insightsoftware

SimbaEngine

Creating an Entity Framework Provider

Version 10.3

August 2024

Copyright

Contact Us

insightsoftware

www.insightsoftware.com

Introduction

This document describes how to take the existing Simba Client for ADO.Net Provider and incorporate it into an ADO.Net Entity Framework Provider which can expose entities for consumption by entity framework applications such as data services which expose entities as OData.

The creation of the example referred to throughout this document (here on referred to as just "EFProvider") involved creating a new C# assembly project, adding in components from a Microsoft Entity Framework Provider example which targets SQL Server, and then writing a DbProviderFactory class which returns various SimbaDotNet Client objects and a new Entity Framework DbProviderServices class. In this sense the EFProvider example can be thought of as a wrapper around SimbaDotNetClient. In practice, the reader will create a wrapper around SimbaDotNetClient for different data sources with different SQL generation and metadata. In practice, the reader will likely create an Entity Framework Provider by starting with an existing Simba ADO.net Framework Provider.

Note:

It will be up to the reader to determine how those components in EFProvider that interact with SQL Server need to be modified in order to interact with a custom database solution. No specific information on doing this is provided in this document.

For information on how to create and install an ADO.Net Provider from scratch see Simba's *Build an ADO.NET Provider in 5 Days.pdf* document.

For information and instructions on how to download the Microsoft's Entity Framework SQL Server example that was used in the creation of EFProvider see: http://msdn.microsoft.com/en-us/library/ee789835.aspx.

The EFProvider example referenced in this document is available from Simba and has been packaged up with two components for testing:

- **EFTest.exe**: a C# console application which connects to a server using the EFProvider directly. This app runs a simple query against the server using the Entity Framework to retrieve C# entities representing entities in the database (e.g. a collection of customer objects). In this document the D2O Server will be run on the local machine configured with the default port to connect to a SQL Server instance with the Northwind data set.

- **EFOData.dll**: a WCF data services assembly which consumes entities from the EFProvider like EFTest.exe does, but exposes the entities as OData. This assembly can then be used by other applications such as web services which handle OData. This assembly can be tested using a web browser during development to prove out that entity data is being consumed and properly exposed as OData.

Building EFProvider

This section details the steps that went into the creation of the EFProvider assembly and describes some of the key components.

1. Create a new .NET assembly project in Visual Studio.

2. Add the following references to the project:

   - System.Data.Entity (from the .NET Framework)
   - Simba.DotNetClient
   - Simba.DotNetDSI
   - Simba.ADO.Net

3. Add a new class/file called *EFFactory* and create an implementation of *DBProviderFactory* as shown below. This implementation wraps the Simba ADO.net provider and thus acts as the "entry point" to obtain the various components.

   In addition to this, the class must implement *IServiceProvider* which is required in order to return the *EFProviderServices* instance (described below) via the *GetService()* method:

```
public sealed class EFFactory : DbProviderFactory, IServiceProvider
{
public static EFFactory Instance = new EFFactory();
public override DbCommand CreateCommand()
{
return DotNetClientFactory.Instance.CreateCommand();
}
public override DbCommandBuilder CreateCommandBuilder()
{
return DotNetClientFactory.Instance.CreateCommandBuilder();
}
public override DbConnection CreateConnection()
{
return DotNetClientFactory.Instance.CreateConnection();
}
public override DbConnectionStringBuilder CreateConnectionStringBuilder()
```

```
{
return DotNetClientFactory.Instance.CreateConnectionStringBuilder();
}
public override DbDataAdapter CreateDataAdapter()
{
return DotNetClientFactory.Instance.CreateDataAdapter();
}
public override DbParameter CreateParameter()
{
return DotNetClientFactory.Instance.CreateParameter();
}
public object GetService(Type serviceType)
{
if (serviceType == typeof(DbProviderServices))
return EFProviderServices.Instance;
else
return null;
}
}
```

4. Add the following source code from Microsoft's SQL Server EF example to the project.

   Note:

   The prefix "EFProvider" that is used in the class and file names below was created manually; the prefix did not originate from Microsoft's example.

