Entity Framework [Code First]

Contents

Entity Framework [Code First] 1
Models, mapping and navigation properties 1
Nullable data types 1
DbContext 2
Connection string 2
Accessing the database 3
Locating the database 3
Never change anything through the Server Explorer! 5
Working with Entity Framework in layered applications 6
Eager Loading 6
Workflow with Entity Framework Code First 7
Database Strategy 7
Problems and fixes 9
1. The Database is used by another process 9
2. An object with the same key already exists in the ObjectStateManager 9
3. The operation cannot be completed because the DbContext has been disposed. 10
4. DbUpdateException 10
1. An overflow occurred while converting to datetime. 11

Models, mapping and navigation properties

First define the models (domain objects) that your application will use, to define a model just write a regular class that represents the model. Entity Framework (EF) will use these classes to make the tables, relations etc. in the database. Below is an example of the models Book and Author.

```
class Book
{
    public int Id { get; set; }
    public string Title { get; set; }

    // A book has a referens to an author-object and then the
    // Book-table will contain a foreign key to the Author-table
    public virtual Author Author { get; set; }
}

class Author
{
    [Key]
    public int Id { get; set; }
    [Required]
    public string Name { get; set; }

    // A author may have written several books, one-to-many
   public virtual ICollection<Book> Books { get; set; }
}
```

Each class will map to a database table, the scalar properties (properties with types such as string, int, double, etc.) will be mapped as columns in the table. Entity Framework uses a lot of conventions to achieve the mapping automatically and one of those conventions is that a property with the name “Id” will become the primary key column in the database table, for instance the Book-class above. For a full list of these conventions, check out: https://msdn.microsoft.com/en-us/library/jj679962(v=vs.113).aspx

If Entity Framework can’t resolve the mapping or if you want to override it yourself, you have the possibility to add Data Annotations. Data Annotations is attributes to your classes and properties that denote a specific behavior, for instance in the Author-class above we have included the Key-attribute to the Id-property and the Required-attribute to the Name-property. Similarly, if a foreign key cannot be automatically resolved by the entity framework, there is a [ForeignKey(“ForeignID”)] attribute that can specify which field in the foreign object is the key. Web resource for mapping and Data Annotations: https://msdn.microsoft.com/en-us/library/jj591583(v=vs.113).aspx

Properties declared as a type of another model is called navigation properties and they will map to relations between tables. If the navigation property is declared as a collection of another model it will create a one-to-many relationship between the tables in the database, if it just contains a reference it will map to a foreign key in the table.

Nullable data types

The classes we define will map to database-tables, in some scenarios it's needed to have an optional field i.e a field that could contain null, like a DateTime or integer. These data types can be assigned null in the database but not in C# code (because they are value types) which could be problematic. Nullable data types can be helpful in these situations. Defining a nullable type is very similar to defining the equivalent non-nullable type with the addition of the '?' type modifier. To define a nullable-integer, you would do the following declaration: **int? myNumber**, and for a date field: **DateTime? myDate**. These variables can be assigned a normal integer or DateTime value but they can also contain null (web resource for nullable data types: http://msdn.microsoft.com/en-us/library/2cf62fcy(v=vs.80).aspx)

DbContext

The DbContext is a class which represents a session with the database, allowing us to query and save data. To use the DbContext-class you need to define a derived context i.e. define a context that derives (inherits) from System.Data.Entity.DbContext and exposes a typed DbSet<TEntity> for each class in our model. The LibraryContext-class below is an example of such a class.

```
    public class LibraryContext : DbContext
    {
        public DbSet<Book> Books { get; set; }
        public DbSet<Author> Authors { get; set; }
    }
```

It is only the classes that are included as a DbSet-property in this class that is considered as models i.e. the classes that will generate tables in the database.

Connection string

When working with Entity Framework Code First we let the models define the database structure but we haven't defined where the actual database should be created. In the app.Config file you need to add a connection-string like below. The name-attribute maps to the name of our LibraryContext-class, the connectionString-attribute points to where the database will be created and the providerName-attribute decides what kind of database that should be used.

```
<connectionStrings>
  <add name="Library.Models.LibraryContext"
       connectionString="data source=(LocalDb)\MSSQLLocalDB;
                         initial
catalog=Library.Models.LibraryContext;
                         integrated security=True;
                         MultipleActiveResultSets=True;
                         App=EntityFramework;
                         AttachDBFilename=|DataDirectory|\Library.mdf"
       providerName="System.Data.SqlClient" />
```

The database is not yet created though it is created only first when it's accessed for the first time, see section below for how accessing the database through the LibraryContext-class.

