Getting Started with Telerik Data Access

CONTENTS

OVERVIEW

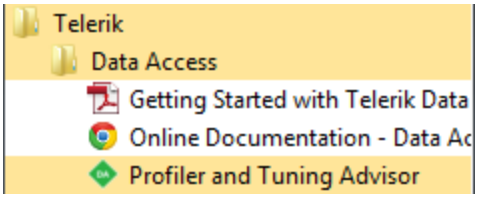
Telerik Data Access is a powerful framework for mapping the objects in an object-oriented model to your relational database tables, views, and/or stored procedures, and vice versa. Mapping is done within Visual Studio and is independent from source code and database. Operating as a datastore-agnostic virtual layer, Telerik Data Access can be used from within the programming language to access and manipulate data.

The Telerik enterprise-grade tool works equally well across all .NET development platforms: ASP.NET AJAX, ASP.NET MVC, WPF, Windows Forms, and Azure. Furthermore, Telerik Data Access provides built-in support for a wide range of databases (MS SQL Azure, MS SQL Server, Oracle, PostgreSQL and etc.) to allow you to connect to one or more databases without the need to install additional components. It also makes it easy to switch from using one database engine to another. Retrieving data is abstracted using the most widely used querying language – LINQ.

Through its feature-rich API, Telerik Data Access allows you to code models that can either **access directly existing databases**, or serve as both **conceptual models from which the structure of the databases is derived** and **data access layers**. The mapping between the model and the database is performed with the Fluent Mapping API that you can start using right away, through the Data Access NuGet packages.

The Telerik.DataAccess.Fluent and Telerik.DataAccess.Core NuGet packages deliver directly in your project everything you need for the development and the deployment of a Data Access model. Telerik.DataAccess.Fluent is designed to be used in the data access layer of your application - where your Telerik Data Access model will reside. It is dependent on the Telerik.DataAccess.Core package, which has to be used within projects which are consuming an already existing Data Access model and do not define persistent classes themselves.

Furthermore, Telerik Data Access offers means for monitoring and profiling the performance of the models – Profiler and Tuning Advisor. This tool is distributed through the Telerik installer (available for free download in your account area on Telerik.com or from the Telerik Data Access product page). Once you run the installer it will walk you through the installation steps and will create a **Data Access** folder under the **Telerik** folder in the **Windows Start** menu. There it will add a shortcut to Profiler and Tuning Advisor and to this document.



To demonstrate Telerik Data Access to you, this tutorial will walk you through the basics of model creation and consumption. It explores the following topics:

- What components does a Data Access model have and how they work together?
- How to configure the projects inside a solution?
- How to derive the database schema from the model and how to deploy it on the server?
- How to execute CRUD operations through the model?
- How to profile the applications that consume a Data Access model?

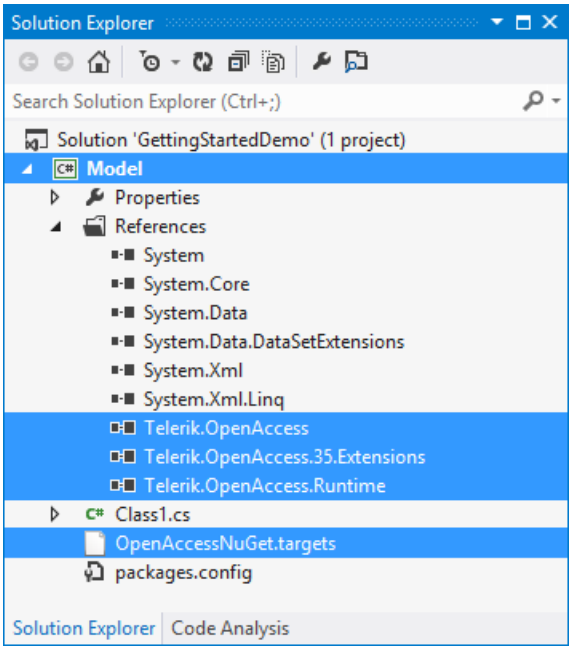
CREATING A DATA ACCESS MODEL

The Data Access model consists of three logical parts: **persistent classes**, a **metadatasource** class, and a **context** class. The main responsibility of the persistent classes is to serve as a shape for the database objects you retrieve, insert or modify. The **context** is the channel used for retrieval of database objects and updating the database with the changes you make. And the **metadatasource** class holds the mapping information about the persistent classes.

Telerik Data Access makes it very simple to start coding your model. All you need to do is create a class library project in your solution and integrate it with the Telerik.DataAccess.Fluent package.

Let's look at a demo application in which the solution *(GettingStartedDemo)* has a class library *(Model)* that will hold a new Data Access model.

Before adding any code, the *Model* project should be integrated with the Telerik.DataAccess.Fluent package. The package will ensure that the project is configured properly for the development process.



DEFINING THE PERSISTENT CLASSES

In your code the persistent classes should look like POCO classes with no knowledge about Telerik Data Access at all.

