http://www.clipcode.net

Clipcode's Code-First Entity Framework Reference Library

Table of Contents

1: Overview....3
1.1: Introduction....3
1.2: Code-First Namespaces....4
2: System.Data.Entity....5
2.1: Overview....5
2.2: CreateDatabaseOnlyIfNotExists....6
2.3: Database....7
2.4: DbContext....12
2.5: DbModelBuilder....19
2.6: DbModelBuilderVersion....22
2.7: DbModelBuilderVersionAttribute....22
2.8: DbSet....22
2.9: DbSet<TEntity>....24
2.10: DbExtensions....26
2.11: DropCreateDatabaseIfModelChanges....28
2.12: DropCreateDatabaseAlways....29
2.13: IDatabaseInitializer....30
2.14: IDbSet....31

1: Overview

1.1: Introduction

Code-First Entity Framework is an evolution of the Entity Framework towards an architecture preferred by developers interested in domain-driven design.

Developers write code to create plain old (regular) CLR objects (POCO), and Code-First EF can use these to create a database and exchange data between the application and the database, all using a minimum of new APIs.

Microsoft delivers Code-First Entity Framework as part of "Microsoft Entity Framework 4.1". The installer can be downloaded form here:

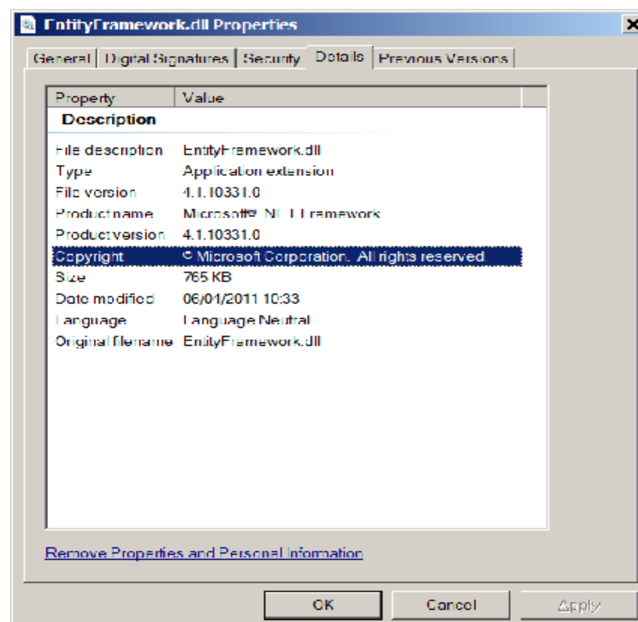
http://www.microsoft.com/downloads/en/details.aspx?FamilyID=b41c728e-9b4f-4331-a1a8-537d16c6acdf&displaylang=en

1.1.1: The Installation

When you install EF 4.1, a folder is created (assuming you are using 64-bit Windows) under:

C:\Program Files (x86)\Microsoft ADO.NET Entity Framework 4.1\Binaries

The root folder contains one sub-folder, called *Binaries* and one file called *License.rtf*. The *Binaries* folder has two files, called *EntityFramework.dll* (with the File properties below) and EntityFramework.xml (code comments).



1.2: Code-First Namespaces

There are the following namespaces in this DLL:

- System.ComponentModel.DataAnnotations
- System.Data.Entity
- System.Data.Entity.Infrastructure
- System.Data.Entity.ModelConfiguration
- System.Data.Entity.ModelConfiguration.Configuration
- System.Data.Entity.Validation

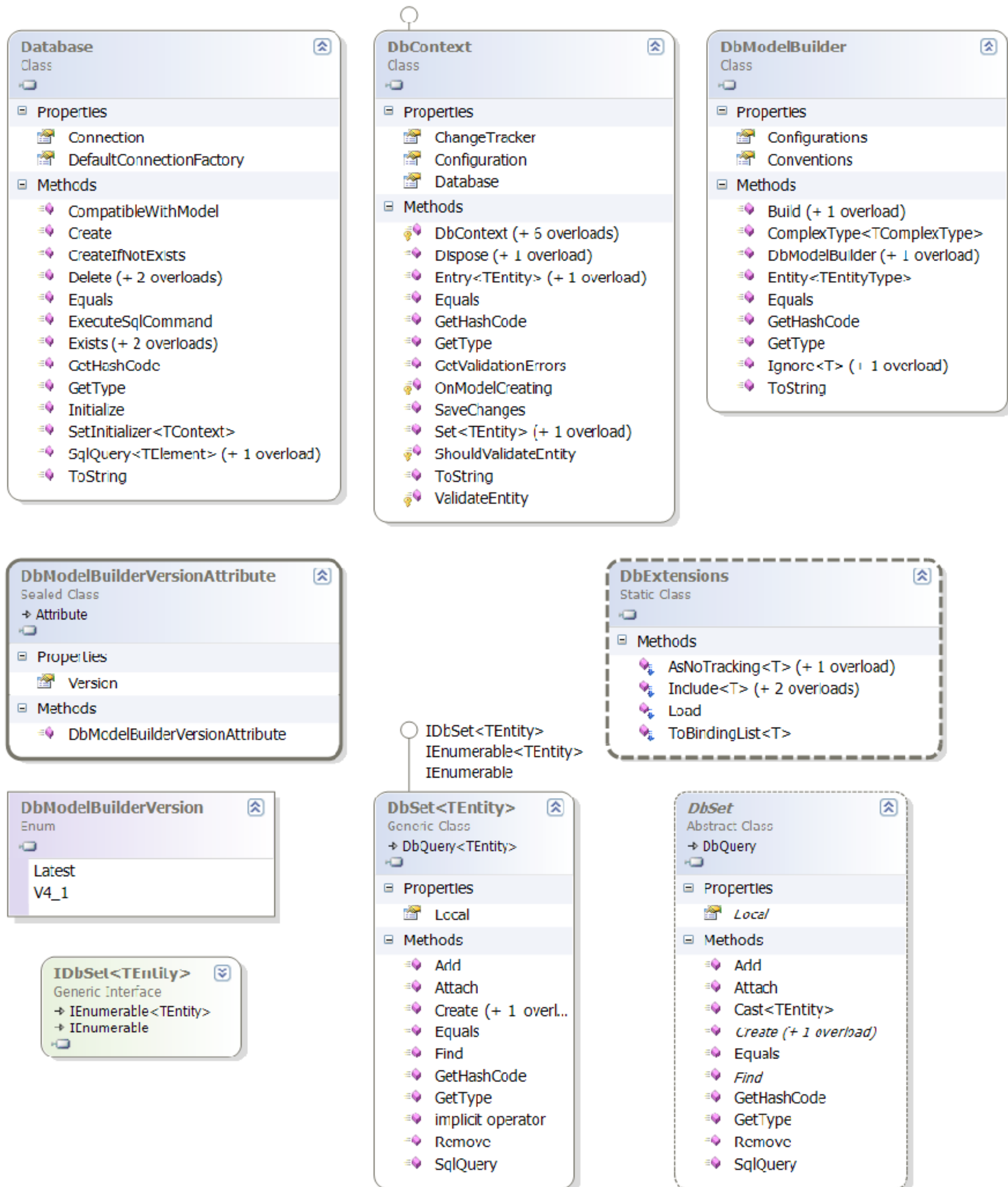
To get started, most application developers only need to use the types from the System.Data.Entity namespace and perhaps the System.ComponentModel.DataAnnotations namespace. The other namespaces are for more advanced/specialist applications.

