Working with Plug-ins

Release 7.3

Legal Notices

Verivo Software, Inc.
1000 Winter Street
Waltham, MA 02451
781.795.8200
www.verivo.com

Table of Contents

1. Working with Plug-ins . . . 4
1.1 Introduction to Plug-ins . . . 4
1.2 Connecting Plug-ins to a Data Source . . . 7
1.3 Using Plug-ins . . . 10
1.3.1 Using the REST Plug-in . . . 11
1.3.1.1 Installing the REST Plug-in . . . 11
1.3.1.2 Creating a Metadata File . . . 12
1.3.1.2.1 Metadata Schema Definition . . . 19
1.3.1.3 Connecting to a REST-based Data Source . . . 23
1.3.1.4 Passing Data to a REST Service . . . 24
1.3.1.5 Debugging REST Data Source Connections . . . 31
1.3.2 Using the WSDL Plug-in . . . 32
1.3.2.1 Connecting to a WSDL-based Data Source . . . 36
1.3.2.2 Passing Data to a WSDL-based service . . . 37
1.3.2.3 Debugging WSDL Data Source Connections . . . 42
1.3.2.4 Limitations of the WSDL Plug-in . . . 45
1.3.2.5 Using a Custom Template . . . 48
1.3.3 Using the WCF Plug-in . . . 49
1.3.3.1 Debugging WCF Data Source Connections . . . 54
1.3.4 Using the SharePoint Plug-in . . . 55
1.3.5 Using the Salesforce 9 Plug-in . . . 58
1.4 Using an HTTP Proxy Server . . . 58
1.5 Checking Plug-in Status . . . 59
1.6 Developing Custom Plug-ins . . . 60
1.6.1 About the Plug-in SDK . . . 61
1.6.2 Plug-In Application Framework . . . 62
1.6.3 Creating a Custom WSDL Plug-in . . . 65
1.6.4 Creating a Custom Plug-in . . . 67
1.6.4.1 Creating a Class Library Project . . . 68
1.6.4.2 Describing the Data Source . . . 69
1.6.4.3 Processing RequestMessage Objects . . . 70
1.6.4.4 Connecting a Custom Plug-in to a Data Source . . . 74
1.6.4.5 Using Database Connections . . . 78
1.6.4.6 Creating ResponseMessage objects . . . 80
1.6.4.7 Logging Plug-in Activity . . . 82
1.6.4.8 Testing Connections . . . 83
1.6.5 Compiling a Custom Plug-in . . . 84
1.6.6 Deploying a Custom Plug-in . . . 84

Working with Plug-ins

Plug-ins facilitate the connection between a mobile application and a data source.

This section describes how to develop and deploy custom plug-ins as well as use pre-existing extensible plug-ins:

- Introduction to Plug-ins
- Connecting Plug-ins to a Data Source
- Using Plug-ins
- Using an HTTP Proxy Server
- Checking Plug-in Status
- Developing Custom Plug-ins

Introduction to Plug-ins

Plug-ins facilitate the connection between a mobile application and a data source.

This section describes the following topics:

- Types of plug-ins
- Plug-in workflow
- Downloading and installing non-core plug-ins

Types of plug-ins

There are three types of plug-ins:

- **Pre-built:** The Verivo platform includes a number of pre-built plug-ins for connecting to common data sources. Some plug-ins are pre-installed with your Verivo AppServer and AppStudio. Most others must be downloaded and installed from the Verivo website.

  There are two flavors of pre-built plug-ins: *core* and *downloadable (*or *non-core)*. *Core* plug-ins are included with the platform installers. When you install AppServer and AppStudio, you do not need to install the core plug-ins separately. *Downloadable* plug-ins are those that can be downloaded from the Verivo FTP site. They are not installed as part of the default AppServer and AppStudio installs.

  The following table lists the core and non-core plug-ins:

Core plug-ins	Downloadable (non-core) plug-ins
• REST • SQL-92 • SQL Lite • SQL Server	• DB2 • HTTP • Oracle • Oracle Shared Procedure • SalesForce9 • SharePoint • SiebelOnDemand • SiebelWse3 • SQL-92 Stored Procedure • SQL Stored Procedure • SQL Stored Procedure Auth • Sybase • WCF • WSDL

  The core plug-ins are updated with each new release of the platform. When you upgrade the platform, these plug-ins are upgraded. The non-core plug-ins are updated regularly. You can download the latest versions of these plug-ins from Verivo. For instructions on downloading and installing the non-core plug-ins, see Downloading and installing non-core plug-ins.

- **Extensible plug-ins:** Several plug-ins can be extended to work with a particular data source. These typically require some customization but do not typically need to be recompiled. For more information on extensible plug-ins, see:
  - Using the WSDL Plug-in
  - Using the REST Plug-in

  For the purposes of upgrading and installing plug-ins, the WSDL and REST plug-ins are considered non-core.

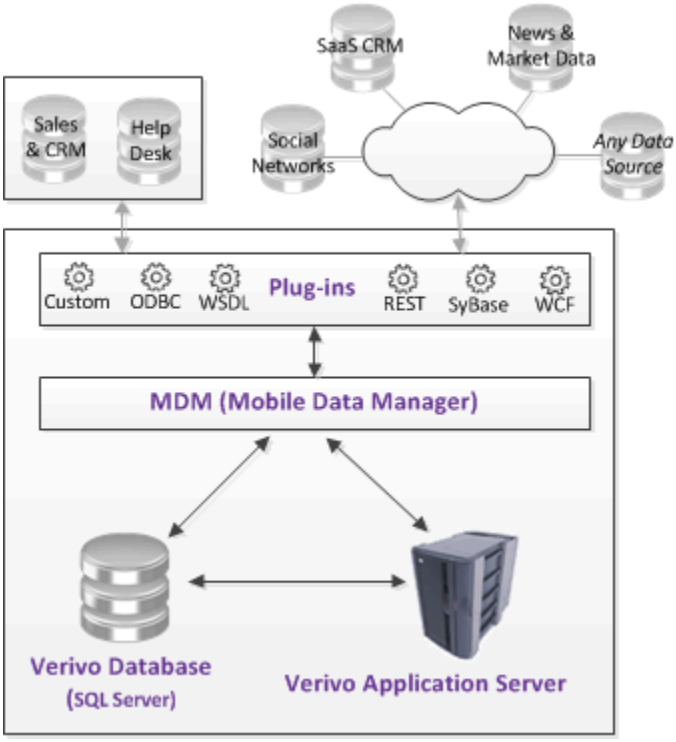
- **Custom:** You can write custom plug-ins that connect to any data source you want, such as a custom Oracle CRM backend. You can use the PAF API to write your own plug-ins: either extend an existing plug-in, customize a sample plug-in, or write a custom plug-in in any .NET language such as Java, C++, and C#. This is for experienced developers only. For more information, see Developing Custom Plug-ins.

You can view a list of currently-installed plug-ins, including generated WSDL plug-ins, in the **Plug-in Manager**. For more information, see Checking Plug-in Status.

Plug-in workflow

Verivo applications query and update back-end data sources through the Mobile Data Manager (MDM) and one or more data source-specific plug-ins. The MDM can provide access to back-end data through any plug-in that is written to work with the Verivo Plug-in Application Framework (PAF). The plug-ins make it possible for MDM to manage queries, updates, and integration of data including enterprise CRM and order management systems, as well as other third-party data sources such as Reuters, Dow Jones, and data warehouses.

The following image shows the relationship among AppServer, MDM, and plug-ins:



When the AppServer receives a request for data from a client, it forwards the request to the MDM. The MDM then determines which plug-in to call, and calls the appropriate method on that plug-in. The plug-in formats the request and connects to the data source. The plug-in then receives a response from the data source and formats the resulting data. It returns the results to the MDM which returns the results to the client.

AppStudio references the plug-in through connection parameters. From AppStudio, you can test connections to back-end data sources and obtain data for Verivo clients.

Downloading and installing non-core plug-ins

Some pre-built plug-ins are not included with the Verivo platform installer. For these non-core plug-ins, you must download and install them on the AppServer and the AppStudio client.

The Plugin.zip file contains the non-core plugins, plus supporting files. The supporting files include several DLLs that are used by more than one plug-in, including:

- DynamicPlugin.dll
- HTTPPlugin.dll

Some plug-ins also include additional files and DLLs. For example, the SalesForce9 plug-in includes a PluginsExclusions.xml file, and the WSDL plug-in includes the Plugins.WSDL.xml file. Each plug-in has its own directory inside the ZIP file, and each directory contains the required files.

To download non-core plug-ins:

1. Log in to the Verivo FTP site:
   ftp://ftp.verivo.com
2. Download the Plugin.zip file.

To install non-core plug-ins on the AppServer:

1. Open the ZIP file in an archiver utility such as WinZIP.
2. Determine the location of the AppServer's "plugins" directory; for example:
   `C:\Inetpub\wwwroot\verivo\bin\plugins`
3. Extract the plug-ins' DLL files and any additional files to the "plugins" directory.
4. Restart the AppServer.

To install non-core plug-ins for AppStudio:

1. Open the ZIP file in an archiver utility such as WinZIP.
2. Determine the location of AppStudio's "Plugins" directory; for example:
   `C:\Program Files\Pyxis Mobile\Application Studio\Plugins`
3. Extract plug-ins' DLL files and any additional files to the "Plugins" directory.
4. Restart AppStudio, if it is already running.
5. Repeat this process for each workstation on which AppStudio is running.

To upgrade and install core plug-ins, you must reinstall the relevant version of the platform.

Connecting Plug-ins to a Data Source

You connect to data sources through Verivo plug-ins. It is through these data sources that Verivo entities—and through the entities, application screens—obtain their data.

> **Note**
>
> The information in this section applies to all plug-ins. For additional information about connection settings or properties that can be set on specific plug-ins, see that plug-in's documentation.

To connect a data source to a plug-in through AppStudio:

1. Click **Data Source** on the main toolbar.
2. At the bottom of the **Data Source Manager**, click **Add**.
3. Set the following properties on the new data source:

Field	Description
Name	Any name you choose to connote the connection's server, database, URL or login settings in AppStudio's **Data Sources** panel; this name will optionally display on the user authorization screen on the mobile application.
Shared	Check this box to configure your data source once for all applications. AppStudio adds the data source to any existing and future applications. Uncheck the **Shared** box to keep the data source within the current application and remove it from all others. Deleting a shared data source removes all mappings to that data source for all applications.
Plug-in Type	Select the plug-in from the drop-down list; this selection determines connection parameters specific for the plug-in. To add a WSDL or WCF plug-in, click **+** (add) next to the drop-down list.
Requires User Authentication	Ensures that device users must be authenticated to access the data source. When not checked, requires login by the AppStudio administrator only, disabling the option for device users to authenticate.
Lockout Minutes	Duration of locked access to the data source after exceeding the maximum number of failed login attempts.
# of Retries	The number of login retries allowed for a successful connection to this data source.
Last Attempt	The date and time of the last login attempt for this plug-in.
Failed Attempts	The number of attempts during the last login for this plug-in. If this number exceeds "# of Retries", then access to the plug-in is locked. You can unlock the plug-in by clicking the Unlock button.

4. Specify the Connection Settings for the plug-in's data source. These settings are defined by the plug-in itself. The required connection settings are in red. The following table describes connection settings that are common to most plug-ins. For information about connection settings that are specific to a single plug-in, see

the documentation for that plug-in.

Field	Description
Base Offset	The timezone setting that the database resides in. This affects timezone handling within the application.
Daylight Saving	Checkbox to indicate if the database adheres to Daylight Savings conventions.
Date Format	Dropdown to indicate what Date Format the database uses.
Time Format	Dropdown to indicate what Time Format the database uses.
Time Stamp Format	Dropdown to indicate what Time Stamp Format the database uses.

Plug-ins that support HTTP protocol, such as WSDL, REST, and WSF, also include a **Proxy Settings** tab under Connection Settings. This tab lets you create a proxy so that you can view the traffic to and from the server. For more information, see Using an HTTP Proxy Server.

For each connection setting, you have the option to select the **Encrypt**, **Global**, or **Prompt User** checkboxes. The following table describes these options:

Option	Description
Encrypt	Encrypts the parameter setting on this screen and, optionally, also on the the mobile application's authorization screen.
Global	Shares this connection parameter setting with other data sources that define the same connection parameter string and also set this parameter as shared. If one of these data sources changes this parameter value, the change is propagated to the other data sources.
Prompt User	Prompts users to supply a value, typically a user name and password. You can set this attribute only if you enable the data source property **Requires User Authentication**.

5. Click the Test Connection button.

   If you successfully connected to the data source, AppStudio displays a Success alert message indicating. If you receive a Failed alert message, click **OK** and check the following:
   - You selected the correct plug-in in the **Plug-in Type** list.
   - You entered your user ID and password precisely.
   - Firewalls are set properly.
   - The user ID you entered has permissions to access the data source.
   - If the data source is a Web service, ensure that you can connect to it through a browser.
6. After you successfully test the data source connection, click **Save**.

AppStudio alerts you if it cannot open a connection with an entity, and when saving settings. If your connection fails at any time, you can reconnect through AppStudio's **Data Sources** panel. You can reconnect by double-clicking the data source from the list.

Mobile users have potential access to the data of third party data sources. AppStudio administrators restrict user access to data through screen workflows.

For more information about connecting specific plug-ins to a data source, see:

- Connecting to a REST-based Data Source
- Connecting to a WSDL-based Data Source

Testing authenticated back-ends

AppStudio maintains a list of end users who can access Verivo applications on their mobile devices or from their desktops. AppStudio administrators must add users of Verivo applications in order to manage application configuration and logging for each individual.

For data sources that require authentication to access, after selecting a user from the bottom of the list on the **User Overview** tab of the **User Manager**, click the **Data Sources** tab and enter a user's connection information. You can set the following values:

- **Source User ID** The data source-specific user id for the selected user accessing that data source.
- **Last Attempt** The reported date and time of the last attempted login of the selected user, successful or unsuccessful. This value cannot be set.
- **Failed Attempts** The reported number of unsuccessful login attempts since the last successful login attempt. This value cannot be set.
- **Unlock** The reset for locked access, when attempts exceeded maximum login attempts set for that data source; sets the number of failed attempts back to 0.

Additional User Access: Your database administrator sets the permissions and access rights to enterprise, third-party, and Web service data sources for this list of users. During the installation process, those users who retrieve and update enterprise, third-party, and Web service data through BlackBerry devices are further configured in the BlackBerry Enterprise Server (BES), as well as optionally, in Active Directory. For more on adding users, see Setting Up Users.

Deleting a data source

You can delete a data source so that your apps can no longer connect to it.

> Deleting a shared data source removes all mappings to that data source for all applications.

To delete a data source:

1. In AppStudio, click Data Sources on the main toolbar. The **Data Source Manager** panel appears.
2. Select the existing data source from the list on the left.
3. At the bottom of the **Data Source Manager**, click the Delete button.
4. Click OK to confirm the deletion of the data source.

Using Plug-ins

This section includes documentation for some of the most common plug-ins for the Verivo platform. The following plug-ins are described:

- Using the REST Plug-in
- Using the WSDL Plug-in
- Using the WCF Plug-in
- Using the SharePoint Plug-in
- Using the Salesforce 9 Plug-in

In addition to the plug-ins described in these sections, the Verivo support site has additional troubleshooting and usage information.

Using the REST Plug-in

Use the REST plug-in to obtain data from a REST service. The plug-in itself requires no customization; you simply create an XML metadata file that describes the HTTP request and the desired REST service data.

The REST plug-in conforms to the structure of the base Plugin class as it is defined in the Plug-in Application Framework (PAF). This structure defines how a plug-in describes itself to AppStudio, and how it interacts with the MDM, a key part of the AppServer. You can use the built-in REST plug-in to perform a wide range of services.

The REST plug-in and other key components—Mobile Data Manager (MDM) and AppStudio—perform the following tasks:

1. Execute HTTP GET or POST requests to the REST service.
2. Render the returned data into tabular (.NET DataTable) data.
3. Populate entity fields in an application with the refactored data.

To use the REST plug-in:

1. Install the REST plug-in.
2. Create a metadata file.
3. Connect to a REST-based data source in AppStudio.
4. Pass data to the REST data source.

Installing the REST Plug-in

The REST plug-in is a core plug-in that consists of the following DLL files:

Required Plug-in	Description
RESTPlugin.dll	The REST plug-in.
HTTPPlugin.dll	The REST plug-in implements the HTTP plug-in, which inherits from the abstract Plugin class. The Plugin class is defined in the PAF; it requires all child classes, including the HTTP plug-in, to define methods that describe source objects and fields to AppStudio.
HtmlAgilityPack.dll	The REST plug-in uses this third-party library to scrape screen data and convert HTML content into valid XML.
Newtonsoft.Json.Net20.dll	Converts JSON content into valid XML.

The REST plug-in DLL files are considered part of the "core" plug-in architecture, which means they are included in the AppServer and AppStudio installations. You do not need to download and install them separately.

In addition to the DLLs listed above, the REST plug-in also uses the DynamicPlugin.DLL. This is a required DLL for most plug-ins.

Creating a Metadata File

You must define the REST service to AppStudio and AppServer. This file describes both the request from the plug-in to the REST service and the response from the REST service to the plug-in. The REST service plug-in depends on this metadata file to perform the following tasks:

- Describes the request to the REST service.
- Describes the response to AppStudio and the AppServer.

Before writing the metadata file, you must become familiar with your REST service's data format.

Analyzing the REST service data

The first step in defining a metadata file is to analyze your REST service to see the structure of the data it returns. The easiest way to do this is to request the service in a browser and view the returned data. In most cases, the returned data will be in XML or JSON format.

For example, if you make the following request:

`http://www.boardgamegeek.com/xmlapi/boardgame/42`

You get a result in XML, shown here in truncated format:

XML-formatted REST service

```
<boardgames termsofuse="http://boardgamegeek.com/xmlapi/termsofuse">
  <boardgame objectid="42">
    <yearpublished>1997</yearpublished>
    <minplayers>2</minplayers>
    <maxplayers>4</maxplayers>
    <playingtime>90</playingtime>
    <age>12</age>
    <name primary="true" sortindex="1">Tigris & Euphrates</name>
    <name sortindex="1">Tigre et Euphrat</name>
    <description>Regarded by many as Reiner Knizia's masterpiece, the game is set in the ancient
fertile crescent with players building civilizations through tile placement.</description>
    <thumbnail>http://cf.geekdo-images.com/images/pic168169_t.jpg</thumbnail>
    <image>http://cf.geekdo-images.com/images/pic168169.jpg</image>
    <boardgamecategory objectid="1032">Territory Building</boardgamecategory>
    <boardgamecategory objectid="1002">Civilization</boardgamecategory>
  </boardgame>
</boardgames>
```

A JSON-formatted response might look like the following:

JSON-formatted REST service

```
{boardgames:{
  termsofuse:'http://boardgamegeek.com/xmlapi/termsofuse',
  boardgame:{
   objectid:42,
   yearpublished:1997,
   minplayers:2,
   maxplayers:4,
   playingtime:90,
   age:12,
   name:[
    {value:'Tigris & Euphrates', primary:'true', sortindex:1},
    {value:'Tigre et Euphrat', sortindex:1}
   ],
   description:'Regarded by many as Reiner Knizia\'s masterpiece...',
   thumbnail:'http://cf.geekdo-images.com/images/pic168169_t.jpg',
   image:'http://cf.geekdo-images.com/images/pic168169.jpg',
   boardgamecategory:[
    {objectid:1032, value:'Territory Building'},
    {objectid:1002, value:'Civilization'}
   ]
  } // boardgame
 } // boardgames
}
```

To view other examples of the same services represented in XML and JSON, see http://json.org/example.html.

Before writing the REST plug-in metadata file, identify the following elements within the REST service output:

- The most basic unit of repeated information that can be regarded as a record or row. To reference this unit, you use the *basexpath*. In this example, the *basexpath* is the data inside the pair of `<boardgame></boardgame>` tags (XML) or in the boardgame object (JSON). In many cases, there will be multiple elements that match the basexpath. In some cases,
- The data that is required by the application. These are fields that you will map to your entity in AppStudio. For example, your application might require the following elements:
  - `name` (where primary="true")
  - `yearpublished`
  - `description`
  - `thumbnail`

Writing the metadata file

The metadata file describes a REST service to AppStudio and the plug-in. The file must conform to the schema definition. The elements you specify can differ, depending on whether the response is in XML or JSON format.

Sample metadata file

The following metadata file describes the request and response for the sample REST service (XML):

Sample metadata file

```
<?xml version="1.0" encoding="utf-8"?>
<PluginDataStore>
  <SourceObject>
    <name>GetGameDetails</name>
    <url><![CDATA[http://www.boardgamegeek.com/xmlapi/boardgame/#gameID#]]></url>
    <basexpath>descendant::boardgames/boardgame</basexpath>

    <!— Input fields: -->
    <Field name="gameID" datatype="System.String" required="true"/>

    <!— Output fields: -->
    <Field name="name" datatype="System.String" fieldxpath="name[@primary='true']"/>
    <Field name="yearpublished" datatype="System.String" fieldxpath="yearpublished"/>
    <Field name="description" datatype="System.String" fieldxpath="description"/>
    <Field name="thumbnail" datatype="System.String" fieldxpath="thumbnail"/>
  </SourceObject>
</PluginDataStore>
```

For JSON-formatted services, you typically specify

Required elements

A valid metadata file must use the `<PluginDataStore>` as its root element and one or more `<SourceObject>` c hild elements.

