16 - Interfacing to External Programs

16.1 Overview

Corvid systems can interface to resources and programs external to the Corvid Runtime program to obtain data, perform special actions and integrate into an overall IT environment.

There are a variety of places that a Corvid system can call external resources. Usually this is done to obtain data, such as reading from a database, but can involve many other actions and types of programs. External interface commands can be added to:

- Get the value for a variable(s) at Runtime
- Get the value list for a Dynamic List variable
- Get initialization values for a Collection Variable
- Set the Prompt text for a variable
- Set the text for a Static List variable value(s)
- Execute a command after the end user provides input
- Execute a External Interface command in a Logic or Command Block
- Obtaining a MetaBlock data file

All of these have a similar interface to build the associated command. The types of commands that can be used are:

- URL: Calling a URL or local file to perform actions and return data (Section 16.4)
- XML: XPath commands to read data (Section 16.5)
- Database: SQL Commands to read or write data (Section 16.6)
- PARAM: Reading data from the Applet tag (Section 16.7)
- Applet: Calling a Java Applet to perform actions and return data (Section 16.8)

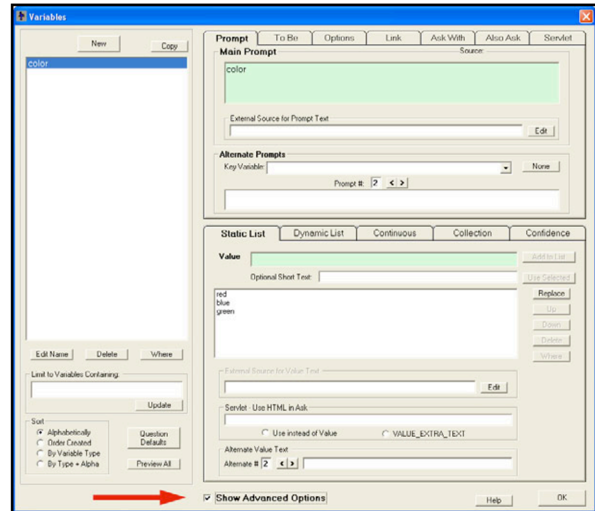
16.2 Where to Use External Interface Commands

There are various places in the Corvid Development environment where external interface commands can be added. In general, any type of external interface command that returns the correct type of data can be used.

Whenever an external interface command can be used, there is an associated “Edit” button, which will display a window to build the command with tabs for XML, Database, Applet, Parameter and URL external interface options. The external interface commands are displayed in text boxes next to the “Edit” button. It is possible to directly edit the command in the edit box, but since most commands have fairly complicated syntax, it is much easier to build/edit the command by clicking the “Edit” button.

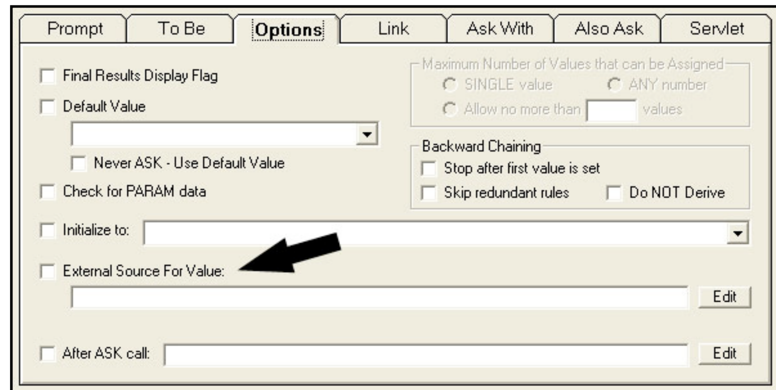
Note: Most external interface commands are found on the Variables window and are considered “Advanced Options”.

To enable these command options, make sure the “Advanced Options” check box is selected at the bottom of the variables window.



Setting the Value for a Variable

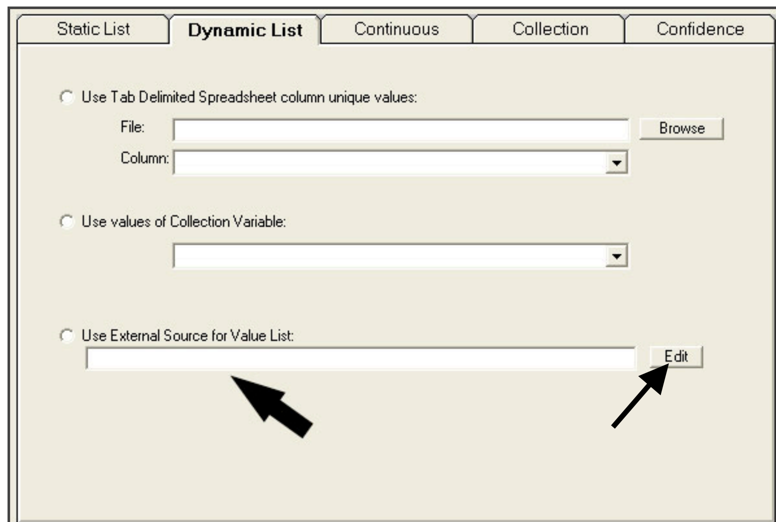
This is the most common use of external data sources. In the Variables window, go to the Options tab. Click the “Edit” button next to “External Source for Value” and build a command.



When the value of the variable is needed, rather than asking the end user, the external source for the data will be used. This allows systems to automatically obtain some, or all, of their values from external sources. The external source/program called must return a value for at least the variable associated with the command, and can return data for other additional variables. The returned value must be consistent with the variable type (e.g. a numeric variable needs to have a numeric value returned, a static list needs to have a value number or short text returned)

Setting the Value List for a Dynamic List Variable

In the Variables window, select a Dynamic List variable. In the “Dynamic List” tab, one of the options for the value list is an external source. Click the “Edit” button next to the “Use External Source for Value List” and build a command.



When the Dynamic List variable is asked or included in a report, the associated value list will be obtained from the external source.

Initializing a Collection Variable

In the Variables window, select a Collection variable. In the “Collection” tab, one of the options is to initialize the value list from an external source. Click the “Edit” button in the “Preload from an External Source at Runtime” and build the command.

Collection variables preloaded with values from an external source will call the external source to initialize the collection before the variable is used in the system. This can be convenient to provide starting content for a report, or to load the collection with items that will be analyzed in by the Corvid system. The value returned from the external program can be a single string, or multiple strings separated by a tab or line feed character. When there are multiple strings, they will be assigned in order as multiple items in the collection value list. A single string will be assigned as a single value in the collection value list.

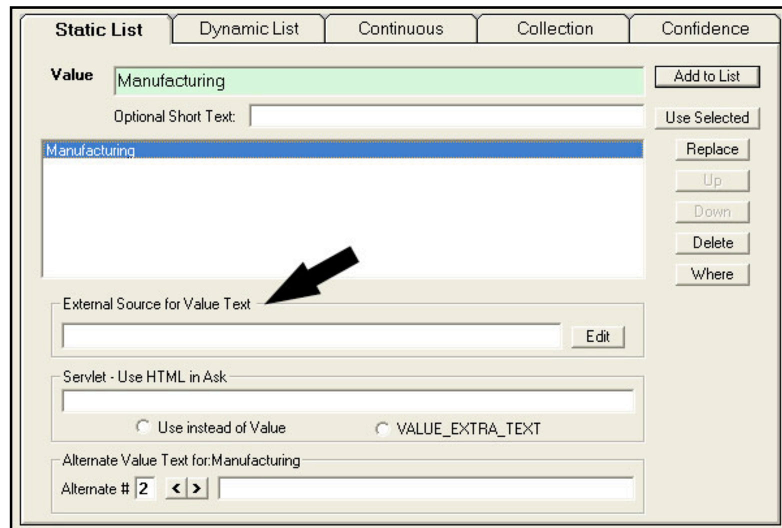
Prompt Text for a Variable:

In the Variable Window, select the Variable that will get its Prompt from the external source. Go to the Prompt tab. Click “Edit” in the “External Source for Prompt Text” and build a command.

When the prompt text for the variable is needed to ask the user a question or to include the variable in a report, the text will be obtained from the external source. The external source should return a single string for the associated variable.

Systems that run in multiple languages, or need to have prompts that change dynamically, can use an external source for the Prompt text. XML can be a convenient way to handle systems with multiple prompts. Corvid also provides internal support for up to 5 prompts based on a key variable, and resource files are often a more convenient way to handle systems that run in multiple languages.

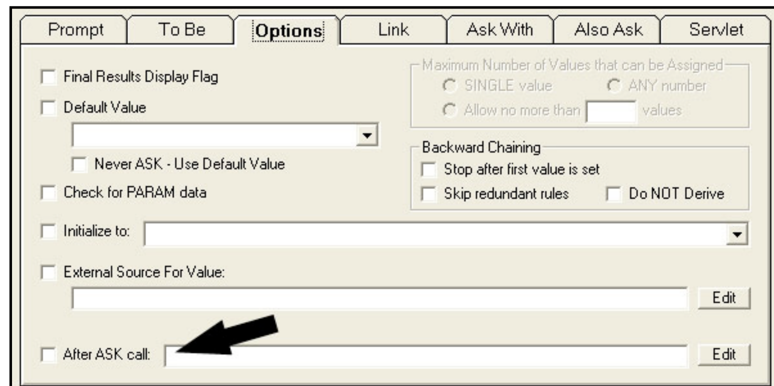
Static List Variable Value Text



In the Variable Window, select a Static List Variable. On the Static List tab, select a value. An external source can be specified for each value individually. Click “Edit” in the “External Source for Value Text” section and build the command.
When the variable value text is needed to ask the user a question or to include the variable in a report, the value text will be obtained from the external source. The external source should return a single string for the associated value.

Systems that run in multiple languages, or need to have value text that change dynamically, can use an external source for the text of the values. XML can be a convenient way to handle systems with multiple value texts. Corvid also provides internal support for up to 5 value strings based on a key variable, and resource files are often a more convenient way to handle systems that run in multiple languages.

After Ask Calls

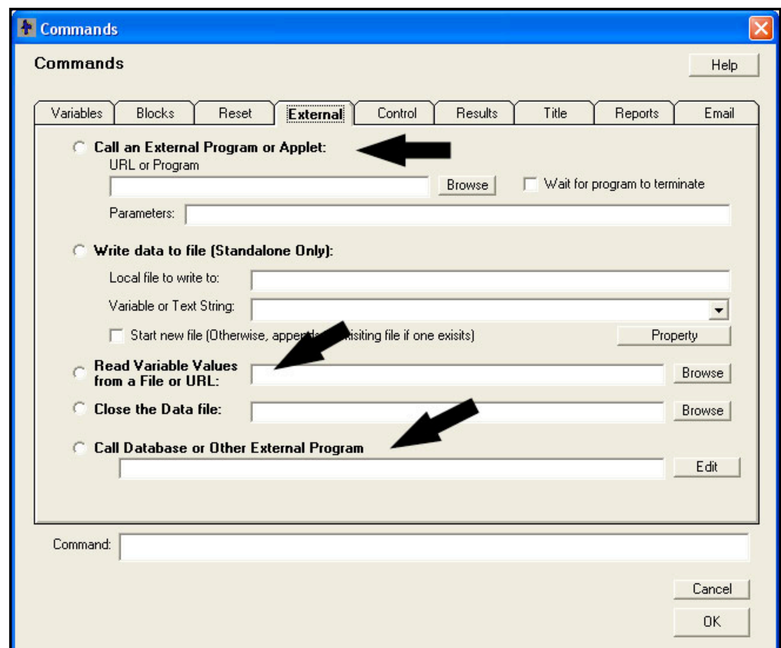


In the Variables window, go to the Options tab. Click the “Edit” button next to “After ASK call” and build a command.

The “After Ask” command is executed after the end user is asked to provide the value for the variable. Normally this is used to write out user’s input to a database either to log the users interaction, or to save the “state” of a session so that the user can exit and return to the system later. After Ask commands are limited to commands that can write data either to a database or other external program.

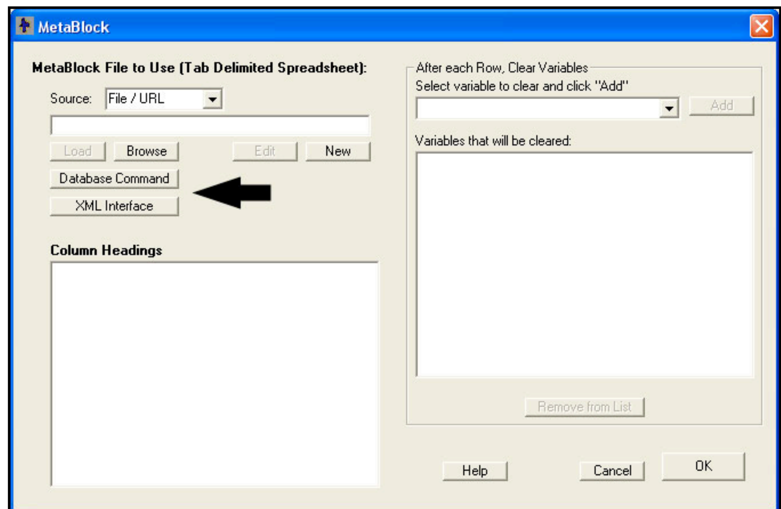
External Interfaces in Commands

In addition to the options for using external interface commands with variables, they can also be used as commands, either in a Command Block or Logic Block. When building a command in the Command Builder window, the “External” tab provides various options for external commands.



External Sources for MetaBlock Data

When building MetaBlock systems, the MetaBlock data file is normally a static tab delimited spreadsheet file, however, such systems can incorporate dynamic data by using an external source for the MetaBlock data.   This is done from the MetaBlock window, which allows adding database and XML sources for the data.

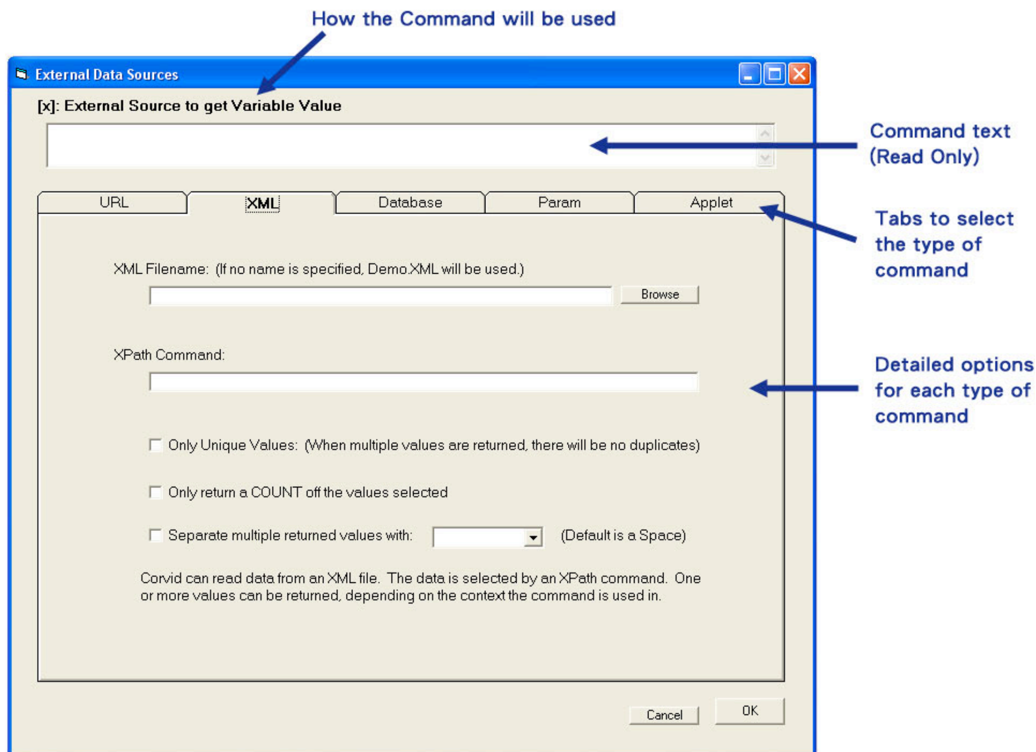


16.3 Types of External Data Commands

Exsys Corvid supports 5 types of external interface commands:

External Interface Command Window

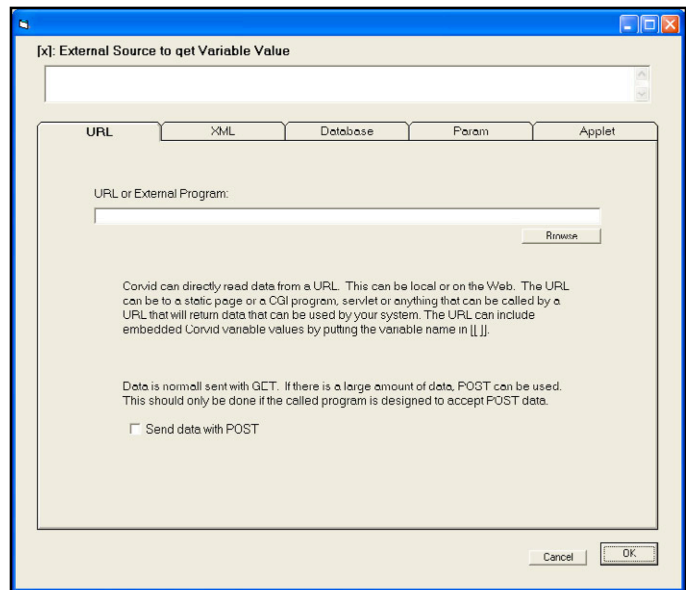
All of the external data commands are built from the External Data Sources window.   This is displayed by clicking the "Edit" button next to any of the options that can use external data sources.



- This window displays how the external command will be used, such as "[X]: External source to get variable value".
- The text of the command in an edit box. This text is read only and cannot be directly edited. All command parameters and changes are entered in the edit boxes under each command type tab. This makes it easier to build commands and reduces the chance for syntax errors.
- The tabs allow selecting the various types of commands.
- Under each tab are the options for building that type of command.