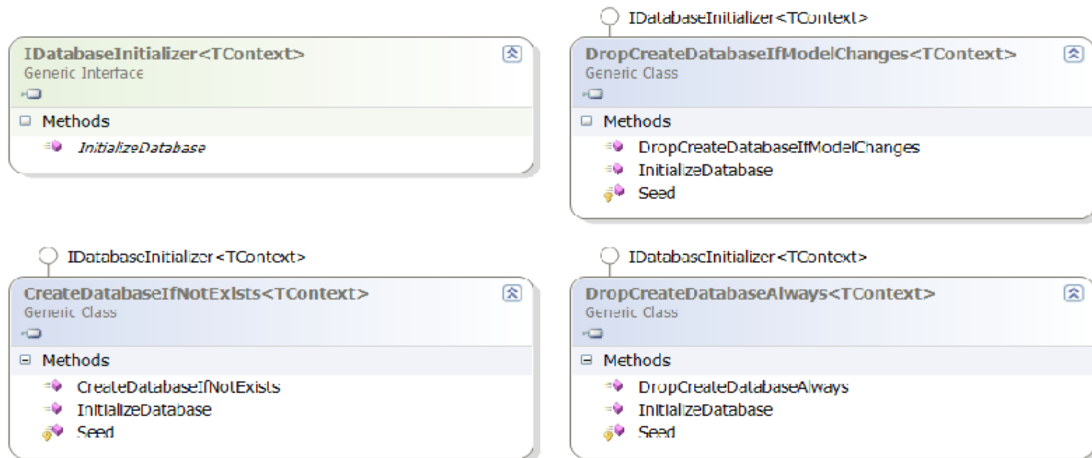
A number of data annotations are defined in the main .NET Framework (also in the System.ComponentModel.DataAnnotations namespace), and what is delivered as part of Code-First EF are additional annotations.

2: System.Data.Entity

2.1: Overview

System.Data.Entity has the following types:





2.2: CreateDatabaseOnlyIfNotExists

The database is only created if it does not exist.

```
public class CreateDatabaseOnlyIfNotExists<TContext> :
  DatabaseInitializerWithSeeding<TContext>
  where TContext : global::System.Data.Entity.DbContext {

    public CreateDatabaseOnlyIfNotExists();

    public override void InitializeDatabase(TContext context);

    protected virtual void Seed(TContext context);

}
```

Remarks

When no initializer is specified for a context, this is the default.

2.2.1: CreateDatabaseOnlyIfNotExists

Constructor for CreateDatabaseOnlyIfNotExists.

```
public CreateDatabaseOnlyIfNotExists();
```

2.2.2: InitializeDatabase

Initializer which creates the database if it does not already exist, and calls the virtual Seed method to add seed data.

```
public override void InitializeDatabase(TContext context);
```

Parameters

- context (TContext) – the context for which the database needs to be initialized

2.2.3: Seed

May be overridden to add Seed (initial) values to a newly created database (by default, none get added).

```
protected virtual void Seed(TContext context);
```

Parameters

- context (TContext) – the context for which the database needs to be seeded

2.3: Database

Represents a database.

```
public class Database {
    public DbConnection Connection { get; }
    public static IDbConnectionFactory DefaultConnectionFactory{get;set;}
    public bool CompatibleWithModel(bool throwIfNoMetadata);
    public void Create();
    public bool CreateIfNotExists();
    public bool Delete();
    public static bool Delete(DbConnection existingConnection);
    public static bool Delete(string nameOrConnectionString);
    public int ExecuteSqlCommand(string sql, params object[] parameters);
    public bool Exists();
    public static bool Exists(DbConnection existingConnection);
    public static bool Exists(string nameOrConnectionString);
    public void Initialize(bool force);
    public static void SetInitializer<TContext>(
     IDatabaseInitializer<TContext> strategy) where TContext : DbContext;
    public IEnumerable<TElement> SqlQuery<TElement>(string sql,
     params object[] parameters);
    public IEnumerable SqlQuery(Type elementType, string sql,
     params object[] parameters);
}
```

Remarks

Each DbContext has a Database property which represents the database that the context works with. It can be accessed via DbContext.Database.

Once a DbContext is instantiated, the Database property is set, but the Database.Connection has its ConnectionState set to Closed. It will be opened when needed.