At minimum, they have to expose public properties, which correspond to the columns in the database tables. Their types should conceptually match the types of the table columns. Later on, when you are creating the metadatasource class of your model, the names of the properties and their types will provide the foundation of the mapping logic.

Besides the basic property-to-column mapping, when you code the persistent classes, you need to take into account the participation of the database tables in relationships of some kind (one-to-one, one-to-many, or many-to-many). You may need to consume these associations in your applications, and exposing them as navigation properties of the persistent classes is the way to do so. To model a general one-to-many association between two classes, you need to add a property of type *IList<TChild>* to the parent class, and a property of type the parent class to the child. The specific details about the underlying constraint in the database will be coded later in your metadatasource class.

NOTE: The navigation properties are not part of the database tables' structure. Their sole purpose is to ease the consumption of related objects.

Additionally, whenever your scenario requires it, you could enhance the classes with private fields and apply dedicated logic in the getters and the setters of the properties.

In the demo model we will add two classes (*Category* and *Product*) with a one-to-many association between them.

1. In the *Model* project, rename the *Class1* file to *Category*, and in it paste the sample definition of the *Category* class provided below. It has an **integer** property (*CategoryID*) for the value of a table's primary key, a **string** property (*CategoryName*) for the name of a category object, a **datetime** property (*DateCreated*) for the date when the given category was first saved to the database, a **byte array** property (*Picture*) for an image that suits best to the given category, and a read-only **IList<Product>** property (*Products*) for the products that belong to a given category.

```
public class Category
{
    public int CategoryID { get; set; }
    public string CategoryName { get; set; }
    public DateTime DateCreated { get; set; }
    public byte[] Picture { get; set; }

    private IList<Product> _products = new List<Product>();
    public IList<Product> Products
    {
        get
        {
            return this._products;
        }
    }
}
```

2. Analogically, let's create the *Product* class. It will have properties that best describe the product entity. The special thing about *Product* is that because it is the child class in the one-to-many association, from relational perspective it has to expose a property for the column that will hold the foreign key value (*CategoryID*). In addition to that, it will also have the *Category* navigation property.
   Add a new class file to the class library. Name it *Product* and in it paste the sample definition of the *Product* class.

```
public partial class Product
{
    private int _productID;
    public int ProductID
    {
        get { return this._productID; }
        set { this._productID = value; }
    }

    private string _productName;
    public string ProductName
    {
        get { return this._productName; }
        set { this._productName = value; }
    }

    private int? _categoryID;
    public int? CategoryID
    {
```

```
        get { return this._categoryID; }
        set { this._categoryID = value; }
    }

    private decimal? _unitPrice;
    public decimal? UnitPrice
    {
        get { return this._unitPrice; }
        set { this._unitPrice = value; }
    }

    private short? _unitsInStock;
    public short? UnitsInStock
    {
        get { return this._unitsInStock; }
        set { this._unitsInStock = value; }
    }

    private Category _category;
    public Category Category
    {
        get { return this._category; }
        set { this._category = value; }
    }
}
```

3. The persistent classes are ready. You can save the two class files and move on to define their mapping.

DEFINING THE MAPPING CONFIGURATION

At this point, you are going to implement the **metadatasource** class of the model. It is coded through the Fluent Mapping API and supplies the context class with details like:

- Which persistent class corresponds to a given table?
- Which class property corresponds to a given table column?
- Which class property corresponds to the primary key column of the table?
- Which SQL type corresponds to the type of the class property?

The model-specific **metadatasource** class should derive from the abstract **FluentMetadataSource** class of Telerik Data Access and can be coded in either one of the two approaches supported by Telerik Data Access - Default Mapping and Explicit Mapping. With Default Mapping, the abstract **FluentMetadataSource** class of Telerik Data Access is responsible for every detail of the property to column mapping, while Explicit Mapping means that the client code provides the configuration through the API. These approaches can be used side by side, meaning that in the same model you can have default mapped classes and explicitly mapped classes. Furthermore, within the mapping configuration of a single class, you can mix the two approaches as well.

The specific **metadatasource** class has to provide implementation for the **PrepareMapping CreateModel** and **SetContainerSettings** methods. **PrepareMapping** serves as an entry point for working with the Fluent Mapping API. Later on, this method will be called when you create instances of the context and the mapping needs to be obtained. **CreateModel** will provide the configuration necessary when the database schema is generated based on the model. **SetContainerSettings** will provide the naming settings consumed by the metadatasource. They are necessary when the conceptual representation of the classes is instantiated in-memory during runtime.

In the metadatasource class of the demo model, the *Category* class will be mapped via Default Mapping and the configuration of the *Product* class will be performed via Explicit Mapping. The association between the classes will be modeled explicitly, which means that the *Category* class will mix Default Mapping and Explicit Mapping as a result.

