

# 2

## *The Application Definition File*

This chapter covers:

- A Detailed Description of The Application Definition File
- Application Definition file tools
- Common Problems
- Walk through of creating an application definition file

Our first chapter was a great introduction into the business problems the Business Data Catalog is trying to solve, and how you can use your Line of Business (LOB) data within SharePoint. We briefly mentioned that before you can use your LOB data within SharePoint that you need to create a XML application definition (Application Definition) file that lets SharePoint know how to connect to your LOB system and how to get the data from it. This chapter will focus on that XML Application Definition file in detail. Although there are tools now available for creating the Application Definition Files it is still important to understand what it is these files are doing in case you need to edit or trouble shoot an error.

### ***2.1 Application Definition File Introduction***

Initially when going through our Application Definition file we'll concentrate on a database system. In appendix 3 you'll find how to go about using web services - both creating BDC friendly web services and also changes required to the application definition file.

Do we really need to write all this XML?

When the BDC was first marketed by Microsoft it was touted as being a no-code solution for data integration with SharePoint that administrators could setup. Many people however consider writing a long XML application definition file by hand as writing code, and who can really blame them. We decided creating a tool to assist with the development of these application definition files would be a good idea and something people would love, and so

BDC Meta Man was born! Microsoft recently also brought The BDC ADF Schema part of the MOSS 2007 SDK. Although these tools alleviate a lot of the manual XML crafting, it is still very important to understand the structure of the Application Definition files to assist you with trouble shooting and manual edits. Rather than listing a whole Application Definition file and expecting you to page through reams of XML we're going to break it down to smaller manageable chunks.

### Listing 2.1 - Illustrates an example of how to create a LOB System in your Application Definition File

```
LOBSystem
The opening element of our Application Definition file is the LOBSystem
element
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<LobSystem xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://schemas.microsoft.com/office/2006/03/BusinessDataCatalog BDCMetadata.xsd" Type="Database"
Version="1.0.0.0" Name="Northwind"
xmlns="http://schemas.microsoft.com/office/2006/03/BusinessDataCatalog">
  <Properties>
    <Property Name="WildcardCharacter" Type="System.String">%</Property>
  </Properties>
</LobSystem>
```

The Wildcard Character for the Data Source e.g. % for SQL.

### **LOBSystem**

LOB stands for Line of Business System and as we'll see this LOB acronym is reused in a number of other places in our Application Definition files. There are a few standard attributes here such as the XML schema definition, however Type, Version and Name are very important.

**Type** – the Lob system type can be either Database or Web Service.

**Version** – The version here is important especially if you are already using the BDC in a live system and wish to make changes to it. Functionality such as the BDC web parts and the Business Data Column will always utilize the latest Application Definition version so if you want to make changes to a LOBSystem that is already being used then simply increase the version number and re-import. If you do this however ensure your latest version has all of the entities you had originally defined otherwise you may find web parts already consuming BDC data stop working.

**Name** – This name must be unique, but is not actually displayed within the SharePoint UI.

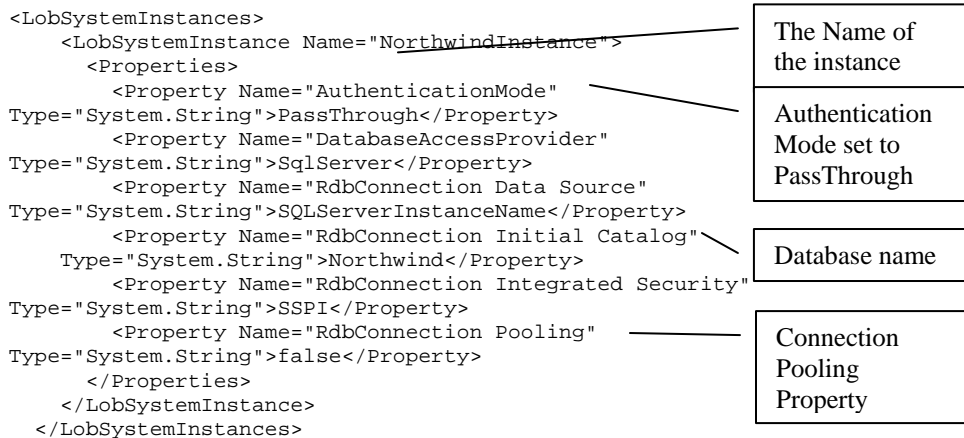
For our LOBSystem element you can also see that we can have properties defined. From experience the only property ever used here is the WildcardCharacter. This describes which

character you'd like to use for Wildcard searches – we'll dig deeper into this when we discuss entities and filters.

## LOBSystemInstances

So we have the beginnings of our Application Definition file, now we need to describe how SharePoint can connect to this LOBSystem. Within our LobSystem XML elements we have the following fragment:

### Listing 2.2 - Illustrates how to create a LOB System instance within your Application Definition File



It seems here with the opening Element being LobSystemInstances that you can actually define many Line of Business System within a single Application Definition file. With MOSS 2007 this isn't the case and if you tried to do that you will receive an error when trying to import the file to SharePoint. So within our LobSystemInstances Element we can define how we are going to connect to a single Line of Business System. We've already indicated that this is going to be a database connection and so therefore SharePoint will expect a number of things to appear here. As you can see our connection is actually made up of a number of properties. SharePoint will actually use a number of these properties to build a connection string when it comes to getting data.

#### AuthenticationMode

The authentication mode SharePoint is going to use is a biggie! ☺ Lots of problems and issues can occur when people are trying to configure the BDC if they do not correctly understand how each authentication mode works and what it means to your setup.