The Database class have many methods to create, delete and check if the database actually exists in the data store. Often there is no need to call these directly, as they are called for you by provided implementations of database initializers. You may wish to use them directly when writing custom database initiializers.

2.3.1: Connection

The connection property represents the connection to the actual database.

```
public DbConnection Connection { get; }
```

2.3.2: CompatibleWithModel

Checks whether the model in the code is compatible with the model int eh database, based on the model hash.

```
public bool CompatibleWithModel(bool throwIfNoMetadata);
```

Parameters

- throwIfNoMetadata (bool) – Whether to throw an expception if no metadata is available

Return Value

- bool – Do the context model and database model match

2.3.3: DefaultConnectionFactory

Specifies the configurable convention to be used to create connections when needed. The connection factory represents a factory which creates connections, such as to a SQL Server database or a SQL Server CE database or a custom database.

```
public static IDbConnectionFactory DefaultConnectionFactory { get; set; }
```

Remarks

By default, this is set to SqlConnectionFactory.

A context has a number of ways in which to establish the connection. DbContext has a number of constructors, such as one which is the default constructor, one which takes a string and one which takes a DbConnection.

The static DefaultConnectionFactory is maintained once per AppDomain and defines the connection factory that is used to create connections.

2.3.4: Create

Creates a new database.

```
public void Create();
```

2.3.5: CreateIfNotExists

Creates a new database if it does not already exist.

```
public bool CreateIfNotExists();
```

Return Value

- Bool – true if the database was created.

2.3.6: Delete

Delete a database for a context.

```
public void Delete();
```

2.3.7: Delete(DbConnection)

Delete the database to which the DbConnection relates.

```
public static void Delete(DbConnection existingConnection);
```

Parameters

- existingConnection (DbConnection) – The DbConnection to the database to be deleted.

Remarks

This static method deletes the database to which the DbConnection relates.

2.3.8: Delete(string)

Delete a named database.

```
public static void Delete(string nameOrConnectionString);
```

Parameters

- nameOrConnectionString (string) – the name of the database or its connection string

Remarks

This static method deletes the database identified by the string parameter.

2.3.9: ExecuteSqlCommand

Executes a SQL command.

```
public int ExecuteSqlCommand(string sql, params object[] parameters);
```

Parameters

- sql (string) – The SQL command to execute
- parameters (params object[]) - Parameters for the command

Return Value

- int – Result from database

2.3.10: Exists

Determines if the database exists.

```
public bool Exists();
```

Return value

- bool – true if the database exists, false otherwise.

Remarks

As an alternative to this instance method, there are two static Database.Exists

that work on a DbConnection instance or a name or connection string.

2.3.11: Exists(DbConnection)

Determines if the database identified by the DbConnection exists.

```
public static bool Exists(DbConnection existingConnection);
```

Parameters

- existingConnection (DbConnection) – The DbConnection to the database whose existence is to be determined.

Return value

- bool – true if the database exists, false otherwise.

Remarks

This static method determines if the database to which the DbConnection relates exists.

2.3.12: Exists(string)

Determines if a database identified by a name or connection string exists.

```
public static bool Exists(string nameOrConnectionString);
```

Parameters

- nameOrConnectionString (string) – the name of the database or its connection string

Return value

- bool – true if the database exists, false otherwise.

Remarks

This static method deletes the database identified by the string parameter if it exists.

2.3.13: Initialize

Initializes a database.

```
public void Initialize();
```

2.3.14: SetInitializer

Sets the strategy for database initialization.

```
public static void SetInitializer<TContext>
    (IDatabaseInitializer<TContext> strategy) where TContext :
    System.Data.Entity.DbContext;
```

Type Parameters

- TContext – specifies which DbContext this initializer applies to.

Parameters

- strategy (IDatabaseInitializer<TContext>) - an implementation of IDatabaseInitializer which specifies how database initialization is to be performed (may be set to null, in which case no initialization occurs).

Remarks

The strategy design pattern (see page 315 of Design Patterns by Gamma et al.) enables different algorithms to be used interchangeably with client code. Here it is being used to allow database initialization to be delegated to custom code and to allow varying strategies to be used.

When there is a need to initialize the database, the database initializer's InitializeDatabase method will be called.

This should be called per app domain before any instance of the DbContext to which it applies is accessed. It may be called multiple times, but only the most recently set initializer at the time of the first DbContext usage of the database is called. Any initializer set after first DbContext access is ignored. By “access”, we mean when the context needs to access the database (e.g. in an Add operation), which will be after the context is instantiated.

The default initializer is CreateDatabaseOnlyIfNotExists (from the System.Data.Entity.Infrastructure namespace), which creates the database only if it does not already exists. If the database does exist, it does nothing.

If you do not wish anything to happen during database initialization, then you can pass in null as the strategy. In this case, it is your responsibility to both initially manually create and later manually update as needed the database so it matches the entity model. Exceptions may be thrown if the database and entity model get out of sync.

Note the database initializer is not responsible for setting up the database connection that is used later to read from and update the database.

See the entry for IDatabaseInitializer for further discussion of initializers.

2.3.15: SqlQuery<TElement>(String, Object[])

Constructs a SQL query that is executed when the returned enumerable is accessed.

```
public IEnumerable<TElement> SqlQuery<TElement>(string sql,
 params object[] parameters);
```

Type Parameters

- TElement – The type of the elements

Parameters

- sql (string) – The command
- parameters (params object[]) - The parameters for the SQL query
- **Return Value**
- IEnumerable<TElement> - The enumerable which returns the results

2.3.16: SqlQuery(Type, String, Object[])

Constructs a SQL query that is executed when the returned enumerable is accessed.

```
public IEnumerable SqlQuery(Type elementType, string sql,
 params object[] parameters);
```

Parameters

- elementType (Type) – The query returns elements of this type
- sql (string) – The command
- parameters (params object[]) - The parameters for the SQL query