For most options using external data, any of the command types can be used. However, system architecture and delivery mode will determine what is best for a particular application. The commands "Param" and "Applet" can only be used with the Corvid Applet Runtime. Some command types require additional server programs or resources.

16.3.1 URL / External Program



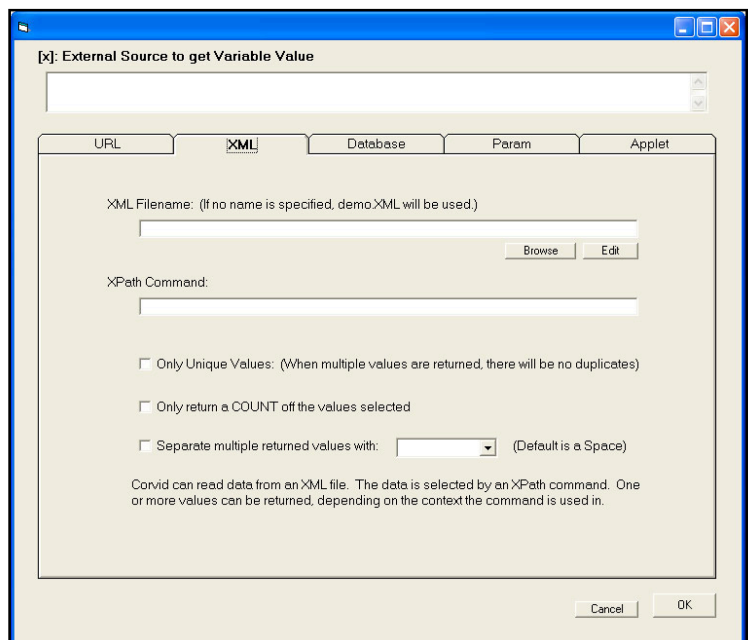
URL commands can be used to read data from any source that can be referenced by a URL, and can also be used to call local programs.

The source of the data can be a simple text file stored locally, or the URL of a server program (Java Servlet, CGI program or any other dynamic resource that can be accessed by a URL) that will return data. The URL can include embedded Corvid variables (using double square brackets) allowing the URL to be set dynamically, and allowing Corvid data to be passed to the program. By default, the URL approach uses GET to send small amounts of data, but POST can also be used when larger amounts of data need to be sent.

The URL approach is often a very quick, easy and reliable way to get external data into a Corvid session. External server programs can be created to add special functionality to Corvid as needed.

For command details see Section 16.4

16.3.2 XML



Corvid's XML interface allows reading item(s) of data from XML files using XPath commands. XML provides a powerful and standard way to make complex data structures available to the Corvid system, without the overhead of a database manager. XML data is a common format for data and XPath is a very effective way to parse the data for use by Corvid.

The XML "file" is actually just a URL. It can be a static file or call a server program that dynamically returns data in an XML form. The same options found in the URL approach to calling external programs can be used to call an external XML resource, however, instead of requiring that the program return data in Corvid's format, it should be returned in the standard XML form and the XPath command will parse out the data your system needs.

Corvid supports standard XPath commands, which are evaluated using the Java XPath command parser. This provides excellent support for current and future XPath syntax. Corvid adds some special options that make it easier to use the data returned by XPath in Corvid systems, especially for those not familiar with advanced XPath options.

For command details see Section 16.5

16.3.3 Database

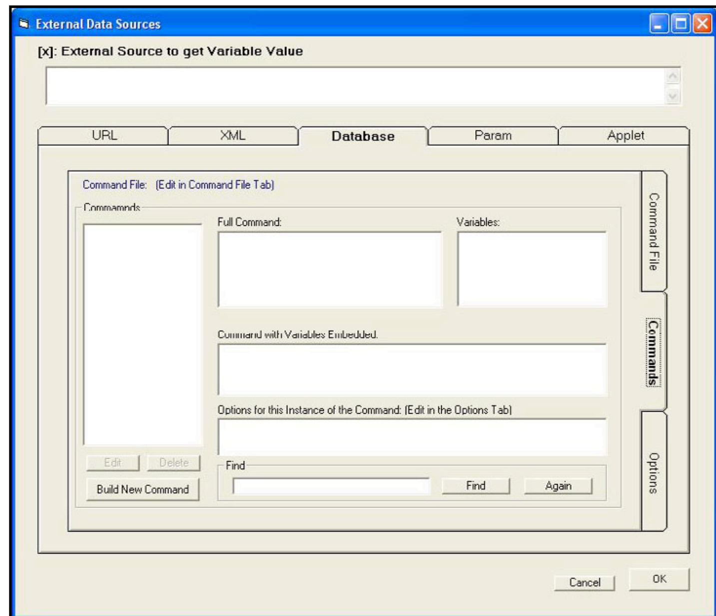
Database commands are some of the most flexible and commonly used external data commands. Corvid can interface with any ODBC/JDBC compliant database to read and write data.

Standard SQL commands can be used to select and read data. The values of Corvid variables can be sent to the database to be stored, or used in commands that select items of data to return to Corvid.

Database commands can be used with both the Corvid Servlet Runtime and the Corvid Applet Runtime, however the Applet Runtime requires server support for database commands in the form of a Java Servlet provided with Corvid.

The Corvid database interface provides a convenient way to integrate Corvid expert systems into corporate databases and IT infrastructure.

For command details see Section 16.6



16.3.4 PARAM Data

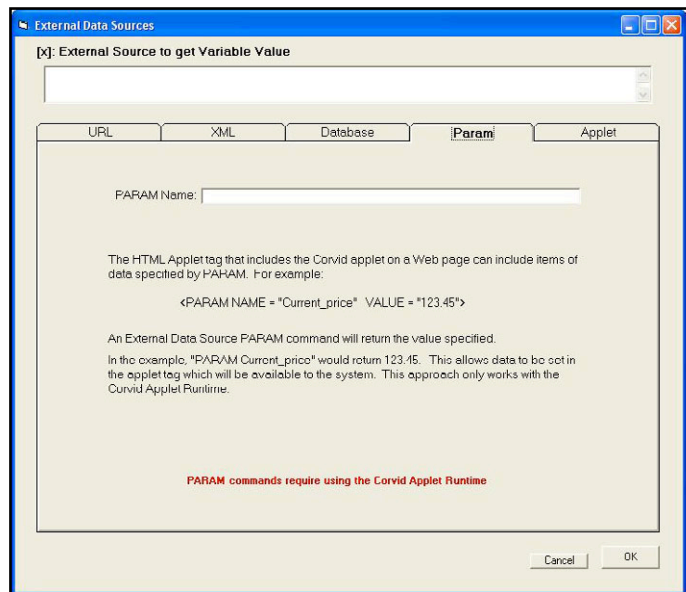
The Corvid Applet Runtime provides a way to run Corvid expert systems in a browser window. An HTML "Applet" tag defines where and how the Corvid Runtime will be displayed in the HTML page. This applet tag can also contain "PARAM" data to set the value for Corvid variables.

The PARAM external interface makes it easy to access this PARAM data from within the system.

PARAM data is most useful for systems that dynamically build the HTML pages that contain the Corvid Runtime, such as with Cold Fusion, but can be used anytime there is data that should be sent to specific Corvid sessions or which should be used to configure the sessions to reflect changing situations.

The PARAM approach can only be used with the Corvid Applet Runtime.

For command details see Section 16.7



16.3.5 APPLET Data

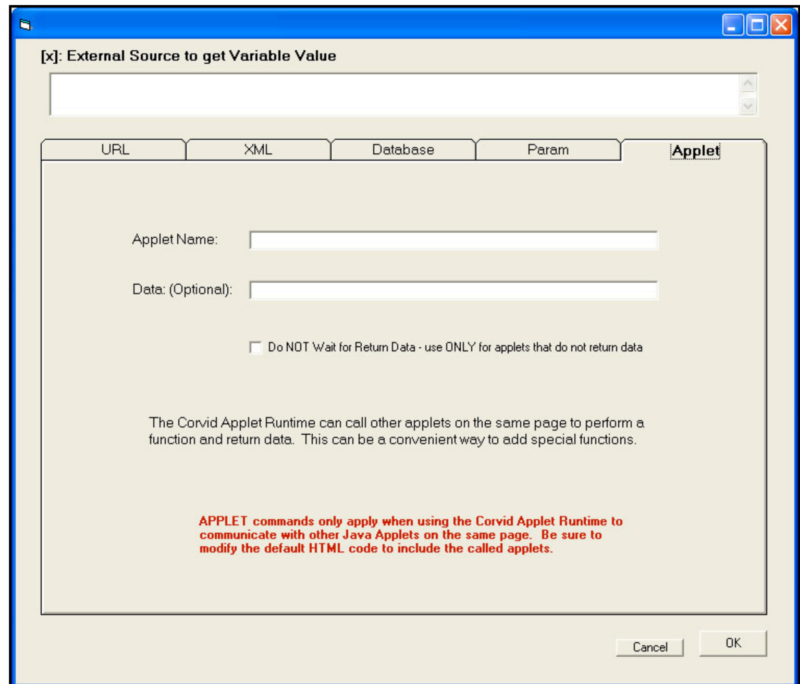
A Corvid session running the Corvid Applet Runtime can call other Java applets on the same page and send / receive data.

This allows creating custom Java applets that work with Corvid to perform special functions, display data or access data via Java API commands.

The Corvid Trace applet uses this technique to display the trace information from a session. The Corivd XML interface for the Applet Runtime also uses an external applet to add the functionality.

Obtaining data from an external applet requires knowledge of Java to build the applet. Applets can be built using Oracle Sun's free NetBeans development tools or other Java development tools.

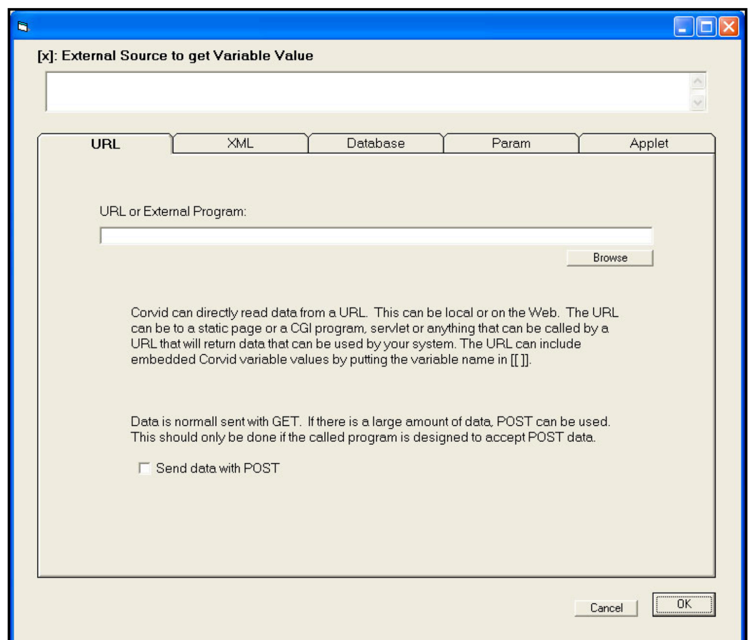
For command details see Section 16.8



16.4 URL / External Program Command Details

URL commands can be used to read data from any source that can be referenced by a URL, and can also be used to call local programs. Embedded Corvid variables can be used to build the URL dynamically or to pass data to the called program.

To add a URL external data source command, open the External Data Sources window and click the URL tab.



16.4.1 URL Syntax

The URL / External Program tab is used to build commands to:

- Read a static file of data
- Call a web program (CGI, Servlet, ASP, etc) that returns data
- Pass data to a web resource

Reading a static file of data is the simplest type of external interface URL and is the basis for many other external interface techniques. In this case, the file is just a simple text file containing the content to return. Generally, the text file is located in the same folder as the system CVR file and can be referenced just by name.

> **For example:** If there is a data file, MyData.txt, that the system needs, it can be put in the same folder as the CVR file. In the system, the URL to reference to it is just: **MyData.txt**

If the file is relative to the system CVR file, Corvid automatically builds the full URL for it. URL addresses that do NOT start with "http" are automatically taken as relative to the folder that the system CVR file is in. Data files can also be in a subfolder by including the subfolder name, such as DataFiles/MyData.txt

If the file is not in the same folder or a subfolder of the CVR file, it must be referenced by a full URL address, starting with "http://". Any address that starts with "http://" is taken as an absolute address and not relative to the CVR file. These addresses must be full URL addresses, the same as if they were entered in a browser program.

In most cases, static files of data needed by a system should be kept in the same folder as the system CVR file. This makes it easy to keep the files together and to move them between servers or run standalone. Full URL addresses starting with "http://" are needed for servlet, CGI and other programs on a different server.

When calling server programs such as Java servlets or CGI programs, the URL would be the same as would be entered in a web browser. The URL can include parameters to pass data to the called program, such as calling a servlet to get the price of a specific part:

http://www.myServer.com/PriceServlet?part=X123

To pass the value of Corvid variables in the URL, include them in double square brackets, such as:

http://www.myServer.com/PriceServlet?part=[[Selected_Part]]

Any Corvid variable names in double square brackets will be replaced by the variable's value before the URL is called. This can be used for parameters passing data, or even part of the base URL address itself.

> **Remember:** Double square bracket embedding can lead to backward chaining to derive the variable's value. To use the immediate value without backward chaining, use [[*varname]]. The asterisk before the name will use the current value without any attempt to derive or backward chain to get a "final" value.

Any URL that could be entered in a browser window can be used. Normally, the called URL will return data, however some external interfaces such as After Ask do not expect anything to be returned and the URL may be used to just send data.

GET and POST

When Corvid sends data to a program via a URL, it uses the GET approach. This sends the data on the URL, but only allows a limited number of characters to be sent. The limit depends on both the browser and server, but is generally 1000-2000 characters or more. For most uses, this limit is not an issue. However, if a system needs to send a large amount of data to the server, check the "Send data with POST" check box. POST allows any amount of data to be sent, but **the receiving program must be designed to receive data using POST**.

Unless it is necessary to send a very large amount of data, and the receiving program is designed to work with POST, it is best to use the default GET approach.

16.4.2 Format of Returned Data

To Corvid, the data returned is just text returned from calling a URL. It does not matter to Corvid if the data is from a static file or dynamic content returned from a servlet, CGI program, ASP or other on-line resource. All that matters is that the content is in the correct form.

The returned data MUST just be text. It should not have a header or that will be interpreted as part of the data. Likewise, a HTML page should not be used because it will have various tags wrapped around the text of the data.

To create static files of data, use a program such as Notepad that produces simple text files. Do not use a word processor such as MS Word unless you make sure to save the file as "text".

Most external interface calls expect to receive a returned data string that will be used for a single purpose, such as the prompt text for a variable or the value to assign to a specific variable. Some calls expect to receive back multiple items of data, but still for a single purpose - such as the list of possible value options for a Dynamic List variable or multiple items to add to a Collection variable. In these cases, the format of the data returned is just the content needed with no additional identifier information. This can be in the static file, or returned from the program called by the URL.

External calls to set the value of a variable can set the value of a single variable or multiple variables in the same call. The returned data can be either:

- A single value with no identifier, in which case it will be assigned to the variable associated with the external call.

- One or more name / value pairs that identify a variable followed by the data to assign to that variable. The variable identifier is the name of the variable in square brackets, followed by a space and the value to assign to that variable. There can be multiple name / value pairs, one per line. The returned data can set the value for any variables in the system, but should also always set the value of the variable associated with the external call.

Example: an external URL call to set the value for numeric variable [X] could return just

123

which would set the value of [X] to 123. Alternatively, it could return

[X] 123

which would also set [X] to 123. However, using the second form, it could return

[X] 123
[Temp] 77
[Price] 99.99

which would set the values of [X], [Temp] and [Price] in a single call.

If [X] was associated with the external call, returning

456
[Temp] 77
[Price] 99.99

which would set the values of [X] to 456. Since that data does not have an identifier it would be assigned to the associated variable. The values of [Temp] and [Price] would also be set.

16.4.3 Types of Returned Data

The format of the data returned for a variable depends on the type of variable.

Static List Variables

- The number of the value. (FIrst value = 1, second = 2, etc)
- The short text, or if there is no short text, the full text of the value
- More than one value number or short text separated by the TAB character
- A numeric expression that evaluates to the number of a value

Example: Static List variable [Color] with values “Red”, ‘Blue” and “Green”

Returning	Sets
1	Red
[Color] 1	Red
[Color] Blue	Blue
2 3 (separated by Tab)	Blue, Green
[Color] Red Green (separated by Tab)	Red, Green
[Color] 1+1	Blue

Dynamic List Variables

- The number of the value. (FIrst value = 1, second = 2, etc)
- More than one value number separated by the TAB character
- A numeric expression that evaluates to the number of a value

Example: Dynamic List variable [Car] with values “Chevy”, “Ford” and “Honda”

Returning	Sets
1	Chevy
[Car] 1	Chevy
2 3 (separated by Tab)	Ford, Honda
[Color] 1+1	Ford

Numeric Variables

- A numeric value.
- An expression that evaluates to a numeric value

Example: Numeric variable [Price]

Returning	Sets
45.23	45.23
[Price] 99.95	99.95
[Price] 90 + 10	100

String Variables

- A string value without quotes

Example: Numeric variable [Name]

Returning	Sets
abcde	abcde
[Name] Exsys	Exsys

Date Variables

- A string date value. This can be a full date “July 5, 2010” or shorter forms such as “1/5/99” or the number of milliseconds since Jan 1, 1970. Dates can also include the time.

Remember: All date values are interpreted by the local settings for dates on the machine running Corvid. This is the server settings for the Servlet Runtime and the client machine settings for the Corvid Applet Runtime. Different countries use different formats for dates. Dates such as “1/2/10” can be ambiguous since the 1 is the month in some countries and the 2 is the month in others. When assigning dates, the long form (e.g. July 5, 2010 1:23PM) is best to use since it is unambiguous. To set the variable to the current date, “now()” can be used, but this can also be done in Corvid and does not require an external call. Using "now()" with the Servlet Runtime will set the time for the server. The Applet Runtime will set the time for the client PC.