   - **EFProviderServices.cs**: contains an implementation of DbProviderServices for building commands which work with the underlying database. This code was taken from Microsoft's example and more information can be found here: http://msdn.microsoft.com/en-us/library/ee789836.aspx. The reader will need to investigate how to modify this code to work with a custom database.

   - **SQL Generation (folder)**:contains 12 classes with code that performs SQL generation and related functionality. This is used by EFProviderServices.

   - **EFProviderManifest.xml**: works in conjunction with *EFProviderManifest.cs* (described next) to provide information about the database. This file must be included in the project as an embedded resource. The reader will need to make modifications to the metadata in this file in

order to get it to work with a custom database.

- **EFProviderManifest.cs**: contains the *EFProviderManifest* class containing methods which describe how to get information about the database. The namespaces in following methods within this class had to be modified to ensure that the correct embedded resources could be found. Note: this will only be required in the reader's code if the namespaces are modified.

  private XmlReader GetStoreSchemaMapping()

  {

  return GetXmlResource("EFProvider.EFProviderServices.StoreSchemaMapping.msl");

  }

  private XmlReader GetStoreSchemaDescription()

  {

  return GetXmlResource("EFProvider.EFProviderServices.StoreSchemaDefinition.ssdl");

  }

  private static XmlReader GetProviderManifest()

  {

  return GetXmlResource("EFProvider.EFProviderManifest.xml");

- **EFProviderServices.StoreSchemaDefinition.ssdl and EFProviderServices.StoreSchemaDefinition.msl**: describes how to get metadata about tables, columns, relationships, etc. from the data source during model building. These files were taken from Microsoft's example. Both of these files must be included in the project as embedded resources.

  Note:

  The schema namespace and provider in *EFProviderServices.StoreSchemaDefinition.ssdl* must be adjusted to reflect the respective elements of the EF project as follows:

  ```
  <Schema Namespace="EFProvider" Provider="Simba.EFProvider"
  ProviderManifestToken="2005" Alias="Self"
  xmlns="http://schemas.microsoft.com/ado/2006/04/edm/ssdl">
  ```

5. Sign the assembly in Visual Studio by navigating to the project's properties and clicking on the Signing tab. This is required for the next step where the assembly needs a public token for installation into the Global Assembly Cache ("GAC"). This will also result in the addition of a .snk (no password) or .pfk (password based) key file being added to the Visual Studio project.

6. Install the EFProvider.dll assembly into the GAC and add entries for the assembly to .NET's machine.config files as described in the *SimbaClientServer User's Guide* and *SimbaEngine Developer Guide*.

7. Ensure that the entries into the machine.config files specify the *EFProvider.dll* assembly, its public key token, and the namespace/class names used in that assembly.

   The public key token can be acquired by running *sn -T* against the assembly on the command line; the sn.exe tool is part of .NET's set of tools.

Building a Model

With the EFProvider built and installed, it can now be used to generate an entity model. The goal here is to end up with two files:

- an .edmx file containing the entity model of the data which can be visually modified in Visual Studio.
- a .cs file corresponding to the .edmx file which contains an *ObjectContext* implementation for accessing the entities in .NET and classes for each entity type.

Both of these files will eventually be embedded into Visual Studio projects for applications which will consume entity data. For the purposes of this document, the EFTest and EFOData projects will use these files to consume entity data from the Northwind database.

Note:

Before a model can be generated for the EFProvider example, the D2O Server must be running with the configuration as described in the introduction.

These model files can either be created by hand or generated from an existing database and it's the latter approach that is described next.

Generating a model from the database is best done using Visual Studio which normally automates the creation of the .edmx and .cs files. However a bug in the SQL Server ODBC driver prevents this from working properly for SQL Server databases and therefore a process involving .NET's edmgen.exe tool (described next) was used to generate intermediate files, the contents of which were then manually migrated into the test project's .edmx and .cs files. This bug should not be an issue for other database solutions/ODBC drivers that the reader is developing for, and therefore model creation in Visual Studio should be possible.