Accessing the database

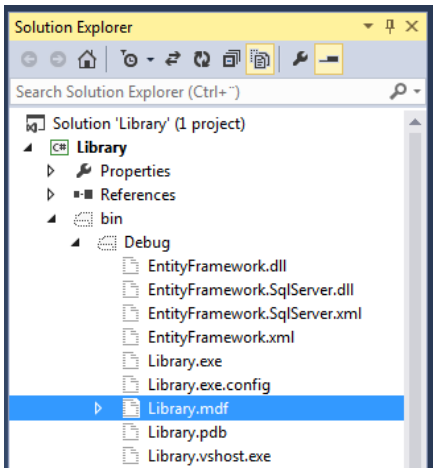
To access the database through the LibraryContext-class we need to instantiate an object from it and then access the database tables through the DbSet-properties. To add rows of data to a table is as easy of adding a new object to a collection of items, se example below:

```
LibraryContext db = new LibraryContext();

// Create the author-object
Author alexDumas = new Author()
{
    Name = "Alexandre Dumas"
};

// Create the book-object and assign the
// Author-property with the author
Book monteCristo = new Book()
{
    Title = "The Count of Monte Cristo",
    Author = alexDumas
};

// Add the book to the DbSet of books.
db.Books.Add(monteCristo);

// Persist changes to the database
db.SaveChanges();
```

In the code above first an author-object is created and then a new book-object is created. The author-object is then assigned to the Author navigation property of the book-object. Entity Framework will notice that the book-object has a new author attached to its navigation property and will also store the author-object to the database.

Locating the database

When accessing the database for the first time Entity Framework will create the actual database, it will be located the bin/Debug-folder if the connection-string given above was used.



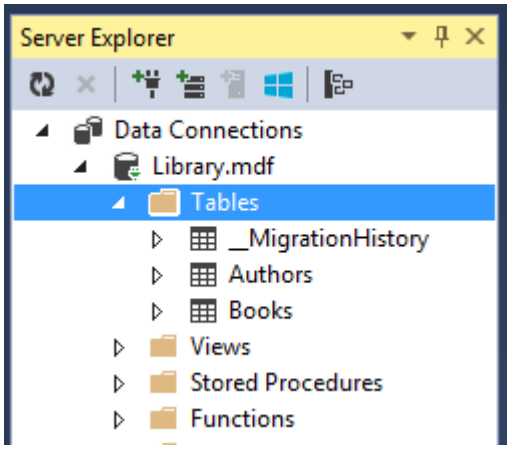
To be able to see the bin/Debug-folder in the solution explorer first click the “Show All Files” button highlighted in the picture.

To open the database in the Server Explorer right click the database file -> Open.

The database location can differ on your system because it depend on some paths set in the OS. With the connectionstring described above you can force the location of the database to this location by adding the following line of code as the first code line your applications main entry point i.e. the “static void Main()” method:

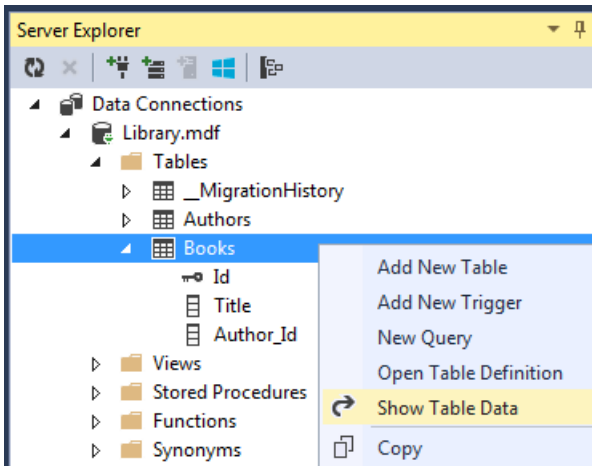
```
// Set the DataDirectory (used in the connection-string) to be the executing app directory
```

The database will open in the Server Explorer and you could browse through the tables that have been generated by Entity Framework and see that it contains the desired columns, data types, primary keys, foreign keys and relationships.