1. In the *Model* project, add a new class file called *DemoModelMetadataSource*. Make the *DemoModelMetadataSource* class **public** and inherit the **FluentMetadataSource** class**.**

2. Let's provide the implementation of the **PrepareMapping** method.
   In general, you can code the mapping configuration of the classes directly in the method body, but for the purpose of the demo example, we are going to separate the mapping of *Category* and the mapping of *Product* in their own methods. *PrepareCategoryConfiguration* will return the mapping of *Category* to the **PrepareMapping** method, and *PrepareProductConfiguration* will return the mapping of *Product*. Inside **PrepareMapping** the two configurations will be added to a list, which will be passed to the context during its instantiation.

   a. The implementation of **PrepareMapping**

```
protected override IList<MappingConfiguration> PrepareMapping()
{
    List<MappingConfiguration> modelConfiguration =
    new List<MappingConfiguration>();

    MappingConfiguration<Category> categoryConfiguration =
    PrepareCategoryConfiguration();
    modelConfiguration.Add(categoryConfiguration);

    MappingConfiguration<Product> productConfiguration =
    PrepareProductConfiguration();
    modelConfiguration.Add(productConfiguration);

    return modelConfiguration;

}
```

   b. The implementation of *PrepareCategoryConfiguration*. In order to create the mapping for *Category*, we are adopting the practice described in the Mixing Default and Explicit Mapping article.

```
private MappingConfiguration<Category> PrepareCategoryConfiguration()
{
    MappingConfiguration<Category> configuration =
    new MappingConfiguration<Category>();

    // Default map the Category class to the Categories table
    configuration.MapType(category => new
    {
        CategoryID = category.CategoryID,
        CategoryName = category.CategoryName,
        DateCreated = category.DateCreated,
        Picture = category.Picture
    }).ToTable("Categories");

    // Specify that CategoryID is mapped to the primary key of the table
    // and that the primary key is autoincremented integer calculated by
    // the server engine.
    configuration.HasProperty(category => category.CategoryID).
        IsIdentity(KeyGenerator.Autoinc);
```

```
        // Mapping of the Products navigation property
        configuration.HasAssociation(x => x.Products).
            HasFieldName("_products").
            WithOpposite(x => x.Category).ToColumn("CategoryID").
            HasConstraint((y, x) => x.CategoryID == y.CategoryID).
            WithDataAccessKind(DataAccessKind.ReadWrite);

        return configuration;
    }
```

c. The implementation of *PrepareProductConfiguration.* The extension methods used here are described in details in the Mapping CLR Types, Properties, and Associations section of the Telerik Data Access documentation.

```
    private MappingConfiguration<Product> PrepareProductConfiguration()
    {
        MappingConfiguration<Product> configuration =
            new MappingConfiguration<Product>();

        // Mapping the Product class to the Products table
        configuration.MapType(x => new { }).
            WithConcurencyControl(OptimisticConcurrencyControlStrategy.Changed).
            ToTable("Products");

        // Mapping the Product properties to the Products columns
        configuration.HasProperty(x => x.ProductID).
            IsIdentity(KeyGenerator.Autoinc).
            HasFieldName("_productID").
            WithDataAccessKind(DataAccessKind.ReadWrite).
            ToColumn("ProductID").
            IsNotNullable().HasColumnType("int").
            HasPrecision(0).HasScale(0);
        configuration.HasProperty(x => x.ProductName).
            HasFieldName("_productName").
            WithDataAccessKind(DataAccessKind.ReadWrite).
            ToColumn("ProductName").IsNotNullable().
            HasColumnType("nvarchar").HasLength(40);
        configuration.HasProperty(x => x.CategoryID).
            HasFieldName("_categoryID").
            WithDataAccessKind(DataAccessKind.ReadWrite).
            ToColumn("CategoryID").IsNullable().
            HasColumnType("int").HasPrecision(0).HasScale(0);
        configuration.HasProperty(x => x.UnitPrice).
            HasFieldName("_unitPrice").
            WithDataAccessKind(DataAccessKind.ReadWrite).
            ToColumn("UnitPrice").IsNullable().
            HasColumnType("money").HasPrecision(0).
            HasScale(0).HasDefaultValue();
        configuration.HasProperty(x => x.UnitsInStock).
            HasFieldName("_unitsInStock").
            WithDataAccessKind(DataAccessKind.ReadWrite).
            ToColumn("UnitsInStock").IsNullable().
            HasColumnType("smallint").HasPrecision(0).
            HasScale(0).HasDefaultValue();

        // Mapping of the Category navigation property
        configuration.HasAssociation(x => x.Category).
```

```
            HasFieldName("_category").WithOpposite(x => x.Products).
            ToColumn("CategoryID").
            HasConstraint((x, y) => x.CategoryID == y.CategoryID).
            WithDataAccessKind(DataAccessKind.ReadWrite);

        return configuration;

    }
```