Example: Date variable [Start_date]

Returning	Sets
July 5, 2010	July 5, 2010
[Start_date] 5/12/10	May 12, 2010 (in the US)
now()	The current date and time
Aug 21, 2008 4:52PM	Aug 21, 2008 4:52PM

Confidence Variables

- A numeric value consistent with the confidence mode
- An expression that evaluates to a value consistent with the confidence mode

Remember: The value assigned must be consistent with the confidence mode settings of the variable. A variable that expects a value between 0 and 1, must be given a value in that range. Confidence values assigned will be combined with any other values assigned to the variable based on the confidence mode settings.

Example: Confidence variable [Conf]

Returning	Sets
5	5 combined with any other values
[Conf] .25	.25 combined with any other values
[Conf] 5 + 10	15 combined with any other values

Collection Variables

- A string value without quotes
- Multiple string values separated by the TAB character
- If returning the value for the collection, multiple values on separate lines

Items added to a Collection variable are always added to the end of the list. To add items to a collection using the other methods such as addfirst or sort, read the returned data into a string variable and then add the value to the collection using a SET command with a method in the Command file.

Example: Collection variable [Coll]

Returning	Sets
aaa	Adds “aaa” to the end of the list
[Coll] aaa bbb (separated by Tab)	Adds “aaa” and “bbb” to the list
aaa bbb ccc (each on a separate line)	Adds “aaa” “bbb” and “ccc” to the list

END Marker

When reading a static file of data using the READ command in a Command block, a file can contain multiple batches of data which will be read and processed sequentially. These batches are separated by the END command. This ONLY should be used with the READ command and is discussed in the Command Block chapter section on the READ command.

16.5 XML Command Details

16.5.1 Intro to XML and XPath

To select item(s) of data in an XML file, Corvid uses standard XPath commands that are processed by the Java’s built in command parser. This provides a very standard and powerful way to utilize XML data.

XPath commands are often compared to SQL commands for databases. It provides a way to identify specific data in the XML file. An XPath command may return a single item of data or multiple items, depending on the nature of the command and the XML data file. Like SQL, relatively simple commands to read simple data structures are easy, but commands to select items out of complex data structures can be complex.

Since XPath is a standard command syntax by W3C, there are many on-line XPath tutorials that can be found by Googling “XPath Tutorial”. There are also various technical books on XPath. These can assist you in understanding advanced XPath syntax and building complex XPath commands when that is needed. However, most Corvid uses of XPath will be relatively simple XPath commands reading from simple XML data structures, and this introduction should get you started.

Simple XPath Commands

XPath commands provide a way to specify one or more items in an XML data file. XML data is organized by nested “tags”, much like the tags in HTML, but far more flexible. XML is a complex subject, and there are many books devoted to it. This is only a VERY simplistic look at how data can be stored in XML and how XPath commands can read that data.

XML data is defined by elements that structure the data. An element is indicated by a tag, a name in < > such as <myData>, <Name>, <Price>, etc, similar to the way HTML is structured. The data for the element is everything between the tag and its closing tag, which is just the tag name in </ ...>. (Note the backslash **/** in the closing tag). So, a price of 123.45 would be entered as:

```
<price> 123.45 </price>
```

Closing tags must match the immediately preceding unclosed tag.

Elements can contain data or they can contain other elements. This ability to put elements in elements is what allows data to be structured with XML. For example, there might be information on a part:

```
<part>
   <part_ID>X351</part_ID>
   <price> 3.45 </price>
   <color> blue </color>
   <color> green </color>
   <size>  1 inch </size>
</part>
```

Here there is a “part” element with other elements describing part, ID, price, color(s) and size, each within their own tag. Each closing tag matches the preceding unclosed tag. The </part> tag closes the opening <part> tag since all the tags between them are already closed.

Note that there are 2 <color> elements. There can be multiple occurrences of some elements. This allows the <part> element to include multiple color options for the part. XML files have an associated “schema” that defines what elements can occur where, when there can be multiple occurrences of an element, nesting, etc. This simple introduction does not cover schema, but this important part of XML is well covered in XML books.

Programs for building and managing XML files check the XML data against the schema to make sure it is correctly structured. Corvid does not check a file against its schema and it is the developer’s responsibility to make sure that the XML data files provided are structurally correct.

In addition to the nested elements, individual elements can have attributes to provide other information and ways to select data. For example, to add data on the part material, we could either add a tag for it:

```
<part>
     <part_ID>X351</part_ID>
     <price> 3.45 </price>
     <color> blue </color>
     <color> green </color>
     <size>  1 inch </size>
     <material> plastic </material>
</part>
```

or add an attribute inside a the part tag.

```
<part material=”plastic”>
     <part_ID>X351</part_ID>
     <price> 3.45 </price>
     <color> blue </color>
     <color> green </color>
     <size>  1 inch </size>
</part>
```

Both provide information on the material, but are referenced in different ways in the XPath command. Also, the first approach using a tag, would potentially allow adding multiple materials. The second approach would be better if we wanted to have multiple similar parts, some of plastic and some of some other material, each with their own <part> tag.

A full list of parts would be made up of data on many individual parts each in its own <part> tag with its associated elements:

```
<part_list>
    <part>
      ....
    </part>
    <part>
      ....
    </part>
    <part>
      ....
    </part>
</part_list>
```

This nesting of tags can get quite complex depending on the nature of the data. The structure can be recursive, so that there might be a <person> element containing various elements with information on that person. One of those might be <children> containing a list of other <person> elements for each child, each of which could also have <children> elements containing more <person> elements, etc. Such a structure could be very complex, and the concept of “parent”, “child” and “sibling” nodes is key to navigation in such complex structures. Advanced XPath commands can be used to select data even in complex structures, but here we will only look at simple XPath commands.

Simple XPath commands look much like a directory path on a computer. In the simplest form, there are element names separated by “/” traversing the tree-like structure defined by the nested elements.

In the simple part example:

```
<part>
    <part_ID>X351</part_ID>
    <price> 3.45 </price>
    <color> blue </color>
    <color> green </color>
    <size>  1 inch </size>
    <material> plastic </material>
</part>
```

The XPath command **/part/price** would return the value 3.45. **/part/size** would return “1 inch”. However, **/part/color** would return both values “blue green”. This is because multiple values are allowed and the command does not differentiate between them.

Working with a more complex list of parts:

```
<part_list>
    <part>
      ....
    </part>
    <part>
      ....
    </part>
    <part>
      ....
    </part>
</part_list>
```

The command **/part_list/part/price** would return a list of the prices for all the parts. To get the price of a single part would require specifying an individual part. XPath provides many ways to do this. When there are multiple items, specific one(s) can be selected by putting a selection test in [ ]. The simplest selection test is just the number of the element when there are multiple occurrences of the element:

For example:

/part_list/part[1]/price

would return the price value for the first part in part_list. The XPath command /partList/part[2]/price would return the price value for the second part.

The selection test can also be a boolean expression on the associated values.

/part_list/part[part_ID="X351"]/price

would return the price of the part with the part_ID of "X351", regardless of where it was in the list.

Commands can also return multiple values:

/part_list/part[color="blue"]/part_ID

would return a list of the part_IDs for all parts that are available in blue.

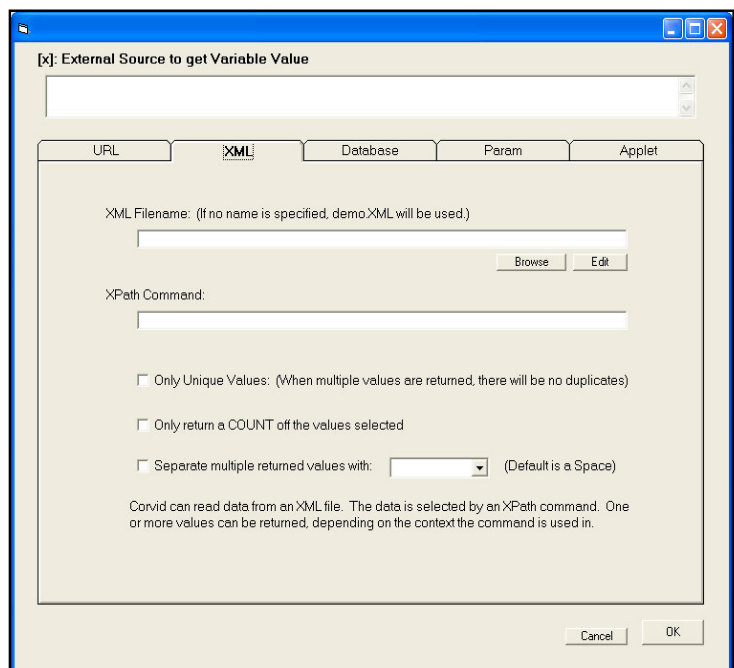
/part_list/part[price > 3.00]/part_ID

would return a list of the part_IDs for all parts that cost more than $3.

There is MUCH more to XPath commands, but even this simple syntax allows many XML files to be parsed and used with Corvid systems.

16.5.2 Building XML Commands

Corvid can read data from XML files or data sources. An XPath command is used to select the data used by Corvid. To add a XML external data source command, open the External Data Sources window and click the XML tab.



Corvid XML commands are made up of 3 parts:

- The name of the XML file / URL to use (in quotes)
- The XPath command to select one or more values
- Modifiers for the returned data, making it easier to use it with Corvid

XML("Filename" XPath_command Modifiers)

16.5.3 XML Filename

The filename for the XML file can be local to the Corvid system CVR file or a URL referencing a file or program somewhere on a network or the web that returns XML data.

The XML filename is optional, however if it is not specified, the file KBName.xml will be used (where KBName is the name of the system CVD/CVR file) and the file must be in the same folder as the system CVR file. The file name must be in quotes.

The filename can be either:

- A static file of XML data
- A URL to a web program (CGI, Servlet, ASP, etc) that returns XML data

Reading XML data from a static file is the simplest type of XML file reference. In this case the file is just a file containing the data in an XML form. If the XML file is located in the same folder as the system CVR file, it can be referenced just by name.

> **For example:** If there is a data file MyXMLData.xml that the system needs, it can be put in the same folder as the CVR file. In the system, the "XML FIlename" is just **MyXMLData.xml**.

All filenames are actually URLs. If the file is relative to the system CVR file, Corvid automatically builds the full URL for it. Filenames that do NOT start with "http" are automatically taken as relative to the folder that the system CVR file is in. XML files can also be in a subfolder by including the subfolder name, such as **DataFiles/MyXMLData.xml**.

If the file is not in the same folder or a subfolder of the CVR file, it must be referenced by a full URL address, starting with "http://". Any address that starts with "http://" is taken as an absolute address and not relative to the CVR file. These addresses must be full URL addresses, the same as if they were entered in a browser program.

In most cases, static XML files needed by a system should be kept in the same folder as the system CVR file. This makes it easy to keep the files together and move them between servers or to run standalone. Full URL addresses starting with "http://" are used for servlet, CGI and other server resources that will return XML data.

To pass the value of Corvid variables in the URL, include them in double square brackets, such as:

> **http://www.myServer.com/PartListXMLData?Model=[[Selected_Model]]**

Any Corvid variable names in double square brackets will be replaced by the variable's value before the URL is called. This can be used for parameters passing data, or even part of the base URL address itself.

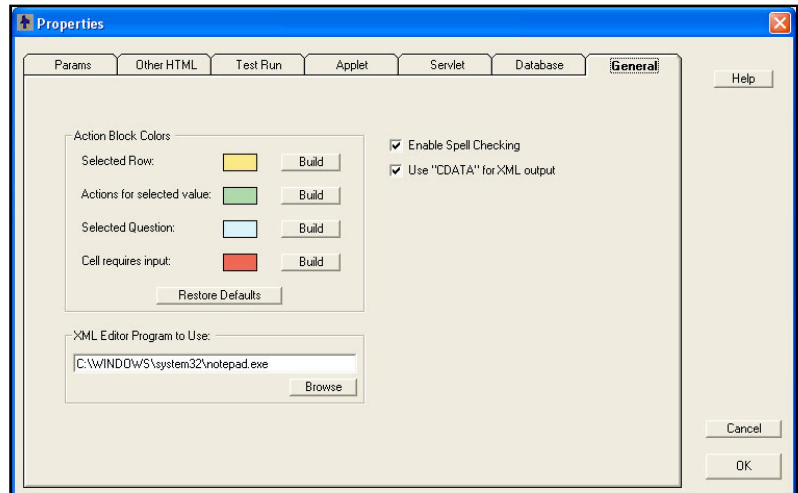
> **Remember:** Double square bracket embedding can lead to backward chaining to derive the variable's value. To use the immediate value without backward chaining, use [[*varname]]. The asterisk before the name will use the current value without any attempt to derive or backward chain to get a "final" value.

Any URL that could be entered in a browser window can be used, however, the called URL MUST return data in an XML form. Corvid does not check the returned data against a schema, but an incorrectly structured file will probably lead to runtime errors or incorrect operation. It is the developer's responsibility to assure that the called program is returning syntactically valid XML data.

Edit: To view or edit the XML file, click the “Edit” button. This will open the file with the XML editor specified in the Properties window. The default is to use Notepad, but later versions of MS Word can be used to display XML with formatting, or a true XML editor such as EditiX can be used.

To specify the program to use to view XML, go to the Properties window “General” tab.

Under “XML Editor Program to Use”, click “Browse” and select a program. The XML programs can be used to assist in building XPath commands, editing data and checking data against schema as needed.



16.5.4 XPath Command

The XPath command can be anything from a simple command to return one item of data to a complex command returning many items. The nature of how the external interface command is used determines how Corvid will handle the returned data, and if Corvid expects one or multiple items of data. An individual command can only return data for one purpose. Unlike some external commands, a single XML command cannot set the value for multiple variables.

The XPath command can include the values of any Corvid variables by simply putting them in the command in double square brackets - [[...]]. Corvid will replace the [[...]] with the value of the variable before evaluating the XPath command. For example, to add a command to find all the parts over a price stored in a Corvid variable [Min_Price]:

/part_list/part[price > [[Min_price]]]/part_ID

Note: when using double square bracket embedded variables within XPath syntax that also uses square brackets, be sure to put a space between the Corvid [[..]] and any XPath [or] characters. Having 3 brackets together can result in ambiguous expression. As long as the Corvid variable [[..]] has spaces around it, it will work correctly. So the above XPath command would be OK, but

/part_list/part[price > [[Min_price]]]/part_ID

would not since it has 3 bracket characters together without any spaces. The Corvid embedded variables will be replaced by their value before the command is parsed as an XPath command, so the Corvid brackets will not confuse the XPath parser.

When using Boolean tests to select items of data, spaces in the XML data can be an issue.

Sample XML data:

```
<part_list>
    <part>
        <part_ID>A111</part_ID>
        <price>  3.45  </price>
        <color> blue </color>
        <material> plastic </material>
    </part>
    <part>
        ...
    </part>
    <part>
        ...
    </part>
</part_list>
```

If the boolean test is numeric, spaces do not matter and

part_list/part[price > 5]/part_ID

will work correctly to select the part_IDs for the part where the price is greater than 5. It does not matter if the data in the <price> tag has spaces around it.

However, for boolean expressions using string matching such as:

part_list/part[part_ID >"A111"]/price

The data in the <part_ID> tag must NOT have spaces around it. This is particularly important when setting the value list for a Dynamic List variable from the XML data and then using the value in other boolean tests to get other tags. Whenever a tag will be used in a boolean expression matching strings, make sure there are no spaces around the data.

When matching a string, the string must be indicated by " or '. Either:

part_list/part[part_ID >"A111"]/price

or

part_list/part[part_ID >'A111']/price

would be legal syntax. If using " causes a parsing problem in some situations, try using **'** instead.

The XPath command is any legal XPath command that can be evaluated by the Java XPath parser. This supports quite complex XPath syntax when needed.

Note for Advanced XPath Users: Corvid expects the data returned from the XPath command to be a string. Because of this, Corvid automatically adds ".text()" to the XPath command. Normally this is correct and allows the developer to just specify the XML elements(s) without needing to explicitly convert it to a string. However, for complex XPath commands, it may not be valid to just add ".text()" to the end of the command. Corvid will NOT add the ".text()" to the XPath command if it ends in ")". So complex commands not needing ".text()" should be put in parenthesis. For example:

/part_list/part[1]/price

would automatically have ".text()" added and would work correctly.

(/part_list/part[1]/price)

would not work since the ".text()" would not be added, but

(/part_list/part[1]/price.text())

would work correctly.

16.5.5 XPath Modifiers

Corvid's special modifiers make it easier to use data returned from XPath commands in Corvid. These are much easier to use than building a complex XPath command to return the same data.

Modifier: Corvid supports 3 modifiers that can be applied to the returned data:

- UNIQUE
- COUNT
- SEP=" "

UNIQUE

UNIQUE will parse the returned data and remove redundant values. This is used when the XPath command may have multiple occurrences of the same value, and what the Corvid system needs is the unique values. For example, you might want to get a list of the materials used for the parts in the list:

/part_list/part/material

would return the list of materials, but it would include "plastic" once for each part made of plastic, potentially many times. This would also apply to each other material. Adding the modifier:

/part_list/part/material UNIQUE

causes Corvid to reduce the list to only include unique materials before returning the data. This could be used to set the value options for a Dynamic List variable, allowing the end user to choose from the materials actually in the parts list without redundancy.

COUNT

COUNT will return a count of the number of data items returned.

/part_list/part/part_ID COUNT

would return a count of the number of parts in the list.

COUNT can be used with UNIQUE to provide a count of the unique values:

/part_list/part/material UNIQUE COUNT

would return a count of the number of materials used for all the parts in the list.

SEP="..."

SEP= provides control of the character/string used to separate individual values when multiple values are returned. This is needed only if the command will return multiple values, and those values are to be used as a single string. When XML commands are used in a context that expects multiple values, such as defining a list of values for a dynamic list variable, Corvid does not require the "SEP=" since the context of the use determines how the values will be used.

The default separation character for multiple values is a new line. To change this to " & ", use SEP=" & ". For example:

/part_list/part/material UNIQUE SEP=" & "

would return a single string, list all the materials in the part list such as:

plastic & aluminum & steel

The separator string is used only between values and does not appear at the end of the string.