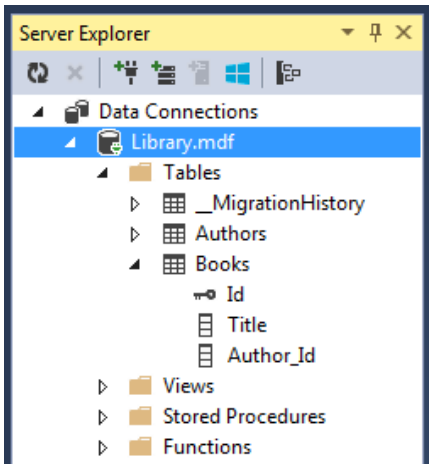
To view more information about a table or view relationships between tables right click a table -> Open Table Definition Properties. Note that a relationship can only be viewed from the foreign key table, for instance to view the relationship that one author may have written several books (one-to-many) is viewed by going through the Books-table.

To show the rows of a table just right click the table > Show Table Data.



Never change anything through the Server Explorer!

Don't modify any columns, relationships, keys etc. through the Server Explorer. All changes made this way will be overwritten when you change the models in your application and Entity Framework will drop and recreate the database to match the models. Also, when you open a connection in the Server Explorer, your application cannot use the database unless you right-click and 'Close connection' in the Server Explorer.



Working with Entity Framework in layered applications

As we discussed, the LibraryContext is the object that maintains the connection to SQL server, and coordinates our objects with the database. Therefore, it is critical to decide the lifecycle of this object. One approach is to create one whenever you need it, and dispose of it afterwards, so it automatically closes its connection, as in the example:

```
    using (LibraryContext db = new LibraryContext())
    {
        // Here we can use the db-object and all
        // resources are released when the object
        // is disposed of at the closing bracket

        Author author = db.Authors.FirstOrDefault(a => a.Name == "Alexandre
Dumas");

        return author;
```

This is an example of a detached solution and NOT how we should use Entity Framework in the

This is ideal when you need to build applications that will handle multiple users at the same time, such as a web site. But it requires that all the data access to be completed within the using statement. This places a constraint on our program since we use a layered architecture consisting of Repositories, Services, and GUI with strict separation of responsibilities.

A different solution is to not mix objects from different contexts and only create and use one context instance. We create this instance early in our, and use that instance throughout the application. This way, our application has a LibraryContext open and running all the time until we close our application.

One way to do this is to create the LibraryContext object, pass it into a RepositoryFactory, which creates the repository objects and for each repository object it creates, it uses the same context.

Eager Loading

It is an optional feature of EF that you can use if you feel the need to. When querying objects from the database navigation properties (relationships) are not loaded automatically. For example when loading all Authors from the database, each authors books-collection will be null (if not explicitly set in the constructor) because the books are not retrieved from the database. To minimize the amount of database connections both the authors and their books could be loaded with the same database query. The book-collection will then be filled with each author's books. This is called Eager Loading and is supported by Entity Framework, to eagerly load navigation properties in a query use the Include-method (**don't forget the "using System.Data.Entity;" statement**).

```
LibraryContext db = new LibraruContext();

// When querying the database we can specify to eagerly
// load navigation properties. This is done by calling
// the Include-method on the query and specify which
// property to eagerly load, several Include-method can
// be chained to load more than one property
var authorsAndBooks = db.Authors.Include(a => a.Books);
```

In the example above all authors and their associated Books-property is retrieved from the database in a single query.

Workflow with Entity Framework Code First

First define a model (class) then add a DbSet-property to the LibraryContext-class so that Entity Framework will include the model when creating the database structure. Don't feel forced to define all the models at once, you could have an iterative approach to developing your application. Code First is an excellent way of working in an iterative fashion because of the direct mapping between the models and the database.