- **PassThrough** – pass through authentication attempts to execute queries against your LOB System as the person who is logged into SharePoint, typically this will be a users active directory account.
- **RevertToSelf** – RevertToSelf authentication takes no notice of the user who is

logged into SharePoint. Instead it will use the active directory account that the SharePoint application pool is running under to try to access the LOB System. RevertToSelf has a number of advantages:

- It can use connection pooling if the LOB System implements it as only one account (the application pools) is ever used to connect to the LOB System. The 'Double Hop' issue never occurs due to only a single hop ever taking place ie IIS app pool -> LOB System
- Although you may think that using RevertToSelf removes the ability to restrict individual users to access the LOB resource, remember these permissions can actually also be set per users for each BDC application via the Shared Services Administration pages.
- **Credentials** – Only ever used for SSO
  - Windows Credentials – same as above.
  - Database Credentials – a word of warning – Database credentials is not actually a value that can be used for the AuthenticationMode, but is more commonly used to describe when your SQL Server is setup to use SQL User Accounts. You may want to connect to SQL Server or Oracle by directly specifying the username and password the BDC should use. To do this the actual AuthenticationMode should be set to PassThrough, but the BDC will actually ignore this and then utilize the username and password set in the Rdb Connection Properties (details following shortly). Having to set the AuthenticationMode to PassThrough to allow you to supply a database user account is counter intuitive, but trust us it does work! Utilizing a database user account is another method of bypassing the Double Hop issue.

### **DatabaseAccessProvider**

The DatabaseAccessProvider is a property that you will only use when connecting to a database LOB System. The name supplied here indicates the ADO.NET client that you will use to connect to the system. SQL Server, OleDb, Oracle and ODBC are the options available. We have always had issues getting OleDb to work however so when integrating with these types of data sources have proceeded with an ODBC connection which is usually also available.

**Rdb Connection** - There are a number of LOBSystemInstance properties that are prefixed with Rdb Connection. The properties are pieced together by the BDC to build the connection string of how to connect. The various properties are:

- **Rdb Connection Data Source** - This is the data source that you want to connect to. If you are using SQL Server it would be the server name and instance eg databaseserver\sqliinstance1
- **Rdb Connection Initial Catalog** – If you are connecting to SQL Server, this will be the actual database that you wish to pull data from.
- **Rdb Connection Integrated Security** – If you are using Pass Through

authentication, setting this property to SSPI indicates that...

- **Rdb Connection Pooling** – set to true if you wish to make use of Connection Pooling. The parameters listed above are the minimum required to enable a connection to SQL Server using Pass Through authentication. There is however additional parameters you may want to make use of
- **Rdb Connection User Id** – if you are wishing to connect using a database user account you can specify the username with this property. If you are using these values SSPI should be set to a value of false. Should be used in conjunction with the next property
- **Rdb Connection Password** – as above, required if you want to connect with a database account.

If you are using the Single Sign On service as an authentication vehicle then there are a few other properties that are required to describe how to connect to the SSO application these are...

**SsoProviderImplementation** – Fully qualified type name of the ISsoProvider implementation that stores credentials used to log in to the database. Used if Authentication Mode is set to Credentials or WindowsCredentials

**SsoApplicationId** - ID of the SSO enterprise application definition that stores credentials used to connect to the database.

Phew....and that's about it. Those are the general settings you'll make use of to connect to your Line of Business database System, now let's move on to how you can actually get at the data via things called Entities.

## **Entities**

Now we need to describe the data from your LOB system in SharePoint. We do this via things called Entities. An entity should be thought of as a real world object e.g. a Customer, Product, or Order. To programmers who are familiar with object orientated programming this concept is really the same as describing objects as classes.

A LOBSystem can contain many entities so we need to be able to define many entities within our application definition file. We also want to be able to give Entities useful names such as Customer or Product as the user will be selecting from a list of entities within SharePoint – a meaningful name makes it a lot easier for people to select the right entity.

Here is an outline of how 3 entities can be described within our LOBSystem element:

### **Listing 2.3 - Explains how to create an Entity within your Application Definition File**

```
<Entities>
  <Entity EstimatedInstanceCount="10000" Name="Product">
  </Entity>
  <Entity EstimatedInstanceCount="10000" Name="Customer">
  </Entity>
  <Entity EstimatedInstanceCount="10000" Name="Order">
```

```
</Entity>
</Entities>
```

Giving our entity a nice name will help our users be able to select the correct data in SharePoint. Each Entity also has an EstimatedInstanceCount. This property can be used by client based applications to change how it displays or pages through your entity data. You do not actually get any problems with your entities if you supply an estimated count of 0, although it is a required field and so must exist and contain some number.

Within each Entity you need to define a number of properties and settings for various bits of BDC functionality to work correctly. First your Entity can have a Title property

#### **Listing 2.4 - Displays how to create the Property Descriptors in the Application Definition File.**

```
<Properties>
  <Property Name="Title" Type="System.String">Name</Property>
</Properties>
```

This Title property is the column that is going to have the drop down actions menu set against it. This column must be returned by our Finder method and when set in our Application Definition file must also be of System.String type. You can however change this to a field of any type through editing the BDC Data List Web Part once you have it displayed on your SharePoint page. Your entity can have many identifiers.

#### **Listing 2.5 - Displays how you can create an Identifier or many identifiers in your Application Definition File.**

```
<Identifiers>
  <Identifier Name="ProductID" TypeName="System.Int32" />
</Identifiers>
```

As with SQL Server database table you should think of an identifier as being the primary key field. The field that is marked as an identifier can be used to uniquely identify each row of data. Note that in the identifiers element all we do is define the name of our identifier and its type. The name does not relate to a column, we need to explicitly tie an identifier to a column later in our XML.

Now we need each entity to be able to pull some data back. We do this by defining methods within our Entity. What method types we create in our entity will actually have an impact on how our Business Data can be used within SharePoint. Let's break this down into a simple table:

<b>Method Type</b>	<b>Where it is used</b>
Finder	BDC Data List Web Part Business Data Column
SpecificFinder	BDC Profile Page Business Data Column

Search  
IdEnumerator Search

The **finder** method is the one to return your general BDC data. In terms of databases it will do a select within your database and return the information to either the BDC Data List Web Part or the Business Data Column entity picker. The finder method can have SQL where clauses to allow users to filter what data is brought back.

The **SpecificFinder** method returns just a single row of data. This means you must pass in the necessary parameters to uniquely identify a row of data from your chosen table or stored procedure. If your SpecificFinder method is configured incorrectly and returns multiple rows of data for a given parameter then you will get errors when you try to crawl your Line of Business system with MOSS search.