Any string can be used in the SEP=" ", but tabs and line feeds cannot be typed into the edit box. To create a string with a TAB between the values, use SEP="TAB". The default separator for multiple values is a new line.

16.5.6 XML Command Samples

This is a small sample XML file and some sample XPath commands:

XML File:

```
<part_list>
<part>
  <part_ID>A111</part_ID>
  <price> 3.45 </price>
  <color> blue </color>
  <color> green </color>
  <size>  1 inch </size>
  <material> plastic </material>
</part>

<part>
  <part_ID>B222</part_ID>
  <price> 7.95 </price>
  <color> black </color>
  <size>  2 inch </size>
  <material> aluminum </material>
</part>

<part>
  <part_ID>C333</part_ID>
  <price> 9.55 </price>
  <color> black </color>
  <size>  4 inch </size>
  <material> steel </material>
</part>

<part>
  <part_ID>D444</part_ID>
  <price> 4.55 </price>
  <color> blue </color>
  <color> green </color>
  <size>  2 inch </size>
  <material> plastic </material>
</part>
</part_list>
```

Sample XPath Commands:

Selecting a part name by its numerical place in the list.

/part_list/part[1]/part_ID

Returns the Part_ID of the first part: A111

Selecting a part price by its part_ID.

/part_list/part[part_ID=”C333”]/price

Returns the price of the part with a Part_ID of “C333”: 9.55

Getting a list of the part_ID of all the parts in the list.

/part_list/part/part_ID

Returns a list of all the Part_IDs:

A111
B222
C333
D444

Getting a list of the part_ID of all the parts in the list as a single string separated by &.

/part_list/part/part_ID SEP=" & "

Returns: A111 & B222 & C333 & D444

Getting a list of the materials used for parts in the list.

/part_list/part/material

Returns:

plastic
aluminum
steel
plastic

Getting a list of the materials used for parts in the list without repeated items.

/part_list/part/material UNIQUE

Returns:

plastic
aluminum
steel

Getting a count of the number of materials used in the list.

/part_list/part/material UNIQUE COUNT

Returns: 3

Getting a count of the number of parts in the list.

/part_list/part/part_ID COUNT

Returns: 4

16.5.7 Embedded XML Commands

In addition to using XML commands associated with specific variable actions such as setting the value of a variable, XML commands can be embedded in any text in a Corvid system. This can be done any place a double square bracket replacement could be used.

The syntax for embedded XML commands is the same as XML commands to get other data, but the command is surrounded by <#<...>#>.

<#<XML “Filename” XPath_cmd Modifiers>#>

The filename is optional, but if not included the KBName.xml will be used, where “KBName” is the name of the system. In this case, the XML file must be in the same folder as the system CVR file. Filenames NOT starting with “http://” will be considered to be an address relative to the folder with the CVR. Names starting with “http://” will be full addresses to files anywhere on the web. If a file name is used, it must be in quotes.

The XPath command MUST be in quotes, and can be any XPath command that could be used to get data.

The optional Modifiers UNIQUE, COUNT and SEP=”..” can be used.

Embedded XML commands can be used in any text in a Corvid system such as Prompts, content in reports, explanations to the end user, etc.

For example, a report might include a comment that:

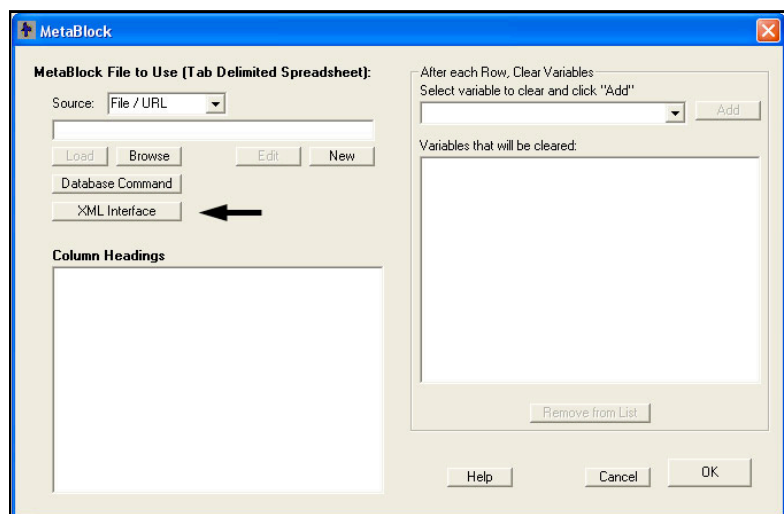
The current part inventory includes parts made of X different materials

This could be added to a report by using the embedded XML command in the text to add the count of the materials currently used:

The current part inventory includes parts made of <#<XML “partInventory.xml” /part_list/part/material UNIQUE COUNT >#> different materials

16.5.8 Using an XML File for MetaBlock Data

XML data can also be used to provide the data for Corvid systems that use MetaBlocks. This allows an XML file to be used in place of a tab-delimited spreadsheet. The XML “file” used can be either a static file or a URL that returns XML data. If a URL is used, it must return the **same** XML data each time it is called during the session.



Selecting to use XML data is done from the MetaBlock properties window. (This is displayed by clicking the “MetaBlock” button on a Logic Block window).

Click the “XML Interface” button to use an XML file. (This will build a MetaBlock source file for the XML Interface. If an XML interface file has already been created, just click the “Edit” button to make any changes to it.

16.5.9 Obtaining MetaBlock Data Using XPath Commands

To use an XML file for the the MetaBlock data, an XML MetaBlock Interface file must be created. This file will tell the Corvid runtime the XML file to use and the XPath commands to use for each of the “columns” in the MetaBlock spreadsheet. This can be a little complicated to visualize at first.

When a MetaBlock is based on an actual tab-delimited spreadsheet, there are actual rows and columns of data. The columns have heading text that is used in the MetaBlock rules to reference the data in that column. The rows are actual rows of data. When using an XML file for the data, commands build individual **columns** of data. The columns are given names so that they can be referenced in the rules, and an XPath command defines the data that goes in that column. Rows do not actually exist as such, but are actually a combination of the nth item of data returned by each of the XPath commands for each column. Because of this, the XPath commands each must return multiple values and they must be synchronized and consistent so that the nth item of each column applies to the same product.

This is easier to see with an example. Using a small XML file:

```
<part_list>
<part>
  <part_ID>A111</part_ID>
  <price> 3.45 </price>
  <color> blue </color>
  <material> plastic </material>
</part>

<part>
  <part_ID>B222</part_ID>
  <price> 7.95 </price>
   <color> black </color>
  <material> aluminum </material>
</part>

<part>
  <part_ID>C333</part_ID>
  <price> 9.55 </price>
  <color> black </color>
  <material> plastic </material>
</part>
</part_list>
```

In the more standard tab-delimited spreadsheet approach to MetaBlocks, the spreadsheet would look something like:

part_ID	price	color	material
A111	3.45	blue	plastic
B222	7.95	black	aluminum
C333	9.55	black	plastic

To create the equivalent data using an XML Interface, requires building it one **column** at a time. The first column is given a name “part_ID” and the data in that column can be obtained from the XML file by the XPath command:

/part_list/part/part_ID

This command would return the list of part_ID values in order. The values are used one per line. This builds the first column of data.

The second column is given the name “price” and the data comes from the XPath command:

/part_list/part/price

This will return the price of the various parts in the same order as the part_ID column.

The same can be done for the "color" and "material" columns. Note, it is NOT required to have the column names match the XML element names, but it is easier and more understandable if this is done.

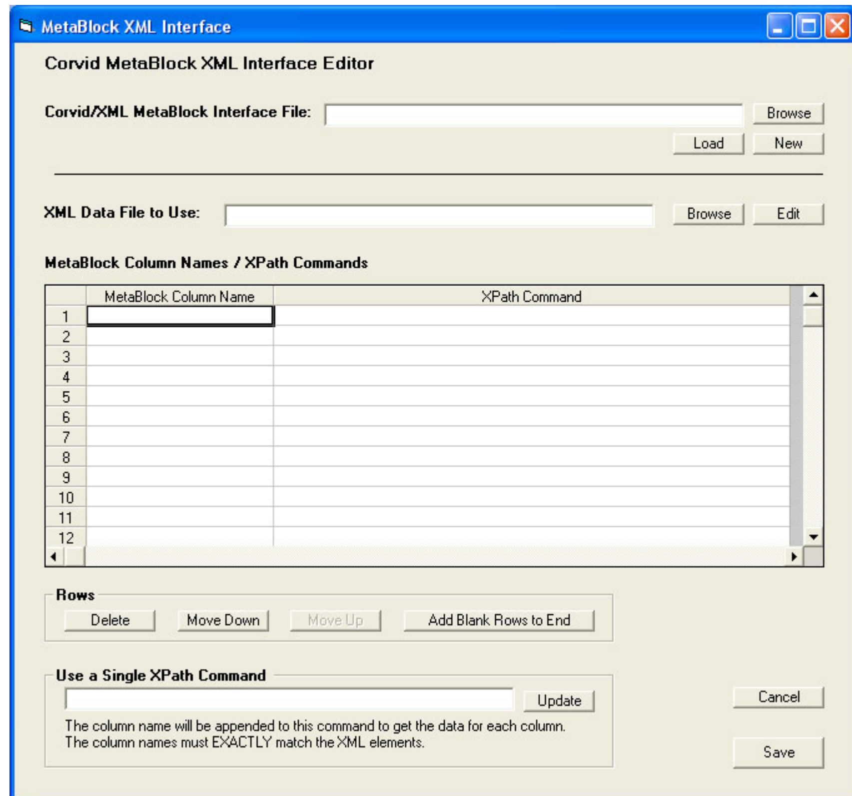
The XML Interface to build the MetaBlock data is:

Column Name	XPath Command for the Column Data
part_ID	/part_list/part/part_ID
price	/part_list/part/price
color	/part_list/part/color
material	/part_list/part/material

This will build exactly the same MetaBlock data for Corvid to use, but the raw data can be maintained, edited and stored as XML data.

Note: Each of the elements under "part" only have a single value. This makes it much easier to use the data for a MetaBlock since if one part had multiple "color" elements, the command would return an extra data element for that column and break the synchronization with the other columns. It is possible to use XML files with multiple occurrences of an element, but it requires a more advanced technique discussed below.

16.5.10 Building MetaBlock XML Interface Files



The MetaBlock XML Interface specification is kept in a text file. This file specifies the XML file to use and the XPath command for each column in the spreadsheet. This file is built and edited from the MetaBlock XML Interface window that is displayed by clicking the "XML Interface" button on the MetaBlock window.

The XML interface specification is kept in a text file. This file specifies the XML file to use and the command for each column in the spreadsheet. Enter/browse the name of the XML data file. Then enter MB column names in the left column and associated XPath commands in the right column.

Interface File

An interface file needs to be selected or created. The recommended file extension for this file is .mbx, but any name can be used. If there is already a interface file created, enter the name and click "Load" or browse to the file and select it. This will load the file and its parameters. If there is no file, click "New" and enter a name for the file.

XML Data File

Select the XML data file that will be used. Either enter the name of the file as a URL or browse to the file. Any XML file or URL reference that could be used in other XML commands can be used. Clicking "Edit" will open the XML file using the XML Editor program specified in "Properties" to view or edit the file.

Column Names

Enter the name of the each of the “columns” in the MetaBlock. These can be directly typed into the boxes on the screen. These will be the column references used in the MetaBlock logic - the names in { }. The names can be anything clear and unique, but it is often easier if the names match the element names in the XML data.

Columns do NOT need to be created for every element in the XML data, only for the ones that will actually be needed in the MetaBlock logic and rules.

XPath Commands

Next to each column name enter an XPath command that will return the data for that **column**. It is VERY important that the XPath commands return consistent data across the columns. That is, the nth item in each column **MUST** all apply to the same product. This is easy to do for simple XML files such as in the example above. Just make sure the XPath command will return data from the same higher-level elements in the XML data.

In the example, the first column XPath command was

/part_list/part/part_ID

This will return one “part_ID” data item for each part element in the part_list. The other XPath commands must match. The “price” column uses the XPath command:

/part_list/part/price

This will return also return one “price” data item for each part element in the part_list.

To have the “rows” of data in the MetaBlock all be related and have meaning, it is very important that this consistency be maintained.

Enter an XPath command to get the data for each of the columns entered.

The XPath command can include embedded Corvid variables and selectors to restrict the data returned such as:

/part_list/part[price > [[Min_price]]]/part_ID

However, these MUST be used consistently in all commands to make sure the data returned is consistent, and the “price” data would have to be:

/part_list/part[price > [[Min_price]]]/price

If the “price” data was instead something like:

/part_list/part/price

it could return more data items and that data might not be in sync with the part_ID command. This would cause the MetaBlock to work incorrectly and give incorrect recommendations.

Multiple Occurrences of an Element

All XPath commands should return single data items for each “record” or higher level XML element. If the element has multiple instances of an internal element, it will return multiple values. For example, in the earlier sample XML data, there were multiple occurrences of the “Color” element. This is legal XML provided it is allowed by the schema for the file, but it will confuse the MetaBlock interface.

If there are multiple elements for some “parts”, it will throw the sync of the returned data. If the data is structured this way, and it needs to be used in the system, build the MetaBlock using only the elements that will return a SINGLE value for each “part”. Then in the actual Logic Block rules use embedded XML data embedding (using <#<XML...>#>) to get the other data that can return multiple values using some unique reference to the record. The <#<XML will return the values as a single string that can be used in expressions or to build reports.

For example, in the MetaBlock specification use:

/part_list/part/part_ID

to get the “part_ID” column data which is a single unique reference to each “part”. This column data is used in the associated Logic Block by the column name in { }

{part_ID}

which will be replaced by the value for the current working “row” in the MetaBlock. The associated “Color” value might have multiple values. Creating a "color" column would be out of sync with the other columns. Instead, use an embedded XML expression in the Logic Block expression:

<#<XML "partdata.xml" /part_list/part[part_ID='{part_ID}']/color sep=" " >#>

This will return the color values for that part as a string that may have one or multiple values. This string can be used in boolean tests to check for particular colors, or can be used as text to add to a collection variable as part of a report.

This makes the actual MetaBlock logic more complex and difficult to read, but can be an effective way to handle more complex XML data. As long as at least one column is defined that returns unique single values, the embedded XML commands can recover any other data in the XML file that is needed.

> **NOTE:** Embedded XML expression for "color" can ONLY be used in the actual MetaBlock rules in the Logic Block. It CANNOT be used in the XPath commands used to select column data in the XML MetaBlock specification file.

Use Single XPath Command

A quick way to build consistent commands for simple XML files is with the “Use Single XPath Command” option. This can only be used if the names of the columns **exactly** match the XML element names, including upper/lower case letters.

Enter a XPath command up to the final element name in the “Use Single XPath Command” edit box and click the “Update” button. Corvid will replace the XPath command for each column with the text entered, followed by the column name.

In the example, enter:

/part_list/part/

and click “Update”. The XPath commands for each column would become “/part_list/part/” followed by the column name entered. These would be:

/part_list/part/part_ID
/part_list/part/price
/part_list/part/color
/part_list/part/material

This works ONLY when the column name matches the XML element name and each item only returns single values; but is a very convenient way to build commands for correctly structured files.

Delete and Move Commands

To delete a row in the XML MetaBlock specification, click on the row to select it and click the “Delete” button.

Rows can be moved and reordered by selecting them and then clicking the “Move Up” and “Move Down” button - however, the order of the columns has no effect on how a MetaBlock will run.

If more rows are needed in the XML MetaBlock specification file, click the “Add Blank Rows to End” button.

Save

When the XML MetaBlock specifications are input, click the “Save” button to return to the MetaBlock window. The columns entered will be displayed in that window and can be used in building the Logic Block.

16.5.11 Using XML Commands with the Applet Runtime

The Corvid Servlet Runtime has the XML command functionality built into it. However, the Corvid Applet Runtime does not. Instead the Applet Runtime uses a special "helper applet" program to process the XML commands. This reduces the overhead both in size and Java requirements for the Corvid Applet Runtime.

In most cases, the special steps Corvid takes to process XML commands in the Applet Runtime are transparent to the developer and users of the system. However, in some cases developers must make special modifications to their HTML pages and the files distributed with their system.

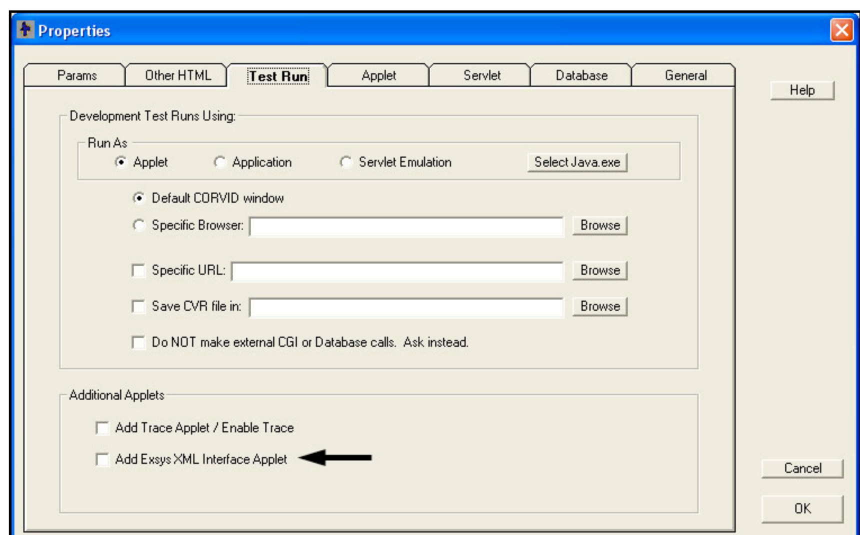
When a system using XML commands is run with the Applet Runtime, Corvid must add a special XML Interface applet to the HTML page used to run the system. The code for this applet is:

```
<APPLET
CODEBASE = "./"
CODE = "EXSYS_XML_Interface.xml_lookup.class"
NAME = "EXSYS_XML_Interface"
ARCHIVE = "EXSYS_XML_Interface.jar"
WIDTH = 0
HEIGHT = 0
HSPACE = 0
VSPACE = 0
ALIGN = middle
>
</APPLET>
```

This can be put anywhere on the HTML page, but just before the Corvid Runtime Applet tag is suggested. The EXSYS_XML_Interface.jar file must also be included in the folder of files for the system, along with the ExsysCorvid.jar, system CVR file and any other associated files.