3. Now let's provide the naming settings consumed by the metadatasource.
   The name of our model will be *DemoModel*. It resides in the default namespace. We would like the generated column names to be based on the property names and to be escaped, if necessary. We used CamelCase for the properties names and the same convention should be kept in the database as well (no word breaks). It will also be good to shorten the column names if they exceed the maximum allowed length for the particular backend and to replace any backend invalid characters with 'string.Empty'.

```
protected override void SetContainerSettings(MetadataContainer container)
{
    container.Name = "DemoModel";
    container.DefaultNamespace = "Model";
    container.NameGenerator.SourceStrategy = NamingSourceStrategy.Property;
    container.NameGenerator.RemoveCamelCase = false;

}
```

4. The last task is to allow the metadatasource class to provide foreign key constrains for the associations in the model. Let's override the **CreateModel** method.

```
protected override MetadataContainer CreateModel()
{
    MetadataContainer container = base.CreateModel();
    container.DefaultMapping.NullForeignKey = true;

    return container;

}
```

5. Once the class file is saved the mapping configuration is done. Let's move on to defining the context.

DEFINING THE CONTEXT

The context is the entry point to all of the Data Access runtime features (LINQ, Bulk Operations, Fetch Strategies, Low Level ADO API, and the rest). This is the place where the **persistent classes** and the **metadatasource** class along with the **connectivity** to the database and the details about the **backend configuration** come together.

The context class of a model should inherit from the **OpenAccessContext** class of Telerik Data Access. It is recommended for the implementation to expose a public read-only property of type IQueryable<T> (a.k.a. an end point) for each of the persistent classes. The proper context functionality during runtime requires it to be aware of the following components:

- The connection string to the database.
- The metadatasource that supplies the mapping configuration.
- The backend configuration which will be used against the server engine.

NOTE: The backend in use for the demo is Microsoft SQL Server 2012 Express.

1. **The connection string to the consumed database**.
   The generally recommended approach regarding connectivity is to specify the connection string in the configuration file of the application that consumes it. This practice suits well to Telerik Data Access, because all the context needs is the name of the connection string entry in the configuration file.
   Let's add an **App.config** file to the *Model* project and specify in it the connection string. For the purpose of the demo, the connection string is named *DemoModelConnection* and the database name is *DemoDatabase*.

```
<connectionStrings>
  <add name="DemoModelConnection"
       connectionString="data source=.\sqlexpress;
                         initial catalog=DemoDatabase;
                         integrated security=True"
       providerName="System.Data.SqlClient"/>
</connectionStrings>
```

2. Now let's move on to the context implementation.
   Add a new class file to the project and name it *DemoModelContext*. Make sure that the *DemoModelContext* class is public, and that it inherits **OpenAccessContext**.

```
public class DemoModelContext : OpenAccessContext
{
}
```

3. To ensure that all instances of our custom context will always have a connection string to work with, the *DemoModelContext* class will contain a private static field that holds the connection sting name.

```
public class DemoModelContext : OpenAccessContext
{
    private static string connectionStringName = @"DemoModelConnection";
}
```

4. **The second requirement of the context is a metadatasource**.
   The availability of metadata for all context instances is ensured by a static filed assigned with an instance of the *DemoModelMetadataSource* class.

```
public class DemoModelContext : OpenAccessContext
{
    private static string connectionStringName = @"DemoModelConnection";

    private static MetadataSource metadataSource =
        new DemoModelMetadataSource();
}
```

5. **The last requirement is a backend configuration**.
   Telerik Data Access has the specially designed **BackendConfiguration** class that provides you with an entry point for configuring backend options like which server engine will be consumed, which provider will be used, how long the command timeout will be, what type of connection pool will be used, and will the Level 2 cache be enabled.
   All the demo model needs at this point is the context to be aware that it will work against an **MsSql** instance, and that it will use the **System.Data.SqlClient** provider. This will happen through another private static field, which will be assigned with a backend configuration object.

```
public class DemoModelContext : OpenAccessContext
{
    private static string connectionStringName = @"DemoModelConnection";

    private static MetadataSource metadataSource = new DemoModelMetadataSource();

    private static BackendConfiguration backend = GetBackendConfiguration();

    private static BackendConfiguration GetBackendConfiguration()
    {
        BackendConfiguration backend = new BackendConfiguration();
        backend.Backend = "MsSql";
        backend.ProviderName = "System.Data.SqlClient";

        return backend;
    }

}
```

NOTE: The complete set of backend configuration options is described in the Backend Configuration section of the Telerik Data Access documentation.