Return Value

- IEnumerable<TElement> - The enumerable which returns the results

2.4: DbContext

The context used to interact with the database.

```
public class DbContext : IDisposable, IObjectContextAdapter {
    protected DbContext();
    protected DbContext(DbCompiledModel model);
    public DbContext(string nameOrConnectionString);
    public DbContext(DbConnection existingConnection,
              bool contextOwnsConnection);
    public DbContext(ObjectContext objectContext,
              bool dbContextOwnsObjectContext);
    public DbContext(string nameOrConnectionString,
              DbCompiledModel model);
    public DbContext(DbConnection existingConnection,
              DbCompiledModel model,
              bool contextOwnsConnection);
    public DbChangeTracker ChangeTracker { get; }
    public DbContextConfiguration Configuration { get; }
    public Database Database { get; }
    public void Dispose();
    protected virtual void Dispose(bool disposing);
    public DbEntityEntry Entry(object entity);
    public DbEntityEntry<TEntity> Entry<TEntity>(TEntity entity)
```

```
      where TEntity : class;
    public IEnumerable<DbEntityValidationResult> GetValidationErrors();
    protected virtual void OnModelCreating(DbModelBuilder modelBuilder);
    public virtual int SaveChanges();
    public DbSet<TEntity> Set<TEntity>() where TEntity : class;
    public DbSet Set(Type entityType);
    protected virtual bool ShouldValidateEntity(
      DbEntityEntry entityEntry);
    protected virtual DbEntityValidationResult
     ValidateEntity(DbEntityEntry entityEntry,
        IDictionary<object, object> items);
  }
```

Remarks

Think of DbContext as a simplified version of ObjectContext. It manages a database connection (Database), model metadata and a cache of (possibly updated) objects, which are sent to the database with a SaveChanges() call. It does this with the assistance of an internally managed ObjectContext instance (accessible via IObjectContextAdapter.ObjectContext).

During model construction, OnModelCreating can be overridden to shape the model being created.

DbContext has seven constructors – two protected and five public. It is common to derive a custom context from DbContext and use the default constructor. For more specialist scenarios, an additional technique to create DbContexts is to manually create a DbCompiledModel, and then call:
DbCompiledModel.CreateObjectContext<myCustomContext>(myConnection);

DbContext uses lazy initialization, in the sense that creation of the underlying object context is deferred until needed (it is not performed in the constructor).

DbContext is highly flexible in how it decides which connection to use. If a DbConnection instance is provided in a constructor, that is used. If a ObjectContext instance is provided in a constructor, then the connection property from the ObjectContext is used. If the default constructor is used, the connection strings section of app.config or web.config is examined, to see if an entry with that name exists (either the simple name of the custom DbContext class (“MyContext”), or the full name (“MyNamespace.MyContext”)). If a DbContext constructor that takes a name is used, a similar check is made for the name in app.config or web.config. If nothing is found, then the connection convention is used. This is a configurable connection factory convention, by default it is set to SqlConnectionFactory which creates a database in the SQLExpress store.

2.4.1: DbContext

Creates a DbContext based on settings that can be accessed via convention.

```
protected DbContext();
```

Remarks

For many developers, this will be the most frequently used of the DbContext constructors. They will derive a custom class from DbContext, add some DbSet instances to it, and simply instantiate the custom context.

2.4.2: DbContext(DbCompiledModel)

Create a DbContext based on an existing DbCompiledModel.

```
protected DbContext(DbCompiledModel model);
```

Parameters

- model (DbCompiledModel) – The DbCompiledModel which will be used to create the ObjectContext

Remarks

When you manually create the DbCompiledModel, then the derived context's OnModelCreating method (if any) is not called (the model is created before the context exists).

2.4.3: DbContext(String)

Creates a DbContext whose connection should be created based on the provided name / connection string.

```
public DbContext(string nameOrConnectionString);
```

Parameters

- nameOrConnectionString (string) – name or full connection string

Remarks

First, the Connection Strings section of App.Config or Web.Config is examined to see if it has an entry that matches nameorConnectionString, and if so, it is used to create the DbConnection. If not, nameOrConnectionString is passed to the configured connection factory, which establishes the connection. A custom factories can be registered using Database.DefaultConnectionFactory and this can also be used to access the current factory.

2.4.4: DbContext(String, bool)

Creates a DbContext whose connection should be created based on the provided name / connection string, and specifying who owns the connection.

```
public DbContext(DbConnection existingConnection,
                 bool contextOwnsConnection);
```

Parameters

- nameOrConnectionString (string) – name or full connection string
- dbContextOwnsObjectContext (bool) – who owns the context (and can close it)

Remarks

First, the Connection Strings section of App.Config or Web.Config is examined to see if it has an entry that matches nameorConnectionString, and if so, it is used to create the DbConnection. If not, nameOrConnectionString is passed to the configured connection factory, which establishes the connection. A custom factories can be registered using Database.DefaultConnectionFactory and this can also be used to access the current factory.

2.4.5: DbContext(ObjectContext, bool)

Create a DbContext based on an existing ObjectContext.

```
public DbContext(ObjectContext objectContext,
             bool dbContextOwnsObjectContext);
```

Parameters

- objectContext (ObjectContext) – an existing ObjectContext
- dbContextOwnsObjectContext (bool) – who owns the context (and can close it)

Remarks

When a different DbContext constructor is used (i.e. one that does not accept an ObjectContext instance), then the object context is constructed internal to DbContext by it calling DbModel.CreateObjectContext().

2.4.6: DbContext(DbConnection, DbModel)

Create a DbContext based on existing DbConnection and DbModel instances.

```
public DbContext(DbConnection existingConnection,
                                  DbCompiledModel model,
                                  bool contextOwnsConnection);
```

Parameters

- existingConnection (DbConnection) - connection to a database
- model (DbModel) - The DbModel which will be used to create the ObjectContext
- dbContextOwnsObjectContext (bool) – who owns the context (and can close it)

Remarks

When you manually create the DbModel, then the derived context's OnModelCreating method (if any) is not called (the model is created before the context exists).