Each `<SourceObject>` element defines one operation in the RESTful service. The `<SourceObject>` element must define the following children:

Child Element	Description	Input/Output
`<name>`	The name of the data source. This is used by AppStudio when you add a new data source to your app.	Input
`<url>`	The location of the REST service. AppServer constructs the URL based on the base URL and sometimes input fields (if they are passed in as part of the URL).	Input
`<basexpath>`	The basic element of the response; roughly corresponds to a row in a database.	Output
`<Field>`	Data that is either passed to the service or received from the service. Input fields are passed to the REST service. Ouput fields map the response data to entities within the app.	Input/Output

The typical structure of a metadata file for an XML service looks like the following:

Metadata file for XML data

```
<PluginDataStore>
  <SourceObject>
    <name>name_here</name>
    <url>url_here</url>
    <basexpath>basexpath_here</basexpath>
    <Field name='field_name_here' datatype='datatype_here' fieldxpath='fieldxpath_here'/>
  </SourceObject>
</PluginDataStore>
```

In a metadata file that describes a JSON service, you typically add the `<originalformat>`, `<converttoxml>`, and `<requestContentType>` elements, as the following structure shows:

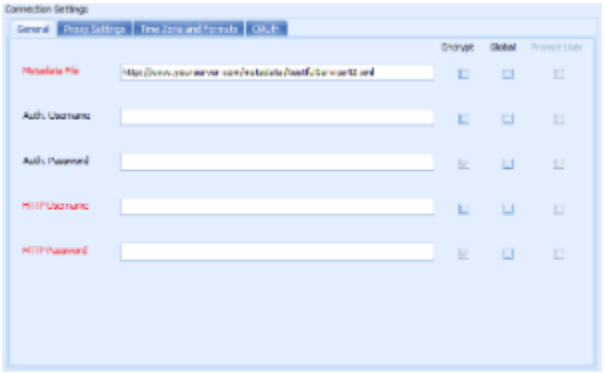
Metadata file for JSON data

```
<?xml version="1.0" encoding="utf-8"?>
<PluginDataStore>
  <SourceObject>
    <name>name_here</name>
    <url>url_here</url>
    <originalformat>JSON</originalformat>
    <converttoxml>true</converttoxml>
    <requestContentType>application/json</requestContentType>
    <basexpath>basexpath_here</basexpath>
    <Field name='field_name_here' datatype='datatype_here'/>
  </SourceObject>
</PluginDataStore>
```

For a complete list of all elements that you can use in the metadata file, see Metadata Schema Definition.

Metadata file location

After you create the metadata file, save it in a location that is accessible via HTTP from the AppServer and all the machines running AppStudio. You define its location to the server and AppStudio in the Metadata File option of the data source's Connection Settings. For example:



You can also store copies of the metadata file in local directories of both the AppStudio machines and the server. For AppStudio, this is the /Application Studio/Plugins directory. For AppServer, this is in the /*app_name*/bin/plugins directory. If you use multiple copies of this file, they must be kept in sync.

If you make a change to the metadata file, you must refresh it in AppStudio to recognize the changes. For more

information, see Updating the metadata file.

Specifying multiple SourceObjects

You can use a single metadata file to perform multiple operations against your RESTful service. To do this, add a second <SourceObject>. Each operation can have a separate URL, basexpath, and input/output fields specified within its own <SourceObject> element.

The following sample metadata file shows two operations, a search and a simple request for the details of a single game that matches the gameID:

Sample metadata file

```
<?xml version="1.0" encoding="utf-8"?>
<PluginDataStore>
  <SourceObject>
    <name>GetGameDetails</name>
    <url><![CDATA[http://www.boardgamegeek.com/xmlapi/boardgame/#gameID#]]></url>
    <basexpath>descendant::boardgames/boardgame</basexpath>
    <!— Input fields: -->
    <Field name="gameID" datatype="System.String" required="true"/>
    <!— Output fields: -->
    <Field name="name" datatype="System.String" fieldxpath="name[@primary='true']"/>
    <Field name="yearpublished" datatype="System.String" fieldxpath="yearpublished"/>
    <Field name="description" datatype="System.String" fieldxpath="description"/>
    <Field name="thumbnail" datatype="System.String" fieldxpath="thumbnail"/>
  </SourceObject>
  <SourceObject>
    <name>SearchBoardgames</name>
    <url><![CDATA[http://www.boardgamegeek.com/xmlapi/search?]]></url>
    <basexpath>descendant::boardgames/boardgame</basexpath>
    <!— Input fields: -->
    <Field name="search" datatype="System.String" required="true" sendinquerystring="true"/>
    <!-- Output fields: -->
    <Field name="name" fieldxpath="name" datatype="System.String"/>
    <Field name="gameID" fieldxpath="@objectid" datatype="System.String"/>
  </SourceObject>
</PluginDataStore>
```

Describing the request elements

HTTP requests to a RESTful service are constructed from the following input elements, which are defined in the metadata file:

- <url> element
- <Field> input elements

<url> element

The <url> element in the metadata file defines the structure of the HTTP request URL. It can contain optional values within hash tags (#) that define user-supplied data and static variables. The value of a typical <url> element looks like the following:

```
<url><![CDATA[http://www.boardgamegeek.com/xmlapi/boardgame/#gameID#]]></url>
```

The URL is wrapped within a CDATA tag to ensure that it is parsed correctly. This example URL includes the variable #gameID#; when the request URL is constructed, this variable resolves to the value of the gameID input field, which is set by the application as described in Input field elements.

Do not include any spaces, tabs, or other characters within the CDATA block that defines the URL. These characters might be interpreted as part of the request.

<Field> input elements

A metadata file can define <Field> elements that obtain values from a Verivo application and pass them to the REST service. These are known as *input fields*. Input fields are used by the REST service to determine what kinds of operation to take or which data to return.

> To define an input field, you set the required attribute to true on the <Field> element. If you do not set this attribute to true, then the field is considered an *output field*.

The metadata file above defines one input field, gameID:

```
<Field name="gameID" datatype="System.String" required="true"/>
```

The value for the gameID field is obtained from the screen view in the application and is referenced by the #gameID # variable that is embedded in the <url> element. Before sending a request, the application replaces the #gameID # variable in the URL with the appropriate value.

For additional information about setting the values of input fields, including how to specify them in AppStudio, see Passing Data to a REST Service.

Describing the response elements

The metadata file specifies how to render data that is returned by the RESTful service into tabular data for the Verivo application, through two definition types:

- A single <basexpath> element that specifies which repeatable elements within the response are atomic values for displaying in tabular format.
- One or more <Field> elements whose fieldxpath attribute maps REST data elements to entity fields in your app.

<basexpath> element

The <basexpath> element identifies the XML element that should be parsed as a recurring row/record in the rendered table data. When specifying the <basexpath> element, you typically prefix the value with "descendant::". This instructs the plug-in to get all children of the specified node. If there is no top-level node, then you can leave this element blank.

The basexpath element must be set to the XML element's fully qualified path—in this example, as follows:

```
<basexpath>descendant::boardgames/boardgame</basexpath>
```

This setting identifies each <boardgame> element as a unique record within the rendered table data.

<Field> output elements

You use the <Field> output element to describe the response data to the plug-in. This lets the application map response data to entity fields so that it can display the data in tabular format on a screen.

> To define an output field, you set the `required` attribute to `false` on the <Field> element (or do not set the value at all). If you set this attribute to `true`, then the field is considered an *input field*.

The sample metadata file defines the following <Field> elements that map to entity fields:

```
<!– Output fields: -->
<Field name="name" datatype="System.String" fieldxpath="name[@primary='true']"/>
<Field name="year_published" datatype="System.String" fieldxpath="yearpublished"/>
<Field name="description" datatype="System.String" fieldxpath="description"/>
<Field name="thumbnail" datatype="System.String" fieldxpath="thumbnail"/>
```

Each <Field> element's `name` attribute identifies an entity field and maps it to an element in the data that is returned by the REST service. The `datatype` attribute specifies the data type for each element returned by the service. The `fieldxpath` attribute specifies the location, relative to the `basexpath`, of the value for each entity field. In some cases, the `fieldxpath` attribute can be blank or omitted.

In this example, the <Field> elements map the following entity fields to their corresponding REST elements:

Entity Field		REST Element
`name`	>	`name` (where the `primary` attribute is set to `true`)
`year_published`	>	`yearpublished`
`description`	>	`description`
`thumbnail`	>	`thumbnail`

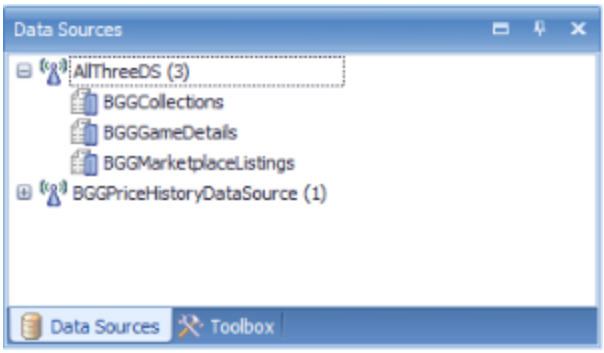
You use XPath syntax to reference XML elements, values, and attributes in the `fieldxpath` attribute of the <Field> element. When matching values in the XML output with XPath, use the following general rules:

- To refer to the root node, use "/".
- To refer to the current node, use "."
- To refer to the parent of the current node, use ".."
- To refer to attributes of an element, use the "@" sign. For example: `@objectid`
- To refer to nested elements, use dot-notation. For example: `nestedelement1.nestedelement2`
- To match values of elements and attributes, use brackets ("[]"). For example, to match all <name> elements that have a `primary` attribute with the value `true`, use: `name[@primary='true']`

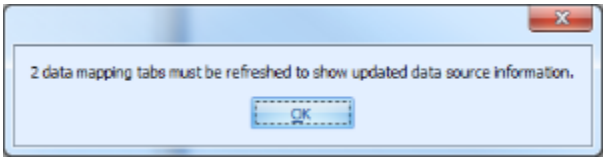
For a tutorial on XPath, see http://www.w3schools.com/xpath/.

Updating the metadata file

If you add, remove, or modify one or more of the input or output fields, you must refresh the data source in AppStudio before you can map those fields. To do this, right-click the data source in the Data Sources panel and click Refresh:



You might also need to refresh the data mapping tabs to use the new fields:



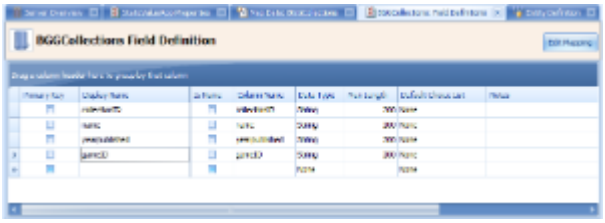
To refresh the data mapping tabs, close the tabs and re-open them.

You can also refresh the tabs and the data source by closing AppStudio and re-opening it.

If these techniques do not cause the new field to show up in the entity definition, then you must manually map it.

To manually map a new field to an entity definition:

1. Double-click the data entity in the Entity Definition screen. The Data Mapping screen appears.
2. Click the Edit Fields button. The Field Definition screen appears:



3. Add the new field by defining the Display Name, Column Name, Data Type, and Max Length. You can optionally specify if the new field is the primary key and if it uses the default choice list, plus add any notes.
4. Save your application.

Metadata Schema Definition

The metadata files that are used by AppStudio and AppServer for the REST plug-in must conform to the following schema definition:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="PluginDataStore" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="PluginDataStore" msdata:IsDataSet="true" msdata:UseCurrentLocale="true">
    <xs:complexType>
      <xs:choice minOccurs="1" maxOccurs="unbounded">
        <xs:element name="SourceObject">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name" type="xs:string" minOccurs="0" maxOccurs="1"/>
              <xs:element name="url" type="xs:string" minOccurs="0" maxOccurs="1"/>
              <xs:element name="useragent" type="xs:string" minOccurs="0" maxOccurs="1"/>
              <xs:element name="usemultipartpost" type="xs:boolean" minOccurs="0" maxOccurs="1"/>
              <xs:element name="skipreplacemap" type="xs:boolean" minOccurs="0" maxOccurs="1"/>
              <xs:element name="binaryonly" type="xs:boolean" minOccurs="0" maxOccurs="1"/>
              <xs:element name="baseregex" type="xs:string" minOccurs="0" maxOccurs="1"/>
              <xs:element name="htmldecode" type="xs:boolean" minOccurs="0" maxOccurs="1"/>
              <xs:element name="forceatleastonerow" type="xs:boolean" minOccurs="0" maxOccurs="1"/>
              <xs:element name="converttoxml" type="xs:boolean" minOccurs="0" maxOccurs="1"/>
              <xs:element name="originalformat" type="xs:string" minOccurs="0" maxOccurs="1"/>
              <xs:element name="baseurlforbinary" type="xs:string" minOccurs="0" maxOccurs="1"/>
              <xs:element name="httpmethod" type="xs:string" minOccurs="0" maxOccurs="1"/>
              <xs:element name="stripxmlns" type="xs:boolean" minOccurs="0" maxOccurs="1"/>
              <xs:element name="basexpath" type="xs:string" minOccurs="0" maxOccurs="1"/>
              <xs:element name="requesttemplate" type="xs:string" minOccurs="0" maxOccurs="1"/>
              <xs:element name="requestContentType" type="xs:string" minOccurs="0" maxOccurs="1"/>
              <xs:element name="Field" minOccurs="1" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:attribute name="name" type="xs:string"/>
                  <xs:attribute name="datatype" type="xs:string"/>
                  <xs:attribute name="required" type="xs:boolean"/>
                  <xs:attribute name="sendinpostheader" type="xs:boolean"/>
                  <xs:attribute name="sendinquerystring" type="xs:boolean"/>
                  <xs:attribute name="sendinhttpheader" type="xs:boolean"/>
                  <xs:attribute name="fieldxpath" type="xs:string"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="replacemap">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="replaceitem" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="find" type="xs:string" minOccurs="1"/>
                    <xs:element name="replace" type="xs:string" minOccurs="1"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The root node <PluginDataStore> identifies this metadata file to the REST plug-in. It requires at least one named <SourceObject> element, which identifies an object used by AppStudio to represent a child of the REST data source. Each <SourceObject> typically corresponds to a single web service call and minimally contains the following elements:

- name
- url
- basexpath
- One or more Field elements

SourceObject sub-elements

The following elements are children of the <SourceObject> element:

Option	Description
name	(Required) The name of the source object. This is how the source object will appear in the object table in AppStudio. For example, BGGSearch.
url	(Required) The URL of the HTTP request to the web service. Because URLs allow special characters that can disrupt well-formed XML, wrap the URL in a CDATA block; for example: <url><![CDATA[http://www.boardgamegeek.com/xmlapi/boardgame/#gameID#]]></url> Do not include any spaces between the <url> element and the <CDATA> element, or between the <CDATA> element and its value.
httpmethod	The method used in the HTTP request, either GET or POST. If omitted, the default method is GET.
basexpath	(Required) The XML element to be parsed as a recurring data row/record in the rendered data. The XML parser uses this tag as a starting point to identify fields within the SourceObject. The syntax conforms to XPath conventions. For more information about XPath, see http://www.w3.org/TR/xpath.
Field	(Required) An XML element that is parsed as an input or output data field: • *Input fields* typically obtain data from the Verivo application and are used to construct URL elements. To enforce data input for a field, set the field's required attribute to true. • *Output fields* map data from the service to entity fields in the Verivo application. Each output field maps the source XML to a column in the rendered table through the fieldxpath attribute. Each SourceObject must contain at least one Field element. For each defined Field element, AppStudio displays a field within the SourceObject. Each Field element contains one or more required and optional attributes as described in Field attributes.

originalformat	Specifies the format of data returned from the web service; one of the following: • XML (default) • JSON • HTML
converttoxml	Specifies whether to convert data returned from the web service into XML. Set to true or false. The default value is false. For a JSON service, you typically set the originalformat element to "JSON", then set converttoxml to true. This lets you use XPath expressions to traverse the results as you would with an XML service.
baseregex	A regular expression that indicates which portion of the returned data represents records, used only for HTML or other data formats where basexpath does not apply.
joinsRequireLiterals	An optional element. Set to true or false. The default value is false.
stripxmlns	Specifies whether to strip XML name space information from data that is returned from the web service. Set to true or false. The default value is false. Setting this to true can reduce server load as it results in less information being transferred during the request.
stripnewlines	Specifies whether to strip new line characters from data that is returned from the web service. Set to true or false. The default value is false.

Field attributes

The following attributes can be set on a <Field> element.

Option	Description
name	(Required) The name of the field.
datatype	(Required) The type of data contained in the field. For example: System.String or System.DateTime.
fieldxpath	The path to web service data. Each field is mapped to a column in the rendered table that is displayed in AppStudio. You can omit this attribute for JSON services where the object properties are at the top level.
sendinquerystring	Specifies whether to include this field as a query string parameter in the HTTP request; valid only when the parent SourceObject's httpmethod element is set to GET. Set to true or false. The default value is false.
sendinpostheader	Specifies whether to include the field as a parameter in the HTTP request; valid only when the parent SourceObject's httpmethod element is set to POST. Set to true or false. The default value is false.

`required`	Specifies whether the field is an input field or not. Set to true to pass the value in with the request; otherwise, set to false. The default value is false. All fields are considered output fields unless this attribute is set to true.

Connecting to a REST-based Data Source

To add the REST service data to Verivo applications:

1. Install the REST plug-in and create a metadata file, as described in Installing the REST Plug-in and Creating a Metadata File.
2. Add the REST plug-in as a data source in AppStudio.
3. Create entities from the REST-based data source.

Add the REST plug-in as a data source

To add the REST service as a data source in AppStudio:

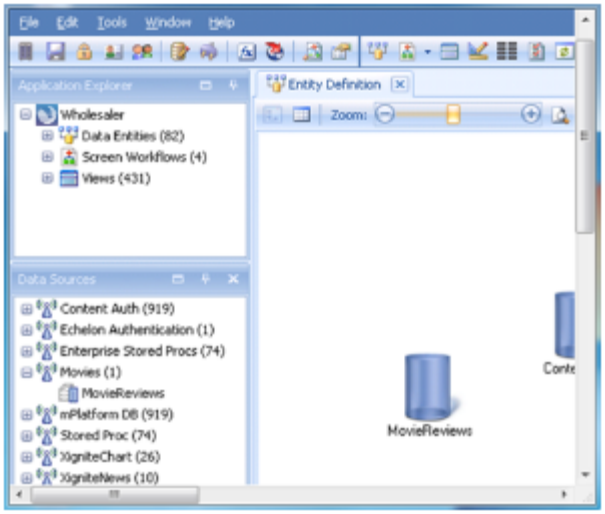
1. Open the **Data Source Manager**.
2. To add a new data source, click **Add.**
3. Configure the new data source with the following properties:
   - **Name**: The name you assign to this data source.
   - **Plug-In Type**: REST Plugin. If the REST plugin is not in the list of available plugins, be sure you copied all the DLLs to the AppStudio plugins directory and that you restarted AppStudio, as described in Check the Verivo installation for required files.
   - Under Connection Settings, enter the name of the **Metadata File** that is used to access the REST service. The location of this file is relative to the AppStudio /Plugins directory. For example, if the MyRestService.xml file is in the /Plugins directory, enter MyRestService.xml.
     You can also enter a network-accessible location. For example: `http://myserver.com/rest/MyRestService.xml`
4. Validate your settings by clicking **Test Connection**. Clicking **Test Connection** does not actually test the service, but it does determine if all required fields have been entered.

After adding the new data source, you generally map its output fields to fields in the entities that your app uses. For more information, see Create entities from the REST-based data source.

Create entities and screens from the REST-based data source

You can quickly create entities and their associated screens in AppStudio from a REST-based data source as follows:

1. In AppStudio, open the **Entity Definition** window.
2. From the **Data Sources** list, expand the REST data source.
3. Select the data source—in this example, MovieReviews—and drag it to the **Entity Definition** window:



4. Double-click the MovieReviews entity in the **Entity Definition** panel. The **Data Mapping** screen appears. The fields should automatically be mapped: the entity's **Data Fields** map to the source object's **Source Fields**. If they are not, select a **Source Object** and **Source Field** (on the right) for each **Data Field** (on the left).
5. To create List, Find, and Summary screens for your new entity, right-click on the entity in the **Entity Definition** panel and select **Create Screens For > Default**.
6. Save the application.

Passing Data to a REST Service

REST-based services often key off of the HTTP request type (such as a GET, POST, or DELETE) to determine what operation to perform. But in many cases, the service also requires some input from your application, either in the form of a query string parameter in the URL, or as part of the URL itself. Common operations include:

- Getting a list of data from the service. This is typically performed by specifying optional parameters, such as a date range, in a GET request.
- Deleting a particular record. In some cases, you send a DELETE HTTP request with a parameter such as an ID to indicate which record to delete.
- Updating records with an HTTP POST request.
- Adding new records with an HTTP PUT request.
- Checking a license key. Some services require a static value such as a license key.
- Searching for records. Some services perform a search on the server side, and you need to pass the search terms to the service.

This section describes how to pass fixed values as well as variables from your application to the REST-based service.

> The instructions in this section change the requests that you send to a REST service. When working with request/response data, it is very helpful to use a proxy server so you can see the headers and request URLs. For more information, see Using an HTTP Proxy Server.

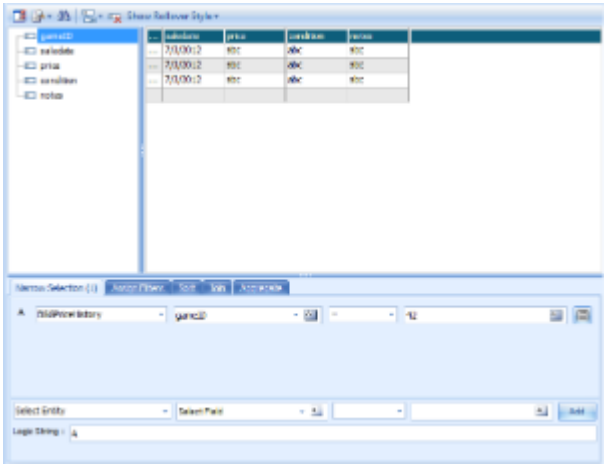
For additional information on connecting to data sources and displaying data in an app, see the Connect to Data tutorial.