6. At this point, all the information needed by the context is available. Let's consume it.
The default constructor for the *DemoModelContext* class will call the base constructor of **OpenAccessContext** and will pass to it the name of the connection string, the backend configuration, and the metadatasource.

```
public class DemoModelContext : OpenAccessContext
{
    private static string connectionStringName = @"DemoModelConnection";

    private static MetadataSource metadataSource = new DemoModelMetadataSource();

    private static BackendConfiguration backend = GetBackendConfiguration();

    private static BackendConfiguration GetBackendConfiguration()
    {
        BackendConfiguration backend = new BackendConfiguration();
        backend.Backend = "MsSql";
        backend.ProviderName = "System.Data.SqlClient";

        return backend;
    }

    public DemoModelContext()
        : base(connectionStringName, backend, metadataSource)
    { }

}
```

7. The last steps in the implementation of *DemoModelContext*, is to expose the persistent classes through **IQueryable<T>** endpoints.

```
public class DemoModelContext : OpenAccessContext
{
    private static string connectionStringName = @" DemoModelConnection";

    private static MetadataSource metadataSource = new DemoModelMetadataSource();
```

```
        private static BackendConfiguration backend = GetBackendConfiguration();

        private static BackendConfiguration GetBackendConfiguration()
        {
            BackendConfiguration backend = new BackendConfiguration();
            backend.Backend = "MsSql";
            backend.ProviderName = "System.Data.SqlClient";

            return backend;
        }

        public DemoModelContext()
            : base(connectionStringName, backend, metadataSource)
        { }

        public IQueryable<Category> Categories
        {
            get
            {
                return this.GetAll<Category>();
            }
        }

        public IQueryable<Product> Products
        {
            get
            {
                return this.GetAll<Product>();
            }
        }
    }
```

8. Save the context file and build the project.

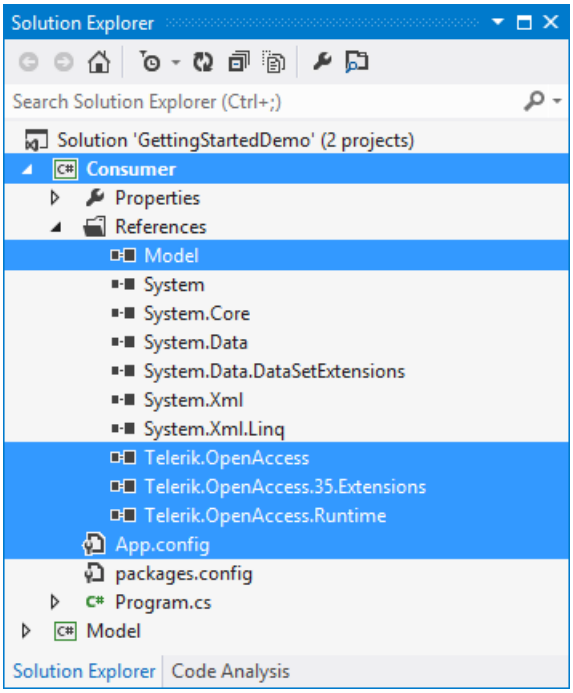
The model is ready for providing data to an application.

WORKING WITH THE DATA ACCESS MODEL

Telerik Data Access data models can be used with just about any .NET platform. The model can be used directly in Web and Windows applications, and for rich clients as well.

For this guide we will create and configure a basic console application to interact with the *DemoModelContext* type; however, the same concepts apply regardless of the application platform (WinForms, WPF, MVC, ASP.NET, etc.) being used.

1. Add a new console application project, called *Consumer*, to the *GettingStartedDemo* solution.
2. Copy, and paste the **App.config** file from the *Model* project to the console application.
3. In the *Consumer* project, add a reference to the Model project.
4. Integrate the Consumer project with the Telerik.DataAccess.Core NuGet package, in order to bring the necessary Data Access assemblies.
5. Set the *Consumer* project as a startup project for the solution.
6. As soon as you build *Consumer*, it is ready to work with the model.



MIGRATING THE MODEL TO THE DATABASE

Telerik Data Access exposes a convenient API that, based on the mapping configuration provided in the metadatasource class, generates DDL script for the database schema. It also has an API that allows you to execute the script automatically every time your model changes and the modifications have to affect the database.

Let's see it in action with the demo model.

1. Open the *DemoModelClass* file that holds the context class.

2. Inside the body of the class, paste the following code of the *CreateUpdateSchema* method.

```
public void CreateUpdateSchema()
{
    var handler = this.GetSchemaHandler();
    string script = null;
    try
    {
        script = handler.CreateUpdateDDLScript(null);
    }
    catch
    {
        bool throwException = false;
        try
        {
            handler.CreateDatabase();
            script = handler.CreateDDLScript();
        }
        catch
        {
            throwException = true;
        }
        if (throwException)
            throw;
    }
    if (string.IsNullOrEmpty(script) == false)
    {
        handler.ExecuteDDLScript(script);
    }
}
```

NOTE: If you place a breakpoint **after** this code line `script = handler.CreateDDLScript();` you will be able to review the generated scrip before it is executed.

