeUsage)

{

return typeUsage == null ? null : GetTypeName(typeUsage.EdmType, \_ef.IsNullable(typeUsage), modelNamespace: null);

}

public string GetTypeName(EdmType edmType)

{

return GetTypeName(edmType, isNullable: null, modelNamespace: null);

}

public string GetTypeName(TypeUsage typeUsage, string modelNamespace)

{

return typeUsage == null ? null : GetTypeName(typeUsage.EdmType, \_ef.IsNullable(typeUsage), modelNamespace);

}

public string GetTypeName(EdmType edmType, string modelNamespace)

{

return GetTypeName(edmType, isNullable: null, modelNamespace: modelNamespace);

}

public string GetTypeName(EdmType edmType, bool? isNullable, string modelNamespace)

{

if (edmType == null)

{

return null;

}

var collectionType = edmType as CollectionType;

if (collectionType != null)

{

return String.Format(CultureInfo.InvariantCulture, "ObservableCollection<{0}>", GetTypeName(collectionType.TypeUsage, modelNamespace));

}

var typeName = \_code.Escape(edmType.MetadataProperties

.Where(p => p.Name == ExternalTypeNameAttributeName)

.Select(p => (string)p.Value)

.FirstOrDefault())

?? (modelNamespace != null && edmType.NamespaceName != modelNamespace ?

\_code.CreateFullName(\_code.EscapeNamespace(edmType.NamespaceName), \_code.Escape(edmType)) :

\_code.Escape(edmType));

if (edmType is StructuralType)

{

return typeName;

}

if (edmType is SimpleType)

{

var clrType = UnderlyingClrType(edmType);

if (!IsEnumType(edmType))

{

typeName = \_code.Escape(clrType);

}

typeName = FixNamespaces(typeName);

return clrType.IsValueType && isNullable == true ?

String.Format(CultureInfo.InvariantCulture, "Nullable<{0}>", typeName) :

typeName;

}

throw new ArgumentException("edmType");

}

public Type UnderlyingClrType(EdmType edmType)

{

ArgumentNotNull(edmType, "edmType");

var primitiveType = edmType as PrimitiveType;

if (primitiveType != null)

{

return primitiveType.ClrEquivalentType;

}

if (IsEnumType(edmType))

{

return GetEnumUnderlyingType(edmType).ClrEquivalentType;

}

return typeof(object);

}

public object GetEnumMemberValue(MetadataItem enumMember)

{

ArgumentNotNull(enumMember, "enumMember");

var valueProperty = enumMember.GetType().GetProperty("Value");

return valueProperty == null ? null : valueProperty.GetValue(enumMember, null);

}

public string GetEnumMemberName(MetadataItem enumMember)

{

ArgumentNotNull(enumMember, "enumMember");

var nameProperty = enumMember.GetType().GetProperty("Name");

return nameProperty == null ? null : (string)nameProperty.GetValue(enumMember, null);

}

public System.Collections.IEnumerable GetEnumMembers(EdmType enumType)

{

ArgumentNotNull(enumType, "enumType");

var membersProperty = enumType.GetType().GetProperty("Members");

return membersProperty != null

? (System.Collections.IEnumerable)membersProperty.GetValue(enumType, null)

: Enumerable.Empty<MetadataItem>();

}

public bool EnumIsFlags(EdmType enumType)

{

ArgumentNotNull(enumType, "enumType");

var isFlagsProperty = enumType.GetType().GetProperty("IsFlags");

return isFlagsProperty != null && (bool)isFlagsProperty.GetValue(enumType, null);

}

public bool IsEnumType(GlobalItem edmType)

{

ArgumentNotNull(edmType, "edmType");

return edmType.GetType().Name == "EnumType";

}

public PrimitiveType GetEnumUnderlyingType(EdmType enumType)

{

ArgumentNotNull(enumType, "enumType");

return (PrimitiveType)enumType.GetType().GetProperty("UnderlyingType").GetValue(enumType, null);

}

public string CreateLiteral(object value)

{

if (value == null || value.GetType() != typeof(TimeSpan))

{

return \_code.CreateLiteral(value);

}

return string.Format(CultureInfo.InvariantCulture, "new TimeSpan({0})", ((TimeSpan)value).Ticks);

}

public bool VerifyCaseInsensitiveTypeUniqueness(IEnumerable<string> types, string sourceFile)

{

ArgumentNotNull(types, "types");

ArgumentNotNull(sourceFile, "sourceFile");

var hash = new HashSet<string>(StringComparer.InvariantCultureIgnoreCase);

if (types.Any(item => !hash.Add(item)))