Passing static values to a REST service

In many cases, REST services require a static value, such as a license key, from the client. You can either pass a value as part of the URL or as a query string parameter. In most cases, static values such as license keys will be passed as query string parameters in GET requests.

To pass a static value to a REST service:

1. Edit the List or Summary screen for your data entity.
2. Select the Narrow Selection tab.
3. Add a new item to the list:
   - **Select Entity:** Select the entity that is mapped to the REST service.
   - **Select Field:** Select the field that the application will pass to the REST service.
   - **Operator:** Select "=".
   - **Value:** Enter the data that you want to pass to the REST service.
4. Click the Add button to add the new Narrow Selection:



5. Save your application.
6. Open the REST service's metadata file.
7. Edit the `<url>` element to ensure that the URL ends with a "?" or a "&". For example:

   ```
   <url><![CDATA[http://www.boardgamegeek.com/xmlapi/boardgame?]]></url>
   ```

   If there are already one or more query string parameters, add a "&" to the end of the query string. For example:

   ```
   <url><![CDATA[http://www.boardgamegeek.com/xmlapi/boardgame?name=fred&]]></url>
   ```
8. Add a new input `<Field>` element.
   a. Set its `name` and `datatype` attributes. The `name` is the field name in the entity. For input fields, the `datatype` is usually System.String.
   b. Set the `required` attribute to `true`.
   c. Set the `sendinquerystring` property to `true`. For example:

      ```
      <Field name="gameID" datatype="System.String" required="true" sendinquerystring="true"/>
      ```

      At runtime, the application appends the Narrow Selection as a query string parameter; for example:

      ```
      http://www.boardgamegeek.com/xmlapi/boardgame?gameID=42
      ```
9. Save the metadata file. If you have separate metadata files, be sure to save the changes in both AppStudio's and AppServer's plugins directories.
10. Reload your app's config file before using the new setting.

Send static values as part of the URL

You can also add a static value to the URL by adding the field to the <url> element, as the following example shows:

```
<url><![CDATA[http://www.boardgamegeek.com/xmlapi/boardgame/#gameID#]]></url>
```

In this case, **do not** set the value of the <Field> element's sendinquerystring attribute to true. Setting this attribute would append the value to the URL as a query string parameter. Instead, you want it to be part of the URL. You can set it to false, or omit it.

The resulting GET request looks like the following:

```
http://www.boardgamegeek.com/xmlapi/boardgame/42
```

Send static values as POST headers

To send a static value as a header in a POST request, set the value of the <Field> element's sendinpostheader attribute to true, as the following example shows:

```
<Field name="gameID" datatype="System.String" required="true" sendinpostheader="true" />
```

In this case, you do not need to add a "?" to the end of the URL because no query string parameters are sent when the request in a POST request.

You must also set the <httpmethod> element to POST:

```
<httpmethod>POST</httpmethod>
```

The resulting POST request looks like the following:

```
POST http://www.boardgamegeek.com/xmlapi/boardgame HTTP/1.1
User-Agent: Mozilla/4.0 (...)
Content-Type: text/xml
Host: www.boardgamegeek.com
Content-Length: 10
Connection: Keep-Alive

gameID=42&
```

Passing variable data to a REST service

For REST services, you commonly pass variable by changing the URL string. A service might let you access different types of content, depending on the URL path. A service might also take additional parameters, either as part of the URL or as query string parameters.

The following examples show different URLs. The first accesses the "mkgray" collection. The second accesses a boardgame with the ID 42 and passes an additional parameter to instruct the service to return different information. The third executes a search on the server side:

```
http://www.boardgamegeek.com/xmlapi/collection/mkgray
http://www.boardgamegeek.com/xmlapi/boardgame/42?pricehistory=1
http://www.boardgamegeek.com/xmlapi/search?search=knizia
```

If the REST API accepts GET requests in this way, you would typically set up three separate SourceObjects in your metadata file. Each one corresponding to a different type of request. A List screen with multiple games might pass a variable value for the `gameID` so that the user can navigate to a Summary screen for an individual game. The optional parameter, `pricehistory`, would be added in the SourceObject. A Find screen might let a user select a `collectionID` from a Choice Field control (similar to an HTML drop-down box). Another Find screen might ask the user to input a search term which is executed on the server.

The variables in these requests are the `gameID` ("42"), `collectionID` ("mkgray"), and search term ("knizia"). You define variable data in AppStudio, and map it to the input parameters defined in the metadata file. When the request is "built" by the plug-in, the app replaces the variables with actual values.

The following example metadata file defines multiple SourceObjects. Each SourceObject has its own URL and `basexpath`, as well as input and output fields:

SampleMetadataFile.xml

```
<?xml version="1.0" encoding="utf-8"?>
<PluginDataStore>
  <!-- Collections -->
  <SourceObject>
    <name>BGGCollections</name>
    <url><![CDATA[http://www.boardgamegeek.com/xmlapi/collections/#collectionID#]]></url>
    <basexpath>descendant::items/item</basexpath>
    <!-- Input field: -->
    <Field name="collectionID" datatype="System.String" required="true"/>
    <!-- Output fields: -->
    <Field name="gameID" fieldxpath="@objectid" datatype="System.String"/>
    <Field name="name" fieldxpath="name" datatype="System.String"/>
    <Field name="yearpublished" fieldxpath="yearpublished" datatype="System.String"/>
  </SourceObject>
  <!-- Game Search -->
  <SourceObject>
    <name>BGGSearch</name>
    <url><![CDATA[http://www.boardgamegeek.com/xmlapi/search?]]></url>
    <basexpath>descendant::boardgames/boardgame</basexpath>
    <!-- Input field: -->
    <Field name="search" datatype="System.String" required="false" sendinquerystring="true"/>
    <!-- Output fields: -->
    <Field name="name" fieldxpath="name" datatype="System.String"/>
    <Field name="gameID" fieldxpath="@objectid" datatype="System.String"/>
  </SourceObject>
  <!-- Marketplace Listings -->
  <SourceObject>
    <name>BGGMarketplaceListings</name>
    <url><![CDATA[http://www.boardgamegeek.com/xmlapi/boardgame/#gameID#?pricehistory=1]]></url>
    <basexpath>descendant::boardgames/boardgame/marketplacehistory/listing</basexpath>
    <!-- Input field: -->
    <Field name="gameID" datatype="System.String" required="true"/>
    <!-- Output fields: -->
    <Field name="condition" datatype="System.String"/>
    <Field name="saledate" datatype="System.DateTime"/>
    <Field name="price" datatype="System.String"/>
    <Field name="notes" datatype="System.String"/>
  </SourceObject>
</PluginDataStore>
```

The examples in this section use this metadata file. You can use this same approach for your own RESTful services.

If you need additional information about adding data sources, creating entities, or creating default screens for entities, see Connect to Data.

Passing variable data to a REST service with a Choice Field

You can create a Choice Field that passes the selected value to a REST service for processing. You do this by mapping the Choice List to the data source.

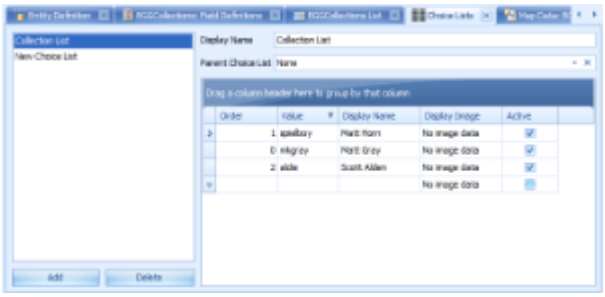
Prerequisite

Before you can complete this procedure, you must perform the following steps:

1. Add a new data source that points to your metadata file.
2. Create a new entity in the **Entity Definition** window.
3. Create **Find**, **List**, and **Summary** screens and views for this entity.

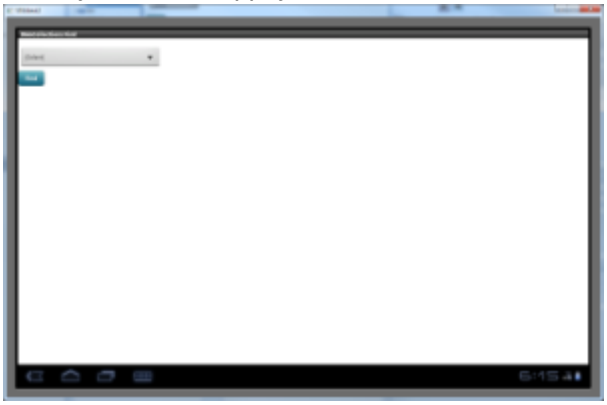
To create a choice field that passes variable data to a REST service:

1. Go to the Choice Lists screen in AppStudio and create a new Choice List with static data; for example:

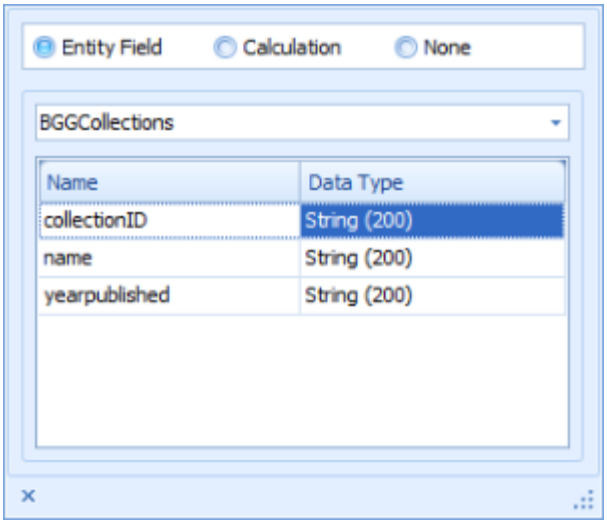


   You can also use dynamic data. For more information on creating choice lists, see Choice List Controls.
2. Add a Choice Field control to the **Find** screen. For the List Source, select the Choice List you created in the previous step.
3. Add a Button control. This button will trigger the REST service request and navigate to the List screen that displays the results.
   a. Select "Find" for the **Action**.
   b. Select your entity's **List** screen for the **Target Screen**.
   c. Select your entity's **List** screen for the **Target View**.

   When you run the app, your screen should look similar to this:



4. Return to the **Find** screen in AppStudio and select the Choice Field.
5. Map the entity field to the Choice Field.
   a. Click the field next to Data. The data mapping popup appears:



   b. Select the **Entity Field** option.
   c. Select the entity that you want to map to your Choice Field (in this case, BGGCollections).
   d. Select the value that you want to map to the Choice Field (in this case, `collectionID`).
   e. Click X to close the popup screen.
6. In the metadata file, modify the `<url>` element to include the input field. Use the BGGCollections SourceObject in the SampleMetadataFile.xml example above as a guide.
7. Save the app.
8. Run the app, select an item from the choice list, and click the Find button. If you use a proxy tool such as Fiddler, you should see a URL request that looks like the following:
   `www.boardgamegeek.com/collection/aldie`

   The plug-in replaces `#collectionID#` in the URL with the *value* of the selected item (not the Display Name) in the Choice Field. For more information on using an HTTP proxy tool, see Using an HTTP Proxy Server.

If the URL does not include a value after /collection/, be sure you mapped the correct entity field to the Choice List. Also, be sure that its name (in this case, `#collectionID#`) matches the name that is used in the SourceObject's `<url>` element.

If you get the error **"An item with the same key has already been added"**, you likely have a redundant Narrow Selection defined. When you map an entity field to a Choice Field, you should not add a Narrow Selection. Remove the Narrow Selection from the **List** screen, save the app, and run it again.

Passing List item data to a REST service

A common use of a **List** screen is to select an item and then navigate to a another **List** screen rather than a **Summary** screen. The second **List** screen might show more information about the item that was requested from the server and then displayed. You can do this with any data source, including REST services.

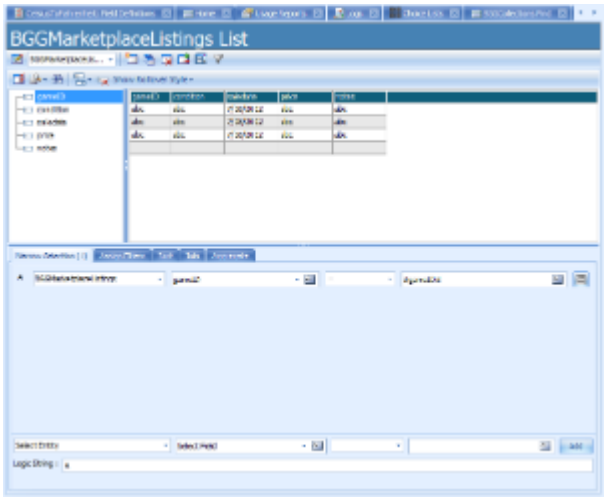
Prerequisite

Before you can complete this procedure, you must perform the following steps:

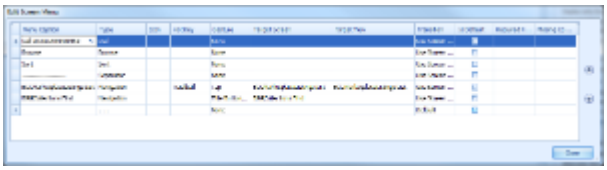
1. Add two data sources that point to your metadata file. Each **List** screen requires its own data source.
2. Create two new entities in the **Entity Definition** window.
3. Create **List** screens and views for each entity.

To create a List screen that passes data to a REST service:

1. On the *first* **List** screen, set the entity field to be a Filter Field. You do this by opening the List screen in the editor and clicking Filter Fields. In the Assign Filter Field dialog, move the field from the left side to the right. For example, move the `gameID` field from the left to the right.
2. On the *target* **List** screen, add the filter field as a Narrow Selection:
    a. Open the *target* **List** screen.
    b. Add a new item to the list:
        - **Select Entity:** Select the entity that is mapped to the REST service.
        - **Select Field:** Select the field that the application will pass to the REST service.
        - **Operator:** Select "=".
        - **Value:** Enter the field, surrounded by hashmarks (#), that you want to pass to the REST service. For example, `#gameID#`.
    c. Click the Add button to add the new Narrow Selection:



3. Change the target screen for the first **List** screen's **Tap** gesture. To do this:
    a. On the first **List** screen, click Edit Menu. The Edit Screen Menu popup appears:



    b. Find the **List** screen that you want to change the behavior for. The default target is a **Summary** screen .
    c. Change the **Target Screen** value to the *second* **List** screen.
    d. Click the Close button.
4. Save your app.
5. In the metadata file, modify the `<url>` element to include the input field. Use the BGGMarketPlaceListings SourceObject in the SampleMetadataFile.xml example above as a guide.
6. Run your app. Be sure to reload the config.

Passing user input to a REST service

A common use of a REST service is to pass a user-entered string that is evaluated on the server side. The service then returns results based on that input. The most common scenario when this happens is for searching.

To create a search input field in AppStudio, use a Text field on a **Find** screen. In this case, using a **Find** screen

makes sense based on its name, but this is also the kind of screen you use for passing just about any user-defined value to a REST service. The service typically reads the value and performs some kind of lookup.

> **Prerequisite**
>
> Before you can complete this procedure, you must perform the following steps:
>
> 1. Add a new data source that points to your metadata file.
> 2. Create a new entity in the **Entity Definition** window.
> 3. Create **Find**, **List**, and **Summary** screens and views for this entity.

To pass user input to a REST service:

1. Edit the **Find** screen for your entity.
2. Add a Text Box control to the **Find** screen. Your users will enter their search terms in this Text Box control.
3. Enable the Text Box control's **Editable** property.
4. Add a Button control. This button will trigger the call to the REST service and navigation to the List screen that displays the search results.
    a. Select "Find" for the **Action**.
    b. Select your entity's **List** screen for the **Target Screen**.
    c. Select your entity's **List** screen for the **Target View**.
5. In the metadata file, modify the `<url>` element to include the input field that is the search term. Use the BGGSearch SourceObject in the SampleMetadataFile.xml example above as a guide.

    There is no need to edit the **List** screen and add a Narrow Selection or Filter Field. The REST service is searching and returning all results that match the pattern.
6. Save and run the app. Be sure to reload the config.
7. Enter a search string and click the Button control. The app navigates to the **List** screen and displays the results. If you enter a search string with spaces or other special characters, the app automatically URL encodes the string. For example, if you enter "Reiner Knizia", the resulting URL passed to the REST service looks like the following:

    ```
    http://www.boardgamegeek.com/xmlapi/search?search=reiner%20knizia
    ```

Adding, removing, and updating data

A common way to implement RESTful services on the server is to support PUT and DELETE HTTP methods. The user submits a PUT request and that adds a new record. A DELETE request deletes a record.

The REST plug-in does not explicitly support PUT and DELETE methods, so you must use the GET or POST method to pass the instruction as well as the required parameters (a DELETE request usually requires some value that defines what to delete, and a PUT request usually requires data to add to the record).

The easiest way to manage this is to pass a static value that acts as the operation.

Debugging REST Data Source Connections

If no data appears when you run your application, ensure that:

- The service is available. For example, see if you can access your RESTful service's URL with a web browser.
- All the proper DLLs have been added to both the AppServer and AppStudio's plugins directories.
- You mapped the data fields to your entity. To check this, double click the entity in the Entity Definition screen. Select each data field and ensure that mapping is enabled.
- The request is being sent to the right URL and that it contains the proper information. To do this, use an

HTTP proxy tool such as Fiddler or Paros. For more information, see Using an HTTP Proxy Server.

- There is an identical metadata XML file in both the AppStudio plugins directory and the AppServer plugins directory (if you specify just a filename). If you specify a network location for the metadata file, ensure that the file is reachable from both the machines running AppStudio and the app server.
- All required input methods are being specified to pass data into the REST service.
- You do not have a redundant Narrow Selection defined. This results in a **"An item with the same key has already been added"** error. When you map an entity field to a Choice Field, you should not add a Narrow Selection. Remove the Narrow Selection from the **List** screen, save the app, and run it again.
- The URL is properly formed. If the URL contains an input parameter as a query string parameter (for example, `http://www.boardgamegeek.com/xmlapi/boardgame/?gameID=42`), rather than as part of the URL (for example, `http://www.boardgamegeek.com/xmlapi/boardgame/42`), set the `<Field>` element's `sendinquerystring` attribute to `true` in the metadata file.
- There are no other errors or warnings that are logged by the AppServer. Review the logs in the AppStudio **Logs** window, as described in Logging.

If the REST plug-in causes the AppServer's CPU to increase dramatically, try setting the `<stripxmlns>` property in the manifest file to false; for example:

```
<stripxmlns>true</stripxmlns>
```

This can reduce the size of the metadata file when it is processed, which reduces server load.

Using the WSDL Plug-in

The WSDL standard provides a common method of defining web services for programmatic use. XML files define the operations that can be performed on those web services, define the format for data transfer, and enforce validation of the requests and responses.

The Verivo WSDL plug-in reads WSDL files and generates a custom plug-in DLL based on that file. This on-the-fly creation of a plug-in lets you connect to WSDL-based data sources and test connections to your back-end all from within AppStudio.

To use data from a WSDL-based data source, you:

1. Install the WSDL plug-in.
2. Generate a custom WSDL DLL.
3. Add the generated DLL as a new data source.
4. Map the new data source to your entities.

Installing the WSDL Plug-in

The WSDL plug-in is not part of the core Verivo plug-ins. As a result, you must download and install it separately. For more information on installing the plug-in, see Downloading and installing non-core plug-ins.

In addition, you must have the .NET SDK or Microsoft Visual Studio installed to use this plug-in.

Generating a WSDL plug-in DLL

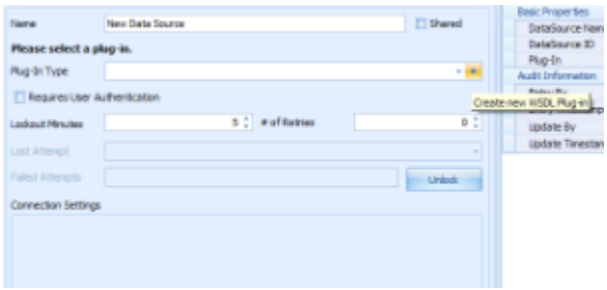
AppStudio automatically generates a WSDL plug-in based on the contents of your data source's WSDL file. The generated WSDL plug-in DLL is based on a WSDL plug-in template. The default template is appropriate for most circumstances. If you want to extend your WSDL plug-in to perform more advanced functionality, you can compile a custom WSDL plug-in to use as a template. For more information, see Developing Custom Plug-ins.