The **IdEnumerator** method is only ever used for crawling process of MOSS search. When you set a crawl going MOSS will first of all execute the IdEnumerator method. It is a simple method that returns a list of identifier values that MOSS should crawl. The crawler will then call the SpecificFinder method for each identifier value returned, passing this identifier in as the necessary parameter to uniquely identify the record. As the SpecificFinder is used in this manner, any field that you would like your users to be able to search on should be returned in the SpecificFinder method.

We will initially study the Finder method as this is generally the most complex of the three described above. Currently within our Entity we have the following:

**Listing 2.6 - Displays the Code to create an Entity with a number of methods.**

```
<Entity EstimatedInstanceCount="10000" Name="Product">
// properties
// identifiers
<Methods>
<Method Name="GetProducts">
  // Properties
  // FilterDescriptors
  // Parameters
  // MethodInstances
</Method>
</ Methods>
</Entity>
```

Unique Finder Method Name and Properties

Firstly our method needs a unique name. A meaningful name that explains the functionality is always a good idea. Within our method we've labeled the different elements that can make up our Method - Properties, FilterDescriptors, Parameters and MethodInstances.

**Listing 2.7- Demonstrates how to create properties in your Application Definition File**

```
<Properties>
  <Property Name="RdbCommandText" Type="System.String">
    SELECT ProductID, Name, ProductNumber, ListPrice FROM
```

Please post comments or corrections to the Author Online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=500>  
Licensed to Manning Marketing <stho@manning.com>  
Regular Select Statement to query database.

```

Product WHERE (ProductID = @ProductId) AND (Name LIKE @Name) AND
(ProductNumber LIKE @ProductNumber)
    </Property>
    <Property Name="RdbCommandType"
Type="System.Data.CommandType">Text</Property>
    <!-- For database systems, can be Text, StoredProcedure,
or TableDirect. -->
</Properties>

```

As we've seen already within our application definition file Properties are used to add values to particular objects within our file. A method for a database needs two properties, RdbCommandText and RdbCommandType. RdbCommandType can give values as described in the above comments Text, StoredProcedure and TableDirect. RdbCommandText will have the value associated with the RdbCommandType eg a stored procedure name, some SQL select statement or the name of table. You will notice that in our example here our SQL select statement has a number of parameters added in the WHERE clause. These parameters need to be defined in the Parameters section, but also if you want your users to be able to use these parameters as filters they need to be described as FilterDescriptors.

### ***FilterDescriptors***

The BDC Data List Web Part and the Entity Data Picker used with the Business Data Column will be executing whatever SQL statement or stored procedure that is defined in our finder methods properties section. This select statement could in theory return 1,000's of rows which is not generally good for users or the backend data source or infrastructure. It would be a lot more useable and a better solution to allow users to be able to filter the data that they can view. To do this a method generally needs three things for each column that you want to be able to filter – a FilterDescriptor, a Parameter and also the SQL statement or stored procedure to make use of the parameter/filter.

### **Listing 2.8 - Describes how to create a Filter Descriptor in your Application Definition File.**

```

<FilterDescriptors>
    <FilterDescriptor Type="Comparison" Name="ProductId" />
    <FilterDescriptor Type="Wildcard" Name="Name">
    <FilterDescriptor Type="Wildcard" Name="ProductNumber" />
</FilterDescriptors>

```

You can see from our SQL select statement earlier we have a FilterDescriptor defined for each of the parts that make up our WHERE clause. The Type of Filter has a lot to do with how the user interface looks. If you choose the Type to be Comparison you are presented with the filter options in the Business.

Data List Web Part:

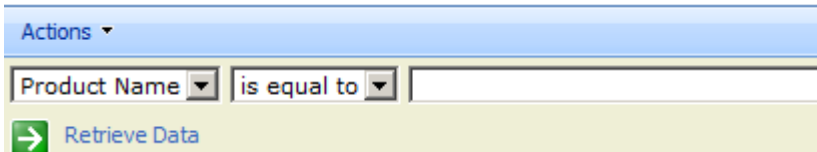


Figure 2.1 - Illustrates the use of the Filter Descriptor in a Data List Web Part

However if you choose Wildcard you are presented with the following:

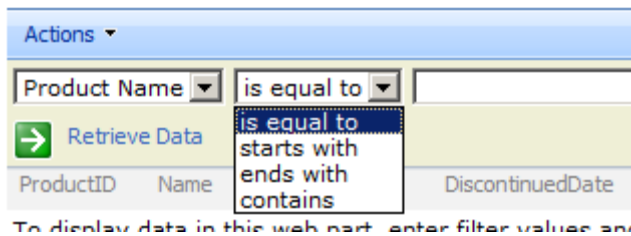


Figure 2.2 - The above image illustrates the effect of using a wildcard filter in your application definition file.

You may think that choosing Wildcard for all string filters would make sense, but there seems to be a problem with using a Wildcard filter, and the general wildcard character. If you have your Filter set to 'Contains' and you enter the filter value to be % you would expect it to return all the data. This isn't the case however and you will find that no data is visible. If you do want this type of functionality you should choose Comparison as the type of filter. With this filter for a string column you'll find that the wildcard symbol will work as expected with the small caveat still that % will not return all data.

### Parameter

Parameters are a big thing within Application Definition files as they describe the values you are passing in to filter at the backend data source, and also the data that the backend is returning to you. We'll therefore break this up further to Input Parameters and Output Parameters.

#### INPUT PARAMETERS

For each Parameter described in SQL statement or stored procedure we need to define it in our definition file. These parameters may or may not also be FilterDescriptors – more on the why not later! Let's take a look at our first parameter.