3. Save the context file and build the *Model* project.

4. Open the *Program* file in the *Consumer* project. Inside the *Main* method, we are going to execute the *CreateUpdateSchema* method. Paste the next few code lines and run the console application.

```
using (DemoModelContext dbContext = new DemoModelContext())
{
    dbContext.CreateUpdateSchema();
}
```

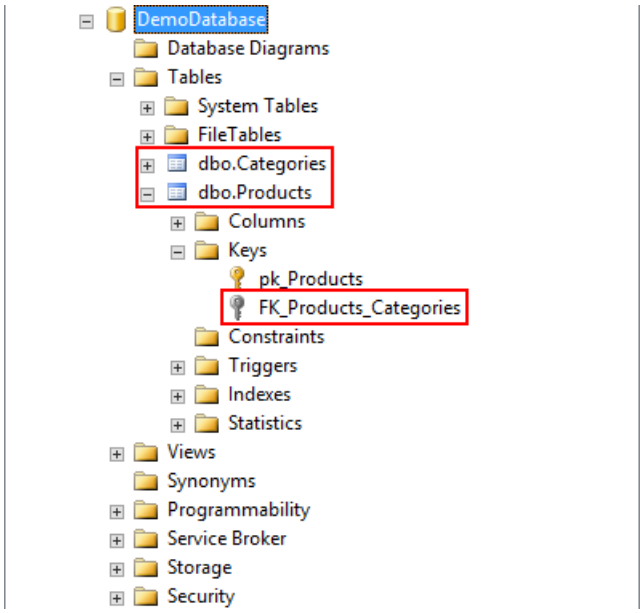
5. If you did place the breakpoint mentioned in step 2, the application will stop its execution on it and you can check the DDL script in the `script` variable. It should be the as the following one.

```
-- Model.Category
CREATE TABLE [Categories] (
    [CategoryID] int IDENTITY NOT NULL,      -- <CategoryID>
    [CategoryName] varchar(255) NULL,        -- <CategoryName>
    [DateCreated] datetime NOT NULL,         -- <DateCreated>
    [Picture] image NULL,                    -- <Picture>
    CONSTRAINT [pk_Categories] PRIMARY KEY ([CategoryID])
)
```

```
go

-- Model.Product
CREATE TABLE [Products] (
    [ProductID] int IDENTITY NOT NULL,      -- _productID
    [ProductName] nvarchar(40) NOT NULL,    -- _productName
    [UnitPrice] money NULL,                 -- _unitPrice
    [UnitsInStock] smallint NULL,           -- _unitsInStock
    [CategoryID] int NULL,                  -- _category
    CONSTRAINT [pk_Products] PRIMARY KEY ([ProductID])
)
go

CREATE INDEX [idx_Products_CategoryID] ON [Products]([CategoryID])
go

ALTER TABLE [Products] ADD CONSTRAINT [FK_Products_Categories]
FOREIGN KEY ([CategoryID]) REFERENCES [Categories]([CategoryID])
go
```

6. Allow the application to finish, so that the script is executed.

7. Currently, the SQL Server instance specified in the connection string should contain a database called *DemoDatabase* with two tables (*Categories* and *Products*) and a foreign key constraint between them.



CRUD OPERATIONS AGAINST THE MODEL

A renowned runtime feature of Telerik Data Access that comes out-of-the-box is Automatic Change Tracking. This means that without further development effort on your side, Telerik Data Access ensures that the data manipulations you perform against a model will be persisted to the database. Combined with the automatic transaction handling, Telerik Data Access brings to you neat and simplified workflows for executing CRUD operations:

1. To insert a new record in the database, all you need to do is:

a. Create a new instance of a persistent class.

b. Initialize its properties.

c. Pass it to the OpenAccessContext.Add method and invoke the SaveChanges method.

2. Reading is as easy and flexible as the Data Access LINQ support.

3. For an update the steps are:

a. Obtain an existing object from the database (through a reading operation with LINQ, for example).

b. Modify its properties according to your needs, and

c. Invoke the SaveChanges method.

4. And in order to delete an object, you need to:

a. Get a reference to the target object.

b. Pass the object to the OpenAccessContext's Delete method.

c. Invoke the SaveChanges method.

Let's look at an example that exercises the demo model. The next code snippet executes several tasks:

1. Ensures the existence of the database,

2. Inserts a new Category object to the database,

3. Reads the first Category row in the database,

4. Updates the retrieved category and preserves the changes, and

5. Retrieves a specific category from the database and deletes it.

```
class Program
{
    static void Main(string[] args)
    {
        using (DemoModelContext dbContext = new DemoModelContext())
        {
            // 1. Check if the database is available on the server
            // Create/update it if necessary.
            if (!dbContext.GetSchemaHandler().DatabaseExists())
            {
                dbContext.CreateUpdateSchema();
            }

            // 2. Add a new category.
            // Category is configured to use the AutoInc key generator. The backend
            // will take care for the value of CategoryID.
            Category newCategory = new Category
            {
```

```
                CategoryName = "Office Materials",
                DateCreated = DateTime.Now,
                Picture = GetCategoryImage(),
            };
            dbContext.Add(newCategory);

            // Commit changes to the database.
            dbContext.SaveChanges();

            // 3. Get the first category using LINQ and modify it.
            var category = dbContext.Categories.FirstOrDefault();

            // 4. Update the retrieved category.
            category.CategoryName = "Office Supplies";

            // Commit changes to the database.
            dbContext.SaveChanges();

            // 5. Use LINQ to retrieve a category named "Office Supplies".
            var categoryToDelete = dbContext.Categories.
                Where(c => c.CategoryName.StartsWith("Office Supplies")).
                FirstOrDefault();

            // Mark the category for removal from the database.
            dbContext.Delete(categoryToDelete);

            // Commit the changes to the database.
            dbContext.SaveChanges();
        }
        Console.Write("All changes executed properly, " +
            "press any key to close.");
        Console.ReadKey();
    }

    /// <summary>
    /// Gets the byte[] representation of an image named office- materials.jpg.
    /// It should be available in the bin\Debug folder of the Consumer project.
    /// </summary>
    private static byte[] GetCategoryImage()
    {
        Image image = Image.FromFile("office-materials.jpg");
        byte[] categoryImage;
        using (MemoryStream ms = new MemoryStream())
        {
            image.Save(ms, ImageFormat.Jpeg);
            categoryImage = ms.ToArray();
        }
        return categoryImage;
    }
}
```

NOTE: In your real-life scenarios you will need to work with related objects. This requires specific configuration of the navigation property mapping. The relevant details are available in the Manage Navigation Properties section of the Data Access documentation.

PROFILING THE MODEL PERFORMANCE

Profiler and Tuning Advisor is a graphical user interface for monitoring of all the Telerik Data Access activity in your application. It makes it easy for you to see how Telerik Data Access is working behind the scenes. You can see all SQL queries sent to the database server, and the LINQ statements that generated them. In addition, the profiler has a built in alert system that will notify about potential issues, and suggest resolutions.

The Profiler is delivered on your machine through the Telerik installer (available for free download in your account area on Telerik.com or from the Telerik Data Access product page). It allows you to monitor your applications in two modes: real-time and offline.

REAL-TIME MONITORING

1. First, you need to install the Telerik.OpenAccess.Profiler NuGet package in the *Consumer* project.

2. Next is the configuration of the context class. In the *DemoModelContext* class, you need to find the *GetBackendConfiguration* method, and add a few settings in it:

    a. In order to see any events and metrics, you need to adjust the **Log Level** setting to **Normal**.

    b. We are going to allow the logging facilities to include the stack trace information. This happens when the **StackTrace** property is set it to **True**.

    c. Telerik Data Access produces two different kinds of data - **metrics** and **events**. **Metrics** are similar to the operating system counters. They produce snapshots of the system status like counting insert, update and delete statements per second. **Log events** contain the real operation information, including query parameters and stack traces. **Tracking the log events can slow down the application**. To keep track of the events and the metrics, you need to provide values for the **EventStoreCapacity**, **MetricStoreCapacity**, and **MetricStoreSnapshotInterval** properties.

    d. The last required settings are those of the log downloader background thread. You need to provide values for the **EventPollSeconds** and **MetricPollSeconds** properties.

    The code of the method should look like this:

```
private static BackendConfiguration GetBackendConfiguration()
{
    BackendConfiguration backend = new BackendConfiguration();
    backend.Backend = "MsSql";
    backend.ProviderName = "System.Data.SqlClient";

    backend.Logging.LogEvents = LoggingLevel.Normal;
    backend.Logging.StackTrace = true;
    backend.Logging.EventStoreCapacity = 10000;
    backend.Logging.MetricStoreCapacity = 3600;
    backend.Logging.MetricStoreSnapshotInterval = 1000;
    backend.Logging.Downloader.EventPollSeconds = 1;
    backend.Logging.Downloader.MetricPollSeconds = 1;

    return backend;

}
```

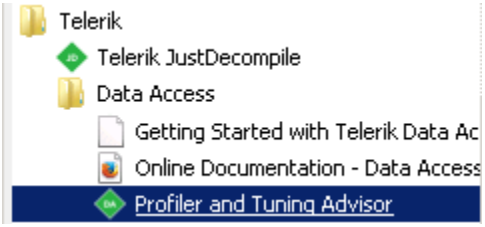
3. Next, the web service host has to be started from the application. In the *Consumer* project, open the *Program* file and paste the following code as the first line of the *Main* method. This will start the service communication on port 15555.

```
Telerik.OpenAccess.ServiceHost.ServiceHostManager.StartProfilerService(15555);
```