The DbConnection must be in a closed state.

2.4.7: DbContext(string, DbCompiledModel)

Create a DbContext which connection should be created based on the provided name / connection string and based on the provided DbModel.

```
public DbContext(string nameOrConnectionString,
          DbCompiledModel model);
```

Parameters

- nameOrConnectionString (string) - name or full connection string
- model (DbModel) - The DbModel which will be used to create the ObjectContext

Remarks

When you manually create the DbModel, then the derived context's OnModelCreating method (if any) is not called (the model is created before the context exists).

2.4.8: ChangeTracker

Returns the System.Data.Entity.Infrastructure.ChangeTracker object.

```
public DbChangeTracker ChangeTracker { get; }
```

2.4.9: Configuration

Returns the DbContextConfiguration object used by this DbContext.

```
public DbContextConfiguration Configuration { get; }
```

2.4.10: Database

Returns the System.Data.Entity.Database this context works with.

```
public Database Database { get; }
```

Remarks

Note this property has a getter only. It is created internally by the DbContext, based on connection information passed in in a constructor or via conventions.

2.4.11: IObjectContextAdapter.ObjectContext (Explicit Interface)

Returns the ObjectContext used to interact with the underlying database.

```
System.Data.Objects.ObjectContext IObjectContextAdapter.ObjectContext { get; }
```

2.4.12: Dispose

Dispose of this DbContext instance.

```
public void Dispose();
```

2.4.13: Dispose(bool)

Dispose of this DbContext instance.

```
protected virtual void Dispose(bool disposing);
```

Parameters

- disposing (bool) – Should both unmanaged and managed resources be released (if true) or only the unmanaged resources (false)

2.4.14: Entry

Returns a DbEntityEntry for the specified entity.

```
public DbEntityEntry Entry(object entity);
```

Parameters

- entity (object) – The entity for which a DbEntityEntry is required

Return Value

- DbEntityEntry – The entity's entry

2.4.15: Entry<TEntity>

Returns a DbEntityEntry for the specified entity.

```
public DbEntityEntry<TEntity> Entry<TEntity>(TEntity entity)
    where TEntity : class;
```

Type Parameters

- TEntity – The type of the entity

Parameters

- entity (object) – The entity for which a DbEntityEntry is required

Return Value

- DbEntityEntry – The entity's entry

2.4.16: GetValidationErrors

Causes validation of tracked entities.

```
public IEnumerable<DbEntityValidationResult> GetValidationErrors();
```

Return Value

- IEnumerable<DbEntityValidationResult> - Validation results

2.4.17: OnModelCreating

Allows custom shaping of the model.

```
protected internal virtual void OnModelCreating(ModelBuilder modelBuilder);
```

Parameters

- modelBuilder (ModelBuilder) – the Builder for the model.

Remarks

If you wish to make adjustments to the model that will be created above and beyond what is available via conventions and data annotations, then you can use the Fluent API, and such calls are usually placed n a derived DbContext and its custom implementation of OnModelCreating.

The model builder is collecting a set of configurations represents what should be in the model. With the Fluent API, you can customize these.

The base method is blank, so you do not have to call it.

2.4.18: SaveChanges

Saves any cached changes back to the database.

```
public virtual int SaveChanges();
```

Remarks

Since this is a virtual method, if creating a derived implementation, you could override SaveChanges and from the ObjectContext.ObjectStateManager determine the altered entities (if there is a need for custom logging, etc.).

This represents the context's unit of work.

2.4.19: Set<TEntity>

Creates a DbSet<TEntity> as part of the model.

```
public DbSet<TEntity> Set<TEntity>() where TEntity : class;
```

Type Parameters

- TEntity – The entity type

Return Value

- DBSet<TEntity> - the instantiated DbSet

Remarks

Note the different between this method and ModelBuilder.Entity<TEntity>().

```
public EntityConfiguration<TEntity> Entity<TEntity>();
```

DbContext.Set<TEntity> returns a DbSet which can be used for CRUD operations using an entity set. In contrast, ModelBuilder.Entity merely adds an EntityConfiguration to the model.

2.4.20: Set(Type)

Creates a DbSet as part of the model.

```
public DbSet Set(Type entityType);
```

Parameters

- entityType (Type) – The entity type

Return Value

- DBSet - the instantiated DbSet

2.4.21: ShouldValidateEntity

Allows whether validation is required to be specified in derived classes.

```
protected virtual bool ShouldValidateEntity(
   DbEntityEntry entityEntry);
```

Parameters

- entityEntry (DbEntityEntry) – What needs to be validated

Return Value

- bool – Should validation occur

2.4.22: ValidateEntity

Allows custom validation in derived classes.

```
protected virtual DbEntityValidationResult
  ValidateEntity(DbEntityEntry entityEntry,
      IDictionary<object, object> items);
```

Parameters

- entityEntry (DbEntityEntry) – What needs to be validated
- items (IDictionary<object, object>) - Custom validation information

Return Value

- DbEntityValidationResult – validation result

2.5: DbModelBuilder

Utility class used to construct data model from code.

```
public class DbModelBuilder {
    public DbModelBuilder();
    public DbModelBuilder(DbModelBuilderVersion modelBuilderVersion);
    public virtual ConfigurationRegistrar Configurations { get; }
    public virtual ConventionsConfiguration Conventions { get; }
    public virtual DbModel Build(DbConnection providerConnection);
```

```
    public virtual DbModel Build(DbProviderInfo providerInfo);
    public virtual ComplexTypeConfiguration<TComplexType>
            ComplexType<TComplexType>() where TComplexType : class;
    public virtual EntityTypeConfiguration<TEntityType>
            Entity<TEntityType>() where TEntityType : class;
    public virtual DbModelBuilder Ignore<T>() where T : class;
    public virtual DbModelBuilder Ignore(IEnumerable<Type> types);
  }
```

Remarks

An easy way to understand the relationship is to consider that ModelBuilder is to DbModel what StringBuilder is to String.