#### Listing 2.9 - Shows how to create a Input Parameter for an Entity within your Application Definition File.

```
<Parameter Direction="In" Name="@ProductId">
  <TypeDescriptor TypeName="System.Int32" >
```

Direction and  
Name of the  
Parameter

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=500>

Licensed to Manning Marketing <stho@manning.com>

```

        Name="ProductID"
        AssociatedFilter="ProductId"
        IdentifierName="[ProductID]">
    <DefaultValues>
        <DefaultValue MethodInstanceName="dbo.[Product]Finder"
            Type="System.Int32">1</DefaultValue>
    </DefaultValues>
    </TypeDescriptor>
</Parameter>

```

While this input parameter initially seems complex, once you have an understanding of it will seem easy! The first part to mention is the name of the parameter. This maps to the name given to a parameter in either our SQL statement or stored procedure. The direction is marked as 'In', meaning this is a value we are going to pass in/to our query/stored procedure.

Within our Parameter we have a TypeDescriptor element. A TypeDescriptor is a way of defining the data type of a Parameter. For input parameters they are generally only a single level deep, however we'll soon see for 'Out' parameters that TypeDescriptors can be multi-level and get quite complex. Back to looking at our particular type descriptor, we can see here we have a TypeName which casts the values as a particular .NET framework Type. If the parameter you are passing in is being used against a column that is also marked as an Identifier then you must mark it as so by adding the IdentifierName in the TypeDescriptor. This is not necessarily used in the Finder method but we'll see why it is very important in a SpecificFinder method shortly.

If this parameter is something that a user can dynamically supply a value for then this will be achieved by matching the parameter with a FilterDescriptor. We do this by adding an AssociatedFilter attribute to the parameters TypeDescriptor with the value matching a FilterDescriptor.

Within our TypeDescriptor element for our input parameter we have some DefaultValues elements. Our sql statement could contain one, two or three parameters eg Username, age, city. The user may only want to filter by Username however, so to allow the statement to be executed with just this one user entered filter, default values are described that can be substituted in if they are not supplied by the user. You'll see our DefaultValue element has a MethodInstance attribute. MethodInstances are defined at the end of a method which we'll cover shortly – for now just remember that your DefaultValue element will need a MethodInstance element. Our DefaultValue also has to have a Type eg System.Int32. Again this is a .NET type the same as TypeName of our main Parameter element was.

Input parameters will not be directly mapped to FilterDescriptors for SpecificFinder methods. This is because the parameter that is used to return a unique row by the SpecificFinder is only really ever used internally by the BDC, and not in the user interface. In fact it only really makes sense to create FilterDescriptors at all for the Finder method.

So that covers Input Parameters, pretty simple huh? Well now we come to output parameters – this is quite a scary beast!

## OUTPUT PARAMETERS

The Business Data Catalog utilizes the ADO.NET data access classes to retrieve data and return it to the SharePoint front end. The classes it utilizes are the DataReader and DataRecord, and these must be defined in return parameter. Let us take a look at the piece of XML

### Listing 2.10 - An example Output Parameter for an Entity within your Application Definition File.

```
<Parameter Direction="Return" Name="dbo.Product">
  <TypeDescriptor TypeName="System.Data.IDataReader, System.Data,
Version=2.0.3600.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
  IsCollection="true"
    Name="dbo.ProductDataReader">
    <TypeDescriptors>
      <TypeDescriptor TypeName="System.Data.IDataRecord,
System.Data, Version=2.0.3600.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"
        Name="dbo.ProductDataRecord">
        <!-- Listing 2.11 goes here -->
      </TypeDescriptor>
    </TypeDescriptors>
  </TypeDescriptor>
</Parameter>
```

Defining the data that is being returned by your query/stored procedure/web method is about the most complex looking piece of XML in your application definition files, how complex it will be also depends on the type of BDC method. For example if you are defining a Finder method it will be returning many records of data so you need to define a DataReader and DataRecord in your XML. However a SpecificFinder method will only be returning a single row of data and therefore only needs the DataRecord level.

Once you have gotten down the DataRecord level in your XML you need to define each individual field that is being returned. At the most basic level you need to define the name of the field coming back, and the datatype.

### Listing 2.11 - How to create a DataRecord within your Application Definition File.

```
<TypeDescriptors>
  <TypeDescriptor TypeName="System.Int32"
    Name="ProductID"
    IdentifierName="ProductID" />
  <TypeDescriptor TypeName="System.String" Name="Name" />
  <TypeDescriptor TypeName="System.String" Name="ProductNumber" />
  <TypeDescriptor TypeName="System.Decimal" Name="ListPrice" />
</TypeDescriptors>
```

If the data field that is coming back is one of your identifiers columns then you need to mark it with the correct identifier name, this is as you can see with the ProductId field.

There are a number of other attributes that can be set for each TypeDescriptor field being returned:

- **Name** – this should match the name of your field as it is being returned from the data source.
- **TypeName** – the .NET data type as returned by your data source. Depending on your data source you may need to translate the data source type to the relevant .NET data type.
- **IdentifierName** – as detailed above if the column being brought back is an identifier it should be marked as so. This is so the BDC web parts know which columns to pass around in association relationships, and for search functionality.
- **IdentifierEntityName** – used within association methods. See later section.
- **AssociatedFilter** – If you have a filter associated with a column, it must be named as so.

Of course not every database column being returned has a nice name that your users would understand, and in a multilingual deployment users in different locations will want localized column names. This is achieved by adding the Localized element to each typedescriptor column being returned:

#### Listing 2.12 - Demonstrates how to create localized names or Aliases

```
<TypeDescriptor TypeName="System.String" Name="Name">
  <LocalizedDisplayNames>
    <LocalizedDisplayName LCID="1033">Product
Name</LocalizedDisplayName>
  </LocalizedDisplayNames>
</TypeDescriptor>
```

As you can hopefully tell you can add as many different LocalizedDisplayName elements to a TypeDescriptor as you like, modifying the Locale ID (LCID) to the relevant country code. There is one type of Property element you can add to a TypeDescriptor that is being used to describe a returned column and this is ShowInPicker.

#### Listing 2.13 - Demonstrates how to create a Showinpicker property

```
<TypeDescriptor TypeName="System.String" Name="ProductNumber">
  <Properties>
    <Property Name="ShowInPicker"
Type="System.Boolean">true</Property>
  </Properties>
</TypeDescriptor>
```

This is used in the data picker that is part of the Business Data Column used in lists and libraries. By default this picker will only show the identifier columns for an entity but if you mark a TypeDescriptor with the ShowInPicker property it will appear here. You also only

need to make use of the ShowInPicker property in your Finder method. In other BDC method types it has no relevance.