Database Strategy

Entity Framework Code First comes with some handy database strategies that will control how and when the database will actually be created. There are two already defined strategies: DropCreateDatabaseIfModelChanges and DropCreateDatabaseAlways. The first one will only drop and recreate if you change any of your models, the latter one will always drop the entire database (and all data stored within) and recreate the tables. There is a third option that is very useful during development and that is to inherit from either the DropCreateDatabaseIfModelChanges-class or the DropCreateDatabaseAlways-class and then override the Seed-method. In the Seed-method we have the possibility to initialize objects and store them to the database.

```
class LibraryDbInit : DropCreateDatabaseAlways<LibraryContext>
{
    protected override void Seed(LibraryContext context)
    {
        base.Seed(context);

        Author alexDumas = new Author()
        {
            Name = "Alexandre Dumas"
        };

        Book monteCristo = new Book()
        {
            Title = "The Count of Monte Cristo",
            Author = alexDumas
        };

        context.Books.Add(monteCristo);
        context.SaveChanges();
    }
}
```

We highly recommend creating a derived strategy with a Seed-method, it is very beneficial to have initial data to work with during development and testing of the application. The strategy needs to be registered with the SetInitializer-method before you instantiate any LibraryContext-object in your application. The constructor of your main form-class is a good place to register the strategy or even the constructor of the DbContext-class itself (remember to include “using System.Data.Entity”).

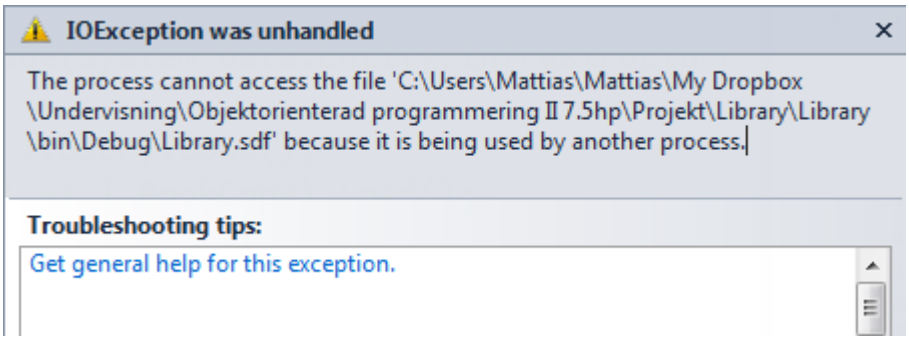
```
// Use a derived strategy with a Seed-method
Database.SetInitializer<LibraryContext>(new LibraryDbInit());

// Recreate the database only if the models change
Database.SetInitializer<LibraryContext>(
    new DropCreateDatabaseIfModelChanges<LibraryContext>());

// Always drop and recreate the database
Database.SetInitializer<LibraryContext>(
    new DropCreateDatabaseAlways<LibraryContext>());
```

Problems and fixes

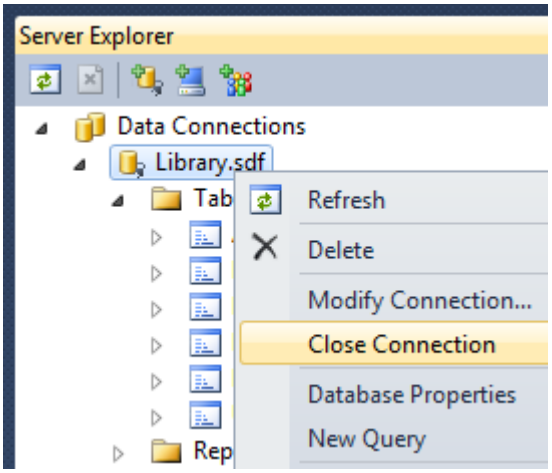
This section will include some problems and fixes associated with Entity Framework Code First.

1. The Database is used by another process

If you get an IOException stating that another process is using your database:

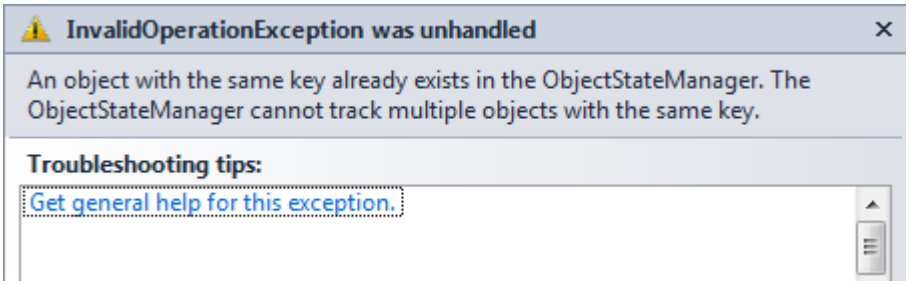


Then you need to close the connection from the Server Explorer to the database by right clicking the database -> Close Connection.