When Corvid builds the default system HTML file to run the system during development, it checks to see if XML commands are used. If they are, Corvid automatically adds the XML Interface applet tag and copies EXSYS_XML_Interface.jar to the correct folder.

For most systems, no special additional actions are needed to use XML commands with the applet runtime. However, while Corvid checks the places where XML commands would typically be used, it does not check for embedded XML commands in strings or external files. If XML commands are used in such places and Corvid does not automatically include the XML Interface Applet, go to the "Properties" window "Test Run" tab.



Check the "Add Exsys XML Interface Applet" check box. With this checked, Corvid will always add the XML applet.

If a custom HTML page is created to field the system, the XML Interface Applet tag must be added to that window.

XML Interface System Requirements: The Corvid Applet Runtime will work with virtually any version of Java, however the XML Interface Applet requires Java v1.6 or higher. When Applet systems using the XML commands are distributed, the end user must have Java 1.6 or higher or the XML Interface Applet will not work. For the vast majority of machines, this will not be a problem, but end users with older versions of Java that have not upgraded may not be able to run systems with XML Commands. If this may be a problem for your target end users, the best solution is to use the Corvid Servlet Runtime, which puts no limits on the end user's machine.

16.6 Database Command Details

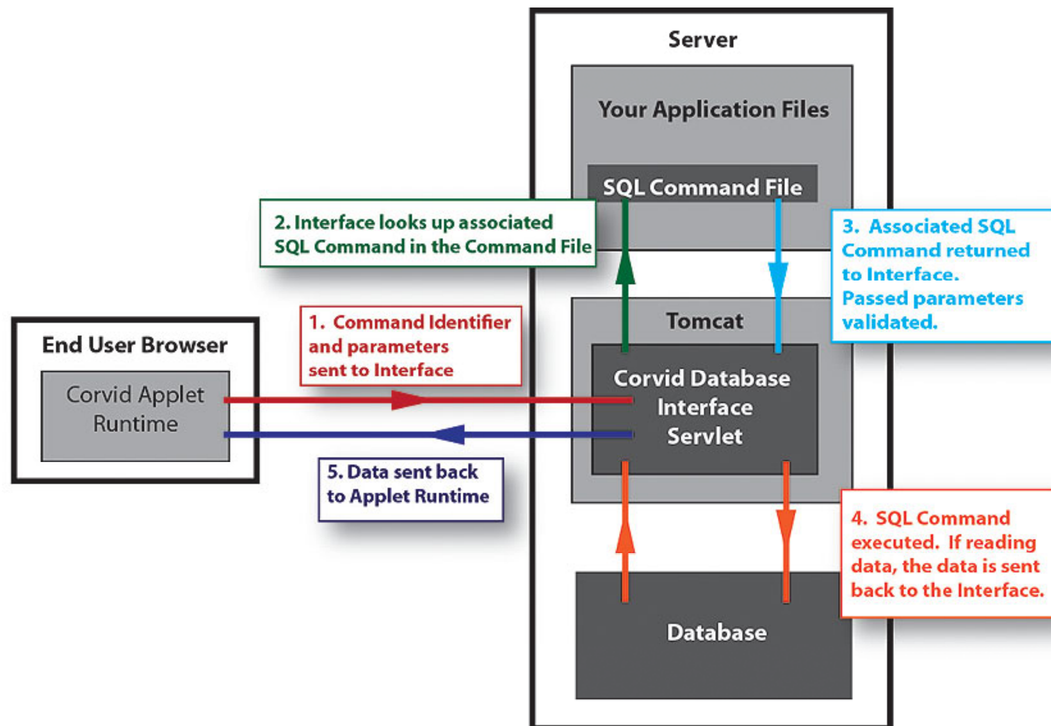
16.6.1 How the Corvid Database Interface Works

The Corvid Database Interface is a little more complicated than the other external interfaces due to Java and security issues that have to be addressed in allowing the Corvid Applet to communicate with a server database. While these issues really only apply to the Corvid Applet Runtime, to make systems more portable between the Corvid Applet Runtime and Corvid Servlet Runtime, the same basic approach is used in both delivery modes.

Servlet Support Requirement: The Corvid Applet Runtime cannot directly access a database on a server or execute SQL commands. The actual execution of the database commands is done by a special Corvid Database Interface servlet running on the server. This Java servlet program is provided with Exsys Corvid, but does require a server that supports Java Servlets via Tomcat, Glassfish, Websphere or other "Servlet Container". Without this server-side program, the Corvid Applet Runtime can not use database commands.

> **Note – Another factor to consider:** Since a Java Servlet and a "servlet container" such as Tomcat are required for systems using database commands from the Applet Runtime, it may make more sense to just use the Corvid Servlet Runtime for systems using database commands. The Corvid Servlet Runtime provides the higher levels of security for systems that use database commands and provides many more user interface and delivery options.

Database Command File: The Corvid Applet Runtime calls the server-side Corvid Database Interface program to execute SQL commands. If the Corvid Database Interface program simply accepted SQL commands sent to it and executed them, this would open serious security vulnerability since anyone knowing the correct URL could execute any SQL command on a database. To prevent this, the Corvid Database Interface only will execute specific SQL commands stored in a file on the server. These commands should be limited to only what the system needs. The Corvid Database Interface is passed the identifier for a command in the file along with any parameters that command needs. The Corvid Database Interface looks up the SQL command in the command file associated with the identifier, checks that the parameters passed match correctly and executes the SQL command.



1. The Corvid Applet Runtime running on the end user's computer sends a Corvid Database command identifier and parameters to the Corvid Database Interface Servlet running on the server. This is only the identifier for a SQL command and not the command itself.

2. The Corvid Database Interface program looks the identifier up in the SQL Command File to find the associated SQL command

3. The actual SQL command and any restrictions on parameters is returned to the Corvid Database Interface program. Any parameters passed from the Corvid Applet Runtime are validated against the parameter restrictions.

4. The parameters passed are combined with the SQL Command to produce a full SQL command that can be executed against the database. In some cases, this may add data to the database, but often it obtains data from the database.

5. The data is returned to the Corvid Runtime Applet.

Important Security Concerns: The SQL Command File approach used with the Corvid Applet Runtime greatly limits the SQL commands that can be executed on your database. However, someone with the correct knowledge COULD execute the commands in the SQL Command File.

> ***If this presents an unacceptable security problem for your database, you should run your system with the Corvid Servlet Runtime and not install the Corvid Database Interface (CorvidDB.war).***

The Corvid Servlet Runtime will only execute your Corvid application and cannot be called to execute any SQL commands directly, even if they are in the SQL command file.

16.6.2 Server Files Required for the Database Interface

Various files are needed depending on your server, database and which Corvid Runtime is being used (Servlet or Applet).

1. There must be some database software installed on the server. It can be any brand that supports ODBC or JDBC connections, such as MySQL, Access, Oracle, SQL Server and almost all other standard databases.

2. There must be a database file that you will be using with your Corvid system.

3. The server needs to support Java Servlets. This requires a "Servlet Container" such as Tomcat, Glassfish or IBM Websphere. Java servlets are required for the database interface even if the rest of your system is being run with the Corvid Applet Runtime.

4. If the system is being run with the Corvid **Applet** Runtime, the servlet interface CorvidDB must be installed. This is done by having the Servlet Container deploy the file CorvidDB.war. In the case of Tomcat, this is done by putting CorvidDB.war in the Tomcat webapps folder. (If unsure how to deploy java servlets on your system, contact your system administrator.) **NOTE: If your system is being run with the Corvid Servlet Runtime, CorvidDB.war is NOT needed and SHOULD NOT BE INSTALLED on the server.**

5. Your KB system files (.CVR) and any associated files. If you are running with the Corvid Applet Runtime, the KB system files MUST be on the same server as you have running the CorvidDB servlet. (Java security prevents you from having the Corvid KB system files on one server, but calling the CorvidDB servlet on another server.) The system files should be in a subfolder off "root" or other folder that can serve normal HTML web pages. The standard location of "root" on some popular servers are:

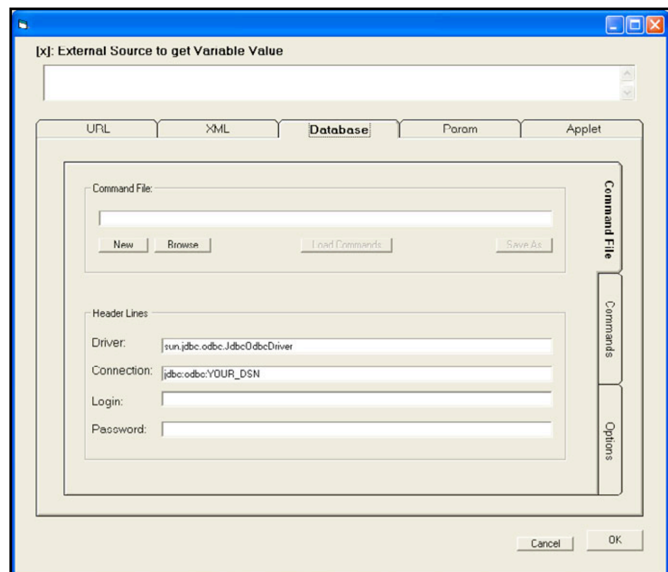
IIS	\inetpub\wwwroot
Apache	wwwroot
Tomcat	webapps/ROOT

16.6.3 Building Corvid Database Commands

Because of the SQL Command file approach Corvid uses, building Corvid database commands has 2 steps:

1. Defining the SQL command with its identifier, parameters and restrictions. This goes in the Corvid SQL Command File that will be put on the server.

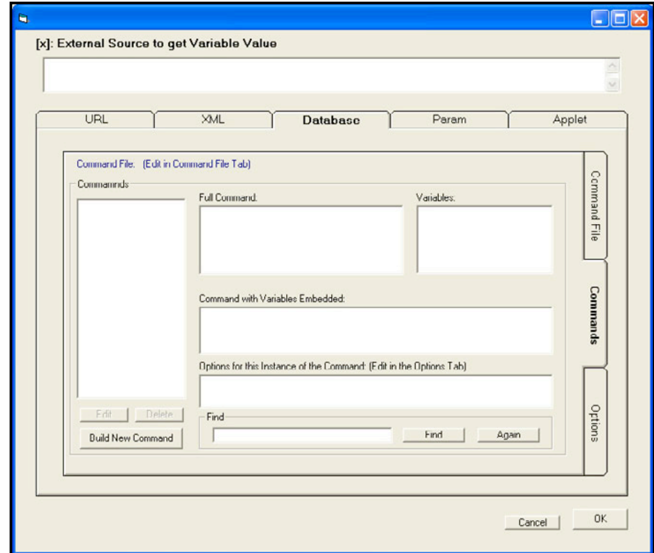
2. Building the command used in the actual Corvid system that refers to the command by its identifier and sets the parameters using the values of Corvid variables.



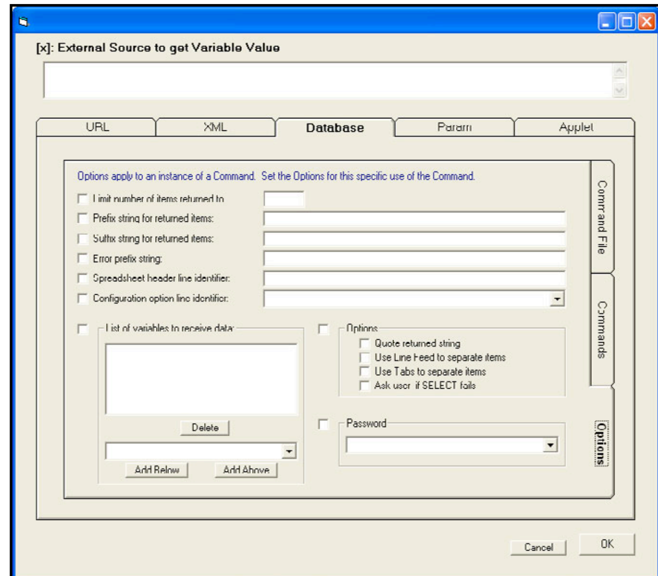
In addition, the overall database interface properties such as ODBC/JDBC driver, DNS and sometimes passwords need to be setup.

These operations are done from the External Interfaces window on the "Database" tab. The "Database" tab has 3 additional tabs on the right side. First "Command File" is to define the name of the SQL Command File that will hold the commands and be put on the server, along with the Driver, connection DSN and password information when that is needed.

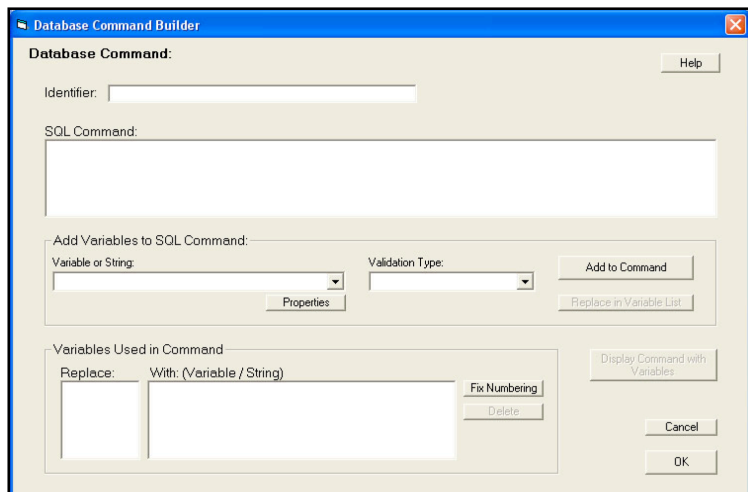
The second tab, “Commands” displays the commands that have been added along with the Corvid variables that will be used to fill in the parameters for the command. This displays the commands in a readable form, though they will actually be broken into 2 portions one in the Corvid system and one in the SQL Command file on the server.



The 3rd tab is for special options that can be added to some commands.



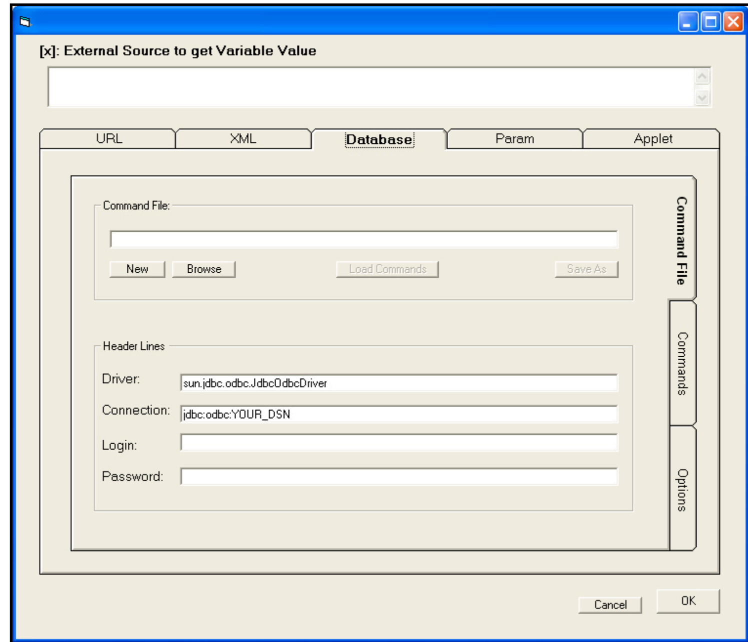
New commands are added on the Command tab by clicking the “Build New Command” button. This opens the command builder window:



This makes it easy to build SQL commands that have replaceable parameters based on Corvid variables.

16.6.4 The Database “Command File” Tab

The database “Command File” tab is used to set the name of the SQL Command file and connection information, that will be used by all of the database commands for the system.



The information on the Command File tab is required whenever adding database commands to a system.

Command File

The first step is to create or open the command file that will hold the connection information along with the allowed SQL commands. All the commands and options will write to this file so the file must be created or selected before going to the other tabs.

Most systems only have a single command file. However, multiple command files can be used for systems that connect to multiple databases using different DSNs.

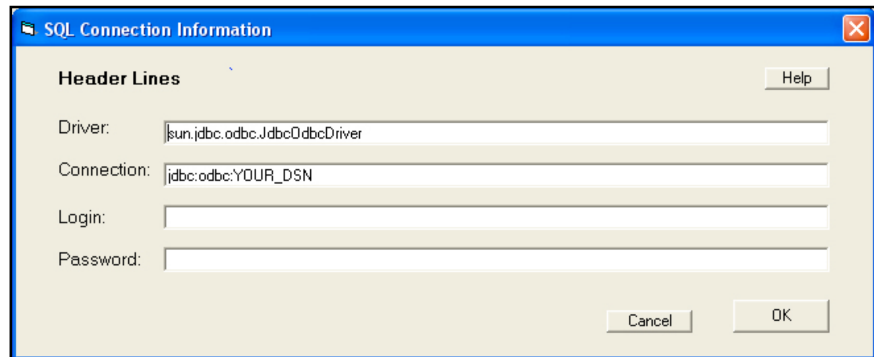
Remember, the command file will be moved to the server.

When using the Corvid Applet Runtime, the command file should be put in the CorvidDB folder created when CorvidDB. war deploys. If using the Corvid Servlet Runtime, the command file is put in the same folder as the system CVR file.

In most cases, the copy of the file being edited is a local copy. If there is already a file on the server, you may need to download it to have the latest version of the command file. After changes are made, be sure to move the edited file up to the server for the changes to take effect.

- To open an existing file, click “Browse” and select it, or enter the name of the file and click “Load Commands”.
- Once a file is opened, to save it with a different name, click the “Save As” button.
- To create a new file, click “New”. A dialog will be displayed asking for the connection information:

Fill in the connection information as described below. This information can later be edited on the Command File tab.



Command files normally have a .cdb file extension.