Before generating a WSDL plug-in DLL, be sure that your WSDL:

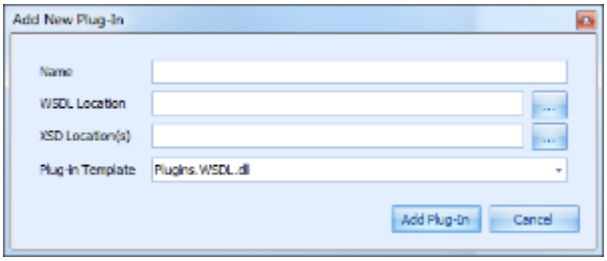
- Uses only supported data types
- Avoids known limitations

To generate a WSDL plug-in DLL from the default template:

1. Open the **Data Source Manager** in AppStudio.
2. Click the Add button to add a new data source.
3. In the **Name** field, enter a name for your new data source. For example, "My WSDL DS". This is the data source that you will use when mapping entities to the web service's output fields.
4. Click the + symbol next to the **Plug-In Type** drop-down list box:



The Add New Plug-In dialog box appears:



5. Enter details about the new plug-in DLL that is to be created. All fields are required, unless otherwise noted. Use the following table to determine the value of the fields:

Field	Description
Name	Enter a name for the plug-in; for example, MyWSDLPlugIn.
WSDL Location	Enter the location of the *.wsdl file; for example: `http://www.w3schools.com/webservices/tempconvert.asmx?WSDL` The WSDL file must be WSE 3.0 compliant.
XSD Location(s)	(Optional) Enter the location of the *.xsd (XML Schema Definition) files. XSD files are used to validate XML files such as WSDL files. For example: `http://www.yourcompany.com/wsdl/ComplexTypes.xsd` If there are more than one XSD files, separate them with commas; for example: `http://www.yourcompany.com/wsdl/ComplexTypes.xsd, http://www.yourcompany.com/wsdl/MoreComplexTypes.xsd` XSD files are optional, but you are encouraged to use them to ensure that the data types being transmitted are clearly defined and meet the requirements of the web service.

Plug-In Template	Select the appropriate DLL. Unless you have created a custom WSDL plug-in template, select Plugins.WSDL.dll. For information on creating a custom template, see Using a Custom Template.

6. Click the Add Plug-In button to generate the new DLL.

   AppStudio generates a new plug-in DLL based on the WSDL file you provided. The name of the plug-in is the name you entered in the **Add New Plug-In** dialog box (for example, MyWSDLPlugIn.dll). This DLL is located in AppStudio's /Plugins directory. In that same directory, AppStudio also generates a *.pdb file. This file is a Program Debug Database file and is used for debugging the plug-in in Visual Studio. For more information, see Program Database Files.
7. Copy the new DLL from the AppStudio /Plugins directory to your server's /bin/plugins directory.
8. (Optional) Copy the *.pdb file from the AppStudio /Plugins directory to your server's /bin/plugins directory. While not required for deployment, this file is necessary if you want to step through the plug-in when debugging in Visual Studio.
9. Save your application.

After the DLL is generated, the WSDL is no longer required. It does not need to be accessible at run-time by the server or the client devices.

Supported data types

The WSDL plug-in supports the data types listed in the WSDL Plug-In Supported XSD Data Types table.

XSD Data Type	WSDL Plug-in Data Type	Additional Information
anyURI	System.Uri	
Boolean	System.Boolean	Requires `minOccurs="0"`
Byte	System.SByte	
Date	System.DateTime	Requires `minOccurs="0"`
dateTime	System.DateTime	Requires `minOccurs="0"`
decimal	System.Decimal	Requires `minOccurs="0"`
Double	System.Double	Requires `minOccurs="0"`
duration	System.TimeSpan	
ENTITY	System.String	
Float	System.Single	Requires `minOccurs="0"`
gDay	System.DateTime	Requires `minOccurs="0"`
gMonthDay	System.DateTime	Requires `minOccurs="0"`

gYear	System.DateTime	Requires `minOccurs="0"`
gYearMonth	System.DateTime	Requires `minOccurs="0"`
ID	System.String	
IDREF	System.String	
int	System.Int32	
integer	System.Decimal	
language	System.String	
long	System.Int64	
month	System.DateTime	Requires `minOccurs="0"`
Name	System.String	
NCName	System.String	
negativeInteger	System.Decimal	
NMTOKEN	System.String	
nonNegativeInteger	System.Decimal	
nonPositiveInteger	System.Decimal	
normalizedString	System.String	
NOTATION	System.String	
positiveInteger	System.Decimal	
QName	System.Xml.XmlQualifiedName	
short	System.Int16	
string	System.String	
time	System.DateTime	Requires `minOccurs="0"`
timePeriod	System.DateTime	Requires `minOccurs="0"`
token	System.String	

unsignedByte	System.Byte	
unsignedInt	System.UInt32	

For additional information about types that require that the `minOccurs` attribute be set to "0", see Limitations of the WSDL Plug-in.

Connecting to a WSDL-based Data Source

To add the WSDL-based data to Verivo applications:

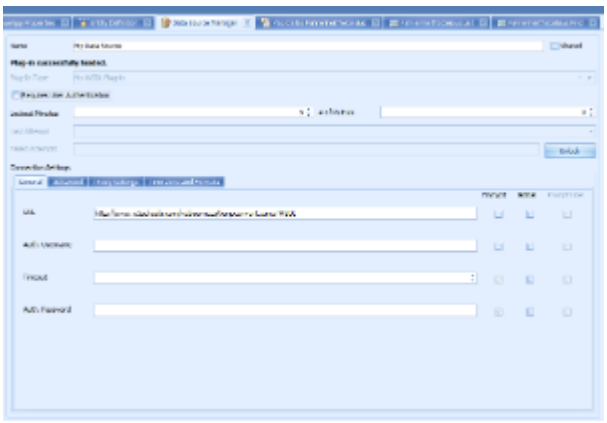
1. Generate a plug-in DLL for your WSDL web service, as described in Using the WSDL Plug-in.
2. Add the WSDL plug-in as a data source in AppStudio.
3. Create entities from the WSDL-based data source.
4. Call the operations of the data source.

Adding the WSDL plug-in as a data source

After you generate a custom DLL for your WSDL data source, you add it as a data source.

To add the WSDL plug-in as a data source:

1. Open the **Data Source Manager** in AppStudio.
2. To add a new data source, click the **Add** button.
3. Enter a name for the new data source in the **Name** field.
4. Select the new DLL from the list of plug-ins under **Plug-In Type**.
5. Under Connection Settings, enter the location of the WSDL in the **URL** field. For example, http://www.w3schools.com/webservices/tempconvert.asmx?WSDL.



   The **URL** field is not actually required. The location of the WSDL is compiled into the generated DLL, as are all the details about the service. However, it is useful to add the URL here for reference.

   You can ignore the settings on the **Advanced** tab.
6. Click the **Add** button.

After adding the new data source, you generally map its output fields to fields in the entities that your app uses. For more information, see Creating entities from the WSDL-based data source.

Creating entities from the WSDL-based data source

Operations are methods of the web service that a client calls. Like methods, they typically take some input and return some value based on the input. In AppStudio, you map web service operations to data entities so that you can display their output on a screen.

The WSDL that you compiled your WSDL against defines these operations and the valid data formats.

For example, the Temperature Conversion WSDL defines the **CelsiusToFahrenheit** and a **FahrenheitToCelsius** o perations:

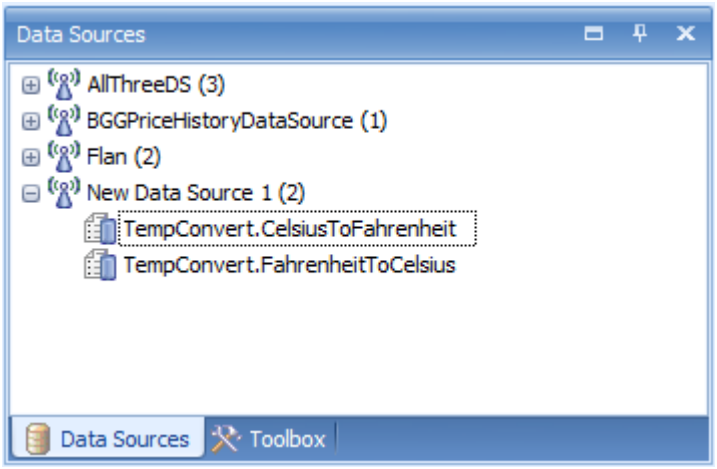
TempConvert operations

```
<!-- From the http://www.w3schools.com/webservices/tempconvert.asmx?WSDL -->
...
<!-- Defines operations for SOAP requests -->
<wsdl:portType name="TempConvertSoap">
  <wsdl:operation name="FahrenheitToCelsius">
    <wsdl:input message="tns:FahrenheitToCelsiusSoapIn"/>
    <wsdl:output message="tns:FahrenheitToCelsiusSoapOut"/>
  </wsdl:operation>
  <wsdl:operation name="CelsiusToFahrenheit">
    <wsdl:input message="tns:CelsiusToFahrenheitSoapIn"/>
    <wsdl:output message="tns:CelsiusToFahrenheitSoapOut"/>
  </wsdl:operation>
</wsdl:portType>
...
```

You can quickly create entities in AppStudio from a WSDL-based data source. Each entity is mapped to an operation of the web service.

To create entities from your WSDL-based web service:

1. In AppStudio, select **Data Entities** to open the **Entity Definition** panel.
2. Select the **Data Sources** panel in the lower left corner of the screen.
3. Click the + next to the web service's data source.
4. View the items in the list. For example, the Temperature Conversion WSDL defines **CelsiusToFahrenheit** an d a **FahrenheitToCelsius** operations, as the following image shows:



5. Drag the operation you want to add as an entity to the **Entity Definition** panel.
6. Save your application.

Passing Data to a WSDL-based service

This section describes the following topics:

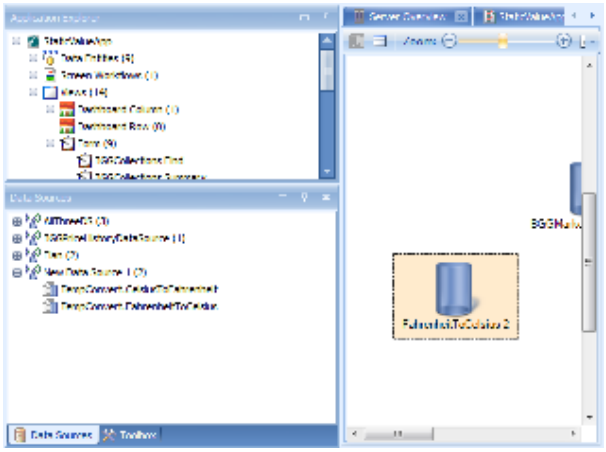
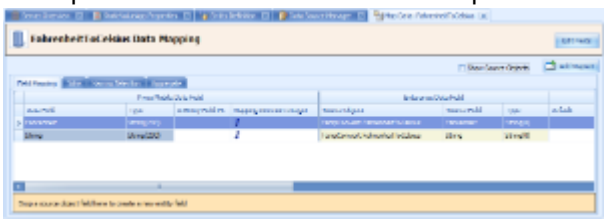
- Identifying inputs
- Passing static values
- Passing variable data to a WSDL service
- Tips for using WSDL-based data sources

Identifying inputs

The inputs for a given web service are defined by the WSDL file that you compiled the plug-in against. Each operation in the WSDL defines an input and an output. Each input and output is separately defined as a message part.

To determine which inputs are available, you can either look at the original WSDL that the generated DLL is based on, or use the Entity Definition in AppStudio. Parsing the WSDL is beyond the scope of this document.

To entities and screens for a given WSDL operation in AppStudio:

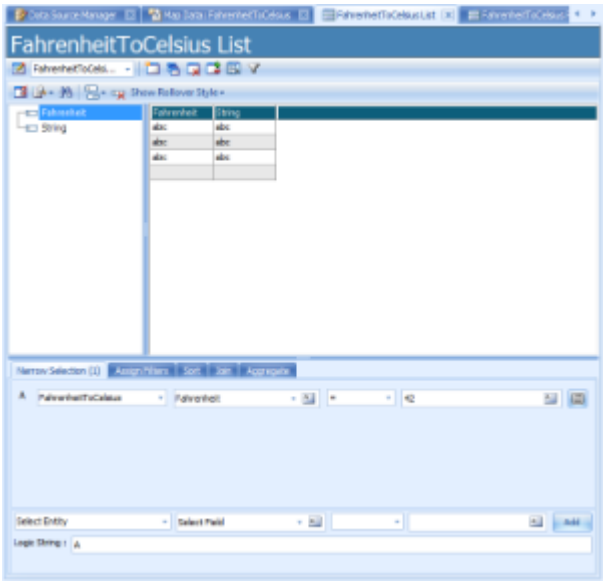
1. Select **Data Entities** from the toolbar.
   The **Entity Definition** window displays.
2. From the **Data Sources** list, expand the WSDL data source.
3. Select the operation – in this example, FahrenheitToCelsius – and drag it from the data source to the Entity Definition window:
   
4. Double click on the entity. The Data Mapping panel displays.
   The inputs are defined as **Data Fields** on the **Field Mapping** tab. In this example, Fahrenheit and String are two inputs to the FahrenheitToCelsius operation:
   
   The fields should automatically be mapped: the entity's **Data Fields** map to the source object's **Source Fields**. If they are not, select a **Source Object** and **Source Field** (on the right) for each **Data Field** (on the left).
5. To create List, Find, and Summary screens for your new entity, right-click on the entity in the **Entity Definition** panel and select **Create Screens For > Default**.
6. Save your application.

Passing static values

In many cases, web services require a static value, such as a license key, from the client. To call a web service, you typically specify the inputs as Narrow Selections or Joins on a List or Form screen.

To pass a static value to a WSDL-based data source:

1. Navigate to the entity's **Form** or **List** screen. For example, the FahrenheitToCelsius List.
2. Select the **Narrow Selection** tab.
3. Add a new item to the list:
    - **Select Entity:** Select the entity that is mapped to the service. For example, FahrenheitToCelsius.
    - **Select Field:** Select the field to pass to the service. For example, Fahrenheit.
    - **Operator:** Select "=".
    - In the value field, enter the static value that you want to send to the web service. For example, enter a number that you want to convert from Fahrenheit to Celsius such as "42".
4. Click the Add button to add the new Narrow Selection:



5. Save and run your app. Be sure to reload your app's configuration before using the new service.

When you run your application, the request URL looks like the address of the WSDL (the parameters are all passed as part of an HTTP POST request so they are not visible in the URL):

```
http://www.w3schools.com/webservices/tempconvert.asmx?WSDL
```

The request contains POST data in XML format. The raw request looks something like the following:

SOAP Request

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
  xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
  <soap:Header>
    <wsa:Action>http://tempuri.org/FahrenheitToCelsius</wsa:Action>
    <wsa:MessageID>urn:uuid:0afe4e9b-cdda-4a93-ada2-c597222e7fe6</wsa:MessageID>
    <wsa:ReplyTo>
      <wsa:Address>http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</wsa:Address>
    </wsa:ReplyTo>
    <wsa:To>http://www.w3schools.com/webservices/tempconvert.asmx</wsa:To>
    <wsse:Security>
      <wsu:Timestamp wsu:Id="Timestamp-c7d77828-5837-4213-afe6-f278a3bc7aca">
        <wsu:Created>2012-07-17T16:16:35Z</wsu:Created>
        <wsu:Expires>2012-07-17T16:21:35Z</wsu:Expires>
      </wsu:Timestamp>
    </wsse:Security>
  </soap:Header>
  <soap:Body>
    <FahrenheitToCelsius xmlns="http://tempuri.org/">
      <Fahrenheit>42</Fahrenheit>
    </FahrenheitToCelsius>
  </soap:Body>
</soap:Envelope>
```

The Temperature Conversion service returns a numeric value embedded within a SOAP envelope. In the following example response, the service returns 5.56, which is 42 degrees Fahrenheit converted to Celsius:

SOAP Response

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <FahrenheitToCelsiusResponse xmlns="http://tempuri.org/">
      <FahrenheitToCelsiusResult>5.56</FahrenheitToCelsiusResult>
    </FahrenheitToCelsiusResponse>
  </soap:Body>
</soap:Envelope>
```

To view the requests and responses, you can use an HTTP proxy server such as Fiddler. For more information, see Using an HTTP Proxy Server.

Passing variable data to a WSDL service

Rather than pass a static value to a WSDL-based web service, you can pass variable data or user input. You can define this data in any control that can be mapped to a data entity. For example, you can pass in a selection in a Choice Field (like an HTML drop-down box), an item from a **List** screen, or user input in a Text Box control. Each of these tasks is described in the following sections.

Passing variable data in a Choice Field

Choice Field controls (similar to HTML drop-down lists) present users with a list of options, from which the user can select one.

The process for creating a Choice Field control and passing the selected value to a WSDL-based service is the same as passing the value to a *REST*-based web service, except that you do not need to edit a metadata file.

For detailed instructions, see Passing variable data to a REST service with a Choice Field.

Passing user input

A common use of a web services is to pass a user-entered string that is evaluated on the server side. The service then returns results based on that input. The most common scenario when this happens is for searching.

To create a search input field in AppStudio, use a Text Box control on a **Find** screen. In this case, using a **Find** scre en makes sense based on its name, but this is also the kind of screen you use for passing just about any user-defined value to a web service. The service typically reads the value and performs some kind of lookup.

> **Prerequisite**
>
> Before you can complete this procedure, you must perform the following steps:
>
> 1. Add a new data source that is based on a generated WSDL DLL.
> 2. Create a new entity in the **Entity Definition** window.
> 3. Create **Find**, **List**, and **Summary** screens and views for this entity.

To pass user input to a WSDL-based web service:

1. Edit the **Find** screen for your entity.
2. Add a Text Box control to the **Find** screen. Your users will enter their search terms in this Text Box.
3. Enable the Text Box control's **Editable** property.
4. Add a Button control. This button will trigger the call to the web service and navigate to the List screen that displays the search results.
    a. Select "Find" for the **Action**.
    b. Select your entity's **List** screen for the **Target Screen**.
    c. Select your entity's **List** screen for the **Target View**.
5. Save and run the app. Be sure to reload the config.
6. Enter a value in the Text Box control and click the Button. For example, enter "42". The app navigates to the **List** screen and displays the results.

Tips for using WSDL-based data sources

Naming conventions

Changing the name of input fields will give you a naming convention that you can refer to when you need to do screen configuration. For example if you have an input of `UserID` you can add in the prefix of `Input_` to indicate that this will be an input into the service (`Input_UserID`).

Maximum number of rows

One thing to watch out for is if the web service returns a max number of rows, the search will be done only on the rows returned. The max rows for the screen will not affect this, however, as that is applied after/at the same time as the query on the output. While the ideal solution would still be to add a new input to the web service for any field on

which you want to search, this functionality will be useful in cases where that is not possible/not easy.

Input/output parameters

When using web services, the Verivo AppServer runs a WHERE clause against results returned from a web service if the entity field is not an input to the service. For example, if a web service has an input of company and outputs a list of rep names associated with the company, you can make a find screen with the company input field as a control, and the rep name output field as a control. The server will send the company name into the web service, and then run a sql query against the results to pull back reps who fit the name parameter.

Using multiple services

You might run into a scenario where a single web service response does not return all the data needed for the screen, or that a web service requires specific values from another method to work correctly. In these cases you can use the **Join** tab to use multiple web services on a single screen request.

The first thing you need to identify is what method needs to be called first and which need to be called subsequently. For each of these subsequent methods, modify each entity and enable the **Joins Require Literal** property.

On the view itself, send inputs to the first web service by using the **Narrow Selection** tab. Each subsequent call is handled on the **Join** tab. For example, if you use 3 web services (A, B, and C) to populate a screen, send in inputs to the first web service (A) with instruction(s) on the **Narrow Selection** tab. On the **Join** tab, add 2 Join instructions: one joining web service A and B, and the second joining B and C. For each join instruction, define the output values that must be sent to the subsequent web service.

Debugging WSDL Data Source Connections

This section describes the following techniques for troubleshooting your WSDL connections:

- Tools for debugging
- Generating proxy classes from WSDLs
- SOAP tracing
- Troubleshooting

In addition to the tools and techniques described in this section, you should familiarize yourself with the limitations of the WSDL plug-in.

Tools for debugging

The following tools are useful for debugging WSDL data sources:

Tool	Description
SoapUI	SoapUI is a tool for simulating and testing SOAP services. For more information, see http://sourceforge.net/projects/soapui/files/.
HTTP proxy server	To view the requests/responses of the server and client, you can use an HTTP proxy server tool such as Fiddler or Paros. For more information, see Using an HTTP Proxy Server.

TraceSOAPRequest.dll	The TraceSOAPRequest.dll tool lets you log SOAP messages to the Verivo database. For more information, see SOAP tracing.
WseWsdl3.exe	The WseWsdl3.exe tool is useful for debugging and testing custom WSDL DLLs. For more information, see Generating proxy classes from WSDLs.

Generating proxy classes from WSDLs

When debugging and testing a custom WSDL DLL, it can be useful to look at the generated proxy class – or to attempt to generate the WSDL plug-in directly so that you can examine the output.

You can use the WseWsdl3.exe command-line utility to generate a proxy class from a WSDL. The class can be a web client or a SOAP client. This utility is in the AppStudio/bin directory.

For information about WseWsdl3, including command-line options and detailed usage information, see http://msdn.microsoft.com/en-us/library/aa529578.aspx.

> When using the WseWsdl3 tool, you must have write privileges in the Application Studio directory because the tool outputs a C# file to that directory.

SOAP tracing

To log SOAP requests in the AppServer's logging table, you can use a tool such as TraceSOAPRequest.dll. This tool logs both requests and responses in the table and can be viewed in AppStudio's **Log** window.

To enable SOAP tracing:

1. Ensure the TraceSOAPRequest.dll file is in the AppServer /bin directory. This DLL is installed with AppServer 7.2 and later.
2. Add the following block to the `<system.web/>` node in the web.config file:

```
<webServices>
  <soapExtensionTypes>
    <add
      type="TraceSOAPRequest.TraceSOAP,
        TraceSOAPRequest,
        Version=1.0.0.0,
        Culture=neutral,
        PublicKeyToken =null"
      priority="1"
      group="0"/>
  </soapExtensionTypes>
</webServices>
```

3. If you want to log the entries to the database log table, you are now ready to query the server from the client. Any requests and responses will be written to the database.
4. If you want to log the entries to a local file, add the following key/value pair to the `<appSettings/>`node of the server's web.config file:

```
<add  key="SOAPLogs" value="C:\Logs\" />
```

5. Replace the path with the path to the directory in which you want the logs written.
6. Ensure the ASP.NET worker process and any IIS user accounts have write access to the log folder.