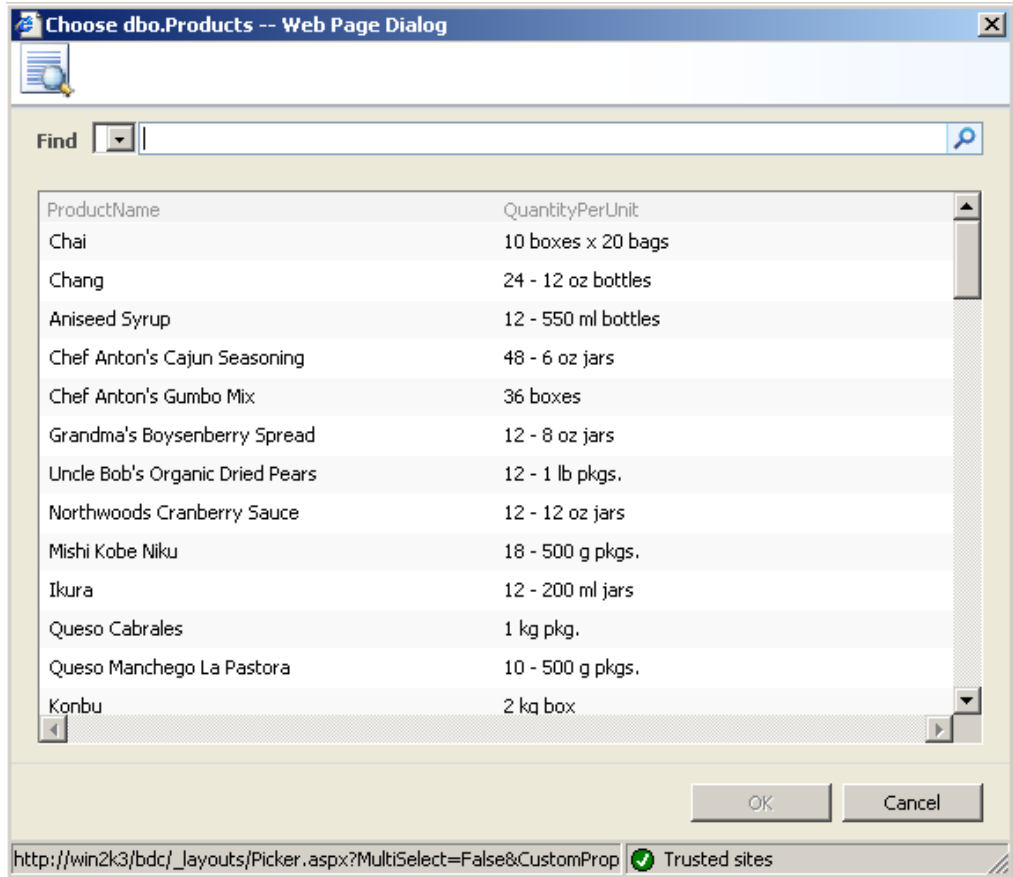


Figure 2.3 - Demonstrates the use of the Show In Picker Property

Another interesting thing about the data picker is the search area at the top. To be able to search your data you must create Filters for your entity. No filters means no filtering of data even though the controls to do this are still displayed – a common support question for us!

#### METHODINSTANCES

The final thing you need to do with your method is let the BDC know what method type you have just described. This is done by adding the MethodInstances element to the end of your method description:

**Listing 2.14 - Demonstrates how to create a Method Instance in your Application Definition File**

```
<MethodInstances>  
  <MethodInstance Name="dbo.ProductFinder" Type="Finder"  
    ReturnParameterName="dbo.Product"  
    ReturnPropertyDescriptorName="dbo.ProductDataReader"  
    ReturnPropertyDescriptorLevel="0" />  
</MethodInstances>
```

As you will see here the XML fragment begins with MethodInstances element. This goes to indicate that a method defined in your Application Definition file can actually implement more than one BDC method. This statement is true, by clever use of parameters and default values for the parameters you could enable a method to act as a Finder and SpecificFinder BDC method. We have found though that an Application Definition file is always much easier to understand if you keep the BDC methods separate.

The most important attribute of our MethodInstance is obviously the Type. As discussed the Types we are interested in that relate to out of the box BDC functionality are Finder, SpecificFinder and IdEnumerator. The ReturnParameterName and ReturnPropertyDescriptorName are the objects described above that will return data to us.

**ASSOCIATION METHODS**

Using the BDC List Web Part and the BDC Related Data (check) Web Part you can display a parent -> child relationship, which is similar to a one to many relationship. A good example of this may be Customer -> Orders relationship where you may want to click on a customer, and see a list of orders that customer made. In database terms this occurs as you have a primary key in Customers that links to a foreign key in Orders. Typically both of these columns would be called CustomerId, and you can see how they are joined in a database by the diagram below:

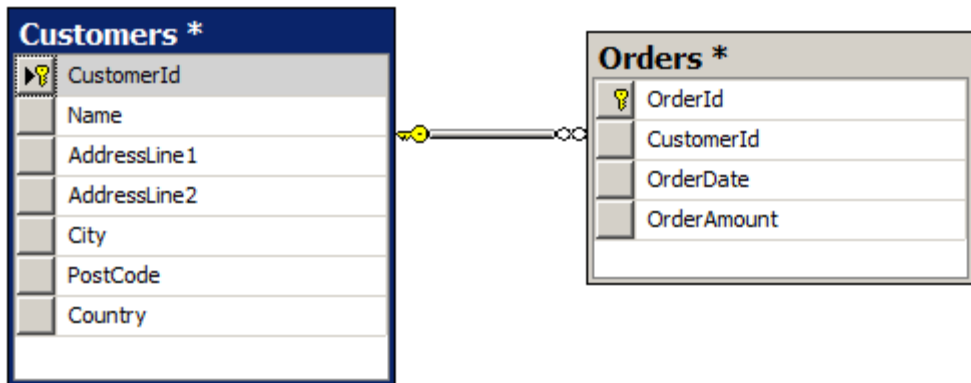


Figure 2.4 - Illustrates a relationship diagram in SQL - and how we should mirror that to create associations in our Application Definition File.

