Contents

This topic explores in depth **how DevForce determines your connection string and when it connects to your database**. The basics were covered in the topic, "Set the database connection string" topic.

Review the basics first

In the topic, "Set the database connection string", we described the mainstream approaches to connecting your application to a database. Please start there before reading this topic.

Here we revisit the same database connection material only this time in more detail and with more advanced scenarios in mind. Connecting to a database isn't complicated. But as you depart from the familiar path and start introduce unusual circumstances, the number of options grows and so does the apparent complexity.

The Code First database connection string

Entity Framework Code First connects to a database with a standard connection string, not an EDM connection string. Since DevForce provides additional connection logic than the Entity Framework - providing data source keys, data source extensions, and data source key resolvers - DevForce uses its own discovery logic in finding the appropriate connection string.

DevForce usually retrieves it from the <*connectionStrings*> section of a configuration file (*app.config* or *web.config*). For example:

```
<add
   name="CodeFirstDemo"
   connectionString="data source=.;initial catalog=CodeFirstDemoDb;integrated
security=True;multipleactiveresultsets=True;App=EntityFramework"
   providerName="System.Data.SqlClient"
/>
```

DevForce can also acquire connection strings dynamically from other, arbitrary sources by calling your custom DataSourceKeyResolver.

The *DataSourceKeyName*

A *DataSourceKeyName* in DevForce identifies symbolically the data source for entities in a model. With DevForce Code First, this name can be specified in several different ways, based upon how you construct your model.

The following table summarizes, in priority order, how DevForce determines the *DataSourceKeyName* (the "key"):

Condition	Result
DbContext decorated with a *DataSourceKeyName* attribute	The key is the name specified in the attribute
DbContext lacks the attribute	The key is the *DbContext* class name
EntityManager decorated with a *DataSourceKeyName* attribute	The key is the name specified in the attribute
EntityManager lacks the attribute	The key is the *EntityManager* class name

As you can see from above, the *DataSourceKeyName* attribute is used to explicitly set the key name, overriding the default naming convention. We explore each of the possibilities in more detail below.

(1) Custom *DbContext* with a *DataSourceKeyName* attribute

You wrote a custom *DbContext* and decorated it with a *DataSourceKeyName* attribute.

```
[DataSourceKeyName("CodeFirstDemo")]
public class ProductDbContext : DbContext {
  // Constructor with connection parameter
  public ProductDbContext(string connection) : base(connection) {...}
}
```

The *DataSourceKeyName* is "**CodeFirstDemo**". It doesn't matter whether you did or did not write an *EntityManager*.

When using the *DataSourceKeyName* attribute you are required to provide a *DbContext* constructor which accepts a connection string parameter; you will receive a build error if you do not.

(2) *DbContext* but no *DataSourceKeyName* attribute

You wrote a custom *DbContext* but didn't decorate it with a *DataSourceKeyName* attribute.

```
public class ProductDbContext : DbContext {
  // Constructor with connection parameter
  public ProductDbContext(string connection) : base(connection) {...}
}
```

The *DataSourceKeyName* is the name of the *DbContext* class, "**ProductDbContext**". It doesn't matter whether you did or did not write an *EntityManager*.

It's always a good idea to provide a constructor that takes a string connection parameter, as in this example. DevForce will use this constructor if present, and perform its own logic to find and resolve the *DataSourceKeyName* and connection string. If this constructor is not present standard Entity Framework conventions are used.

(3) *EntityManager* with a *DataSourceKeyName* attribute

You wrote a custom *EntityManager* decorated with a *DataSourceKeyName* attribute. You didn't write a custom *DbContext*.

```
[DataSourceKeyName("ProductDb")]
public class ProductEntities : EntityManager { ... }
```

The *DataSourceKeyName* is "**ProductDb**".

(4) *EntityManager* but no *DataSourceKeyName* attribute

You wrote a custom *EntityManager* but didn't decorate it with a *DataSourceKeyName* attribute. You didn't write a custom *DbContext*.

```
public class ProductEntities : EntityManager { ... }
```

The *DataSourceKeyName* is the name of the *EntityManager* class, "**ProductEntities** ".

DbContext should have a connection constructor

When you write a custom *DbContext* you should provide it with a constructor that takes a string parameter. That parameter contains connection information which the Entity Framework uses to find ... or possibly create ... the database for your model.