With Code-First EF, developers write code (e.g. in C#) to create POCOs, and something (namely ModelBuilder) needs to be able to extract the model from the code.

2.5.1: ModelBuilder()

Constructor for ModelBuilder using the Latest ModelBuilderVersion.

```
public ModelBuilder();
```

2.5.2: ModelBuilder(DbModelBuilderVersion)

Constructor for ModelBuilder using a specific ModelBuilderVersion.

```
public DbModelBuilder(DbModelBuilderVersion modelBuilderVersion);
```

2.5.3: Configurations

A collection of configurations for structural types, such as entity types.

```
public virtual ConfigurationRegistrar Configurations { get; }
```

Remarks

Configurations can later be added to this list by calling ModelBuilder.Entity<TEntity>().

2.5.4: Conventions

A collection of convention configurations.

```
public virtual ConventionsConfiguration Conventions { get; }
```

2.5.5: Build(DbConnection)

Builds a DbModel based on the DbConnection.

```
public virtual DbModel Build(DbConnection providerConnection);
```

Return Value

- DbModel – the created DbModel

Remarks

Calls to Build() are expensive, so CompiledModel instances should be be created

from DbModel and then cached.

2.5.6: Build(DbProviderInfo)

Builds a DbModel based on the DbProviderInfo.

```
public virtual DbModel Build(DbProviderInfo providerInfo);
```

Return Value

- DbModel – the created DbModel

Remarks

Calls to Build() are expensive, so CompiledModel instances should be be created from DbModel and then cached.

2.5.7: Entity

Registers an entity type.

```
public virtual EntityTypeConfiguration<TEntityType>
                    Entity<TEntityType>() where TEntityType : class;
```

Type Parameters

- TEntityType – the entity type

Return Value

- EntityTypeConfiguration<TEntity> - This is useful for Fluent API usage

2.5.8: Ignore<T>

Ignores a type (so that it is not part of the model).

```
public virtual DbModelBuilder Ignore<T>() where T : class;
```

Type Parameters

- T – Entity type to ignore

Return Value

- DbModelBuilder – A model builder which ignores the specified type

2.5.9: Ignore<IEnumerable<Type>

Ignores a collection of types (so that they are not part of the model).

```
public virtual DbModelBuilder Ignore(IEnumerable<Type> types);
```

Type Parameters

- T – Entity type to ignore

Return Value

- DbModelBuilder – A model builder which ignores the specified collection of type

2.6: DbModelBuilderVersion

An enumeration specifying which version of the DbContext and model builder conventions to use when constructing a model from code.

```
public enum DbModelBuilderVersion {
    Latest = 0,
    V4_1 = 1,
  }
```

Remarks

Latest means use the latest available version, and V4_1 means use EF 4.1. For now, since there is only one version, 4.1, they actually result in the same model builder functionality. In future, when a later version exists and has additional/different functionality, then this enum becomes important. When Latest is specified, then all the conventions of the latest available DbContext & model builder are to be used, whereas when V4_1 is specified, they only conventions defined for that version is to be used.

The DbModelBuilder and DbModelBuilderAttribute types accept DbModelBuilderVersion parameters.

2.7: DbModelBuilderVersionAttribute

An attribute which can be applied to DbContext-derived types that states which version of conventions to use.

```
public sealed class DbModelBuilderVersionAttribute : Attribute {
    public DbModelBuilderVersionAttribute(DbModelBuilderVersion version);
    public DbModelBuilderVersion Version { get; }
}
```

Remarks

When not specified, Latest is used.

2.7.1: Constructor

Creates a new version attribute.

```
public DbModelBuilderVersionAttribute(DbModelBuilderVersion version);
```

2.7.2: Version

Returns the version to use.

```
public DbModelBuilderVersion Version { get; }
```

2.8: DbSet

A non-generic version of DbSet<TEntity>.

```
public abstract class DbSet : DbQuery  {
```

```
        public abstract IList Local { get; }
        public object Add(object entity);
        public object Attach(object entity);
        public DbSet<TEntity> Cast<TEntity>() where TEntity : class;
        public abstract object Create();
        public abstract object Create(Type derivedEntityType);
        public abstract object Find(params object[] keyValues);
        public object Remove(object entity);
        public DbSqlQuery SqlQuery(string sql, params object[] parameters);
    }
```

2.8.1: Add

Adds an entity.

```
public object Add(object entity);
```

Parameters

- entity (TEntity) – The entity to be added

2.8.2: Attach

Attaches an existing entity to a context (in an unchanged state).

```
public object Attach(object entity);
```

Parameters

- entity (TEntity) – The entity to attach

2.8.3: Cast

Casts to an entity type.

```
public DbSet<TEntity> Cast<TEntity>() where TEntity : class;
```

2.8.4: Create()

Creates an instance of the entity.

```
public abstract object Create();
```

Return Value

- TEntity – The newly created entity

2.8.5: Create()

Creates an instance of the entity or a derived entity.

```
public abstract object Create(Type derivedEntityType);
```

Return Value

- TDerivedEntity – The newly created entity

2.8.6: Find

Finds an entity in the context or the database.

```
public abstract object Find(params object[] keyValues);
```

Parameters

- keyValues (params object[]) -

Return Value

- TEntity – The found entity (or null, if not found)

2.8.7: Local

A local view of entities from this set which have been added/unchanged/modified.

```
public abstract IList Local { get; }
```

2.8.8: Remove

Removes an entity.

```
public object Remove(object entity);
```

Parameters

- entity (TEntity) – The entity to be removed.