Microsoft .NET's edmgen.exe application takes in the name of the entity framework provider installed in the GAC as well as the IP address and port of the server hosting the database, along with some other parameters which control the output. It will then generate the following files:

- a .ssdl (storage schema definition language) file which describes the storage model of an Entity Framework application by defining the entity types, associations, entity containers, entity sets, and association sets of a storage model corresponding to a database schema.
- a .csdl (conceptual schema definition language) file describing the entities, relationships, and functions that make up a conceptual model of a data-driven application. The metadata described in this file is used by the Entity Framework to map entities and relationships that are defined in a conceptual model to a data source. The Entity Framework uses conceptual model metadata to translate queries against the conceptual model to data source-specific commands.
- a .msl (mapping specification language) file which maps the conceptual model to the storage model.

- a .cs file containing an ObjectContext implementation for accessing the entities in .NET.
- a .cs file containing the entity views of the database

Note:

edmgen.exe is located in C:\Windows\Microsoft.NET\Framework64\v4.0.30319. For more information on edmgen.exe see: http://msdn.microsoft.com/en-us/library/bb387165.aspx.

The following command line for edmgen.exe performs this generation process:

edmgen /prov:Simba.EFProvider /c:"servers=localhost 12345"

/mode:FullGeneration /outssdl:out.ssdl /outcsdl:out.csdl

/outmsl:out.msl /outobjectlayer:out.cs /outviews:oudviews.cs /namespace:NorthwindModel /entitycontainer:NorthwindContext /pl

Here the */mode* parameter instructs edmgen.exe to use the connection information in the /c parameter, the context name is specified in the */entitycontainer* parameter, and the namespace to apply to the code is specified in the */namespace* parameter.

Tip: executing this command line will generate the files in the current working directory. Therefore it's good practice to first create a folder to hold these output files, and then switch to that directory on the command line before executing this command.

After executing this command, the five files will be generated in the current working directory and can be integrated into Visual Studio projects for applications which consume entity data

(note: the non source files must be set as embedded resources).

Alternatively, a default .edmx file can be added to an existing application project–Visual Studio will automatically generate a corresponding .cs file for the *ObjectContext* implementation–and then the contents of the five files generated above can be manually copied and pasted into the .edmx file and .cs file.

This was the approach taken in the development of the *EFTest* and *EFOData* test applications referenced in this document and involved the following steps.

When these applications were created, an empty ADO.NET entity data model was added in Visual Studio by right clicking on the project, and selecting **Add New Item**. In the *Add New Item dialog*, *ADO.NET entity data model* was selected from the *Visual C# items* list. An empty model was then selected, and upon completing the dialog, Visual studio created a .edmx and corresponding .cs file containing a default *ObjectContext* implementation.

The contents of these two files were then overwritten by manually copying and pasting the content form the five files generated by edmgen.exe as follows:

- **ObjectContext (.cs)** file: all of the content in this file was removed and then replaced with the content from the out.cs file.
- **.edmx file**: this file was opened in a text editor, and the content from the generated .ssdl, .csdl, and .msl files were added to the SSDL content, CSDL content, and C-S mapping content

sections respectively. The following snippet shows where this content needs to be placed in the default .edmx file created by Visual Studio:

```
<edmx:Runtime>
<!-- SSDL content -->
<edmx:StorageModels>
<!--Copy the entire Schema node of the ssdl file here -->
</edmx:StorageModels>
<!-- CSDL content -->
<edmx:ConceptualModels>
<!--Copy the entire Schema node of the csdl file here -->
</edmx:ConceptualModels>
<!-- C-S mapping content -->
<edmx:Mappings>
<!--Copy the entire Mapping node of the msl file here -->
</edmx:Mappings>
</edmx:Runtime>
```

Overview of the EFTest application

EFTest is a C# console application which uses the entity model of the Northwind database. The application runs a query against the context representing the Northwind database to retrieve customers, and then iterates through the customer entity objects returned displaying their names, addresses, and other information to the console window.

The main purpose of this application is to prove that the entity model works by testing it directly. It's recommended that a test application like this be used before creating OData application to isolate and verify that the entity model works.

To see a detailed view of the modified .edmx file, open *NorthwindModel.edmx* in the EFTest project.

Overview of the EFOData application

EFOData is a WPS Data Services test project (.NET DLL assembly project) which consumes entity data exposed by the model, and exposes it as OData. This assembly can be used by web applications which consume OData, and can also be tested in a webrowser.