If you write a constructor with a string parameter ("a connection constructor"), DevForce can provide the proper connection string. If you do not, standard Entity Framework conventions are used. In brief, EF conventions can be summarized as follows:

> If the default DbContext constructor is called from a derived context, then the name of the derived context is used to find a connection string in the app.config or web.config file. If no connection string is found, then the name is passed to the DefaultConnectionFactory registered on the Database class. The connection factory then uses the context name as the database name in a default connection string. (This default connection string points to .\SQLEXPRESS on the local machine unless a different DefaultConnectionFactory is registered.) This is explained further in the "**Remarks**" section of the MSDN documentation for *DbContext*.

If you don't provide a connection constructor:

- The DevForce *DataSourceKeyName* and associated connection string are ignored by the Entity Framework
- EF looks in the configuration file for a connection string with the name of the *DbContext* class
- EF may create a database with the full name of your *DbContext*, e.g., "MyApp.ProductDbContext"

- You will not be able to change the database connection string dynamically,

Automatic database creation

The Entity Framework can (and usually will) create a database that matches your model if it can't find the requested database.

The name of the database EF creates depends on how the *DbContext* is constructed.

1. If the *DbContext* is instantiated with the **constructor that takes a string parameter**, the *DbContext* uses the string argument to look for the database connection string.
    1. If the string argument is just a name, Entity Framework looks in the configuration file for a matching connection string.
        1. If it finds such a string but can't connect to the database with that string, it creates a new database using the database name specified in the connection string.
        2. If it doesn't find a connection string in the configuration file, it creates a database with the **name passed into the constructor**, e.g. "CodeFirstDemo".
    2. If the string argument is a connection string but EF can't connect to the database with that string, it creates a new database using the database name specified in the string.
2. If the *DbContext* is constructed with its **default, parameterless constructor**, Entity Framework looks for a connection string in the configuration file that has the same name as the *DbContext* class.
    1. If it finds such a string but can't connect to the database with that string, it creates a new database using the database name specified in the connection string.
    2. If it doesn't find a connection string in the configuration file, it creates a database with the **full name of the *DbContext* class**, e.g. "MyApp.ProductDbContext".

With these rules in mind, if DevForce can't find a connection string for a *DataSourceKeyName* it will pass this *DataSourceKeyName* into the *DbContext* constructor, allowing EF to follow decision path #1.1.

When EF creates a database it does so on the default database server. The default server is SQL Server Express unless you change it.

Automatic database re-generation

If Entity Framework doesn't find the requested database, its default behavior is to create it such that it matches the entity model.

You can set an alternative database initialization strategy by calling *Database.SetInitializer(...)* in your *DbContext* constructor as in the following example:

```
public ProductDbContext(string connection) : base(connection)
{
  // Do not use in production; for early development only
  Database.SetInitializer(
     new DropCreateDatabaseIfModelChanges<ProductDbContext>());
}
```

Notice the *DropCreateDatabaseIfModelChanges<T>* object passed into the static *Database.SetInitializer* method call.

That’s an initialization strategy object that tells EF to re-create the database if it detects model changes.

Here are the stock initialization strategies that are useful in early development when you don't care about the database schema and data. **All of them are dangerous in production code**:

```
//Default strategy: creates the DB only if it doesn't exist
Database.SetInitializer(new CreateDatabaseOnlyIfNotExists<ProductDbContext>());
//Recreates the DB if the model changes but doesn't insert seed data.
Database.SetInitializer(new RecreateDatabaseIfModelChanges<ProductDbContext>());
//Always recreates the DB every time the app is run.
Database.SetInitializer(new DropCreateDatabaseAlways<ProductDbContext>());
```

You can create your own initialization strategy by inheriting from one of these and overriding the *Seed* method.

Never enable **any** of these database initialization strategies in production code. Never enable them if there is a chance that the Entity Framework could destroy potentially valuable data ... even valuable test data.

You can stop Entity Framework from creating or re-creating the database - **and should do so in production** - by calling the *Database.SetInitializer* static method with a null argument. One possible place to do that is in the constructor as follows:

```
public ProductDbContext(string connection) : base(connection)
{
  Database.SetInitializer(null); // Never create a database
}
```

Convert EDM connection strings

A Code First application uses a standard connection string to connect to a database.

If you are migrating from a "Database First" or "Model First" model to "Code First", you'll have to revise your connection string from an EDM connection string to a standard ADO.NET database connection string.