2.8.9: SqlQuery

Create a SQL query.

```
public DbSqlQuery SqlQuery(string sql, params object[] parameters);
```

2.9: DbSet<TEntity>

A collection of an entity type.

```
public interface IDbSet<TEntity> : IQueryable<TEntity>,
IEnumerable<TEntity>, IQueryable, IEnumerable where TEntity : class {
    ObservableCollection<TEntity> Local { get; }
    TEntity Add(TEntity entity);
    TEntity Attach(TEntity entity);
    TEntity Create();
    TDerivedEntity Create<TDerivedEntity>()
                        where TDerivedEntity : class, TEntity;
    TEntity Find(params object[] keyValues);
    TEntity Remove(TEntity entity);
  }
```

Remarks

When using auto-discovery (the usual case), when you add DbSets to a DbContext, they should be properties, not fields.

```
CORRECT:
        public DbSet<Room> Rooms;
        public DbSet<Building> Buildings;
INCORRECT:
        public DbSet<Room> Rooms { get; set; }
        public DbSet<Building> Buildings { get; set; }
```

Note that DbSet<T> has no public constructors so you cannot directly instantiate them, and you cannot derive from them. There are two ways to create DbSets. One is to define a reference in your DbContext-derived class to either DbSet<T> or IDbSet<T>, and during model initialization, Code-First EF will automatically instantiate the DbSet<T> and set you reference to point to it. Alternatively, you can call DbContext.Set<T> which returns an instantiated DBSet<T>. For test driven development purposes, pay attention to IDbSet also.

2.9.1: Add

Add an entity to a DbSet.

```
void Add(TEntity entity);
```

Parameters

- entity (TEntity) – The entity to be added.

Remarks

DbContext.SaveChanges() must be called to persist the change back to the database.

2.9.2: Attach

Attaches an existing entity to a context (in an unchanged state).

```
void Attach(TEntity entity);
```

Parameters

- entity (TEntity) – The entity to attach.

Remarks

DbContext.SaveChanges() must be called to persist the change back to the database.

2.9.3: Find

Finds an entity in the context or the database.

```
TEntity Find(params object[] keyValues);
```

Parameters

- keyValues (params object[]) -

Return Value

- TEntity – The found entity (or null, if not found)

Remarks

The primary key is supplied as one or more objects to this method. The im-memory objects for this type are first searched, and if a match if a match is found, it is returned. Otherwise the database is searched, and if a match is found, it is returned. If no match is found, null is returned.

2.9.4: Remove

Removes an entity.

```
void Remove(TEntity entity);
```

Parameters

- entity (TEntity) – The entity to be removed.

Remarks

DbContext.SaveChanges() must be called to persist the change back to the database.

2.10: DbExtensions

A class containing extension methods for IQueryable.

```
public static class DbExtensions {
    public static IQueryable<T> AsNoTracking<T>(
            this IQueryable<T> source) where T : class;
    public static IQueryable AsNoTracking(this IQueryable source);

    public static IQueryable<T> Include<T, TProperty>(
            this IQueryable<T> source,
            Expression<Func<T, TProperty>> path) where T : class;
    public static IQueryable<T> Include<T>(
            this IQueryable<T> source, string path) where T : class;
    public static IQueryable Include(
            this IQueryable source, string path);

    public static void Load(this IQueryable source);

    public static BindingList<T> ToBindingList<T>(
            this ObservableCollection<T> source) where T : class;
}
```

These methods extend IQueryable to allow the loading of related entities. The

difference between them is that one accepts a string, and the other a lambda, which is better because of compiler checking, Intellisense support and refactoring.

2.10.1: AsNoTracking

Preventing caching of entities.

```
public static IQueryable<T> AsNoTracking<T>(
          this IQueryable<T> source) where T : class;
public static IQueryable AsNoTracking(this IQueryable source);
```

Parameters

- source (this IQueryable) – the query

Return Value

- IQueryable<T> - Query with no tracking if supported

2.10.2: Include(Expression)

Include a related entity based on a lambda.

```
public static IQueryable<T> Include<T, TProperty>(
          this IQueryable<T> source,
          Expression<Func<T, TProperty>> path) where T : class;
public static IQueryable<T> Include<T>(
          this IQueryable<T> source, string path) where T : class;
public static IQueryable Include(
          this IQueryable source, string path);
```

Type Parameters

- T – the selected entity

Parameters

- source (IQueryable) – The base IQueryable
- path (Expression<Func<T, object>>) - The lambda identifying the property of the related entity which is to be loaded
- path (string) - The name of the property of the related entity which is to be loaded

Return Type

- IQueryable – The IQueryable to be used for the Include to take effect.

Remarks

Multi-level includes are also supported (when the selected entity has a related entity, and the related entity itself has a related entity, and we wish to retrieve all in one go). For a reference property, use dotted notation. For a collection, use Select.

2.10.3: Load

Loads entity instances.

```
public static void Load(this IQueryable source);
```

Parameters

- source (IQueryable) – The query

2.10.4: ToBindingList

Creates a binding list (System.ComponentModel namespace) that is synchronized with the observable collection.

```
public static BindingList<T> ToBindingList<T>(
            this ObservableCollection<T> source) where T : class;
```

Parameters

- source (IQueryable) – The collection

Return Type

- BindingList<T> – The binding list

2.11: DropCreateDatabaseIfModelChanges

Re-create the database in the model changes.

```
public class DropCreateDatabaseIfModelChanges<TContext> :
                     IDatabaseInitializer<TContext> where TContext
                     : global::System.Data.Entity.DbContext {

    public DropCreateDatabaseIfModelChanges();

    public void InitializeDatabase(TContext context);

    protected virtual void Seed(TContext context);

}
```

Remarks

Code-First EF mantains a table in the database called EdmMetadata which has two columns – Id (int, primary key) and ModelHash (nvarchar 4000, nullable).

For each context that is supported, an entry is made in this table. The Id for the first context is set to 1 and increments by 1 for each other context (if any). The model has is calculated over the topology of the model, and stored as 64 characters in hex format.

When a change is made to the model, then the model hash changes too. This is how Code-First EF knows when the code is or is not in sync with the database. If the model changes, then DropCreateDatabaseIfModelChanges will re-create the

database.