At this point all subsequent web service requests will yield additional records in the Pyxis Logging Tables indicating the request and response to a service. You can then use this to compare against any examples provided by the customer of an appropriate request/response. Based on the discrepencies found through this exercise you will be able to determine if the issue on the application side of creating the request or handling the response, or if the service itself is doing something unexpected to the application.

To disable logging, remove or comment out the `<soapExtensionTypes/>` block in the web.config file.

Troubleshooting

The following table describes common errors that you might encounter when working with WSDL-based web services:

Error	Description
"Unknown exception thrown... Retrieve the LoaderExceptions property for more information"	If you get this error, the AppServer does not have WSE 3.0 installed. Install WSE 3.0 on the AppServer's machine.
"Could not find interimplugin.cs"	If you get this error when generating a WSDL DLL, there could be permissions errors or an issue with the WSDL file itself. To find out the source of the error: 1. Open a command window. 2. Navigate to AppStudio's root directory (where WseWsdl3.exe resides). 3. Run the following command: `WseWsdl3.exe /type:webClient c:\Users\ndanger\Desktop\MyService.wsdl` 4. Replace the directory location in that command to the location of your WSDL file. In addition, you must have either the .NET SDK (not just the framework) or WSE 3.0 installed to generate a WSDL DLL.
"Could not find plugin dll"	If you get this error at run-time, ensure that you copied the generated DLL from the AppStudio plugin directory to the AppServer plugins directory. You are not required to restart the server after adding a new DLL to the server's plugins directory.
"Client found response content type of '', but expected 'text/xml'"	If you get this error, the request you are sending to the service is returning a no-XML response back to the application. You should contact the service administrator.

"Errors building interimplugin.cs into interimplugin.dll ..."	You might encounter this error when generating the WSDL DLL. It means that the WSDL returns a DataSet in its response, rather than strongly typed fields. In the WSDL file, you will typically see `<xsd:schema>` and `<xsd:any>` elements. To fix this error, you might need to modify the web service to be strongly typed or create a custom plug-in for that web service. For more information, see Developing Custom Plug-ins.
"Method method_name *cannot be reflected"*	This error is most likely caused because the backend service generated a bad WSDL. For example, two methods in the WSDL might have the same signatures. To resolve this issue, test the WSDL for compliance.

If you do not get expected data, verify the following:

- The request is being sent to the correct URL and that the SOAP message is properly encoded. To do this, use an HTTP proxy tool such as Fiddler or Paros to view the full XML request and response. For more information, see Using an HTTP proxy server .
- The data entity is mapped correctly for download.
- The WSDL describes the same output structure that the web service returns. If there is a discrepency in these two the plugin will not see the appropriate column to bring the value back to the screen.
- Your web service method has distinct input and output fields. The WSDL plug-in does not support methods that have fields that are used as both input and output.

Limitations of the WSDL Plug-in

This section describes the following known limitations for the WSDL plug-in:

- Handling null value responses
- Request/response types
- Array types
- References to multiple WSDL files
- Exposure of SOAP header values as connection parameters
- Recursive object handling
- Upload support

Handling null value responses

For the WSDL plug-in to properly handle null value responses from a Web service for certain data types, the type definition must include the `minOccurs=0` attribute.

The following default XML Schema Definition (XSD) types require this addition:

- `xsd:datetime`
- `xsd:date`
- `xsd:time`
- `xsd:timeperiod`
- `xsd:month`
- `xsd:gYearMonth`
- `xsd:gYear`
- `xsd:gMonthDay`
- `xsd:gDay`
- `xsd:gMonth`
- `xsd:boolean`
- `xsd:float`

- `xsd:double`
- `xsd:decimal`
- All derivative XSD types of `xsd:decimal` including integer, long, int, short, byte, or anything positive, negative, non-positive, non-negative, or unsigned

Request/response types

A WSDL plug-in requires that the request/response types be strongly typed. The following is a sample of a complex type that can be handled successfully by the WSDL plug-in:

```
<xsd:complexType name="Sample_Employee">
  <xsd:sequence>
    <xsd:element minOccurs="0" maxOccurs="1" name="FirstName" type="xsdLocal1:string50"/>
    <xsd:element minOccurs="0" maxOccurs="1" name="LastName" type="xsdLocal1:string50"/>
    <xsd:element minOccurs="0" maxOccurs="1" name="MiddleName" type="xsdLocal1:string50"/>
    <xsd:element minOccurs="0" maxOccurs="1" name="Address1" type="xsdLocal1:string50"/>
    <xsd:element minOccurs="0" maxOccurs="1" name="Address2" type="xsdLocal1:string30"/>
    <xsd:element minOccurs="0" maxOccurs="1" name="EmployeeID" type="xsdLocal1:string30"/>
  </xsd:sequence>
</xsd:complexType>
```

The WSDL plug-in handles nested objects in request and response data types if they are strongly typed.

Array types

The WSDL plug-in can handle requests and responses of array types, but they must be strongly typed.

The following is a sample of a strongly typed array requests and responses that the WSDL plug-in can be handled successfully:

```
<s:complexType name="ArrayOf_xsd_string">
  <s:complexContent mixed="false">
    <s:restriction base="soapenc:Array">
      <s:attribute WSDL:arrayType="s:string[]"ref="soapenc:arrayType"/>
    </s:restriction>
  </s:complexContent>
</s:complexType>
```

References to multiple WSDL files

The WSDL plug-in cannot handle references to multiple WSDL files within the same plug-in. However, you can employ a workaround by creating multiple plug-in assemblies, each with a reference to one WSDL file. In addition, the WSDL plug-in does allow importing of external XSDs as long as they are in the same WSDL.

Exposure of SOAP header values as connection parameters

The WSDL plug-in can handle required values passed in SOAP headers; however, the WSDL plug-in requires code added to the derived class to allow the plug-in to expose those header values as connection parameters in AppStudio. The SOAP headers should be explicitly defined in the WSDL plug-in DLL file.

The following is an example of a header definition within Plugins.WSDL.dll:

```
<xsd:element name="SampleSoapHeader">
  <xsd:complexType name="SoapHeader">
    <xsd:sequence>
      <xsd:element name="applicationID" type="xsd:string"/>
      <xsd:element name="login" type="xsd:string"/>
      <xsd:element name="password" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<WSDL:message name="RequestHeader">
  <WSDL: part name="soapHeader" element="SampleSoapHeader"/>
</WSDL:message>
```

To make the SOAP header a part of the operation, include the following `WSDLsoap:header` tag in every input operation required. Likewise, you can also include the soap header for output operations by using the using the same convention for the WSDL:output.

The following is an example of the SOAP header as part of the operation:

```
<WSDL:binding name="AuthSoapBinding" type="AuthService">
  <WSDLsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <WSDL:operation name="login">
    <WSDLsoap:operation soapAction=""/>
    <WSDL:input name="loginRequest">
      <WSDLsoap:header WSDL:required="true" message="RequestHeader" part="soapHeader"
use="literal"/>
      <WSDLsoap:body use="literal"/>
    </WSDL:input>
    <WSDL:output name="loginResponse">
      <WSDLsoap:body use="literal"/>
    </WSDL:output>
  </WSDL:operation>
</WSDL:binding>
```

Your custom WSDL plug-in is required to expose those SOAP header values as connection parameters so that users can provide values for the SOAP headers at runtime. The following are steps to add connection parameters to the plug-in:

- Override the `DescribeParams()` method in the derived plug-in.
- Add the following code to the overridden `DescribeParams()` method:

```
ArrayList ParamArray = base.DescribeParams();
ParamArray.Add(new ConnParam("applicationID ", "",1,true,false,"Application ID"));
ParamArray.Add(new ConnParam("login","",2,true,false,"User ID"));
ParamArray.Add(new ConnParam("password","",3,true,false,"Password"));
return ParamArray;
```

Recursive object handling

The WSDL plug-in does not support recursive objects.

The following is an example of a WSDL definition of a recursive object that the WSDL plug-in does not support. As

you can see, the <Variations> element contains an <Item> element, and the <Item> element, in turn, contains a <Variations> element:

```
<!-- Not supported! -->

<xs:element name="Variations">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" name="TotalVariations" type="xs:nonNegativeInteger"/>
      <xs:element minOccurs="0" name="TotalVariationPages" type="xs:nonNegativeInteger"/>
      <xs:element minOccurs="0" ref="tns:VariationDimensions"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="tns:Item"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Item">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" ref="tns:VariationSummary"/>
      <xs:element minOccurs="0" ref="tns:Variations"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Upload support

The WSDL plug-in DLL file included with the SDK contains code that lets it execute download requests from any Web service. If your WSDL plug-in requires upload support, you will need to add this as an inheriting class; the WSDLPlugin base class does not currently support uploads.

Using a Custom Template

While AppStudio offers on-the-fly creation of a standard WSDL plug-in connecting to Web service data, WSDL plug-ins with more advanced capabilities require custom development. This section lists some possible characteristics of WSDL plug-ins and their Web services that might prevent successful or robust WSDL plug-in generation.

WSDLs and their web services with the following characteristics require custom WSDL plug-in development using the plug-in SDK:

- WSDLs used for uploading data.
- Web services that require a login to access data.
- Web services that require the base entity and association requests to execute in the same Web service method call; an AppStudio-generated WSDL plug-in treats an association upload as a separate request, executed after the base entity request.
- WSDLs requiring Web service request headers.
- Web service definitions that contain additional parameters specifying comparison operations for any data you intend to filter using >, < or any comparison operator other than =.
- Web service definitions that contain additional parameters specifying logic operations for any data you intend to filter using logic operators other than AND.
- Web service definitions containing enumerations.

These conditions for requiring custom-coded WSDL plug-ins are in addition to the standard requirement of having a well-defined object based WSDL with clearly defined input and output parameters.

You can write WSDL plug-ins using the PAF in any .NET language. All resources provided in the SDK address

writing plug-ins in C# only. This guide provides instructions for creating a WSDL plug-in class library project in Microsoft Visual Studio 2005.

To write a custom WSDL plug-in, you need the following information about your back-end data source, the web service:

1. The names and values for the appropriate connection parameters for the data source, which might include any of the following:
    - Username
    - Password
    - ConnectionString
    - URI
    - Connection protocol
    - Cookies
    - SessionIDs
2. The .NET data type equivalents for the data types of the Web service.
3. The access points, APIs, or ways to discover the available SourceObject classes.
4. The status fields, exceptions, and return codes.
5. The functions, formatting, and logic supported by the Web service data source, specifically which operations it can handle.
6. Any special data type formatting rules needed for queries or requests to work correctly, such as the rule requiring SQL Server to single quote literal values when the QUOTED_IDENTIFIER is set and rule to appropriately format date and time strings.

To format date and time strings, convert DataTime data type values: Date and time formatting are aided by the plug-in properties `DATE_FORMAT`, `TIME_FORMAT`, and `TIMESTAMP_FORMAT`.

For more information about creating custom plug-ins, see Developing Custom Plug-ins.

Using the WCF Plug-in

Verivo includes support for WCF-based data sources through a WCF plug-in. WCF (Windows Communication Foundation) is Microsoft's unified programming model for building service-oriented applications. It combines and extends the capabilities of Distributed Systems, Microsoft .NET Remoting, Web Services, and Web Services Enhancements (WSE), to develop and deliver unified secured systems. WCF was introduced as part of the .NET 3.0 Runtime components in 2006.

This section describes the following topics:

- WCF plug-in overview
- Installing the WCF plug-in
- Generating a WCF plug-in DLL
- Connecting to a WCF data source
- Creating entities from the WCF-based data source

For an introduction to WCF, see http://msdn.microsoft.com/en-us/library/dd936243.aspx.

WCF plug-in overview

WCF services are made up of the following components:

- **Service class** – The service class defines the methods that are accessed by clients. AppStudio generates the service class based on the WSDL file that you define.
- **Host process** – The host process manages the service class and handles requests to the endpoints. AppServer is the host process.

- **Endpoints** – Clients connect to the service through endpoints. Endpoints define an address, a binding type, and a service contract.
- **Service contract** – The service contract defines which methods of the service are available to clients.

The Verivo WCF plug-in is similar to the WSDL plug-in in that it reads a WSDL file and generates a custom DLL based on that file. It also defines data binding and mapping of XML data so that the data can cross the wire and be consumed by any type of application. You deploy the generated DLL and IIS exposes its endpoint so that clients can access it.

About binding

When connecting to a WCF data source, you choose the binding type based on the needs of your application.

The WCF plug-in supports the two most common WCF bindings, BasicHttpBinding and WSHttpBinding. BasicHttpBinding is the simplest binding. WSHttpBinding supports secure and transaction-based messaging. WSHttpBinding is the default WCF binding type. Both use HTTP for the transport layer, but they differ in implementation. The following table summarizes the differences:

Criteria	BasicHttpBinding	WSHttpBinding
Security support	WS-BasicProfile 1.1	WS-*
Compatibility	This is aimed for clients who do not have .NET 3.0 installed and it supports wider ranges of clients. Many of the clients like Windows 2000 still do not run .NET 3.0. So older version of .NET can consume this service.	As its built using WS-* specifications, it does not support wider ranges of client and it cannot be consumed by older .NET version less than 3 version.
SOAP version	SOAP 1.1	SOAP 1.2 and WS-Addressing specification
Reliable messaging	Not supported	Supported
Default security options	None. Data is sent as plain text by default.	WS-Security is enabled. Data is not sent in plain text.
Security options	• None • Windows (default authentication) • Basic • Certificate	• None • Transport • Message (default) • TransportWithMessageCredential • TransportCredentialOnly

Bindings are part of the endpoint definition in the WSDL. The following example defines two endpoints, one for the BasicHttpBinding binding type and one for the WSHttpBinding binding type:

```
<wsdl:service name="ManageContacts">
  <wsdl:port name="wsHttpBinding" binding="tns:wsHttpBinding">
    <soap12:address location="http://www.example.com/ContactsWCF/Contacts.ManageContacts.svc/ws"/>
    <wsa10:EndpointReference>
      <wsa10:Address>
        http://sqlqa2k8.pyxisit.com/ContactsWCF/Contacts.ManageContacts.svc/ws
      </wsa10:Address>
      <Identity xmlns="http://schemas.xmlsoap.org/ws/2006/02/addressingidentity">
        <Dns>localhost</Dns>
      </Identity>
    </wsa10:EndpointReference>
  </wsdl:port>
  <wsdl:port name="basicHttpBinding" binding="tns:basicHttpBinding">
    <soap:address location="http://www.example.com/ContactsWCF/Contacts.ManageContacts.svc/basic"/>
  </wsdl:port>
</wsdl:service>
```

For additional binding support, such as NetTcpBinding and NetNamedPipeBinding, you can develop a custom plug-in.

Note

BasicHttpBinding supports WS-BasicProfile 1.1. As a result, the Verivo WSDL plug-in can consume WCF services that use BasicHttpBinding. For information about using the WSDL plug-in, see Using the WSDL Plug-in.

For more information about bindings, see WCF Bindings In Depth.

About the endpoints

The endpoints that you define for a WCF plug-in define the address, binding type, and name of the WCF service that your app uses to connect to the data source. Typically, a WCF service defines a separate endpoint for each type of binding that is supported. For example, your service might have a "basic" endpoint that supports the BasicHttpBinding type, and a "secure" endpoint that supports the secure, transactional WSHttpBinding type.

The address, binding and contract match the address, binding and contract defined in the WebHost project's Web.config file.

Installing the WCF plug-in

The WCF plug-in is included in the downloadble set of non-core plug-ins. The WCF plug-in is regularly updated with bug fixes and features.

For more information on installing the plug-in, see Downloading and installing non-core plug-ins.

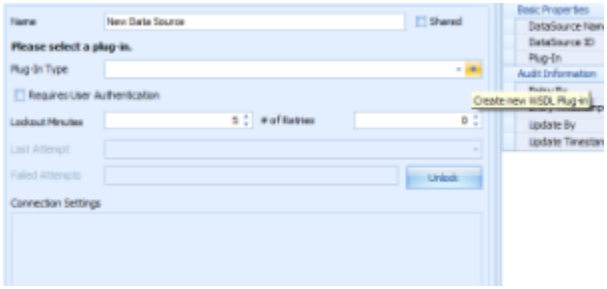
Generating a WCF plug-in DLL

AppStudio automatically generates a WCF plug-in DLL for you when you supply it the location of a WSDL file and additional connection parameters.

When generating a WCF plug-in DLL, you choose the type of binding and set the location of the endpoints. The two supported types of binding are BasicHttpBinding and WSHttpBinding. For more information, see About binding.

To generate a WCF plug-in DLL:

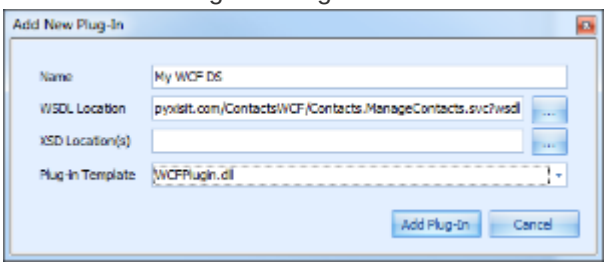
1. Open the **Data Source Manager** in AppStudio.
2. Click the **Add** button add a new data source.
3. In the **Name** field, enter a name for your new data source. For example, "My WCF DS". This is the data source that you will use when mapping entities to the service's output fields.
4. Click the + symbol next to the **Plug-In Type** drop-down list box:



5. The Add New Plug-In dialog box appears.
6. Enter details about the new plug-in that is to be created. Use the following table to determine the value of the fields:

Field	Description
Name	Enter a name for the plug-in; for example, MyWCFPlugIn.
WSDL Location	Enter the location of the *.wsdl file; for example: http://www.example.com/ContactsWCF/Contacts.ManageContacts.svc?wsdl
XSD Location(s)	Leave blank.
Plug-in Template	Select WCFPlugin.dll.

The Add New Plug-in dialog box should look similar to the following:



7. Click the **Add Plug-In** button to generate the new DLL. AppStudio generates a new plug-in DLL based on the WSDL file you provided. The name of the plug-in is the name you entered in the **Add New Plug-In** dialog box (for example, MyWCFPlugIn.dll).

   AppStudio saves the new DLL in its /Plugins directory. In that same directory, AppStudio also generates a *.pdb file. This file is a Program Debug Database file and is used for debugging the plug-in in Visual Studio. For more information, see Program Database Files.
8. Copy the new DLL from the AppStudio /Plugins directory to AppServer's /bin/plugins directory. You do not need to restart the server after copying a new DLL to its /bin/plugins directory.
9. Save your application.

Connecting to a WCF data source

When you connect to a WCF data source, you provide connection parameters that specify the endpoints.

To use the WCF plug-in as a data source:

1. Open the **Data Source Manager** in AppStudio.
2. To add a new data source, click the **Add** button.
3. Enter a name for the new data source in the **Name** field.
4. Select the new DLL from the list of plug-ins under **Plug-In Type**.
5. Under Connection Settings, select the **General** tab.
6. Enter the location of the endpoint in the **URL** field. For example, http://www.example.com/ContactsWCF/Contacts.ManageContacts.svc/basic. Note that this is not the same as the location of the WSDL, but rather the location of the endpoint defined by the WSDL for a particular binding.
7. Select the **WCF Binding** tab. Select the appropriate binding type and security mode. If you use a security mode, also select the client credential types.
8. If you are using Windows credentials with your WCF plug-in, select the **WCF Credentials** tab and enter the appropriate credentials.
9. Test the data source by clicking the Test Connection button. The "Connection Succeeded" message should appear. If the "Connection Failed" message appears, check that the value of the **URL** field is correct and that the service is available.
10. Click the **Add** button.

After adding the new data source, you generally map its output fields to fields in the entities that your app uses. For more information, see Creating entities from the WCF-based data source.

Creating entities from the WCF-based data source

In AppStudio, you map operations in the WSDL to data entities so that you can display their output on a screen.

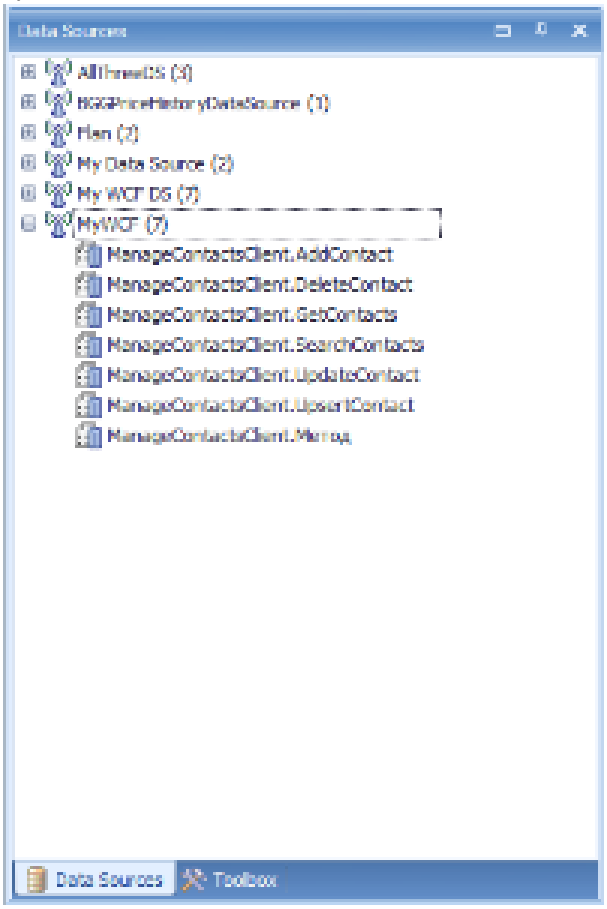
For example, the ManageContacts WSDL defines the inputs and outputs for the GetContacts and SearchContacts operations:

```
<wsdl:operation name="GetContacts">
  <wsdl:input
    wsaw:Action="http://tempuri.org/IManageContacts/GetContacts"
    message="tns:IManageContacts_GetContacts_InputMessage"/>
  <wsdl:output
    wsaw:Action="http://tempuri.org/IManageContacts/GetContactsResponse"
    message="tns:IManageContacts_GetContacts_OutputMessage"/>
</wsdl:operation>
<wsdl:operation name="SearchContacts">
  <wsdl:input
    wsaw:Action="http://tempuri.org/IManageContacts/SearchContacts"
    message="tns:IManageContacts_SearchContacts_InputMessage"/>
  <wsdl:output
    wsaw:Action="http://tempuri.org/IManageContacts/SearchContactsResponse"
    message="tns:IManageContacts_SearchContacts_OutputMessage"/>
</wsdl:operation>
```

To create entities from your WCF-based service:

1. In AppStudio, select **Data Entities** to open the **Entity Definition** panel.
2. Select the **Data Sources** panel in the lower left corner of the screen.
3. Click the + next to the service's data source.
4. View the items in the list. For example, the ManageContacts service defines several operations, including AddContact, DeleteContact, GetContact, and SearchContact. The following image shows the complete list of

operations:



5. Drag the operation you want to add as an entity to the **Entity Definition** panel.
6. You can now create List, Summary, and Search screens for the new entity. On the Entity Definition panel, right-click the new entity and select **Create screens for > Default**. AppStudio creates the three new screens for the entity. For information about working with the various screens, see About Screens and Views.
7. Save your application.

Debugging WCF Data Source Connections

When working with WCF-based data sources, it is useful to see the requests and responses as they are sent across the wire. To view the requests/responses of the server and client, you can use an HTTP proxy server tool such as Fiddler or Paros. For more information, see Using an HTTP Proxy Server.

The following table lists some common errors you might receive while working with WCF plug-in, and what to do about them:

Error	Description
Unknown exception thrown: There was no endpoint listening at <url> that could accept the message.	This is often caused by an incorrect address or SOAP action. Check the URL parameter for your data source. This should be the URL of the endpoint. It should not be your WSDL URL.

Unknown exception thrown: BasicHttp binding requires that BasicHttpBinding.Security.Message.ClientCredentialType be equivalent to the BasicHttpMessageCredentialType.Certificate credential type for secure messages.	The most likely source of this issue is that you are trying to connect to a WSHttpBinding endpoint using BasicHttpBinding. Make sure that your endpoint URL matches the Binding Type. Select Transport or TransportWithMessageCredential from the Security Mode drop-down box in the data source's Connection Settings.
The exception is Content Type application/soap+xml; charset=utf-8 was not supported by service <url>. The client and service bindings may be mismatched.	The most likely cause of this that you are trying to connect to a BasicHttpBinding endpoint using WSHttpBinding. Make sure that your endpoint URL matches the Binding Type.
BasicHttp binding requires that BasicHttpBinding.Security.Message.ClientCredentialType be equivalent to the BasicHttpMessageCredentialType.Certificate credential type for secure messages.	The security settings that you specified in AppStudio do not match to endpoints secuirty settings. Check that the Security Mode selection in the Connection Settings is correct.
The message could not be processed.	This error is most likely because the action <url> is incorrect or because the message contains an invalid or expired security context token. The security context token would be invalid if the service aborted the channel due to inactivity. To prevent the service from aborting idle sessions prematurely increase the Receive timeout on the service endpoint's binding. Another source of this error is that there is a mismatch between bindings. The security settings that you specified in AppStudio do not match the endpoint security settings. Check the Security Mode and Binding Type selections in the Connection Settings. Note that the default selection for BasicHttpBinding is None; for WSHttpBinding, the default should be Message.

Using the SharePoint Plug-in

Verivo provides a connector plug-in for access to a SharePoint Web Service. Below lists basic information including instructions for connecting to the SharePoint environment, capabilities of the plug-in, how to deploy the plug-in in your environment and troubleshooting steps.

This section describes the following topics:

- Capabilities of the SharePoint plug-in
- Installing the SharePoint plug-in

- Using the SharePoint plug-in
- Troubleshooting

Capabilities of the SharePoint plug-in

The SharePoint plug-in lets you do the following to the SharePoint Lists and Views:

- Add
- Edit
- View

The Sharepoint Plugin also has Rich Text Modes for FullHTML and Compatible Mode when communicating with a Sharepoint backend. You can set these options in the Advanced Connection Settings section of the Datasource Manager.

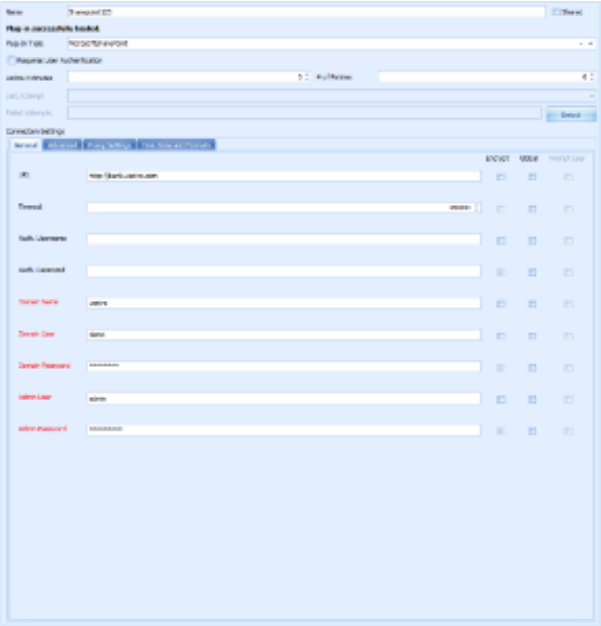
Installing the SharePoint plug-in

The SharePoint plug-in is not part of the core Verivo plug-ins. As a result, you must download and install it separately. For more information on installing the plug-in, see Downloading and installing non-core plug-ins.

Using the SharePoint plug-in

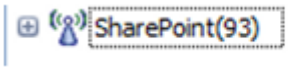
To use a SharePoint data source:

1. Add your new SharePoint data source:
    a. Open the **Data Source Manager**.
    b. Click "Add new data source".
    c. Give the data source an appropriate name and select the Microsoft SharePoint plug-in to connect with.
    d. Select the General tab under Connection Settings.
    e. Fill in the following information:
        - *URL:* The URL of the SharePoint site.
        - *Domain Name:* The Domain which the SharePoint site is hosted.
        - *Domain User:* A user specifically created to connect to the Data Source through Verivo (required to read the data).
        - *Domain Password:* The attached password of the Domain User.
        - *Admin User*: An admin account specifically created to connect to the Data Source through Verivo (required to make edits/uploads).
        - *Admin Password:* The attach password of the Admin User.



2. Save the connection information.
3. Click the Test Connection button.

   The 'Test Connection' option might fail due to the connection protocol Verivo uses to connect to SharePoint. To confirm a successful connection, click Data Sources in the bottom left corner of AppStudio; right click on your data source and click the Refresh button. If your connection parameters are correct you should see the SharePoint object load with its contents:



Troubleshooting

When you refresh your SharePoint data source and you see the below image, it most likely means that there is an issue with your connection parameters.

- Left click this icon to see the error message. This message should help you to determine what the issue could be.
- In many cases, the URL is the cause of the failure. AppStudio calls for the root path to the service and not the full path to the SharePoint web service. SharePoint provides all of its metadata in a subdirectory named "_vti_bin". The SharePoint plug-in appends this suffix plus the service that data is obtained from (lists, views, etc) internally.
- Sharepoint administrators and developers must configure column field types on the backend to use the correct Rich Text Mode option, depending on the version of Sharepoint. For more information, see:

  http://office.microsoft.com/en-us/sharepoint-server-help/site-column-types-and-options-HA010302196.aspx

  http://sharepointkb.wordpress.com/2008/09/12/three-powerful-columns-not-many-people-know-about/

SharePoint Example

For our site, we can use the following URL in AppStudio:

```
https://bank.verivo.com/
```

The plug-in appends to this the path to the asmx file:

```
https://bank.verivo.com/_vti_bin/lists.asmx
```

```
https://bank.verivo.com/_vti_bin/views.asmx
```

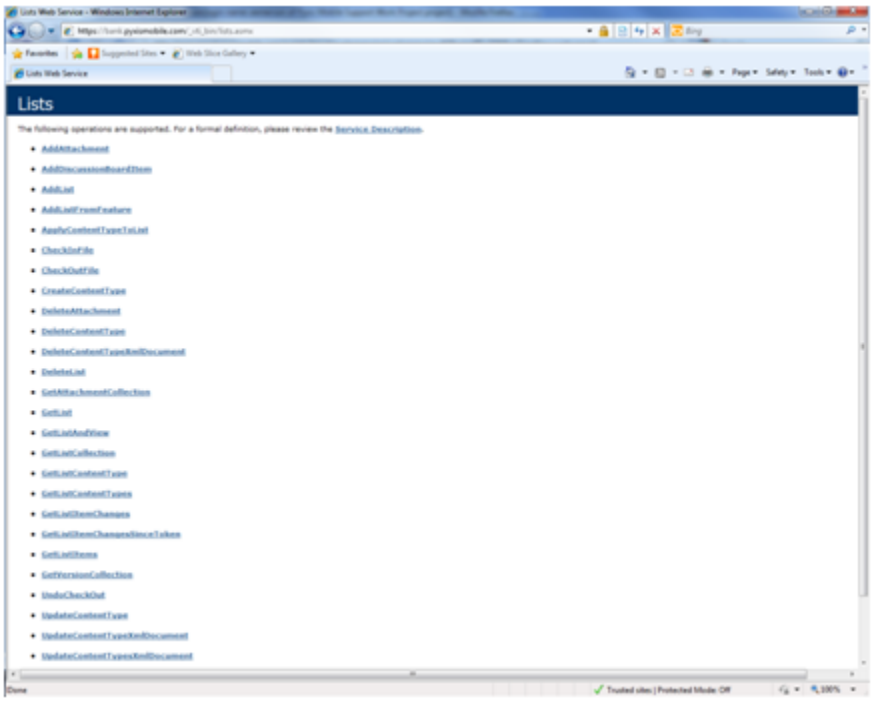
To determine the correct URL, open the URL in a browser and then append:

```
_vti_bin/lists.asmx
```

or

```
_vti_bin/views.asmx
```

If you are using the correct URL, you should see a page similar to the following after you make the request:



In our example, the URL is:

```
https://bank.verivo.com/_vti_bin/lists.asmx
```

Using the Salesforce 9 Plug-in

Salesforce is a CRM product that hosts the application and data online for customers. It provides an API that allows 3rd-party applications to connect and transfer data for a user. Verivo provides a plug-in that connects to Salesforce data from a mobile app.

For more information about using the Salesforce 9 plug-in, see http://support.verivo.com/entries/21143887-salesforce.

Using an HTTP Proxy Server

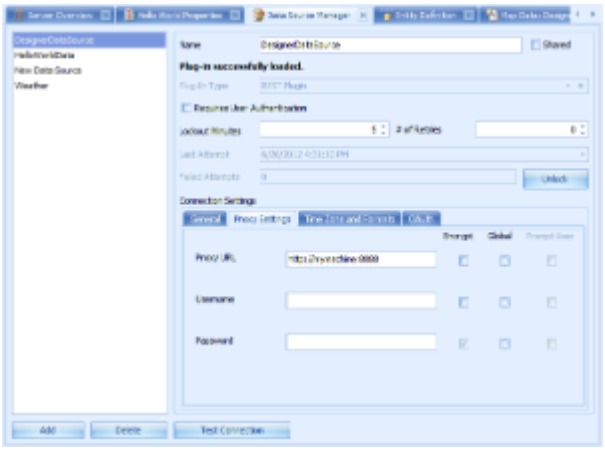
AppStudio supports using third-party HTTP proxy servers such as Fiddler or Paros. HTTP proxy servers let you view all the HTTP traffic coming and going from your machine. This is useful for debugging plug-ins that use web services because you can view the communication between the device or simulator and the back-end data source.

> HTTP proxies work with plug-ins that use HTTP-based web services, such as the WSDL, WCF, and REST plug-ins. It does not work with plug-ins like SQL Server.

When configuring the proxy, be sure to allow remote computers to connect to the proxy. This lets you view the traffic between your mobile device (including simulators) and Verivo server. Also, be sure to enable HTTPS traffic if your app uses secure connections to the web services.

To use a proxy with AppStudio:

1. Open the Data Source Manager.
2. Under Connection Settings, select the Proxy Settings tab:



   If you do not see a Proxy Settings tab, then your plug-in does not support proxy servers.
3. Enter the following information in the fields:

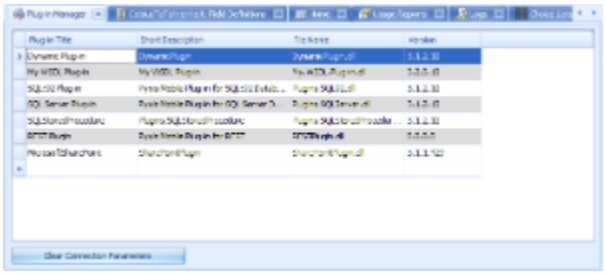
Field	Description
Proxy URL	(Required) Enter the proxy URL. This will typically be your machine name and a unique port such as 8888. This port must match the port that your proxy server is listening on and must not be used by any other services.
Username	(Optional) Enter your user name if the proxy server requires one.
Password	(Optional) Enter your password if the proxy server requires one.

Before you run your application on a device or in a simulator, be sure to reload the config if you just added the proxy information.

If the proxy is configured correctly, then you should be able to view outbound requests and inbound responses between your machine and the remote data source.

Checking Plug-in Status

To view the status and version details of all plug-ins currently included with your Verivo installation, click **Plug-Ins** on the main toolbar. The **Plug-in Manager** appears:



Plug-In file location: All plug-ins are DLL files in the plug-in folder with the other AppStudio program files. The default location is:

```
c:\Program Files\Verivo\AppStudio\Plugins
```

When upgrading Verivo plug-ins, or when the connection parameters have changed for an existing plug-in, you will need to clear both the self-described connection parameters as well as the values entered for those parameters.

To clear plug-in connection parameters:

1. In the **Plug-in Manager**, select the plug-in from the list and click the **Clear Connection Parameters** button.
2. In the **Data Source Manager**, reconnect to the data source for that plug-in (see Connecting to a Data Source).

All configuration settings for that data source are maintained.

Developing Custom Plug-ins

This section describes the following topics:

- About the Plug-in SDK
- Plug-In Application Framework
- Creating a Custom WSDL Plug-in
- Creating a Custom Plug-in
- Compiling a Custom Plug-in
- Deploying a Custom Plug-in

> Before you can create custom plug-ins, you must download and install the plug-in SDK. For more information, see About the Plug-in SDK.

About the Plug-in SDK

Before you can create custom plug-ins for the Verivo platform, you must download and expand the contents of the Plug-in SDK.

Downloading the Plug-in SDK

You can download the Plug-in SDK from the following location:

```
ftp://ftp.verivo.com/Shared/PAF v3.0 Plug-in SDK/
```

Plug-in SDK contents

The plug-in SDK includes the following resources:

Resource	Description
SampleWSDLPlugin.zip	One working and compiling sample of open source WSDL plug-in code for a plug-in available for testing integration with a Web service. The sample WSDL plug-in is useful as a reference for seeing how the MDM makes requests of the plug-in, how the requests can be parsed, and what is expected in the return message. Includes the following DLLs that are required to build WSDL plug-ins: • Plugins.WSDL.dll • Logger.dll • PAF.API.dll
WSDLLibrary.zip	Contains just the DLL files that are in the SampleWSDLPlugin.zip file.
SQLSamplePlugIn.zip	Open source plug-in code for SQL databases, with script built in to test data integration with the SQL NorthWind database. The SQL sample plug-in is useful as a reference for seeing how MDM makes requests of the plug-in, how the requests can be parsed, and what is expected in the return message.
TemplatePlugIn.zip	A basic template for writing a custom plug-in; useful as a starting point for new plug-in development. This template provides stubs for required overrides and notes key steps needed to complete the plug-in development.

PAF-API.chm	Defines the basic API for plug-in classes that you use when writing Verivo plug-ins.
Plugins.pdf	The Working With Plugins document in PDF form.

Plug-In Application Framework

The MDM delivers request messages and returns response messages that contain the requested data from a data source. The MDM sends an array of request messages to a given plug-in. Each RequestMessage is made up of connection parameters and InstructionGroup objects that tell the plug-in about instruction clauses—similar to SQL requests— involving specific data fields.

On completion of each RequestMessage, the plug-in answers with a corresponding response via the ResponseMessage object. In some cases, the MDM sends an array of RequestMessage objects in a single request, and returns a corresponding ResponseMessage object as an array.

The ResponseMessage contains one or more data tables. Each data table contains the results of a single request message, any updated connection parameters, and any unhandled InstructionGroup objects.

This section describes the following topics:

- General Requirements
- Plug-in Classes
- Plug-in Methods

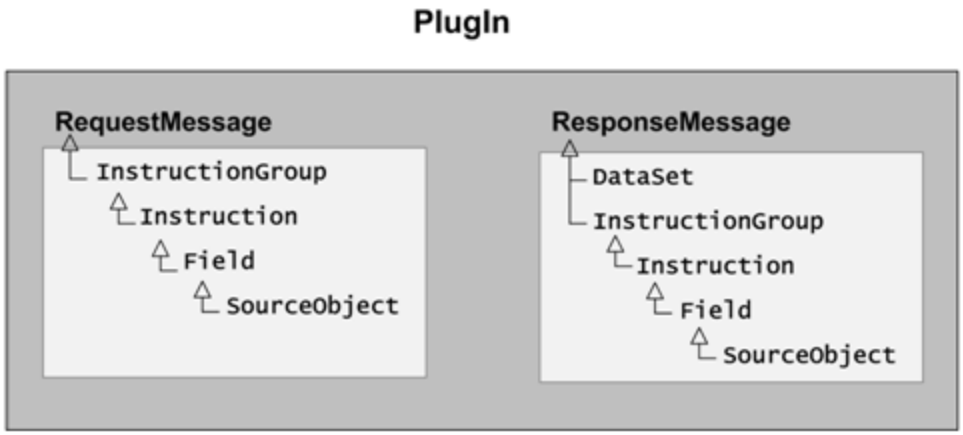
General Requirements

Plug-ins that conform to the PAF API must meet the following requirements:

- Describe the following attributes of its back-end data source:
  - Configurable connection parameters
  - Source objects
  - Required fields to send to the data source objects
  - Required fields returned from the data source objects
- Test its connection to its back-end data source.
- Accept requests from the MDM and return results back to the MDM.

Plug-in Classes

The following diagram shows the hierarchy of classes that a plug-in uses:



The following table describes these classes:

Class	Description
Plugin	The base class for all plug-ins; all plug-ins must extend this class.
RequestMessage	Describes the query or update, composed of connection parameters and InstructionGroup objects.
ResponseMessage	Contains: • One DataSet that can contain one or more DataTable objects, each containing the results of a single RequestMessage. • Any updated connection parameters. • Any unhandled InstructionGroup or Instruction.
InstructionGroup	Represents one clause of a request. An InstructionGroup also contains a list of all Instruction objects for that group. The possible clauses or `GroupLogic` property values for an InstructionGroup include: • SELECT • WHERE • JOIN • ORDER • INSERT • UPDATE
Instruction	Describes a field, including the formatting code for its data. Depending on the InstructionGroup clause in which it appears, an Instruction might also include the following: • Criteria for limiting the results of a WHERE or JOIN clause • A relationship between two SourceObjects in a JOIN clause Instruction objects contain a Field object that describes the field within the instruction. If the Instruction object describes a relationship between two SourceObjects, a second Field describes the relating field.

Field	Represents a single field or parameter of the plug-in's data source and must contain: • Formatting code for the data of that field • Reference to the SourceObject that contains it
SourceObject	Represents an object for the plug-in's back-end data source, including any of the following: • Database table, view, or stored procedure • SOAP or Web service method • API method • Any other programmatic container for data

Plug-in Methods

The main entry points for calls to plug-ins are the following methods:

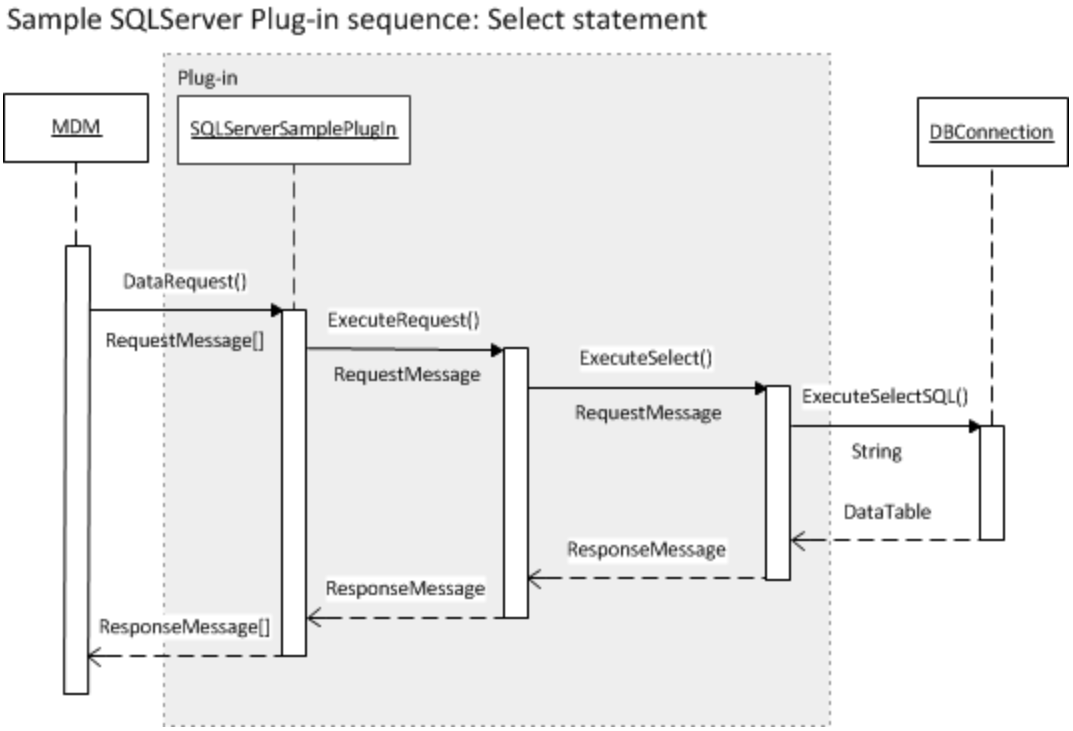
- `DataRequest()`
- `TestConnection()`
- `DescribeParams()`
- `DescribeObject()`
- `DescribeFields()`

The `DescribeParams()`, `DescribeObject()`, and `DescribeFields()` methods are entry points used by AppStudio. They are required by the PAF and are documented in the API documentation that is included in the plug-in SDK.

The `DataRequest()` method is the entry point for all requests for data within the plug-in. The `DataRequest()` me thod calls other methods for each of the different types of supported RequestType properties. These methods typically include the following:

- `ExecuteSelect()`
- `ExecuteInsert()`
- `ExecuteUpdate()`
- `ExecuteDelete()`

The following sequence diagram shows a typical call stack; in this case it is the call stack for the Select action in the Sample SQLServer plug-in:



Creating a Custom WSDL Plug-in

To implement a custom WSDL plug-in, follow most of the general processes for creating any custom plug-in, with some exceptions that are specific to the WSDL plug-ins.

The following sections describe each of these tasks:

- Create a WSDL plug-in class library project
- Edit the plug-in class
- Edit the assembly info file
- Override the PAF methods
- Deploy the WSDL plug-In

Create a WSDL plug-in class library project

The process for creating a WSDL plug-in is very similar to that of creating any custom plug-in. As a result, you should use the instructions in Creating a Class Library Project to create your WSDL plug-in's class library project, with the following differences:

1. For the **Assembly Name**, Verivo recommends naming it Plugins.*service*.dll, where *service* is your Web service name.
2. Add references to the following DLLs to your project (in addition to the PAF.API.dll):
   - Plugins.WSDL.dll
   - Logger.dll

   These DLLs are included in the plug-in SDK. Plugins.WSDL.dll contains the source code for WSDL plug-ins.
3. Import a WSDL document that includes descriptions of required properties and methods of the back-end Web service for your plug-in:

   a. Add a Web reference. Click **Project > Add Web Reference**.
   b. In the Add Web Reference window, enter the URL for the WSDL document.

The WSDL document resides with the backend Web service and is generally exposed and publicly available. To find the WSDL document, add the query string variable, 'WSDL', to the Web service URL.

> **Tip**
>
> After adding the Web reference to your WSDL plug-in project, you might want to rename it to something short as its name will be included in fully-qualified names of any objects and members when described in AppStudio.

Edit the plug-in class

When you create a new library project, Miscrosoft Visual Studio creates a default class file for you. The default name is Class1.cs, but you should change it to be the name of your service. For example, MyWSDLPlugin.cs.

1. Open the project's class file.
2. Import the mPlatform.MDM.PAF.API package with the `using` keyword:

   ```
   using mPlatform.MDM.PAF.API;
   ```
3. Change the project's namespace to conform to your company's standard; for example:

   ```
   namespace my.company.plugins
   ```
4. Extend the WSDLPlugin class:

   ```
   public class MyWSDLPlugin : WSDLPlugin
   ```

   The WSDLPlugin class is the base class for all WSDL plugins; all WSDL plug-ins must inherit from the WSDLPlugin class.
5. Override the following getters:
   - `DefaultDateFormat`
   - `DefaultTimeFormat`
   - `DefaultTimeStampFormat`
   - `GetType`

   You can use the xQuotePlugin.cs file in the SampleWSDLPlugin as a guide. The source code for the SampleWSDLPlugin is available in the plug-in SDK.
6. Override or implement any special processing for your Web service. If the WSDLPlugin class performs no unsupported actions, this may not be necessary.

   The plugins.WSDL.dll reference already implements the `ExecuteSelect()` method. The `ExecuteInsert()`, `ExecuteUpdate()`, and `ExecuteDelete()` methods are not currently implemented by the plugins.WSDL.dll reference.
7. Save your class file.

Edit the assembly info file

The assembly info file in a Visual Studio project defines the build options for a project, including the versioning, general information, and key. You also link the plug-in class into this file.

To edit the assembly info file:

1. Open the AssemblyInfo.cs file.
2. Set the value of the following attributes according to your company information:
   `AssemblyTitle`
   `AssemblyDescription`

`AssemblyCompany`
`AssemblyProduct:` set the value to mPlatform
`AssemblyCopyright`

3. Add the following assembly attribute:

   ```
   [assembly: mPlatform.MDM.PAF.API.Plugin.PluginType("plugin")]
   ```

   where *`plugin`* is the fully qualified name of the class inheriting from the WSDLPlugin class. For example, my.company.plugins.MyWSDLPlugin.
4. Save the AssemblyInfo.cs file.

Override the PAF methods

To enable the plug-in to communicate with AppStudio and the MDM, you must implement certain PAF methods. These methods include:

Method	For More Information
`DescribeParams()`	Connecting a Custom Plug-in to a Data Source
`DescribeObjects()`	Using the DescribeObjects() method
`DescribeFields()`	Using the DescribeFields() method
`TestConnection()`	Testing Connections

Deploy the WSDL plug-In

You deploy the custom WSDL plug-in the same way that you deploy any custom plug-in. You copy the plug-in's DLL to AppStudio's plug-ins directory and the AppServer's plug-ins directory. For more information, see Deploying a Custom Plug-in.

Creating a Custom Plug-in

To implement a custom plug-in, you typically perform the following tasks:

Task	For More Information
Create a plug-in class library project in Microsoft Visual Studio. This project can be based on the plug-in template or be created from scratch.	Creating a Class Library Project
Define the connection parameters that the plug-in uses to connect to the data source.	Connecting Plug-ins to a Data Source
Test the connection to the data source.	Testing Connections
Describe the objects and fields that the data source uses to define the structure of the data.	Describing the Data Source
Build queries with InstructionGroup objects.	Processing RequestMessage Objects

Create a RequestMessage object that the MDM populates with request parameters and sends to a third-party data source.	Processing RequestMessage Objects
Populate the ResponseMessage object with data that the data source returns to the MDM and return it to the AppServer.	Creating ResponseMessage objects
Open, execute, and close the connection to the database.	Using Database Connections

You can also implement server-side calls to modify the data that is sent or returned from a data source. Verivo provides a scripting framework that lets you inject custom scripts at key stages in the data request and response process. It permits you to add business logic to a datasource without having to develop or recompile code. For more information, see Introduction to Echelon.

After you implement a custom plug-in, you should also compile, deploy and test it on the AppServer. For more information, see:

- Compiling a Custom Plug-in
- Deploying a Custom Plug-in

Creating a Class Library Project

To begin writing a custom plug-in, create a new **Class Library Project** in Microsoft Visual Studio by doing the following:

1. Open Visual Studio and select **File > New >Project**.
2. Create a **Visual C# Project**, using the **Class Library** template, named appropriately for your plug-in.
3. Click on **Project > Properties**.
4. Change the **Default Namespace** to the following:

   ```
   mPlatform.MDM.PAF.API
   ```
5. Modify the **Assemby Name** to the following:

   ```
   Plugin.{name}
   ```
6. Add the PAF.API.dll as a reference DLL by clicking **Project > Add Reference** and selecting the following:

   ```
   PAF.API.dll
   ```
7. Rename the Class1.cs and the corresponding class within the file to an appropriate name for your plug-in; for example, SamplePlugin.cs.
8. Change the namespace in the SamplePlugin.cs file to the following:

   ```
   mPlatform.MDM.PAF.API
   ```
9. Ensure that your class extends the base Plugin object by using the following the class definition:

   ```
   public class SamplePlugin : Plugin
   ```
10. Open the AssemblyInfo.cs file and perform these actions:
    a. Import the namespace `mPlatform.MDM.PAF.API` by including a `using` statement for the namespace:

       ```
       using mPlatform.MDM.PAF.API;
       ```
    b. Set the following attributes according to your plug-in, company and data source:
       `AssemblyTitle`
       `AssemblyDescription`
       `AssemblyCompany`

`AssemblyProduct:` set the value to mPlatform
`AssemblyCopyright`

c. Add a new assembly attribute:

```
[assembly: Plugin.PluginType ("mPlatform.MDM.PAF.Plugin.SamplePlugin")]
```

where the value is the fully qualified name of the class.

11. Save changes to the SamplePlugin.cs and AssemblyInfo.cs file.

Describing the Data Source

The data source defines elements that are mapped to SourceObjects by the plug-in. In AppStudio, the SourceObjects are represented by Entities. SourceObjects refer to any method, table, view, stored procedure, or access point available in the data source for the given credentials.

To map data elements within the data source to Entities and Entity Fields within AppStudio, MDM calls the `DescribeObjects()` and `DescribeFields()` methods.

To implement the `DescribeObjects()` and `DescribeFields()` methods in a custom plug-in, you must have access to the details of the target data source.

Implementing the DescribeObjects() method

The `DescribeObjects()` method returns a List of SourceObjects. A SourceObject is a list of names of programmatic containers for data in the data source. For example, it could be a list of object names for a JSON service or a list of method names within a RESTful service. For a SQL-based backend, the `DescribeObjects()` method returns a list of tables in the data source. The SourceObject class is defined in the mPlatform.MDM.PAF.API package.

The following public properties can be set on each SourceObject:

- `Alias`
- `ID`
- `Name`

Not all properties are required. For example, the SQLServer Sample Plugin defines only the `ID` and `Name` properties when creating a new SourceObject.

The actual implementation of the `DescribeObjects()` method is highly dependent on the type and configuration of the data source. For an example of this method, see the SQLServerSamplePlugin.cs class in the plug-in SDK.

Implementing the DescribeFields() method

The `DescribeFields()` method returns a List of Field objects for each SourceObject. A Field is simply a name of a sub-element within a SourceObject. For example, it could be a list of properties defined by an object or the inputs and outputs of a web service method. For a SQL-based backend, the `DescribeFields()` method returns a list of column names within a table (the SourceObject). The Field class is defined in the mPlatform.MDM.PAF.API package.

The following public properties can be set on each Field object:

- `Alias`
- `ID`
- `IsSourceObjectPk`

- `MaxLength`
- `Name`
- `Required`
- `SourceFieldDataType`
- `SourceObjectRef`

Not all properties are required. For example, the SQLServer Sample Plugin defines only `ID`, `Name`, `SourceFieldDataType`, and `SourceObjectRef` when creating a new Field object. The `Alias` and `DefaultValue` properties are set when a new Field is created, but their default values are the empty string and null, respectively.

The actual implementation of the `DescribeFields()` method is highly dependent on the type and configuration of the data source. For an example of this method, see the SQLServerSamplePlugin.cs class in the plug-in SDK.

Processing RequestMessage Objects

When an application loads data, it sends a request to the MDM, and the MDM calls the plug-in's `DataRequest()`method. The MDM passes an array of RequestMessage objects to this method, each of which includes the connection parameters as well as InstructionGroup objects. The MDM also sets the value of the `RequestType` property on the RequestMessage. This property can be used by the plug-in to determine how to call the data source.

This section describes the following tasks:

- Using InstructionGroup objects to build queries
- Using SelectGroup objects
- Using InsertGroup objects
- Using UpdateGroup objects
- Using WhereGroup objects
- Using JoinGroup objects
- Using OrderByGroup objects

Using InstructionGroup objects to build queries

The InstructionGroup objects tell the plug-in what to do after it connects to the data source. For example, an InstructionGroup might consist of a SELECT, WHERE, and LIMIT clause. The `DataRequest()` method extracts these instructions and combines them into a single SQL statement that is then used to query the data source.

For each RequestMessage that the plug-in processes in the `DataRequest()` method:

1. Retrieve the InstructionGroup objects using the following sample code for a WHERE InstructionGroup as a syntactical guideline:

   ```
   InstructionGroup whereGroup = request.InstructionGroups.WhereGroup;
   ```

   You can then iterate over the properties of the group to build a data-source specific query.

2. Call the `AddUnhandeldInstruction()` method for each unhandled instruction. An unhandled instruction is an instruction that the data source does not support. For example, web services do not support JOIN or ORDER BY instructions.

   For example:

   ```
   AddUnhandeldInstruction(Instruction instruction, ResponseMessage message)
   ```

3. Build the queries with the InstructionGroup objects.

InstructionGroups are analogous to SQL clauses, with each clause defined by a separate property on child objects.

You use the properties stored in these child objects to build queries that can then be passed to the data source in a language that the data source expects.

You access each InstructionGroup by referencing the request object's `InstructionGroups` property.

For SQL-based data sources, you build a query from an InstructionGroup that complies with SQL syntax. For example, a SelectGroup could define the SELECT, FROM, and LIMIT parts of a simple SQL statement, which you can then build like this:

Building an InstructionGroup

```
StringBuilder selectClause = new StringBuilder();
StringBuilder fromClause = new StringBuilder();
string limitClause = "";
InstructionGroup selectIG = request.InstructionGroups.SelectGroup;

// Build SELECT clause
foreach ( Instruction inst in selectIG.Instructions.Values ) {
 string field = inst.sourceObjectField.toString();

 // Add a comma if there's another column name.
 if (selectClause.length > 0) {
  selectClause.Append(",{0}", field);
 } else {
  selectClause.Append(field);
 }
}

// Build FROM clause
StringBuilder select = new StringBuilder();
foreach ( Instruction inst in selectIG.Instructions.Values ) {
 string name = SourceObject so = inst.sourceObjectField.SourceObjectRef.toString();
 fromClause.Append(name);

 // Add a comma if there's another table.
 if (fromClause.length > 0) {
  fromClause.Append(",{0}", field);
 } else {
  fromClause.Append(field);
 }
}

// Build LIMIT clause
limitClause = request.InstructionGroups.SelectGroup.MaxRows.toString();