Here's a typical *EDM* string as defined in a configuration file after reformatting for readability:

```
<add
  name="default"
  connectionString=
     "metadata=res://*/Model1.csdl|res://*/Model1.ssdl|res://*/Model1.msl;
      provider=System.Data.SqlClient;provider connection string=&quot;
      data source=.;initial catalog=CodeFirstDemoDb;integrated security=True;multipleactiveresultsets=True;App=EntityFramework
      &quot;"
  providerName="System.Data.EntityClient"
/>
```

Convert it to a regular connection string configuration in four steps:

1. Change the **name** of the string to match your model's *DataSourceKeyName*
2. Locate the inner database connection information, beginning at the words "**data source**"
3. Discard everything else in the "connectionString" segment
4. Change the "**providerName**" from "System.Data.EntityClient" to "**System.Data.SqlClient**" (for SQL Server)

If you were previously using a .NET provider other than "SqlClient" use that providerName here.

After application of these steps our example looks like this:

```
<add
  name="CodeFirstDemo"
  connectionString="data source=.;initial catalog=CodeFirstDemoDb;integrated
security=True;multipleactiveresultsets=True;App=EntityFramework"
  providerName="System.Data.SqlClient"
/>
```

Build vs. Runtime

DevForce generates metadata about your model when you build your model project. DevForce harvests most of *its own* metadata from Entity Framework. So as part of the process, DevForce constructs a *DbContext* - your *DbContext* if you've written one - to get that metadata,

During the build, DevForce looks for a suitable connection string in the *model project*'s configuration file. It does not use a DataSourceKeyResolver to acquire connection information. DevForce then passes the connection (if found) into the *DbContext* constructor (if it has a constructor that accepts a string).

This connection string management will be transparent to you if you've defined your model within your application or web project because such projects tend to have a configuration file that contains a database connection string anyway.

But many developers prefer to keep the model in its own, separate project. If you have a separate model project, the project **must contain its own *App.config* file** [DevForce 6.1.3; this requirement is lifted in subsequent releases]. That *App.config* could have a connection string for metadata generation purposes (see SQL Express discussion, below).

Don't confuse the model project *App.config* with the application configuration file. At runtime, DevForce refers to the proper configuration file, the one defined in the application or the web project. It ignores the model project's *App.config*.

Multiple Databases

DevForce applications can access more than one database. Your customer information might be in one database while your accounting information is in a separate database. You probably would build corresponding "Customer" and "Accounting" models, each with its own entity types.

An Entity Framework *DbContext* can only reference a single database. The "Customer" and "Accounting" models would have their own *DbContext*s, each with its distinct *DataSourceKeyName*. Entities in the "Customer" model might have the "Customer" *DataSourceKeyName* and entities in the "Account" model might have the "Accounting" *DataSourceKeyName*.

An entity type can only be in one model and have one *DataSourceKeyName*.

Unlike the *DbContext*, a DevForce ***EntityManager*** can retrieve entities from multiple models backed by multiple databases. DevForce can save changes to multiple databases in single, distributed transaction. DevForce uses the entity type's *DataSourceKeyName* to determine where to save the entity data.

Multiple Models

You must define multiple models if your application accesses more than one database. You may choose to create multiple models even if all data are stored in one database. A modular application might have a "CRM" module for managing customer relationships and an "Accounting" module for managing the books. You could have separate models for each module. Although all model data might be stored in one database, you'd have some separation in your application's model design.

You can define separate models either by defining separate *EntityManager*s or separate *DbContext*s or both. If you have both, you probably want to coordinate your definitions of *EntityManager*s and *DbContext*s so they pair up, one-to-one.

When DevForce detects multiple *EntityManagers*, it divides the entity classes into separate models based on the types it detects in each manager's *EntityQuery* properties. When you define multiple *DbContexts*, the classes are allocated to separate models based on the types detected in each *DbContext*'s *DbSet* properties.

Each model has its own *DataSourceKeyName*, determined by the same decision rules discussed above.

Living without SQL Server Express

We know that Entity Framework Code First by convention uses Microsoft SQL Server Express. If you don't have SQL Server Express installed you'll first discover the consequences of this convention when you build your Code First model with DevForce: Visual Studio will appear to hang as your project is building, as EF hunts for your installation. Eventually EF times out and reports an error. This happens because DevForce must generate model metadata at **build** time.

One easy resolution is to accept assimilation and install SQL Express. But, you certainly don't have to. It's easy to change the convention and use any database provider supported by the Entity Framework.

There are two workarounds for those of you who can't or won't install SQL Express:

1. Add a connection string to the configuration file
2. Change the convention by setting the *DefaultConnectionFactory*

Add a connection string

This approach may be best if you are connecting to an existing database. The name of the connection string must match your *DataSourceKeyName* and should be fully specified as in the example above.

If you define your model in its *own project*, add an ***App.config*** to that project. The *App.config* needs only a <*connectionStrings*/> section with the connection string. The database named in this *Model project* connection string does not have to exist. At runtime the application uses the connection string defined in the application or web project configuration. At runtime the model project's *App.config* is ignored; it's sole purpose is to provide the database and provider information for gathering metadata.