2.11.1: DropCreateDatabaseIfModelChanges

Constructor for RecreateDatabaseIfModelChanges.

```
public DropCreateDatabaseIfModelChanges();
```

2.11.2: InitializeDatabase

Initializer which re-creates database if model hash has changed.

```
public override void InitializeDatabase(TContext context);
```

Parameters

- context (TContext) – the context for which the database needs to be initialized

2.11.3: Seed

May be overridden to add Seed (initial) values to a newly created database (by default, none get added).

```
protected virtual void Seed(TContext context);
```

Parameters

- context (TContext) – the context for which the database needs to be seeded

2.12: DropCreateDatabaseAlways

Database initializer which always re-creates the database (regardless of whether the model has changed or not).

```
public class DropCreateDatabaseAlways<TContext>
                        : IDatabaseInitializer<TContext> where TContext
                        global::System.Data.Entity.DbContext {

    public DropCreateDatabaseAlways();

    public void InitializeDatabase(TContext context);

    protected virtual void Seed(TContext context);

}
```

Remarks

This is often used for unit testing, those has heavy performance implications.

2.12.1: DropCreateDatabaseAlways

Constructor for DropCreateDatabaseAlways.

```
public DropCreateDatabaseAlways();
```

2.12.2: InitializeDatabase

Initialise the database by always recreating it.

```
public override void InitializeDatabase(TContext context);
```

Parameters

- context (TContext) – the context for which the database needs to be initialized

2.12.3: Seed

May be overridden to add Seed (initial) values to a newly created database (by default, none get added).

```
protected virtual void Seed(TContext context);
```

Parameters

- context (TContext) – the context for which the database needs to be seeded

2.13: IDatabaseInitializer

The interface that represents what needs to be done when a database for a context needs to be initialized.

```
public interface IDatabaseInitializer<in TContext> where TContext :
                                                global::System.Data.Entity.DbContext {
    void InitializeDatabase(TContext context);
}
```

Remarks

You have five options when deciding which DatabaseInitializer to use.

You could use of the three supplied classes – CreateDatabaseIfNotExists, DropCreateDatabaseAlways, or DropCreateDatabaseIfModelChanges – each of which does exactly what its name suggests.

You could also set the initializer to be null, in which case nothing happens.

Finally you could create a custom initializer.

The default is set to CreateDatabaseIfNotExists.

If you do not correctly set up the database initializer, and the database and model diverge, then an exception is raised with the following error message:

> *The model backing the <xyzContext> context has changed since the database was created. Either manually*

> *delete/update the database, or call Database.SetInitializer with an IDatabaseInitializer instance. For example, the DropCreateDatabaseIfModelChanges strategy will automatically delete and recreate the database, and optionally seed it with new data.*

To create a custom initializer, create a class that implements IDatabaseInitializer. In the InitializeDatabase method, first determine if the database exists (Database.Exists) or if the entity model does not match what is in the existing database. If the database needs to be re-created, delete the existing datbase (Database.Delete) and re-create (Database.Create). Then add whatever custom functionality you wish to have in your initializer.

2.13.1: InitializeDatabase

The method that is called when the database needs to be initialized.

```
void InitializeDatabase(TContext context);
```

Parameters

- context (TContext) – the context for which the database needs to be initialized

Remarks

Use Database.SetInitializer to set up an initializer for each context. The best way to think about database initializer is to imagine the Database class maintains a table of triples: initializer, context type and a boolean recording whether the initializer has been called yet in the curent app domain (initially set to false). Every time Database.SetInitializer is called with a context type parameter, the initializer for that context's triple is updated. When the context is first used, the initializer is called (and the boolean set to true). Two important points are firstly that the initializers are called not from Database.SetInitializer, but rather when the context type is first used, and secondly the initializer is only called once per app domain, so later calls to Database.SetInitializer have no effect (but cause no exceptions).

2.14: IDbSet

Represents an interface to a set of entities which allows mapping out implementations of one persistence layer for another (e.g. during testing).

```
public interface IDbSet<TEntity> : IQueryable<TEntity>, IEnumerable<TEntity>,
IQueryable, IEnumerable where TEntity : class {

    ObservableCollection<TEntity> Local { get; }

    TEntity Add(TEntity entity);
```

```
    TEntity Attach(TEntity entity);
    TEntity Create();
    TDerivedEntity Create<TDerivedEntity>() where TDerivedEntity : class, TEntity;
    TEntity Find(params object[] keyValues);
    TEntity Remove(TEntity entity);
  }
```

Remarks

To support testing, you may define an interface to your context, which includes IDbSet<T> references for the various entity types you wish to work with. Provide an implementation of this context interface that derives from DBContext and is used for to actually interact with the database, and a test version that does not. Note that when Code-First EF is checking the context to discover the entities, it checks for both DbSet<T> and IDbSet<T> references.

2.14.1: Add

Adds an entity.

```
void Add(TEntity entity);
```

Parameters

- entity (TEntity) – The entity to be added

2.14.2: Attach

Attaches an existing entity to a context (in an unchanged state).

```
void Attach(TEntity entity);
```

Parameters

- entity (TEntity) – The entity to attach

2.14.3: Create()

Creates an instance of the entity.

```
TEntity Create();
```

Return Value

- TEntity – The newly created entity

2.14.4: Create()

Creates an instance of the entity or a derived entity.

```
TDerivedEntity Create<TDerivedEntity>() where TDerivedEntity : class, TEntity;
```

Return Value

- TDerivedEntity – The newly created entity

2.14.5: Find

Finds an entity in the context or the database.

```
TEntity Find(params object[] keyValues);
```

Parameters

- keyValues (params object[]) -

Return Value

- TEntity – The found entity (or null, if not found)

2.14.6: Local

A local view of entities from this set which have been added/unchanged/modified.

```
ObservableCollection<TEntity> Local { get; }
```

2.14.7: Remove

Removes an entity.

```
void Remove(TEntity entity);
```

Parameters

- entity (TEntity) – The entity to be removed.