When we have our Customer and Orders defined as BDC entities we need to know how we are going to pass a primary key value from one entity to the other. This is done by passing identifier values from the parent entity to the child entity, and the child entity using this (these) identifier values to return related records.

The first important decision is where do you place your Association method? Should it go in the Customer entity or the Orders entity? The answer is it can go in either, best practise however is to place it in the child entity, this being Orders. It becomes impossible to build multiple cascading associations if you place the Association method in the parent entity (eg you want to do Customers -> Orders -> Products)

Let's see an Association method and explain it again

**Listing 2.15 - Demonstrates an example of creating Associations within your Application Definition File**

```

<Method Name="Getdbo.OrdersFordbo.Customers">
  <Properties>
    <Property Name="RdbCommandText" Type="System.String">Select
dbo.[Orders].[OrderId],dbo.[Orders].[CustomerId],dbo.[Orders].[OrderDate],
dbo.[Orders].[OrderAmount] FROM dbo.[Customers], dbo.[Orders] Where
dbo.[Customers].[CustomerId]=dbo.[Orders].[CustomerId] and
dbo.[Customers].[CustomerId]=@CustomerId</Property>
    <Property Name="RdbCommandType" Type="System.String">Text</Property>
  </Properties>
  <Parameters>
    <Parameter Direction="In" Name="@CustomerId">
      <PropertyDescriptor TypeName="System.Int32" Name="[CustomerId]"
IdentifierName="[CustomerId]" IdentifierEntityName="dbo.Customers" />
    </Parameter>
    <Parameter Direction="Return" Name="dbo.Orders">
      <PropertyDescriptor TypeName="System.Data.IDataReader, System.Data,
Version=2.0.3600.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
IsCollection="true" Name="dbo.OrdersDataReader" />
      <TypeDescriptors>
        <PropertyDescriptor TypeName="System.Data.IDataRecord, System.Data,
Version=2.0.3600.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
Name="dbo.OrdersDataRecord">
          <TypeDescriptors>
            <PropertyDescriptor TypeName="System.Int32"
Name="OrderId"
IdentifierName="[OrderId]"
IdentifierEntityName="dbo.Orders" />
            <PropertyDescriptor TypeName="System.Int32"
Name="CustomerId"
IdentifierName="[CustomerId]"
IdentifierEntityName="dbo.Customers" />
            <PropertyDescriptor TypeName="System.String" Name="OrderDate" />
            <PropertyDescriptor TypeName="System.Decimal" Name="OrderAmount" />
          </TypeDescriptors>
        </PropertyDescriptor>
      </TypeDescriptors>
    </Parameter>
  </Parameters>
  <Body>
    <Text>
      <Text></Text>
    </Text>
  </Body>
</Method>

```

Input Parameter  
Return Type Set

Strongly Types Type Descriptors

```
</Parameters>
</Method>
```

In this example we would be following our best practice and placing the Association method in the child entity ie our Order entity.

As you can see our Association method looks like a finder method as in it returns a DataReader and DataRecords. The input parameter is the primary key that the query needs to be able to filter the orders coming back by a particular customer. You'll notice that this has an identifier name, but our Order entity will not have an identifier called this. This is where the attribute IdentifierEntityName comes in as you the BDC knows it wants to get the Identifier value from this particular entity. This also makes it matter how the entities are ordered in your application definition file. The Business Data Catalog imports an application definition file in an interpreter type manner line by line. If during the import the BDC comes across an entity with an association method that refers to an IdentifierEntityName where the entity has yet to be imported – the import process will error. This means your parent entity must be ordered before what will be the child entity.

As well as your input parameter needing to be marked with the parent identifier name, the column name being returned as a typedescriptor which is the foreign key column needs to be marked to match up with the identifier. You can see this on line X of our example.

The one thing you'll notice that is missing from our Method is a MethodInstance element, and this is because Association methods don't have them, simple as that. So if you see a method with no MethodInstance tags it's either meant to be an Association Method or an error!

One final thing to get Associations working

There is one final thing to get Associations working to allow you to display a Parent -> Child relationship. At the end of our application definition files outside of all our entity definitions, we need to define how our entities are linked together with the use of Associations tags

### Listing 2.16 - How to set the parent/child relationship.

```
<Associations>
  <Association AssociationMethodName="dbo.Orders"
AssociationMethodName="Getdbo.OrdersFordbo.Customers"
AssociationMethodReturnParameterName="dbo.Orders"
Name="dbo.CustomersTodbo.Orders" IsCached="true">
  <SourceEntity Name="dbo.Customers" />
  <DestinationEntity Name="dbo.Orders" />
</Association>
</Associations>
```

Association  
Between  
two Entities

This isn't too hard to explain, but we'll break it down into its individual sections

- **AssociationMethodName** – a unique name for this association. This name will appear in the Related data web part so best to give it a name that users will understand

- **AssociationMethodEntityName** – the name of the entity that contains the Association method the BDC should execute.
- **AssociationMethodReturnParameterName** – the name of the return parameter that describes our data that is coming back for our association method.
- Each Association also has at least 2 elements
- **SourceEntity** – the name of the parent entity. Usually this is a single entry, but occasionally it may be a list of entities if you have a 2 parent -> child relationship (eg Customer and Location -> Orders)
- **DestinationEntity** – the name of the child entity

#### **PROBLEMS WITH ASSOCIATIONS**

Associations are great and work really well, that is until you get to an entity that a composite key. Composite keys are a primary key that is made up of multiple columns. Having an entity which is based on a database table that has a composite key is not a problem as the entity will just contain multiple identifiers, each one identifying one of the columns in the composite keys. This becomes a problem when this entity is the parent in an association relationship. This is because the Business Data Catalog will try to pass all the identifiers through to the association method, when your association method may only be expecting one identifier (as usually happens with one to many relationships).