{

\_errors.Add(

new CompilerError(sourceFile, -1, -1, "6023",

String.Format(CultureInfo.CurrentCulture, CodeGenerationTools.GetResourceString("Template\_CaseInsensitiveTypeConflict"))));

return false;

}

return true;

}

public IEnumerable<SimpleType> GetEnumItemsToGenerate(IEnumerable<GlobalItem> itemCollection)

{

return GetItemsToGenerate<SimpleType>(itemCollection)

.Where(e => IsEnumType(e));

}

public IEnumerable<T> GetItemsToGenerate<T>(IEnumerable<GlobalItem> itemCollection) where T: EdmType

{

return itemCollection

.OfType<T>()

.Where(i => !i.MetadataProperties.Any(p => p.Name == ExternalTypeNameAttributeName))

.OrderBy(i => i.Name);

}

public IEnumerable<string> GetAllGlobalItems(IEnumerable<GlobalItem> itemCollection)

{

return itemCollection

.Where(i => i is EntityType || i is ComplexType || i is EntityContainer || IsEnumType(i))

.Select(g => GetGlobalItemName(g));

}

public string GetGlobalItemName(GlobalItem item)

{

if (item is EdmType)

{

return ((EdmType)item).Name;

}

else

{

return ((EntityContainer)item).Name;

}

}

public IEnumerable<EdmProperty> GetSimpleProperties(EntityType type)

{

return type.Properties.Where(p => p.TypeUsage.EdmType is SimpleType && p.DeclaringType == type);

}

public IEnumerable<EdmProperty> GetSimpleProperties(ComplexType type)

{

return type.Properties.Where(p => p.TypeUsage.EdmType is SimpleType && p.DeclaringType == type);

}

public IEnumerable<EdmProperty> GetComplexProperties(EntityType type)

{

return type.Properties.Where(p => p.TypeUsage.EdmType is ComplexType && p.DeclaringType == type);

}

public IEnumerable<EdmProperty> GetComplexProperties(ComplexType type)

{

return type.Properties.Where(p => p.TypeUsage.EdmType is ComplexType && p.DeclaringType == type);

}

public IEnumerable<EdmProperty> GetPropertiesWithDefaultValues(EntityType type)

{

return type.Properties.Where(p => p.TypeUsage.EdmType is SimpleType && p.DeclaringType == type && p.DefaultValue != null);

}

public IEnumerable<EdmProperty> GetPropertiesWithDefaultValues(ComplexType type)

{

return type.Properties.Where(p => p.TypeUsage.EdmType is SimpleType && p.DeclaringType == type && p.DefaultValue != null);

}

public IEnumerable<NavigationProperty> GetNavigationProperties(EntityType type)

{

return type.NavigationProperties.Where(np => np.DeclaringType == type);

}

public IEnumerable<NavigationProperty> GetCollectionNavigationProperties(EntityType type)

{

return type.NavigationProperties.Where(np => np.DeclaringType == type && np.ToEndMember.RelationshipMultiplicity == RelationshipMultiplicity.Many);

}

public FunctionParameter GetReturnParameter(EdmFunction edmFunction)

{

ArgumentNotNull(edmFunction, "edmFunction");

var returnParamsProperty = edmFunction.GetType().GetProperty("ReturnParameters");

return returnParamsProperty == null

? edmFunction.ReturnParameter

: ((IEnumerable<FunctionParameter>)returnParamsProperty.GetValue(edmFunction, null)).FirstOrDefault();

}

public bool IsComposable(EdmFunction edmFunction)

{

ArgumentNotNull(edmFunction, "edmFunction");

var isComposableProperty = edmFunction.GetType().GetProperty("IsComposableAttribute");

return isComposableProperty != null && (bool)isComposableProperty.GetValue(edmFunction, null);

}

public IEnumerable<FunctionImportParameter> GetParameters(EdmFunction edmFunction)

{

return FunctionImportParameter.Create(edmFunction.Parameters, \_code, \_ef);

}

public TypeUsage GetReturnType(EdmFunction edmFunction)

{

var returnParam = GetReturnParameter(edmFunction);

return returnParam == null ? null : \_ef.GetElementType(returnParam.TypeUsage);

}

public bool GenerateMergeOptionFunction(EdmFunction edmFunction, bool includeMergeOption)

{

var returnType = GetReturnType(edmFunction);

return !includeMergeOption && returnType != null && returnType.EdmType.BuiltInTypeKind == BuiltInTypeKind.EntityType;

}

}

public static void ArgumentNotNull<T>(T arg, string name) where T : class

{

if (arg == null)

{

throw new ArgumentNullException(name);

}

}

#>