Driver

Enter the driver for your database. In most cases, this will be the Java ODBC/JDBC driver:

sun.jdbc.odbc.JdbcOdbcDriver

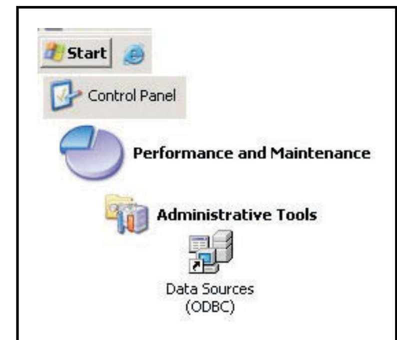
This will be pre-filled in the driver field. Use this unless your database requires something different. (Check with your System Administrator if this driver does not work with your system.) This driver will be used by the Corvid Database Interface. It is not a driver that you need to install.

Connection

To use an ODBC connection to your database, a DSN must be created. **This must be a System DSN.** Other types of DSN will not work. Your DSN must have a name. In the "Connect" edit box enter:

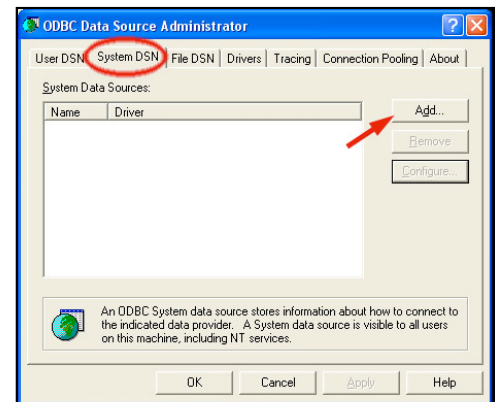
jdbc:odbc:Your_DSN_Name

In MS WIndows, DSNs are created from the control panel. Go to the "Start" button and select "Control Panel". Select "Performance and Maintenance" and then "Administrative Tools" (In XP, just directly select "Administrative Tools" from the Control Panel). Then select "Data Sources (ODBC)"

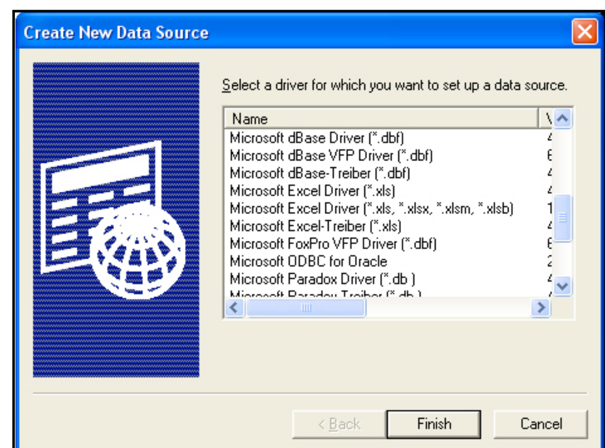


This will open the window for creating a DSN.

1. Click on the "System DSN" tab
2. Click the "Add" button.

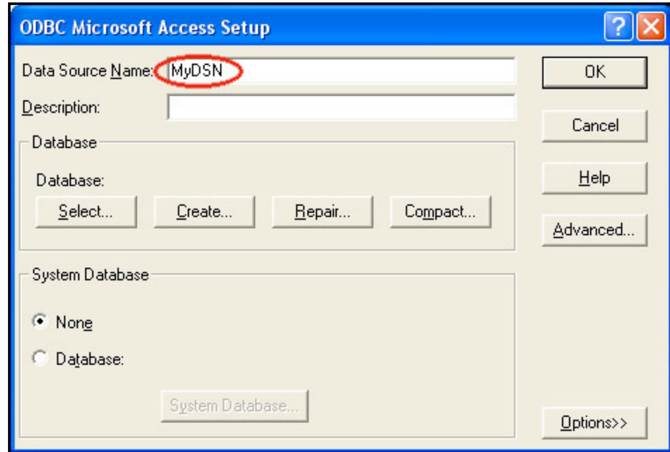


3. Select your database type:



4. Give the DSN a name:

   This can be anything you wish. Remember this name as you will need it in the Corvid Database Connection information. Depending on your database, there may also be various other parameters to set



5. Back in the Corvid Database Interface Connection window, for the "Connection" enter:

   jdbc:odbc:MyDSN

   where "MyDSN" is the name you gave to the DSN.

Login / Password

If the database requires a login and password, enter them, otherwise leave these fields blank. If a login/password is entered, it will be stored in the Database Command file. This file will be on the server, and it should be handled with whatever security limitations and precautions are appropriate for the password.

If there are security concerns about either having the login/password in a file on the server, or in having these essentially hard coded into the system, you can have the end user provide them. To do this:

1. You must be running the system with the Corvid Servlet Runtime. (This technique will not work with the Corvid Applet Runtime.)

2. In your system, create Corvid string variables for the login and/or password.

3. In the database connection window, for the login / password enter the name of the string variable in double square brackets with the .VALUE property.

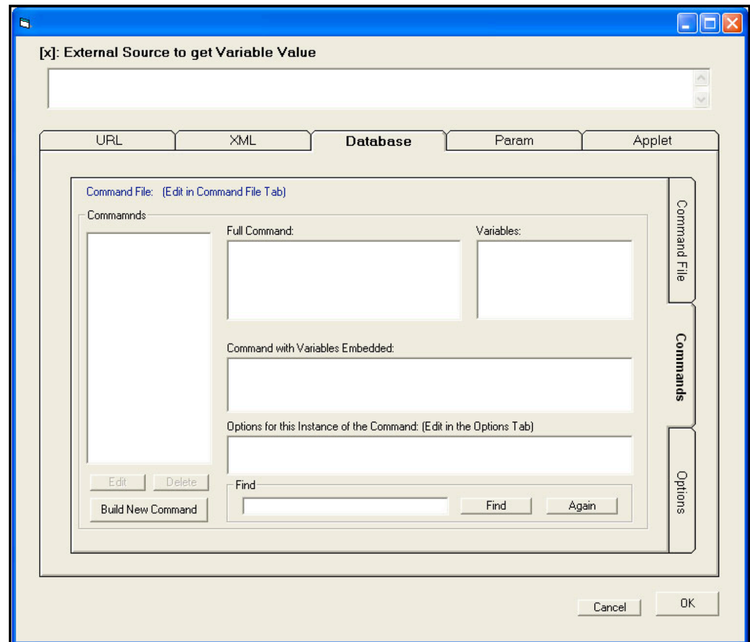
4. In your system, add a command in the command block to force the login / password variables to be asked before any database calls are made. (If this is not done, the variables will be asked when the first database call is made, which may not be the order of questions desired for your end user interface)

This way, the user will be asked to provide the login and/or password and nothing sensitive is hard coded in the file. If they cannot provide the password, the database commands will not execute, but the rest of the system will run.

For example, create Corvid string variables [TheLogin] and [ThePassword] and have these asked of the end user before any database calls are made. In the connection data window, for the login use [[TheLogin.value]] and for the password use [[ThePassword.value]]

16.6.5 The Database "Commands" Tab

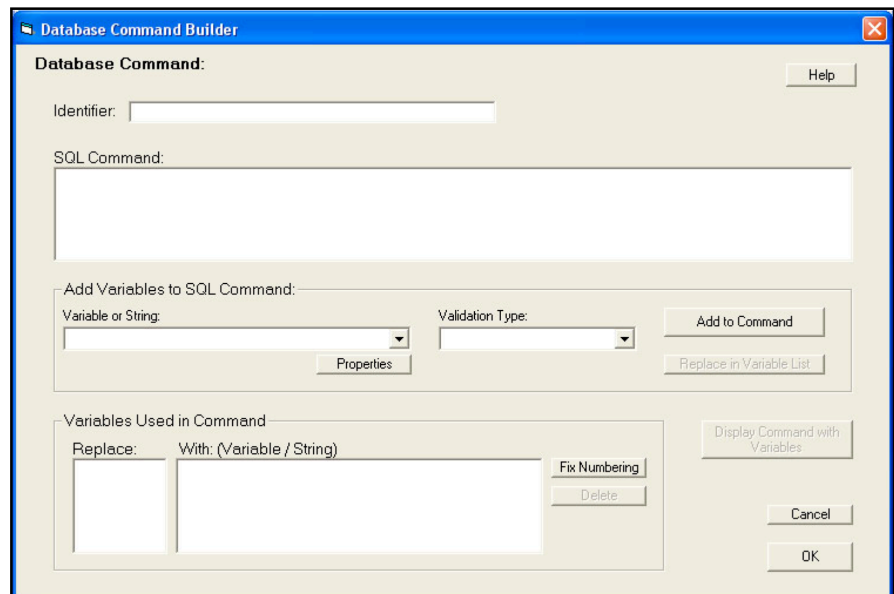
The "Commands" tab displays the SQL commands in the command file. These are the only commands your system will be able to execute.



Commands are made up of standard SQL along with Corvid variables. For a particular database call, one of the commands is selected. Adding database calls to a system is a combination of defining a command that can be used, and then associating it with a particular database action such as getting the value of a variable.

The SQL command to execute can be any SQL command starting with SELECT, UPDATE, INSERT, DELETE, ALTER, CREATE, DROP. In addition, some databases and drivers may allow other SQL commands.

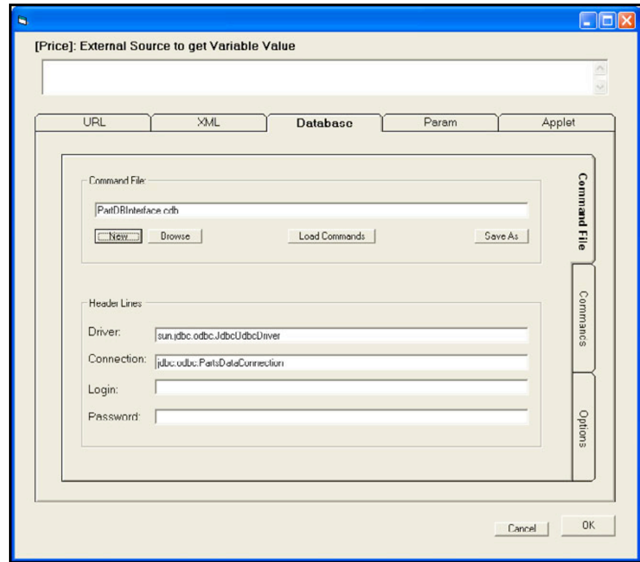
First, a command must be added. This is done by clicking the "Build New Command" button, which displays the Database Command Builder.



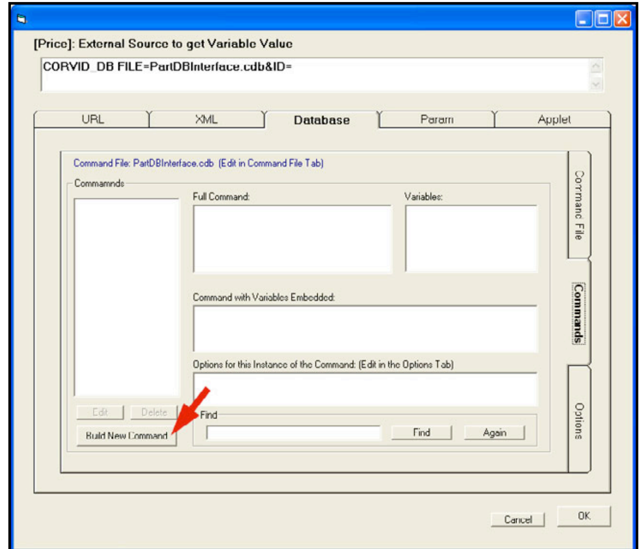
As described at the start of the database interface section, in order to provide a greater level of security when using the database interface from the Corvid Applet Runtime, the actual SQL commands are stored in a file on the server. These commands are a combination of standard SQL and replaceable parameters. The Corvid Applet Runtime only sends the name of the command to execute and the parameter values. It is only possible to execute the specific commands in the server database command file. This allows the database to be accessed by the client-side applet, while limiting the potential for other unauthorized users to execute other commands. (For greater security, it is recommended that systems needing the database interface use the Corvid Servlet Runtime rather than the Corvid Applet Runtime.)

Building a database command calls for entering a combination of SQL and parameters that will get their value from Corvid variables. The Database Command Builder makes this easy. The process is best illustrated by an example of building a command to get the price of a part from the database.

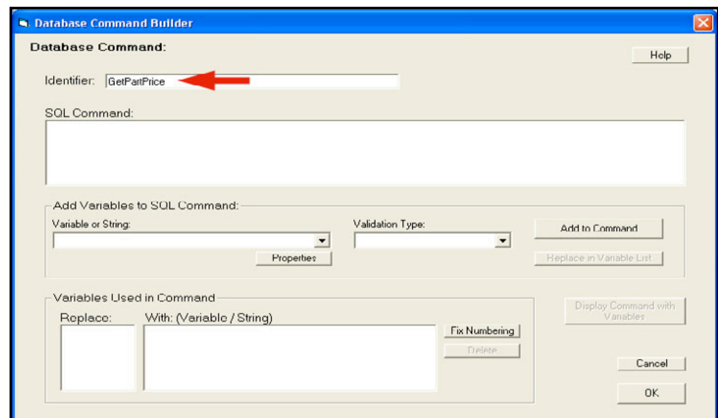
1. Create a new database command file and DSN as described above. Here the command file is **PartDBInterface.cdb**. The DSN is “PartsDataConnection”.



2. Go to the “Commands” tab on the right side.



3. To add a command, click the “Build New Command” button at the lower left. This opens the window for building commands. First enter an identifier. This can be any text, but should be simple and easy to recognize. For example, "GetPartPrice."



4. In the "SQL Command" edit box start entering the SQL command. This can be any SQL command starting with SELECT, UPDATE, INSERT, DELETE, ALTER, CREATE, DROP. (In addition some databases and drivers may allow other SQL commands.)

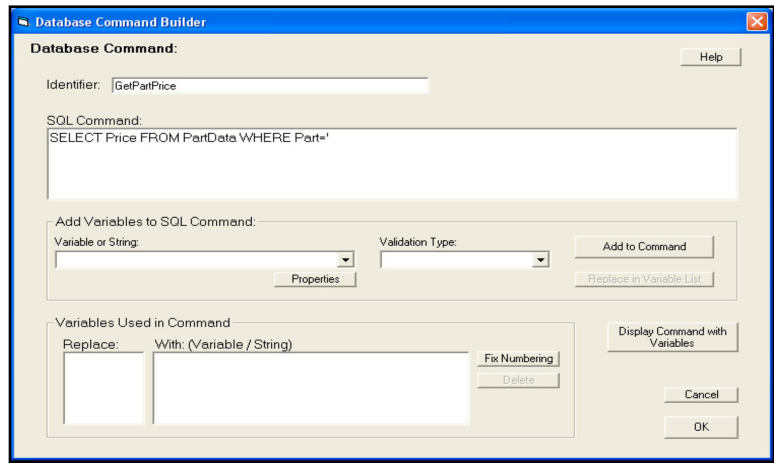
   Here we want the command:

   SELECT Price FROM PartData WHERE Part='partID'

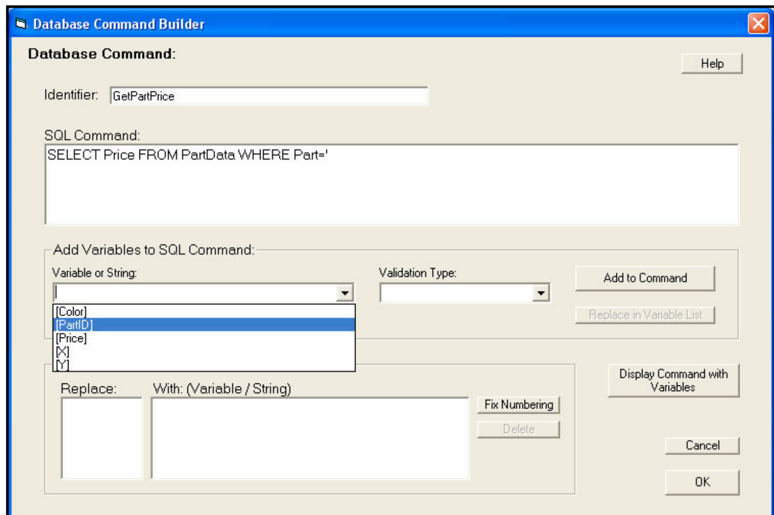
   "partID" will be the name of the part and will come from a Corvid variable.

   Enter the portion of the SQL command up to where the Corvid variable value would be added. Here this will be:

   SELECT Price FROM PartData WHERE Part='

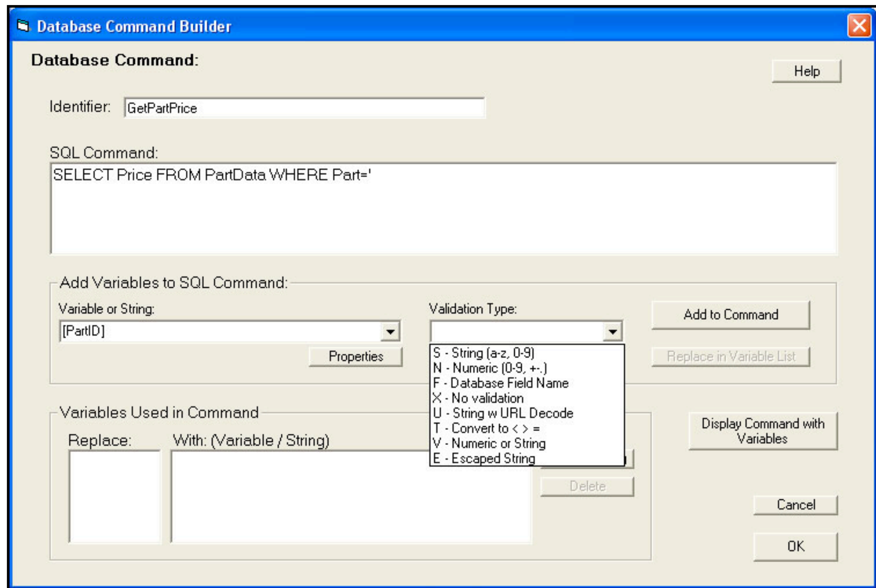


5. Now add a replaceable parameter in the SQL command and associate a Corvid variable with it. The replaceable parameters in the SQL are denoted by curly brackets **{ }**. The brackets contain a letter and a number. The letter indicates the type checking and restrictions that should be applied to the parameter value, and the number is the number of the parameter in the command (e.g. the first parameter is "1", the second is "2", etc). These numbers match the list of variables that Corvid will provide in making the call.