If you are working with an entity that needs to have multiple identifiers, you could do the following steps

1. Create your original entity with multiple identifiers (composite key) – this entity will be used for Search
2. Create a copy of the entity in step one (named differently) but only with a single identifier, basically the one you want to be the filter for your association method.

You could do the above with BDC Meta Man by using the original table, and creating a view for the second step and only having it return the key column you are interested in.

#### ***Business Data Catalog Actions***

Easily displaying your Line of Business data within SharePoint is a fantastic start but you may want to perform certain actions with your data from the SharePoint user interface. Examples of this may be:

- Create or update data
- View location information on a map
- Email a contact

Actions are defined at the entity level after all the Methods have been defined. Here's an example of an Action:

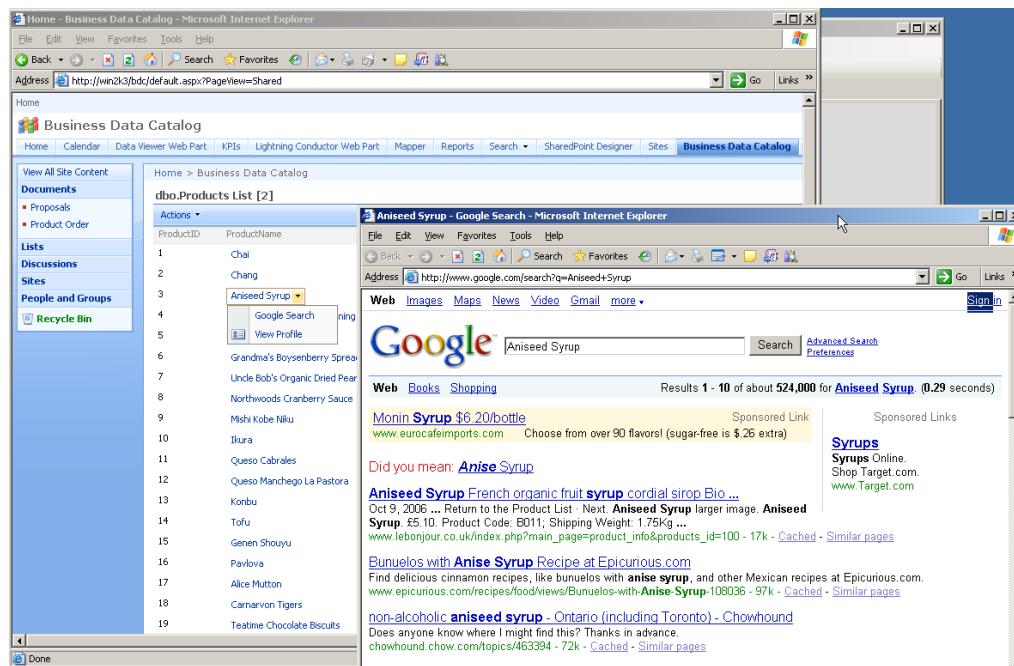


Figure 2.5 - Displays a Custom Action of a Google Search

The attributes are pretty self explanatory. The URL is the most significant as this is where your user will get navigated to once then click on the action link. You may want the link to be dynamically generated so users are navigated to a particular pages record and this is achieved with query strings and ActionParameter definitions. Here's an example of a URL value

<http://crm/customer.aspx?customerID={0}>

The query string value {0} is dynamically replaced by the BDC field that is defined in your action parameter. If you have a url that has a number of query string values their locations within the URL can be defined by {0}, {1}....{n} with the ActionParameters index number relating to the number within the brackets.

## Creating an Application Definition File

Now that we have discussed the Application Definition File, Let us walk through the real world steps on how to create and import your application definition file. In order for these steps to work, you will need the Northwind Database installed in SQL Server. The northwind database can be found here:

<http://www.microsoft.com/downloads/details.aspx?FamilyID=06616212-0356-46A0-8DA2-EEBC53A68034&displaylang=en>

For this exercise to work, we are going to use the free version of BDC Meta Man to generate an application definition file:

1. Open your browser and logon to [www.lightningtools.com](http://www.lightningtools.com)
2. On the Home Page under Latest Releases, download BDC Meta Man.
3. Extract the Zip File, and run the Setup.exe
4. BDC Meta Man will install.
5. When you run the program, choose the 'Try it' option which will give you access to the developer version.
6. When BDC Meta Man Launches, Click the 'Connect to Data Source' button.
7. Choose SQL Server
8. Type your Server Name and Instance for your SQL Server in the format ServerName\Instance Name or Just ServerName if you do not have multiple instances.
9. Choose your Authentication Type - I am using Windows Authentication because my SQL Server is set to Mixed Mode.

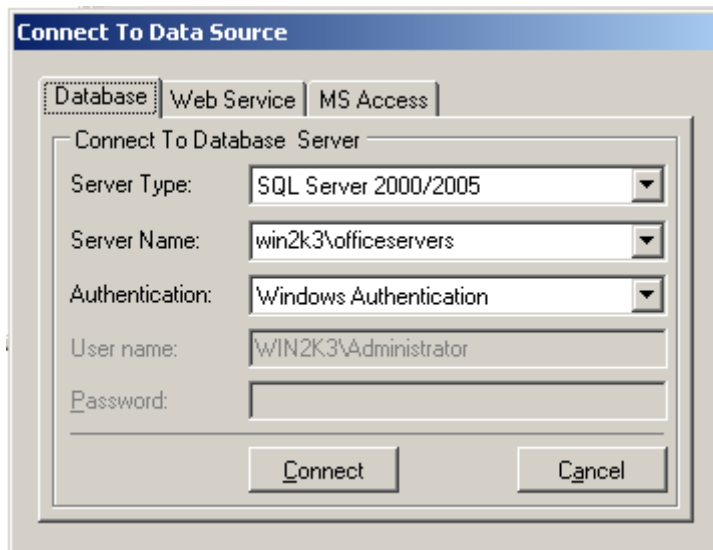


Figure 2.6 Displays the Connect to Data Source Dialog Box with the Authentication set to Windows Authentication.