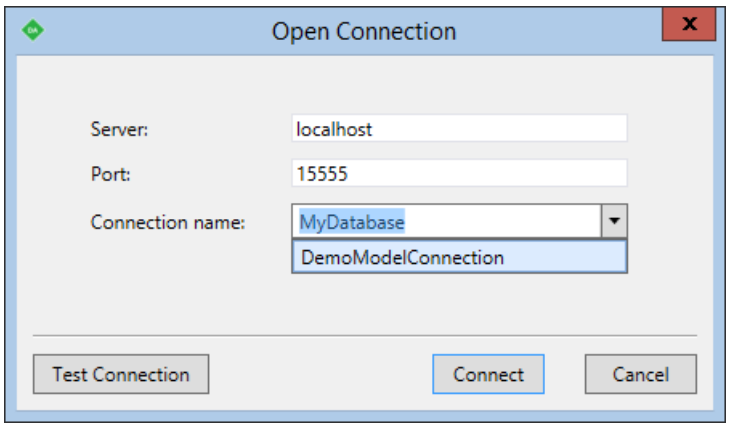
4. The final step before running the application and checking its performance in Profiler and Tuning Advisor is to modify the *Main* method a little. The idea is to ensure that the *Client* project will be running while we are reviewing its performance. For example:

```
static void Main(string[] args)
{
    Telerik.OpenAccess.ServiceHost.ServiceHostManager.StartProfilerService(15555);

    while (!Console.KeyAvailable)
    {
        using (DemoModelContext dbContext = new DemoModelContext())
        {
            // The code for the CRUD operations remains unchanged here.
        }
    }

    Telerik.OpenAccess.ServiceHost.ServiceHostManager.StopProfilerService();
}
```

5. While the application is running, open Profiler and Tuning Advisor. You can do this from the **Windows Start** menu.



6. Click on the **Connect To...** toolbar command and fill the **Open Connection** dialogue as demonstrated.



7. Review the statistics displayed in Profiler.

OFFLINE MONITORING

In offline profiling, all Telerik Data Access metrics and events are written to a log file, which can be reviewed in Profiler and Tuning Advisor at a later time.

Let's look at how to configure the sample model for offline monitoring.

1. In the *DemoModelContext* class, you need to find the *GetBackendConfiguration* method and to add a few settings in it:

   a. **MaxFileSizeKB** - specifies the max size of the log file in KB.

   b. **NumberOfBackups** - the number of old log files that Telerik Data Access keeps on the file system. After that number is exceeded, Telerik Data Access deletes the first log file before creating a new one. By default the maximum size of the log files is 1000KB and the maximum amount of historical files is 3.

   c. **Filename** - specifies the file name for the log output file.

   d. **EventBinary** - specifies if events should be logged in binary form.

   e. **MetricBinary** - specifies if the metric snapshots are logged in binary form.

```
private static BackendConfiguration GetBackendConfiguration()
{
    BackendConfiguration backend = new BackendConfiguration();
    backend.Backend = "MsSql";
    backend.ProviderName = "System.Data.SqlClient";

    backend.Logging.LogEvents = LoggingLevel.Normal;
    backend.Logging.StackTrace = true;
    backend.Logging.EventStoreCapacity = 10000;
    backend.Logging.MetricStoreCapacity = 3600;
    backend.Logging.MetricStoreSnapshotInterval = 1000;
    backend.Logging.Downloader.EventBinary = true;
    backend.Logging.Downloader.MetricBinary = true;
    backend.Logging.Downloader.Filename = "C:\\MyFileName";
    backend.Logging.Downloader.MaxFileSizeKB = 1000;
    backend.Logging.Downloader.NumberOfBackups = 3;
    backend.Logging.Downloader.EventPollSeconds = 1;
    backend.Logging.Downloader.MetricPollSeconds = 1;

    return backend;
}
```

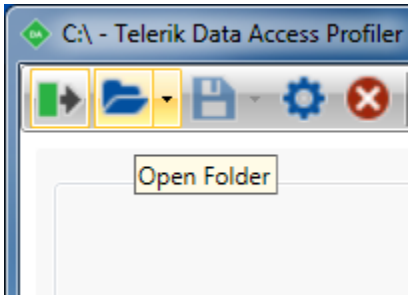
2. Before running the application and checking its statistics in Profiler and Tuning Advisor we have to modify the *Main* method a little. The idea is to ensure that the *Client* project will produce enough data.

```
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    {
        using (DemoModelContext dbContext = new DemoModelContext())
        {
            // The code for the CRUD operations remains unchanged here.
        }
```

```
        }
        Console.Write("All changes executed properly, " +
                "press any key to close.");
        Console.ReadKey();
    }
```

3. Run the application and after if finishes, start Profiler and Tuning Advisor.

4. In order to load the log files in Profiler and Tuning Advisor, you need to use the **Open Folder** toolbar command.



5. Navigate to the log location and open the **.oalog** file. The log should now be ready for you to explore.

FURTHER INFORMATION

Telerik Data Access has a rich set of comprehensive education resources that provide real-life solutions to everyday problems.

- Online Developer's Guide and Feature Reference – here you can find constantly updated information about Telerik Data Access, how-to articles, and best practices.

- Knowledge Base – contains articles you can search for solutions to your problems.

- Code Library – it is a place for exchanging code samples, sharing knowledge and getting hands-on experience.

- Forum – provides a place where you can communicate and consult with other people who use Telerik Data Access.