// Build the final SQL query out of the SELECT, FROM, and LIMIT clauses
string finalSQL = new string.format("SELECT {0} FROM {1} LIMIT {2}", select, from, limit);
```

For examples of building more complex SQL queries from InstructionGroup objects, see the SQLServer Sample plug-in.

You can also use InstructionGroups to build a request object for non-SQL based data sources.

Using SelectGroup objects

The SelectGroup defines one or more fields to be returned from the request.

When using the SelectGroup, adhere to the following guidelines**:**

- Each instruction within a SelectGroup contains a single field that must be returned to the MDM.
- An Instruction object contains only a reference to a single Field object for a single data source field.

- The SourceObjectField object can contain any of the following properties sent as requests from the MDM:
  - The `Alias` property specifying the name to use for the field in the ResponseMessage. This is the most important property to set.
  - `EntityFormat` and `ControlLayoutFormat` properties to format data fields.
  - An `Aggregate` property or value for aggregating this field.
  - The `EntityDataType` property denoting the data type of the field to return in the ResponseMessage.
  - The `SourceObjectRef` property containing the SourceObject for the field.
- Any default value to use for null values of the field. Any Instruction object that contains a field for which the data source cannot apply all the needed formatting should be added to a ResponseMessage object's unhandled InstructionGroup collection or object by calling the `AddUnhandledInstruction()` method. For more on adding UnhandledInstruction objects, see Using InstructionGroup objects to build queries.

> ⚠ Any formatting, such as EntityFormat or ControlLayoutFormat, that is unhandled by the data source causes an exception in the MDM. In most cases, a configuration error causes this error.

- When a field is returned in the ResponseData DataSet object within the ResponseMessage, its `DataColumn` member should be named according to the field `Alias` property and have the specified EntityDataType data type.
- The same data source field may be specified more than once in the SELECT InstructionGroup object with different aliases. Each instance of the field should return in the ResponseMessage with the appropriate alias as the `DataColumn` member.

Using InsertGroup objects

The InsertGroup defines data that will be added to the data source as part of the request.

When using the InsertGroup, adhere to the following guidelines**:**

- The InsertGroup references only one SourceObject for all its Instructions.
- No other InstructionGroup object will be set for an INSERT instruction request.
- Formatting can be present for fields in the INSERT InstructionGroup object, but aggregation will not.
- Set the `ResponseMessage.IsSuccess` property in the ResponseMessage object to indicate success or failure of the request.

Using UpdateGroup objects

The UpdateGroup defines data that replaces existing data in the data source.

When using the UpdateGroup, adhere to the following guidelines**:**

- The UpdateGroup references only one SourceObject for all its Instructions.
- No JoinGroup objects are set for an UPDATE instruction request.
- A WhereGroup object is set.
- Formatting can be present for fields in the UPDATE InstructionGroup, but not aggregation.
- Set the `ResponseMessage.IsSuccess` property in the ResponseMessage object to indicate success or failure of the request.

Using WhereGroup objects

The WhereGroup defines input parameters or WHERE clause query instructions for the request.

When using the WhereGroup, adhere to the following guidelines**:**

- WhereGroup objects contain all WHERE instructions for all source objects in a given RequestMessage.
- Each Instruction in a WhereGroup contains an `Alias` property that corresponds to its place in the InstructionGroup's logic string, named `LogicString`. The `LogicString` property contains the AND/OR logic applied to the request to restrict the results.
- A WhereGroup object contains the following:
  - A reference to a single Field object.
  - An `operationType` comparison operation.
  - A `CompValue` value to compare to the field.
- The SourceObjectField object in a WhereGroup can have all formatting described in the SELECT instruction applied to it.
- If a data source cannot apply the WHERE Instruction or if it does not perform the operation requested in the `operationType`, the plug-in should do both of the following:
  - Add all InstructionGroup and the Instruction objects to the ResponseMessage object's unhandled InstructionGroup collection by calling the `AddUnhandledInstruction()` method. For more on adding UnhandledInstruction objects, see Processing RequestMessage Objects.
  - Ensure that the values for the SourceObjectField object used in instruction return to the caller, either by adding a new Instruction referencing the SourceObjectField to the SELECT InstructionGroup, or by some other method.

Using JoinGroup objects

The JoinGroup links data containers in the data source to each other so that you can construct more complex queries.

When using the JoinGroup, adhere to the following guidelines**:**

- The JoinGroup objects are used for all CROSS, INNER, or LEFT outer joins.
- A single JoinGroup object describes the relationship between two and only two SourceObjects.
- Each Instruction within the JOIN InstructionGroup contains a single relationship between the two SourceObjects, or a single condition by which one of the SourceObject's results should be limited.
- Currently, all Instruction objects in a JOIN InstructionGroup object are joined together using an AND function: a rule reinforced by the InstructionGroup object's `LogicString`.
- Operations performed by an Instruction must do one of the following:

  a. **Relate** the two SourceObjects involved in the InstructionGroup:
     - Two fields will be referenced in the Instruction**:** `SourceObjectField` and `CompValue.`
     - The `CompValueType` value will equal FIELD_REF value.
     - Each field can have source level formatting or aggregation.

  a. **Equate** a Field property in one of the two SourceObject objects in the InstructionGroup object to a constant value:
     - The `SourceObjectField` property could be a member of either SourceObject involved in the JOIN Instruction object.
     - The Field can have source-level formatting or aggregation.
     - Set the value of the `CompValueTypeCD` property to `LITERAL`.

  a. **Relate** one SourceObject to a DataTable passed into the plug-in in the RequestMessage object's `VirtualSourceObjects` property:
     - The `SourceObjectField` property in the Instruction will be a member of a SourceObject in the data source.
     - The value of the `CompValue` property must contain the name of the table and column to use in the relation.
     - The format of the `CompValue` property will be syntactically explicit:

`{DataTable name}.{DataColumn name}`

- Set the `CompValueType` property to DATATABLE.
- The DataRow collection within the DataTable contains the values to use for relating to the `SourceObjectField` property in the Instruction.

- If a data source cannot apply the JOIN instruction, or if it does not perform the `operationType`property requested, the plug-in should:
  - Add all InstructionGroup and Instruction objects to the ResponseMessage object's unhandled InstructionGroup collection by calling the `AddUnhandledInstruction()` method. For more on adding UnhandledInstruction objects, see Processing RequestMessage Objects.
  - Ensure that the values for the `SourceObjectField` property used in instruction return to the caller, either by adding a new Instruction referencing the SourceObjectField to the SELECT InstructionGroup, or by some other method.

Using OrderByGroup objects

The OrderByGroup defines the sort order of the data returned from the request.

When using the OrderByGroup, adhere to the following guidelines**:**

- Each instruction within an OrderByGroup object contains a single field by which the results should be ordered.
- The Instruction contains only a single Field object for a single data source field.
- The Instruction contains an `Order` property. This property contains the zero-based offset for the position of the field in the ORDER InstructionGroup.
- Set the `CompValueTypeCD` property to `LITERAL`.
- The SourceObjectField object can contain any of the following properties sent as requests from MDM:
  - `EntityFormat` and `ControlLayoutFormat` properties to format data fields.
  - An `Aggregate` property or value for aggregating this field.
  - The `SourceObjectRef` property containing the SourceObject for the field.
  - Any default value to use for null values of the field.
- Any Instruction containing a field for which the data source cannot apply all the needed formatting. Add all InstructionGroup and the Instruction objects to the ResponseMessage object's unhandled InstructionGroup collection by calling the `AddUnhandledInstruction()` method. For more information about adding UnhandledInstruction objects, see Processing RequestMessage Objects.
- The same data source field may be specified more than once in the ORDER InstructionGroup with different aliases.

Connecting a Custom Plug-in to a Data Source

To connect to the data source, the plug-in creates a hashtable that contains the connection parameters. These parameters typically include the server name on which the data source resides, the data source name, user ID, password, and default connection timeout.

An AppStudio developer enters the required parameters when setting up the data source in the Data Source Manager's Connection Settings. AppStudio generates the hashtable from these settings and passes them to the plug-in when the developer clicks the Test Connection button. These parameters are also passed by the MDM to the plug-in when an application makes a call to a data source.

In the plug-in, you define the connection parameters that the developer is prompted to enter. You can optionally add validation logic to prevent unnecessary connections to a database.

This section describes the following topics:

- Defining connection parameters
- Validating connection parameters

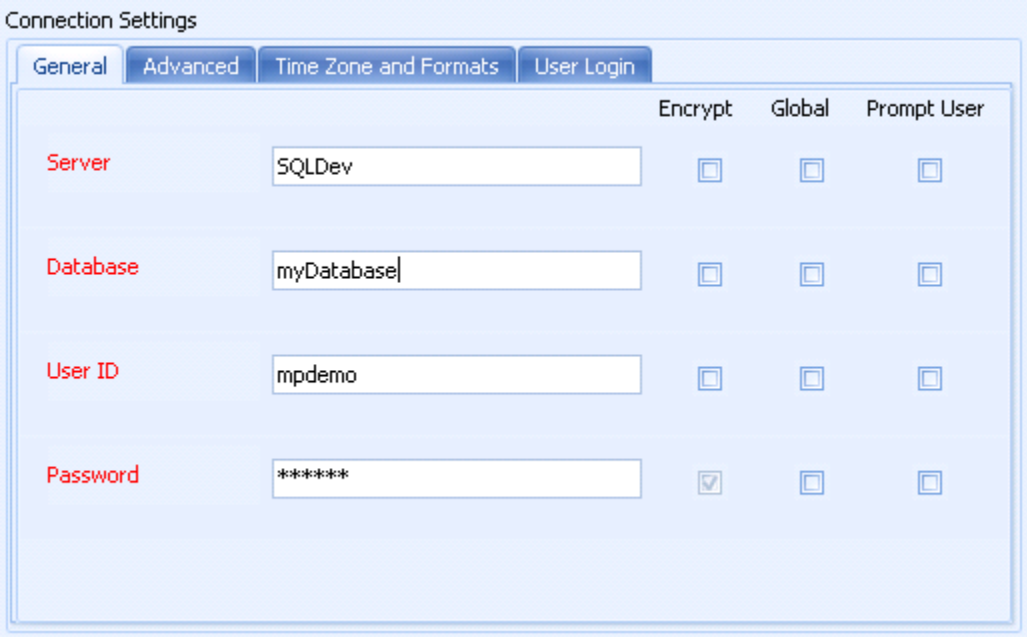
Defining connection parameters

Connection parameters let the plug-in authenticate with the data source. They can also include other optional information.

You define the connection parameters that a custom plug-in supports in the plug-in's constructor. You can specify that some connection parameters are required for authentication and that some are optional. Optional parameters could be something like a timestamp or other information that the data source might use for logging or error checking.

If the data source is a stored procedure, then it might require an additional set of authentication parameters that are separate from the plug-in's required authentication parameters.

In AppStudio, the application developer enters the connection parameters in the Data Source Manager. The following example shows a typical form:



The required authentication parameters are in red.

To get information about which parameters are available, the MDM calls the plug-in's `DescribeParams()` method when the developer configures a plug-in in AppStudio. This method returns all parameters, whether they are required for authentication or not, so that AppStudio can render the Data Source Manager Connection Settings. The developer can then enter values for the parameters.

To define how AppStudio accepts input for the connection:

1. Create two getters in the plug-in: one for required parameters (`m_RequiredAuthParams`) and one for all parameters (`m_AllAuthParams`). The `m_AllAuthParams` property should include the `m_RequiredAuthParams` array, plus any additional parameters that are *not* required.
2. In the `DescribeParams()` method, return the `m_AllAuthParams` array; or return the `m_RequiredAuthParams` if all connection parameters are required.

The `m_AllAuthParams` and `m_RequiredAuthParams` properties each define an array of objects of type

ConnParam. Each object in these arrays is equivalent to a field on the Data Source Manager Connection Settings screen. The type of objects in the array defines how the input control is displayed and what types of input it can take.

The following table describes the types of objects that you can use:

ConnParam Type	Description
`StringConnParam`	Stores strings. When a connection parameter is of type StringConnParam, then AppStudio displays a text input field.
`NumericConnParam`	Limits the input to any number of type double.
`PasswordConnParam`	Similar to the StringConnParam, but this type is exclusively for password parameters. AppStudio hides the value of this parameter with asterisks when a developer enters the value of this parameter in the text input field.
`BooleanConnParam`	AppStudio displays a checkbox for this value. The value of this parameter is a 0 or a 1, depending on whether the box is checked (1) or unchecked (0).
`TimeZoneConnParam`	Used for time zone parameters.
`DropDownConnParam`	Limits the selection of string connection parameter values to a discrete list of items.
`EnumConnParam`	Limits the selection of listed values to an Enum.

All ConnParam types take the same attributes in their constructors. The signature of the constructor is as follows:

`new` *`Type`*`ConnParam(`*`Name`*`:string, `*`Value`*`:string, `*`ParamOrder`*`:int, `*`Required`*`:boolean, `*`IsHidden`*`:boolean, `*`ShortDesc`*`:string)`

The following table describes those attributes:

ConnParam Property	Description
`Name`	A String that defines the name of the parameter. This value appears as the label in AppStudio, but is also used by the plug-in.
`Value`	The value of the parameter that you want to pass.
`ParamOrder`	An int that defines the order in which this parameter appears in the Data Source Manager.
`Required`	A Boolean that determines whether or not the connection parameter is required. If this is set to true, then the AppStudio displays it's label in red and will not let the developer test the connection until some value has been entered for this parameter.

`IsHidden`	A Boolean that determines whether the connection parameter should be visible to the AppStudio developer. Set to true to hide the parameter from the Data Source Manager. Set to false to ensure that the parameter shows up in the Data Source Manager. This is useful for passing information that the developer either should not know about or does not need to know about, such as a timestamp or application-specific value for logging.
`ShortDesc`	A String that describes the connection parameter. This description shows up as tool tip text in AppStudio.

Required authentication parameters should be defined in the `m_RequiredAuthParams` property of the plug-in. Additional connection parameters should be defined in the `m_AllAuthParams` property.

The following example shows getters for the `m_AllAuthParams` and `m_RequiredAuthParams` properties in the plug-in's constructor:

Connection Parameters

```
public ConnParam[] REQUIRED_AUTH_PARAMS
{
 get
 {
  if ( this.m_RequiredAuthParams == null )
  {
   this.m_RequiredAuthParams = new ConnParam[5] {
    new StringConnParam("DataSource", "", 0, true, false, "The network name of this SQL server.",
ConnParam.CATEGORY_GENERAL, "Data Source"),
    new StringConnParam("Catalog", "", 1, true, false, "The database to connect to on this server",
ConnParam.CATEGORY_GENERAL),
    new StringConnParam("UserID", "", 2, true, false, "The SQL server UserID to login with",
ConnParam.CATEGORY_GENERAL, "User ID"),
    new PasswordConnParam("Password", "", 3, true, false, "The SQL server Password to login with",
ConnParam.CATEGORY_GENERAL),
    new NumericConnParam("Timeout", 60, 4, true, false, "The timeout time in seconds on connection
open attempt", ConnParam.CATEGORY_GENERAL, 0, null)
   };
  }
  return this.m_RequiredAuthParams;
 }
}
public ConnParam[] ALL_AUTH_PARAMS
{
 get
 {
  if ( this.m_AllAuthParams == null )
  {
   this.m_AllAuthParams = this.REQUIRED_AUTH_PARAMS;
  }
  return this.m_AllAuthParams;
 }
}
```

The following simple `DescribeParams()` method returns only the required authentication parameters for a data source:

DescribeParams()

```
protected override List<ConnParam> DescribeParams()
 {
  return new List<ConnParam>(this.REQUIRED_AUTH_PARAMS);
 }
```

For information on how to test the connection in AppStudio, see Connecting Plug-ins to a Data Source.

Validating connection parameters

A plug-in should validate the required authentication parameters so that it can short-circuit any invalid calls. At the least, the plug-in should ensure that the minimum number of required parameters is being passed. If one or more of the required parameters is not passed, then the plug-in can refuse to attempt a connection to the data source, reducing network traffic.

To validate that the right number of required authentication parameters were passed and that they were properly named, construct an array of parameters in the custom plug-in's `ValidateAuthParams()` method and pass it to the base PlugIn class's `ValidateAuthParams()` method, as the following example shows:

ValidateAuthParams()

```
public override bool ValidateAuthParams(Dictionary<string,string> AuthParams)
{
 string [] reqParams = new string[this.REQUIRED_AUTH_PARAMS.Length];
 for ( int i = 0; i < this.REQUIRED_AUTH_PARAMS.Length; i++ )
  reqParams[i] = this.REQUIRED_AUTH_PARAMS[i].Name;
 return this.ValidateAuthParams(AuthParams, reqParams);
}
```

If one or more parameters are invalid, the plug-in throws a PluginAuthFailure exception. This class is defined in the mPlatform.MDM.PAF.API package.

Using Database Connections

Custom plug-ins most commonly connect to a data base. This section describes how to create a database connection, open the connection, query the database, and close the connection in the following topics:

- Opening a connection to the database
- Executing the query
- Closing the database connection

Opening a connection to the database

When the plug-in receives a connection request from the MDM, it attempts to open a connecting to the back-end data source. In most cases, the back end data source is a database. In this case, you can use the DBConnection class in the mPlatform.MDM.PAF.API package. This class defines methods and properties that connect, query, and disconnect from a database.

To open a connection to a database, you must first instantiate a DBConnection object. You pass a connection string to the DBConnection's constructor. The connection string is made up of your connection parameters and arranged in a way that the data base expects. For example, SQLServer expects a connection string that matches the following

format:

`provider=SQLOLEDB;data source=`*`datasource`*`;initial catalog=`*`catalog`*`;User ID=`*`user_id`*`;Password=`*`password`*`;Connect Timeout=`*`timeout`*

The following example instantiates a new DBConnection by calling its constructor:

```
public string ConnectionString {
    get {
        return string.Format(
            "provider=SQLOLEDB;data source={0};initial catalog={1};User ID={2};Password={3};Connect
Timeout={4}",
     this.BackendAuthHashtable["DataSource"],
     this.BackendAuthHashtable["Catalog"],
     this.BackendAuthHashtable["UserID"],
     this.BackendAuthHashtable["Password"],
     this.BackendAuthHashtable["Timeout"]);
    }
}


try {
    this.myConnection = new DBConnection(this.ConnectionString);
} catch (ArgumentException e) {
    throw new PluginAuthFailure("Could not create database connection due to connection string.",
this.InstanceID, this.BackendAuthHashtable, e);
}
```

To optimize your plug-in, you can check your connection parameters prior to attempting to create a new DBConnection object. For more information, see Validating connection parameters.

After successfully instantiating a new DBConnection, you can then call its `Open()` method to open a connection to the database. For example:

```
myConnection.Open();
```

Executing the query

After building your query, you can then call the DBConnection's execute methods against the database. If your query is a SELECT statement (and requires response data), then you call the DBConnection's `ExecuteSelect()` method. The method returns a DataTable as a response. For example:

```
DataTable dt = myConnection.ExecuteSelectSQL(myStatement);
```

For INSERT, UPDATE, and DELETE statements, call the DBConnection's `ExecuteTransaction()` method. This method does not require any response data, as the following example shows:

```
myConnection.ExecuteTransaction(myStatement);
```

For full examples of various `ExecuteTransaction()` methods, see the SQL Server sample plug-in. This sample is available in the plug-in SDK.

Closing the database connection

Finally, you close the database connection by calling the DBConnection's `Close()` method.

```
myConnection.Close();
```

You should close the connection whether the query executed successfully or not, and whether or not the query returned any data.

Creating ResponseMessage objects

The plug-in creates a ResponseMessage object from DataTable objects that are returned from the data source in the `DataRequest()` method. The plug-in returns the result of a request to the MDM, and includes any updated authentication properties, unhandled requests, or exceptions.

This section describes the key members of a ResponseMessage object and provides guidelines for writing a ResponsMessage object.

> ⚠ Each ResponseMessage in the ResponseMessage array corresponds to a RequestMessage in the RequestMessage array.

The following table specifies the key members of the ResponseMessage class:

Property	Description
`IsSuccess`	Boolean indicating if the request succeeded. For more information, see Setting the IsSuccess property on a ResponseMessage.
`ResponseData`	A DataSet object that contains the results from the request. For more information, see Parsing response data.
`BackendAuthHashtable`	A Dictionary object that contains updated or changed authentication parameters.
`InstructionGroups`	A collection of any unhandled Instruction or InstructionGroups classes. For more information, see Defining unhandled InstructionGroups in the ResponseMessage.
`RespException`	An Exception that was thrown during the request processing. For more information, see Handling Exceptions in the ResponseMessage.

Setting the IsSuccess property on a ResponseMessage

To indicate to the application that the request for data was successful, you must set the `IsSuccess` property of the ResponseMessage to true before returning it to the MDM. In the SQLServer sample plug-in, this is done in the `ExecuteSelect()` method just before the response is returned.

The following example sets this property upon successful completion of a SQL data source query:

Setting isSuccess Example

```
response = new ResponseMessage();
try {
 DataTable dt = SqlCon.ExecuteSelectSQL(finalSQL);
} catch (PluginException pe) {
 this.PrintToLogger(LogLevel.ERROR, string.Format("{0}: Error processing data request: {1}",
this.PluginTitle, pe.Message));
 throw pe;
} catch (Exception e) {
 this.PrintToLogger(LogLevel.ERROR, string.Format("{0}: Error processing data request: {1}",
this.PluginTitle, e.Message));
 throw new DataSourceError("Unable to process data request", this.InstanceID, e);
}
response.ResponseData = ds;
response.IsSuccess = true;
return response;
```

Parsing response data

The data returned from the data source to the plug-in is set on the `ResponseMessage.ResponseData` property.

Whether or not there is any response data depends on the value of the `RequestType` property of the RequestMessage object. For example, if the request was an INSERT or an UPDATE, then there is no data to return so the `ResponseData` property should not be set.

The following table describes the actions required to properly set the value of the `ResponseData` property based on ReturnType:

ReturnType	Action on ResponseData
SELECT	1. Populate the ResponseData object's DataSet with the results, so that each DataTable in the DataSet is named according to the SourceObject results that it contains, specifically: • The DataTable class name consists of the SourceObject ID property wrapped in brackets ([,]). • If multiple SourceObjects are involved in the result, as is the case of a JOIN having been performed, all SourceObject IDs should be included, each wrapped in brackets and delimited by a comma. • The `Plugin.StringBuilder()` method aids in creating the final DataTable name. 2. Populate the `ResponseData` property with the results, so that if the data source does not perform JOINs and JOIN InstructionGroup classes were present in the ReqestMessage class, multiple DataTable classes should be present in the ResponseData property. Specifically, each DataTable class should be named with the bracketed ID of the SourceObject class involved in populating the DataTable class. 3. Populate the `ResponseData` property with the results, so that the DataColumns in the DataTable are named after the `Field.Alias` property, and have the DataType specified by the `Field.EntityDataType` property, for the Field associated with that DataColumn**.** The same data source field can have multiple Field objects requested in the SELECT InstructionGroup object, but each Field object requested must be returned as its own.
INSERT | UPDATE | DELETE	No ResponseData object result is needed.

Exception	1. Set the `ResponseMessage.IsSuccess` property to false. 2. Set the value of the `RespException` property to the one of the exceptions that were encountered during the request.

Populating a ResponseMessage with properly-formatted data can be very challenging. For an example of doing this with data returned from a SQL source, see the SQLServer sample plug-in.

Defining unhandled InstructionGroups in the ResponseMessage

In some cases, a data source cannot apply an Instruction such as a JOIN operation or specific data formatting. When this happens, the plug-in should return the unhandled instruction in the ResponseMessage. Plug-ins do this by calling the `AddUnhandledInstruction()` method. This method adds the relevant InstructionGroup and Instructions to the ResponseMessage object's unhandled InstructionGroup collection.

For an example of when to call the `AddUnhandledInstruction()` method, see the SQL Server sample plug-in.

Handling exceptions in the ResponseMessage

When an exception is thrown, set the `ResponseMessage.IsSuccess` property to false and the `RespException` property to the text of the exception.

A plug-in should not perform operations that the back-end data source does not have. For example, a plug-in should not perform JOIN operations or specific data formatting if the back-end data source does not. When populating responses, throw the appropriate exception.

The following example is called in the `try...catch` block of the `DataRequest()` method:

Handling Exceptions

```
try
{
 this.InitializePluginPerRequest(requests[i]);
 responses[i] = this.ExecuteRequest(requests[i]);
 this.PrintToLogger(
  LogLevel.DEBUG,
  string.Format("{0}: successfully executed request[{1}] of {2}. Sending response.",
  this.PluginTitle,
  i,
  requests.Length) );
}
 catch(Exception e)
{
 responses[i].IsSuccess = false;
 responses[i].RespException = e;
 this.PrintToLogger(
  LogLevel.ERROR,
  string.Format("{0}: Exception encountered executing request[{1}] of {2}. The exception is {3}",
  this.PluginTitle,
  i,
  requests.Length,
  e.Message) );
}
```

Logging Plug-in Activity

A key component to troubleshooting and testing custom plug-ins is to ensure that they log all activity with the MDM

and the external data source.

To log plug-in activity, use the `PrintToLogger()` method. This method is defined on the base Plugin class. It takes the following arguments:

- `logLevel`– An int that defines the type of event. Valid values are defined by the following constants:
  - `LogLevel.DEBUG`
  - `LogLevel.ERROR`
  - `LogLevel.FLOW`
  - `LogLevel.INFORMATION`
  - `LogLevel.NONE`
- `msg` – A String the describes the event. You typically build this string as a combination of situation-specific information and variables available to the exception handler.

Messages that are logged with the `PrintToLogger()` method show up in the AppStudio logging screen. AppStudio adds a timestamp to the information.

The following example uses several types of messages:

Logging

```
try
{
 ...
 // Success!
 this.PrintToLogger(LogLevel.DEBUG, string.Format("{0}: successfully described parameters.",
this.PluginTitle));
}
catch (PluginException pe)
{
 // Log a Plugin error
 this.PrintToLogger(LogLevel.ERROR, string.Format("{0}: Error describing required parameters: {1}",
this.PluginTitle, pe.Message));
 ...
}
catch (Exception e)
{
 // Log an application error
 this.PrintToLogger(LogLevel.ERROR, string.Format("{0}: Error describing required parameters: {1}",
this.PluginTitle, e.Message));
 ...
}
```

Testing Connections

To let developers test the connection between the plug-in and the data source, override the `TestConnection()` method. An AppStudio developer can call this method by clicking the Test Connection button in the Data Source Manager.

When AppStudio calls the `TestConnection()` method, it passes the connection parameters and returns true if the connection succeeds or false if it fails. In many cases, the `TestConnection()` method also logs activity for debugging purposes. For example:

```
public override bool TestConnection() {
    try {
        return this.OpenConnection();
    } catch {
        this.PrintToLogger(LogLevel.ERROR, string.Format(this.logMessages.GetString("300000"),
"300000"));
        return false;
    } finally {
        this.CloseConnection();
    }
}
```

The `TestConnection()` method should:

- Return true if the connection succeeds; otherwise false.
- Close the connection after completing the test.
- Not return an exception, except in critical events.

Compiling a Custom Plug-in

To compile your custom plugin, you must add the plugin libraries to your project. Typically, you do this when you set up your project.

1. Determine the location of the required plug-in libraries, installed as part of the AppServer installation.
2. Add the location of those libraries to your custom plug-in project.

The sample SQL plug-in and the plug-in project template included with this SDK assume these plug-in library locations. If your Verivo server is installed in an alternative location, adjust your project to specify the location of your plug-in libraries.

Deploying a Custom Plug-in

The ApServer requires access to the plug-in so that it can use it during the request/response workflow. AppStudio requires access to the plug-in so that it can map entities to data fields within the remote data source.

As a result, when you create a custom plug-in, you must deploy it on both the AppServer and on each workstation that is running AppStudio.

To install the plug-in on the AppServer:

1. Determine the location of the "plugins" directory; for example:

   `C:\Inetpub\wwwroot\verivo\bin\plugins`
2. Copy the plug-in's DLL file to the "plugins" directory.
3. Restart the AppServer.

To install the plug-in for AppStudio:

1. Determine the location of the "Plugins" directory; for example:

   `C:\Program Files\Pyxis Mobile\Application Studio\Plugins`
2. Copy the plug-in's DLL file to the "Plugins" directory.
3. Restart AppStudio, if it is already running.
4. Repeat this process for each workstation on which AppStudio is running.