10. Click Connect
11. You should see a list of Databases on the left hand side of your screen.

12. Expand Northwind
13. Drag the dbo.Products Table onto the Design Surface.
14. Drag the dbo.Categories table onto the design surface.
15. Drag the CategoryID from the Categories Table onto the CategoryID field in the Products Table to create an association. As shown below

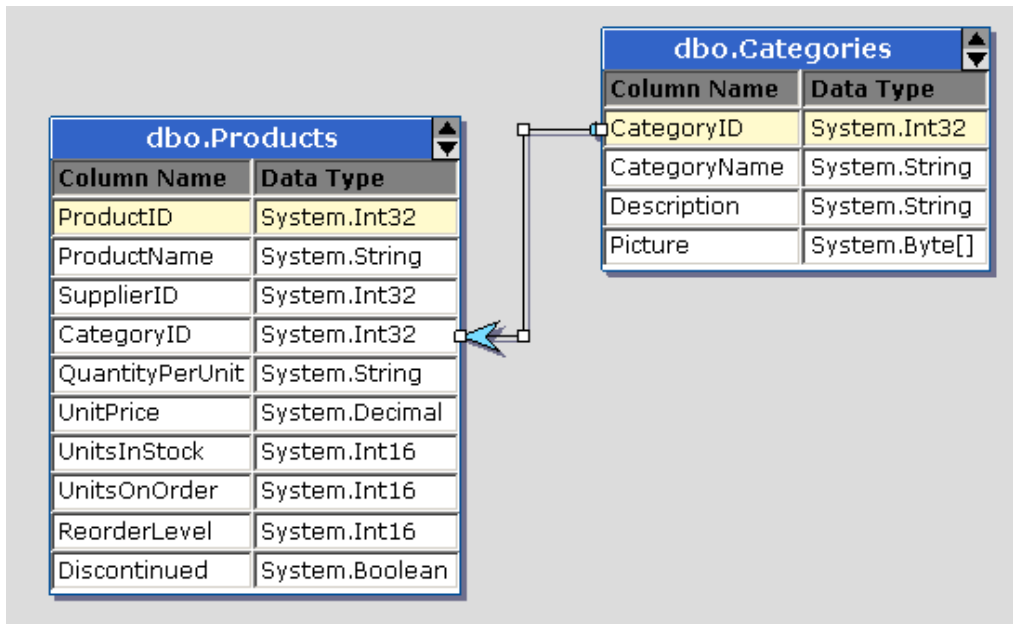


Figure 2.7 displays two Entities with an Association in BDC Meta Man.

16. Right Click the dbo.Products Entity and choose Edit Entity.
17. Click the Actions Tab
18. Click Add Pre Defined Action
19. Choose MSN Live Search and click Select
20. Select the Google Search Action, and then choose your column to search as ProductName as shown below:

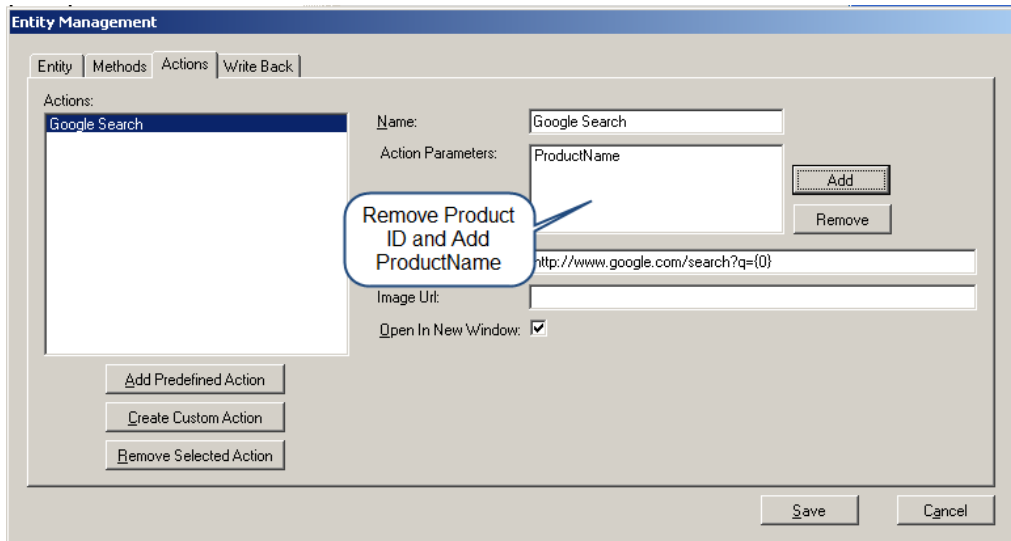


Figure 2.8 displays the Entity Management Dialog Box in BDC Meta Man and also demonstrates how to create a Custom Action.

21. Click Save
22. In BDC Meta Man click Configuration, Settings from the Menu Bar
23. Set your Application Definition File name e.g. c:\AppDefs\Products.xml
24. Click Save
25. Click the Green Run Icon in the top left hand corner to generate the XML.
26. Click Yes to open the XML in the browser. You can now review the Application Definition File.
27. Close the browser, and navigate to SharePoint Central Administration by choosing Start, All Programs, Microsoft Office Server, SharePoint 3.0 Central Administration.
28. In SharePoint Central Administration, on the left hand pane, click your Shared Services Provider. It should be immediately below 'Shared Services Administration'
29. In the Business Data section, click Import Application Definition.
30. Browse to your file that was generated e.g. c:\appdefs\products.xml
31. Click Import.
32. The Application Defintiion File should import successfully.

In a Appendix 3 of this book, we will explore how to generate an ADF file for a Web Service. This will include creating the Web Service to connect to the Data Source.

## ***Summary***

Within this chapter we have explored in detail how to create the Application Definition file that populates the meta database with Business Data Catalog Properties. We explored how to configure database connection information, define Entities, Type Descriptors, Parameters, Methods and Actions.

Throughout this book we will explore other examples on how to connect to different sources including SAP, Siebel, and Web Services.