   To add the replaceable parameter, go to the "Add Variables to SQL Command" section and click the "Variable or String" drop down list. This will display a list of the Corvid variables in the system. Select the variable to use. Here this is the string variable [PartID]. (Note: Instead of selecting a variable, a string can be typed into the edit box. This is used with commands that can be called in different ways and it is easier to just have a parameter be a string. This is illustrated in more detail in the next example.)

6. Now select the Validation Type for the parameter value. Validation types allow you to restrict the type of value that the command will accept. When using the Corvid Applet Runtime this is one more layer of protection against inappropriate use of the command. When using the Corvid Servlet Runtime, strongly limiting the value is less important unless there are ways for the end user to directly type in values that will be passed to the command. However, setting the Validation Type is always a good idea. To set the Validation Type, click the drop down list.



The Validation Types are:

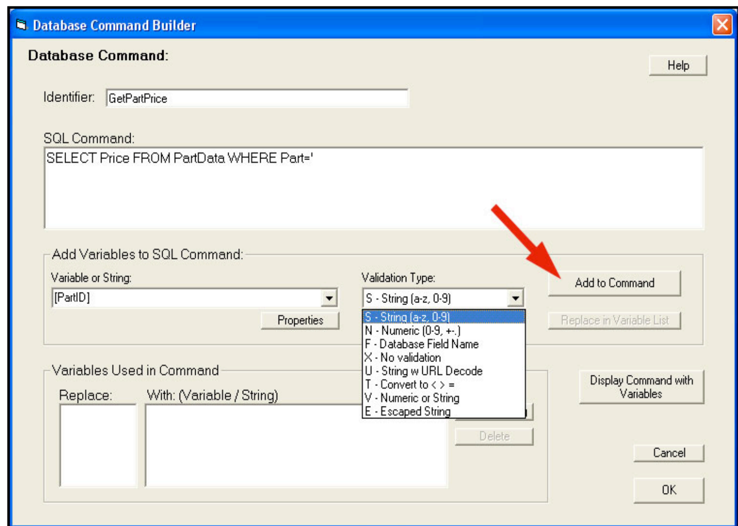
S	**String.** Allows any alphanumeric characters. Quotes and ticks will not be allowed. The value will automatically be URL decoded.
N	**Numeric**. Allows only the characters 0-9, '+', '-" and '.'
F	**Field.** Must be a valid syntax for a database field name. The existence of the field in the database is not checked until the command is actually executed. Allows only alphanumeric characters and ' _').
X	**No validation**. String just passed through.

Some of the letters do not do type checking but instead do some form of conversion:

T	**Convert** EQ,NE,LT,LE,GT,GE to =,<>,<,<=,>,>= respectively.
V	**Unknown value type**. If it contains all numeric characters, it will be handled as a numeric. Otherwise, it will have ticks ' ' put around it and handled as a string. If you use V, you must not use a string that contains only digits or it will think it is a numeric and not tick it. The V is used when you have something like, "WHERE {F#} ={V#}". Since you do not know what field will be used, you do not know what the field type will be. By using V it will automatically tick it if it looks like a string value.

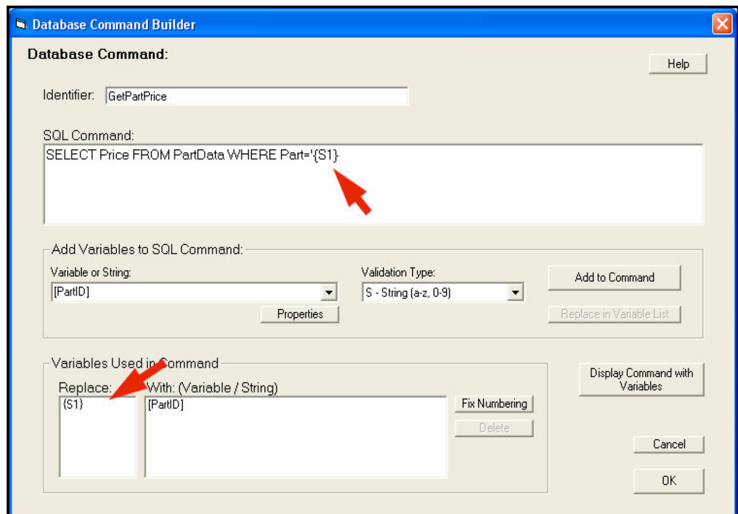
Here we can use the String type since the part name is alphanumeric.

7. Once the Validation Type is selected, click the "Add to Command" button.

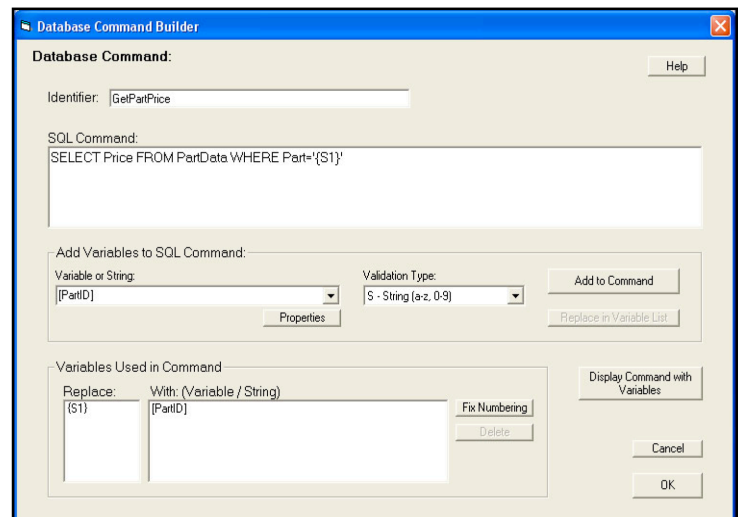


The replaceable parameter {S1} will be added to the SQL command and the lower list box will show that {S1} corresponds to [PartID]. In the {S1} the "S" indicates the validation type is String and the "1" indicates this is the first replaceable parameter in the SQL command.

If the SQL command required additional replaceable parameters, you would continue entering the SQL command up to the next command and then add that Corvid variable the same way.

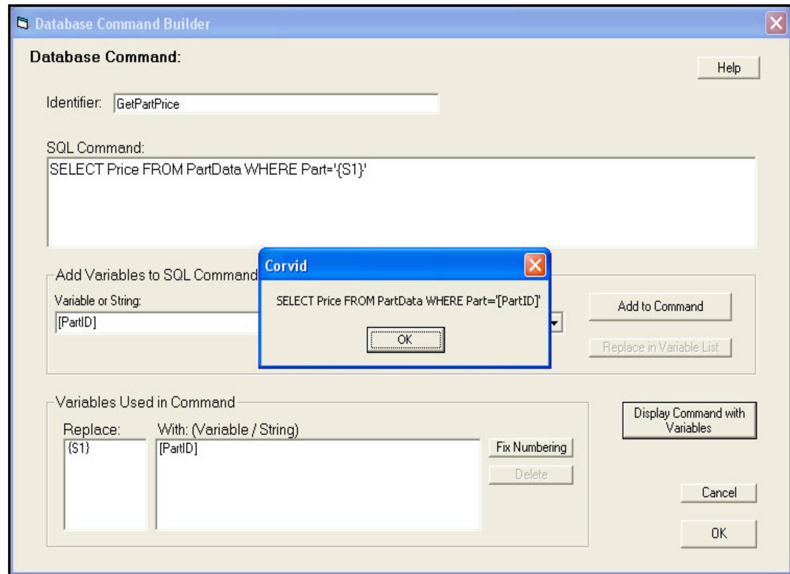


Here all that is needed to close the ' around the {S1}. **Replaceable parameters are just replaced by the value passed. If SQL syntax calls for ticks or quotes around a value, those must be added in the command around the replaceable parameter**.



To view the command with the Corvid variable(s) directly embedded in it, click the "Display Command with Variables" button.

This often makes it easier to check the command and make sure the correct Corvid variables are being used.

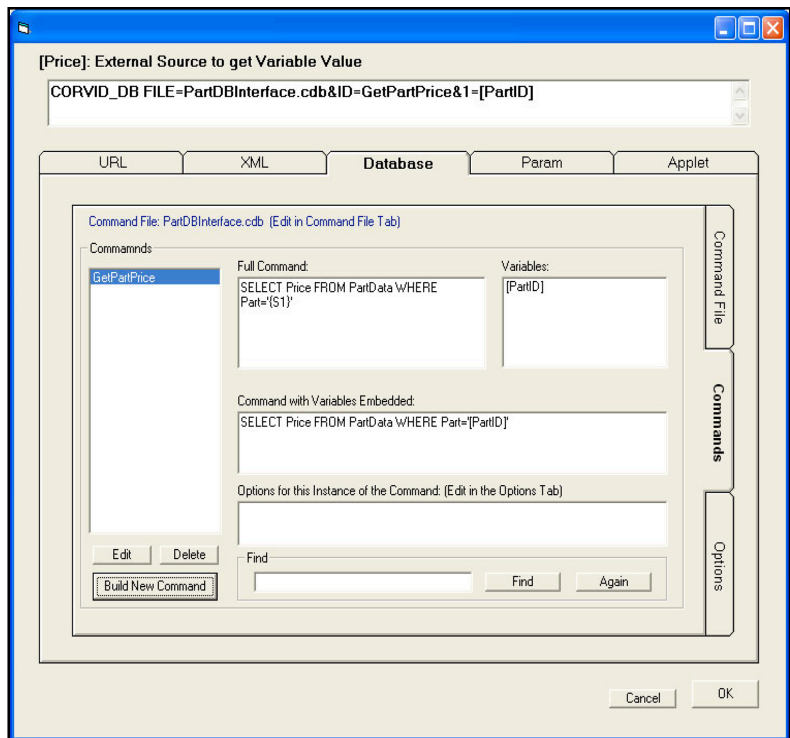


Now that the new command is built, click the "OK" button to return to the Command tab.

The command list now shows the command just added and it is selected in the command list.

- The "Full Command" shows the command with the replaceable parameters.
- The "Variables" list shows the variables associated with the command in order.
- The "Command with Variables Embedded" shows the command with the variables in the command.

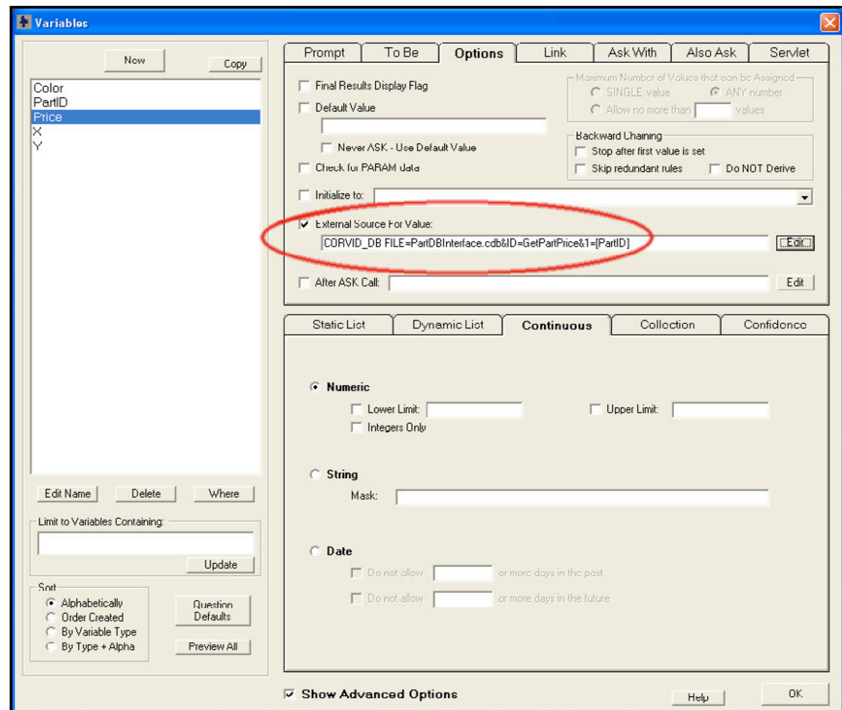
These are to make it easier to find and use commands. They cannot be directly edited in this window. To edit a command, select it and click "Edit" to reopen the command builder window.



The command at the top of the screen is the actual command that will be added to the Corvid system. This is a CORVID_DB command. The "FILE=" is the name of the Database Command File on the server. The "ID=" is the name of the SQL command to use from the command file. The "1=" is the value for first the replaceable parameter and here that value comes from the variable [PartID]. If there were more replaceable parameters, there would be a "2=", "3=", etc. with associated variables.

Clicking “OK” returns the external data interface command to wherever the interface command was built from:

As many database commands as needed may be added to the system in the same way. When done, be sure to move the command file (here, “PartDBInterface.cdb”) to the server. (For the Corvid Applet Runtime, put the Command File in the CorvidDB folder created when the CorvidDB.war servlet deploys. For the Corvid Servlet Runtime, put it in the same folder as the system CVR file.)



16.6.6 Complex Commands with Replaceable Field Names

This example shows how to add a more complex command. Suppose the system needs to look up information on a customer's address, but to help make sure it is the actual customer the system will check both their name and one other item of information that the customer can select. This second item can be their customer ID, home zip code or last order number. Also, the information on the customer’s address is stored in different fields in the database and these should be read individually. This requires building a SQL command that has replaceable parameters for:

- The field to read
- The customer’s name
- The second field to check
- The value for the second field

By building the command with replaceable parameters, this can be used in multiple ways.

The full SQL command will be:

SELECT {F1} FROM customerData WHERE CUSNAME='{S2}' AND {F3}='{S4}'

Remember that the replacement parameters are numbered. The numbers correspond to the parameters passed on the URL line. The letters indicate the type of data. In this case the validations are “F” which checks that the value is a valid field name for the database, and “S” which allows an alphanumeric string.

{F1} The field to return data from. This will be a string based on how the command is used.

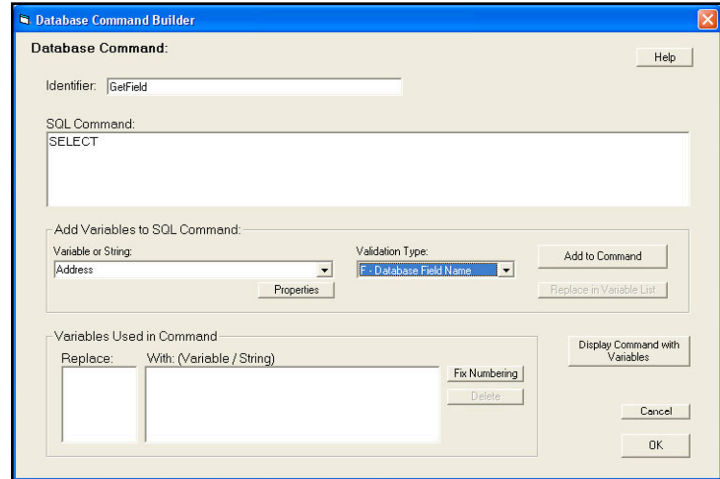
{S2} The name of the customer. This will come from the String variable [Name]

{F3} The name of the second field to check. This will come from a Static List variable [ID_Type]

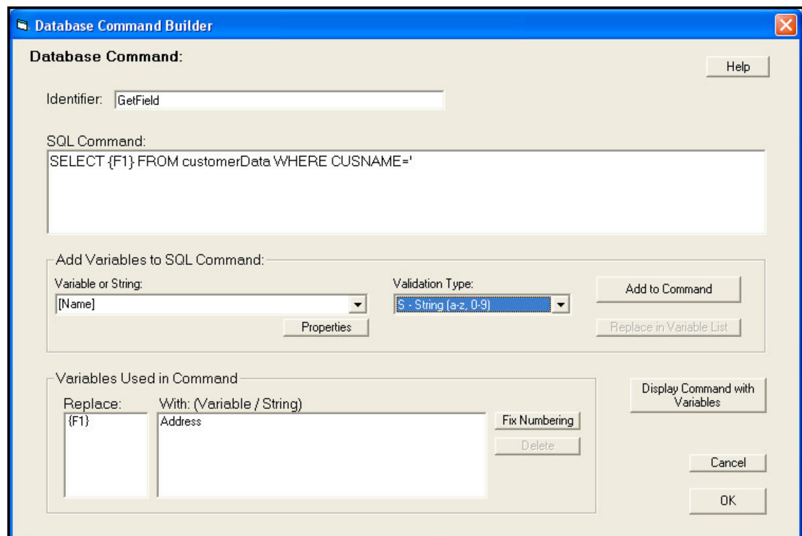
{S4} The value that the second field must match. This will come from the string variable [ID_data]

The first use of this command will be to get the "Address" field, though the same command can then be used later to get other fields from the database.

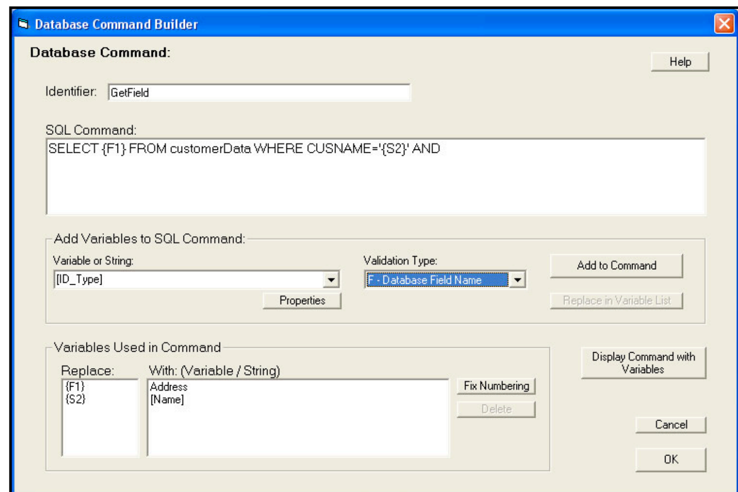
To build the command, go to the External Interface window, Database tab and click “Build New Command”
Enter the portion of the SQL command up to the first replaceable parameter – "SELECT".