2.An object with the same key already exists in the ObjectStateManager

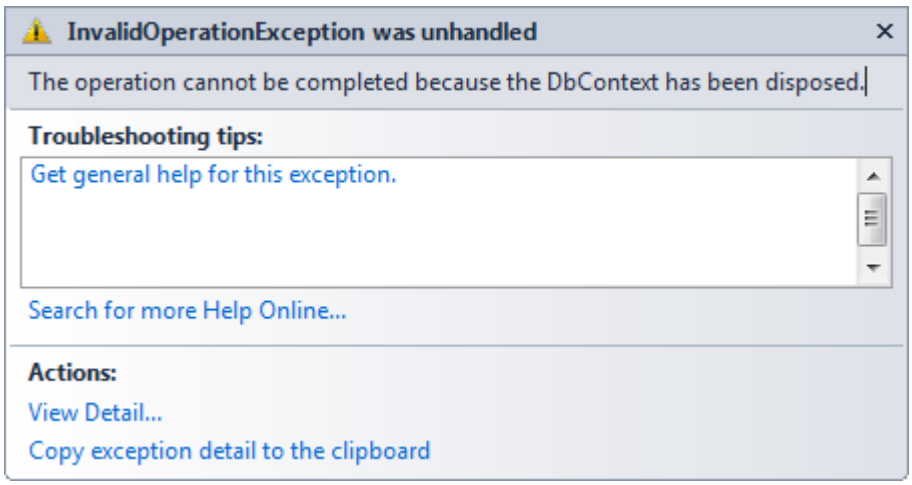
If you get the following exception:



This means you are trying to insert an object that is already in the database, and EF noticed this. Then read the section ”Working with Entity Framework in layered applications” earlier in this document and compare the solutions there with your code.

3. The operation cannot be completed because the DbContext has been disposed.

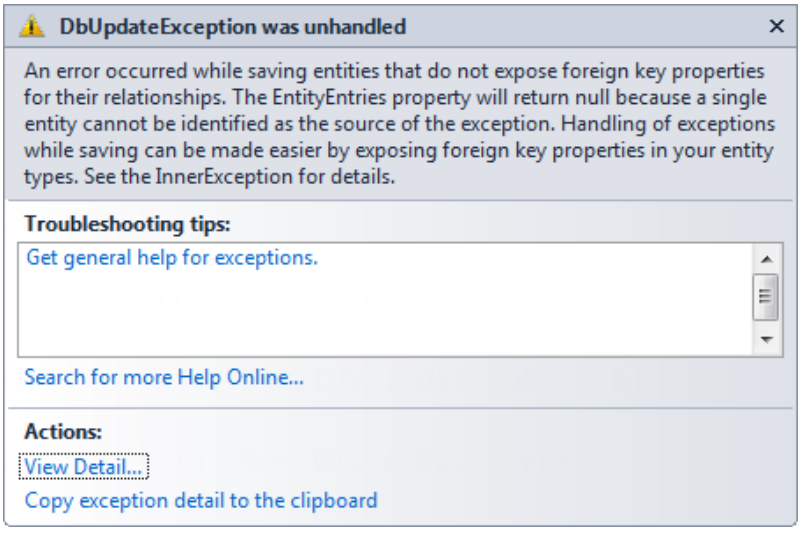
If you get the following exception:



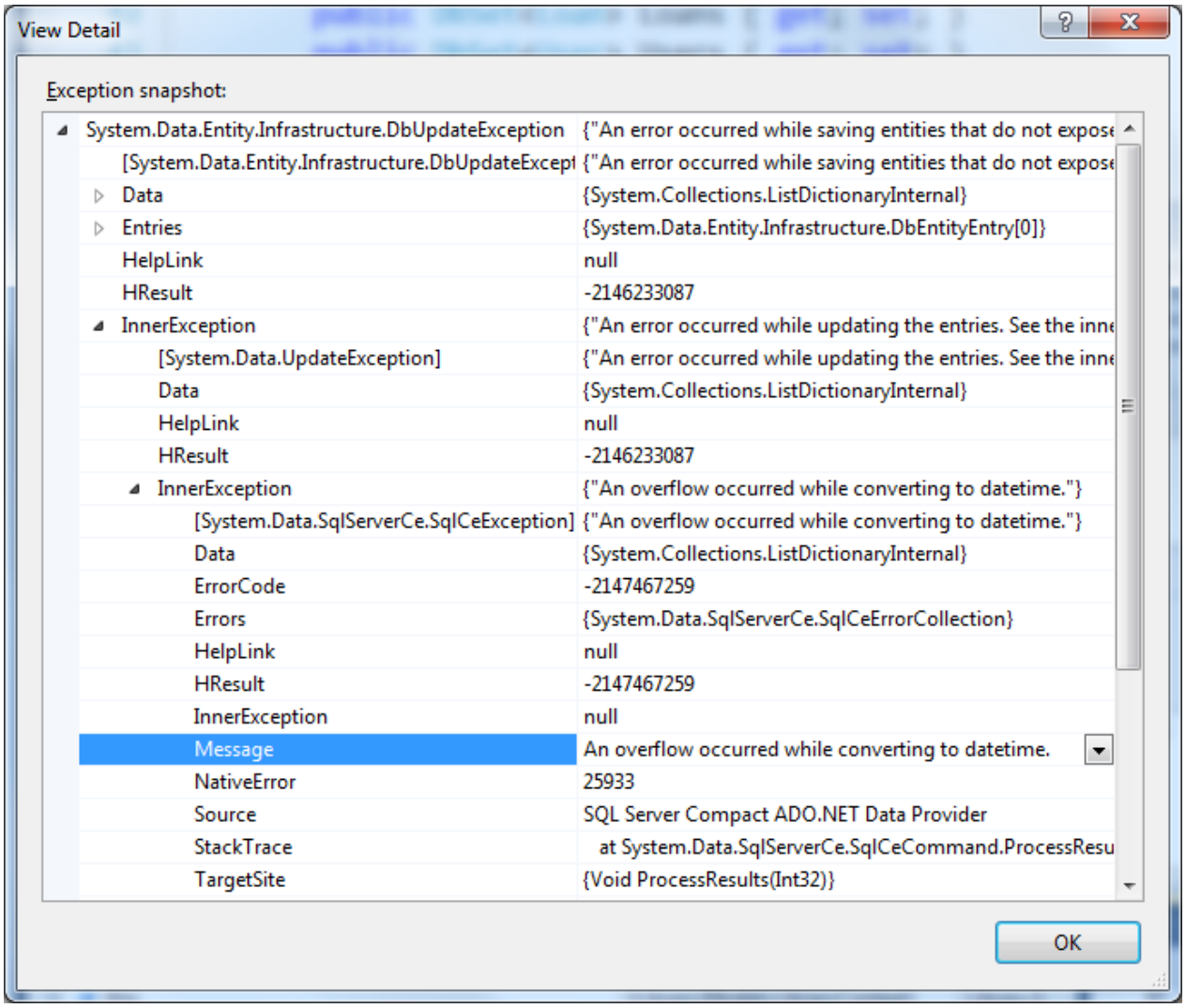
Your context has already been disposed when the query was executed, you need to check where you create your LibraryContext object, and make sure that when you eventually fetch the data, it is still alive.

4. DbUpdateException

The DbUpdateException-exception is thrown when the SaveChanges-method is called and the database refuses the changes because the operations to be performed to the database are breaking some rule or relationship in the database.



To get more information click “View Detail” and then browse as far down the inner exceptions as possible:



This will get you a better idea why the exception is thrown. In this case there is a specific error with the message “An overflow occurred while converting to datetime.” See next section.

1. **An overflow occurred while converting to datetime.**

This is an error thrown because the minimum value of the data type DateTime can be smaller in C# then the corresponding data type in the database. Probably you want an optional DateTime field when getting this error and you could address this issue with introducing a nullable DateTime for this field. More information about nullable data types is given in the section “Nullable data types” earlier in this document.