Set the *DefaultConnectionFactory*

You can override the default convention to use SQL Server Express by setting EF's static *Database.DefaultConnectionFactory* to use the provider and connection information of your choice.

Locate your *DefaultConnectionFactory* configuration code where it will execute *before DevForce makes a request of the Entity Framework*: the static constructor of your custom *DbContext* is a good place.

```
static MyDbContext()
{
   ConfigureForSqlServer(); // See below
}
```

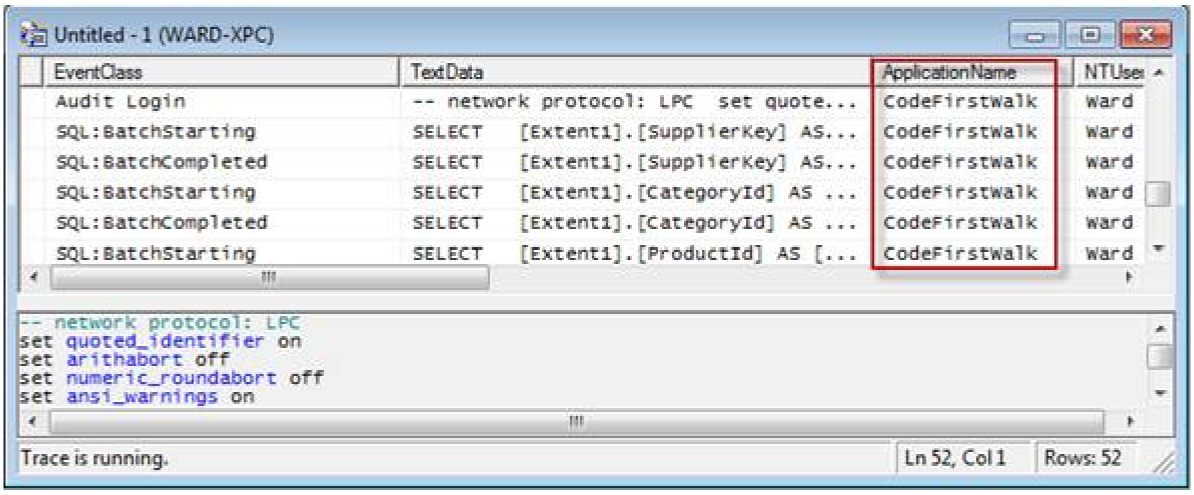
Here's an example of *DefaultConnectionFactory* configuration for SQL Server:

```
static ConfigureCodeFirstForSqlServer()
 {
  // Set base connection string
  const string baseConnectionString =
    "Data Source=.; " +  // my SQL Server name
    "Integrated Security=True; " +
    "MultipleActiveResultSets=True; " +
    "Application Name=CodeFirstWalk"; // change to suit your app
  Database.DefaultConnectionFactory = new SqlConnectionFactory(baseConnectionString);
}
```

The base string is a collection of connection string parts. When EF receives a real connection string, it blends the parts from this base string with the real connection string (the real string's parts take precedence) to produce the final string. If EF can't find a string or can't find the database described in the string … and EF is configured to create a database … EF will create a database on the default database server you prescribed in the "Data Source=" part of the base string.

In this example, "*Data Source=.;*" resolves to SQL Server on the author's machine; if EF creates a database, it will create a SQL Server database. If the base string had no "*Data Source=*" part or specified "*Data Source=./SQLEXPRESS;*", EF would attempt to create the database on SQL Server Express (the default) on the author's machine.

Specifying an *Application Name* in a connection string makes it easier to find the app's database commands in a Profiler trace log; the Application Name appears prominently as seen in this profiler snapshot:



EventClass	TextData	ApplicationName	NTUser
Audit Login	-- network protocol: LPC set quote...	CodeFirstWalk	Ward
SQL:BatchStarting	SELECT [Extent1].[SupplierKey] AS...	CodeFirstWalk	Ward
SQL:BatchCompleted	SELECT [Extent1].[SupplierKey] AS...	CodeFirstWalk	Ward
SQL:BatchStarting	SELECT [Extent1].[CategoryId] AS ...	CodeFirstWalk	Ward
SQL:BatchCompleted	SELECT [Extent1].[CategoryId] AS ...	CodeFirstWalk	Ward
SQL:BatchStarting	SELECT [Extent1].[ProductId] AS [...	CodeFirstWalk	Ward

```
-- network protocol: LPC
set quoted_identifier on
set arithabort off
set numeric_roundabort off
set ansi_warnings on
```