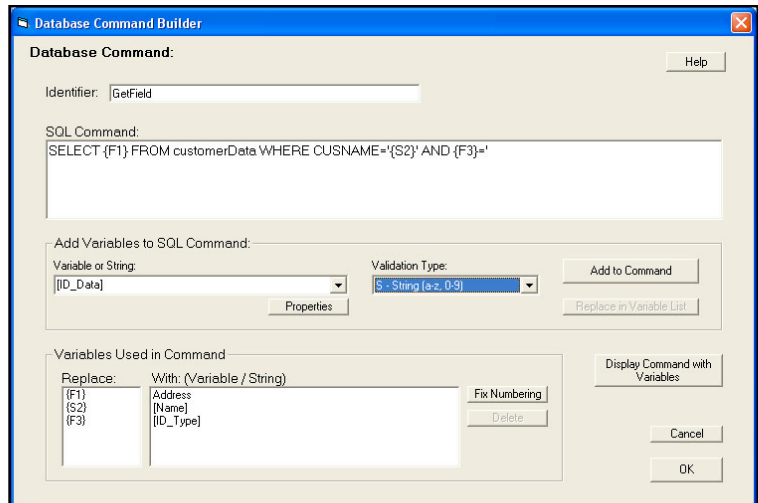
This time, since the field name is static and does not come from a variable (although it could), just type in the field name “Address” and select the “F” validation type since this is a field name. Click “Add to Command”. Continue adding the next portion of the SQL command up to the next replaceable parameter.



This time the replaceable parameter does come from a Corvid variable, so select the variable, [Name], from the drop down list and the “S” type for a string value. Add this to the command and continue out to the next replaceable parameter.

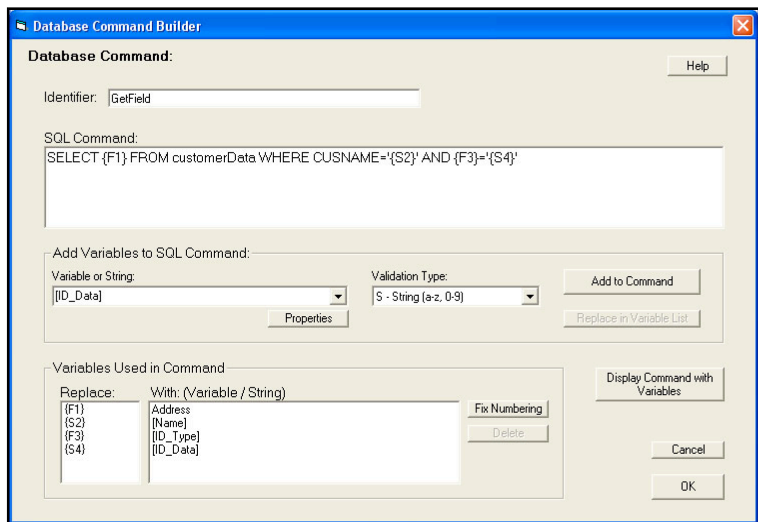


This time the replaceable parameter is again a field name, but it is limited to specific options in the Static List variable [ID_Type]. Select that variable and the “F” validation type for a field. This can continue to the final replaceable parameter.

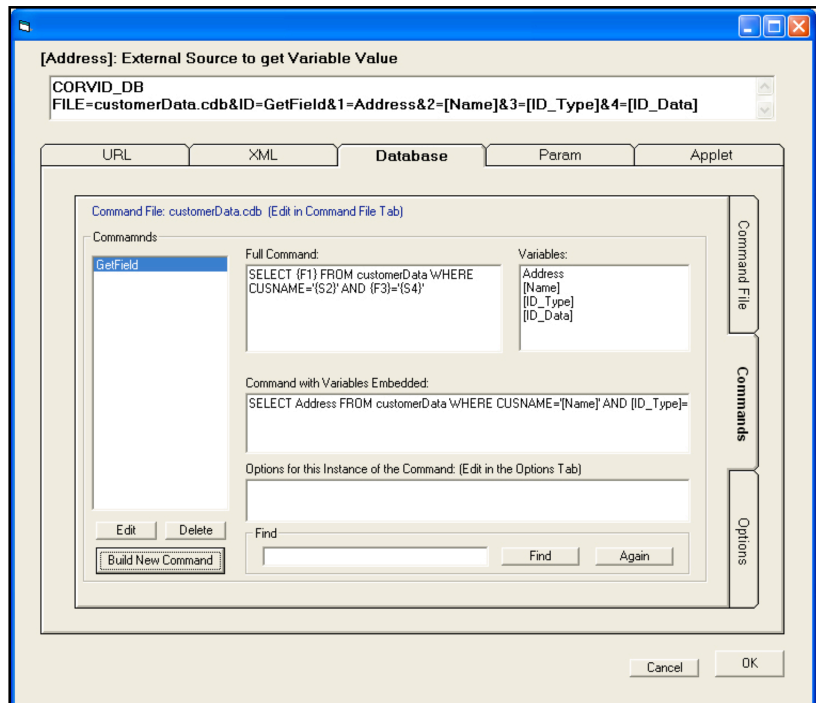


After the end user selects the second field they want to use, the system will ask for the value for that field and store it in the variable [ID_Data]. This is used for the last replaceable parameter in the command.

Click OK to add this command to the command list.

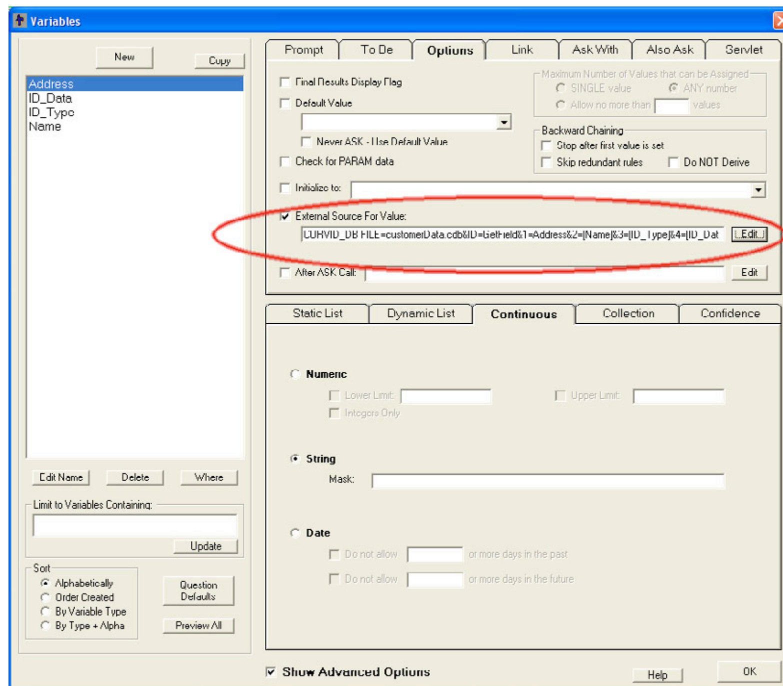


Click OK to add the CORVID_DB command as the external interface command to get the value for the variable [Address].



Notice that the actual command in the system is:

CORVID_DB FILE=customerData.cdb&ID=GetField&1=Address&2=[Name]&3=[ID_Type]&4=[ID_Data]



The command in the SQL Command File can be used to get the value of any field in the database just by making a small change in the command. If there is a variable [City] that should be set by the “City” field in the database, just copy the command, but change “Address” to “City”

CORVID_DB FILE=customerData.cdb&ID=GetField&1=City&2=[Name]&3=[ID_Type]&4=[ID_Data]

Since the replaceable parameters can be changed as needed, SQL commands can be designed so that they can be used in multiple ways.

Important Security Considerations:

If database field names are made into replaceable parameters, and the Corvid Applet Runtime is used, users can potentially read any field in the database. Remember, the URL to the database interface servlet can be typed by hand into a browser, therefore you cannot control what the parameters will be. If replaceable field names are used, it is possible for users to access any field in the database. If this causes security concerns, the field names should be "hard coded" into specific individual commands in the SQL Command Table and called by name rather than using “generic” commands.

You should limit the allowed SQL commands to allow ONLY as much as you are willing to expose to outside users.

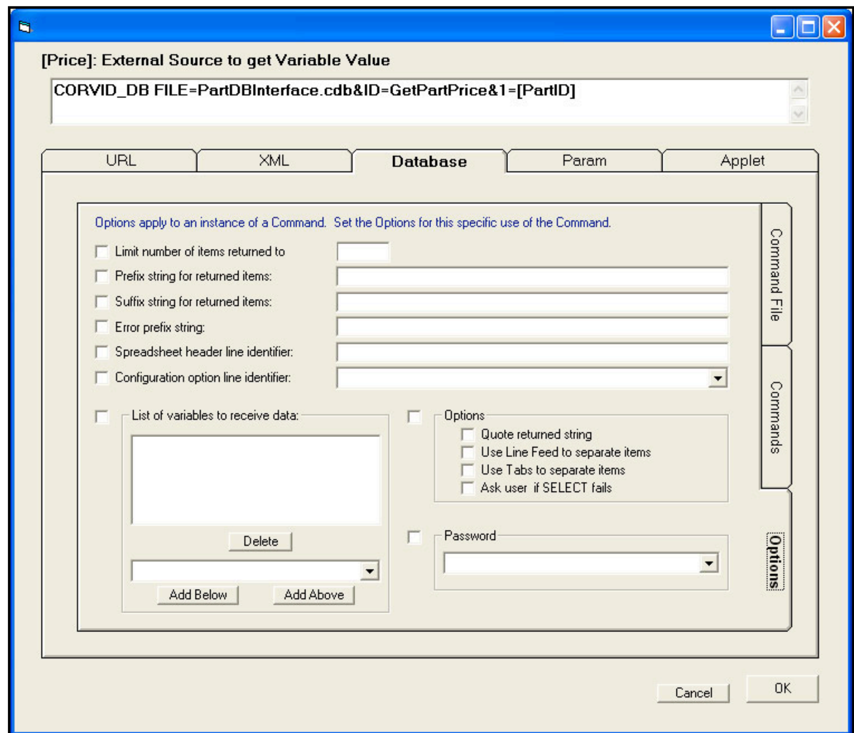
If the database contains sensitive information, the Corvid Servlet Runtime should be used. Even with the Corvid Servlet Runtime, the replaceable field names should be set by logic in the system, such as specific values from a Static List, rather than using a string value that the user types in.

16.6.7 “Options” Tab

Each command can have a variety of options. To set these options click on the Options tab.

Options apply to the individual use of the command, so a single command can be used multiple times with different options.

The command that the options apply to is displayed at the top of the window. To select a different command, click the “Commands” tab, select a command and return to the “Options” tab.



A command can have multiple options.

Limit Number of Items Returned: NR=#

Sets the number of items of data being returned from a SELECT command. The default is to return all data from the command, but in some cases it may be useful to limit this to only a particular number. If there is no NR= command, the default is to return all data items. If NR= is used, the value must be greater than 0. This command should only be used with a SELECT command.

Prefix String: PF=

This is a string that will be added before each row/record's data. It is a way to format data and add text as needed before the returned value. The string will be part of a URL and should be URL encoded. For example, a returned item of data added to a HTML report might need to have an HTML tag wrapped around it. Usually there are other ways to handle this when the variable value is added, but this approach can be convenient.

Suffix String: SF=

This is a string that will be added after each row/record's data. Like the Prefix String, this is a way to format data and is often used in conjunction with a prefix string. The string specified should be URL encoded.

Error Prefix String: E=

Normally if the SQL command produces an error, the returned error message will start with "Error: " but this option allows you to specify an additional prefix for the error string so your system can detect and handle the error in a more customized way. Remember this is a prefix, so you will still get the full text of the error message. For situations where having the database call fail is likely, and not an error (e.g. not all fields in the database are populated), use the O=Z option to return no data and no error. This will cause Corvid to instead ask the value of the variable from the user.

Spreadsheet Header Line: SS=

In some cases, it is desirable to return data from the database in the form of a tab-delimited spreadsheet. This is particularly useful for MetaBlocks that use real-time data from a spreadsheet. A MetaBlock requires a tab-delimited spreadsheet with the first line having only column header information. These column headers are referenced in the MetaBlock logic. A SELECT command can return a number data in tab-delimited form, arranged by field and one record per row. The SS= command provides a convenient way to add a header to this data. Using databases with MetaBlocks is described in section 16.6.10.

Variables to Receive Data: V=

Allows database calls to set the value for multiple Corvid variables. When a SELECT command returns the values from multiple fields, these can be assigned to multiple Corvid variables. This option should only be used with database commands to set the actual value of a variable, and not with other database calls such as those used to set prompt text, or commands not using SELECT.

If the SELECT command returns multiple values and this option is NOT used, the values will all be assigned to the same variable associated with the command. The only type of variable that is usually assigned values this way is a Collection variable.

The Corvid variables that will be assigned values are displayed in the list in the “List of variables to receive data” section. The order of the variables must be the same as the order of the fields specified in the SELECT command.

To add a variable to the list, select the Corvid variables from the lower drop down list. Click the “Add Below” button to add the new variable below the variable currently selected in the list. Click the “Add Above” button to add the new variable above the variable currently selected in the list. If there is nothing in the list, either button can be clicked to add the first item. To delete a variable in the list, select it and click “Delete”.

Make sure the variables in the list are in the same order as the data items that will be returned. Also make sure that the returned data is correct for the variable’s type.

In the command that will be built, the individual Corvid variables will be separated by a "-" and NOT include [ ].
For example,

SELECT Address, City, State FROM …. WHERE ...

will return 3 fields from the record that is found. To put the 3 values in 3 string variables [ADDR], [CITY] and [ST], add these 3 Corvid variables to the list. The configuration option added to the command will be:

V=ADDR-CITY-ST

Other Special Options: O=

Special options are a string of characters that cause different pre-programmed behavior.
For example, **"O=TQZ"** adds 3 behaviors. **Note:** there is no period between individual options.

The available options are:

Q Put quotes around the entire return data string.

T Instead of using line feeds to separate the row/record data use a tab.

N Instead of using a tab to separate the column/field data use a line feed.

Z If the database SELECT statement fails, return an empty string. This will force Corvid to ask the value of the variable

Configuration Option Line Identifier: R=

The Configuration options line can get fairly long, especially when many items of data are being returned to multiple variables. An alternative to having the line in the Corvid_DB command is to put it in the SQL Command File with an identifier string. Then just use R=identifier to add all the Configuration commands on that line.

This configuration options line can be added to the SQL Command File by:

1. Build a command in the Command Builder with all the required command options.
2. Select and copy the options from the command. Save the command.
3. Start a new command by clicking “Build New Command” on the Command tab.
4. Give the new “command” an identifier and paste the configuration options in as the text of the “command”. Click OK to save it.
5. Return to the command from step #2 and go to the Options tab.
6. Turn off all configuration options except “Configuration Option Identifier”, which should be set to the identifier for the option line added to the file in step #4.

16.6.8 Commands that Change the Database

SQL commands like SELECT return the selected value to Corvid. However, some SQL commands such as INSERT, UPDATE, and DELETE make changes to the database itself. These commands will return the string “Success” if the operation could be completed or an error message if it failed.

Usually, commands that change the database are used either in the Command Block or as an After Ask command. It is not required to catch and check the return string, but it is generally a good idea and makes debugging database interfaces much easier.

To check the return string, associate the SQL command with a string variable. This can be done by either:

- Having the command set the value for the string variable, and then using a DERIVE command to force the system to get the value
- Use the “V=” option on the command to put the return value in the string variable.

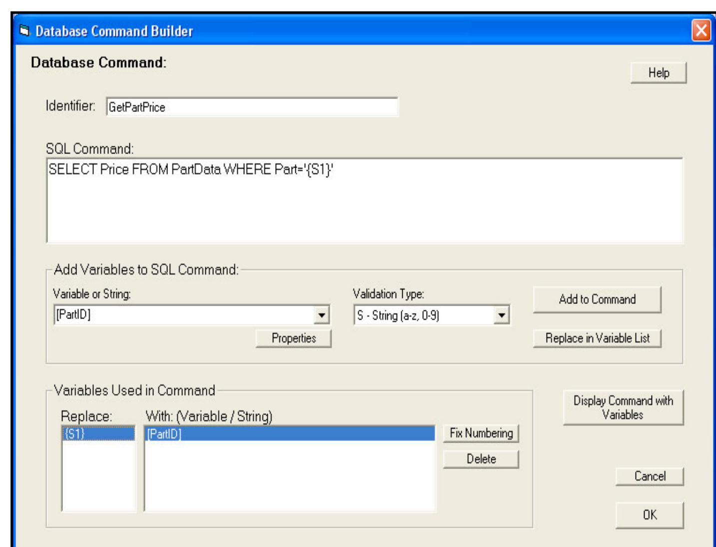
Either way, after the value is set, it can be checked in a Logic Block or with an IF in the Command Block. Unless the return string is “Success”, the SQL command failed. If it fails the string will be an error message, which may help in determining the cause.

16.6.9 Editing Commands

To edit an existing command, go to the External Interface window Database tab. Select the command and click “Edit”.

The same window will open that was used to build the command.

The text of the command can be directly edited in the edit box. Additional replaceable parameters can be added by positioning the cursor in the text, selecting a variable or sting and validation type, and clicking “Add to Command”



Exiting replaceable parameters can be changed by:

1. Select the parameter to change in the “Variables Used in Command” list
2. Select the new variable/string and validation type in “Add Variables to SQL Command”
3. Click “Replace in Variable List”

This will change the selected parameter to the new one.

After editing there can be unused variables left in the “Variables Used in Command” list. To delete unneeded ones, select them and click “Delete”.

Adding new parameters in the middle of the command can make the numbering of the replaceable parameters not be in numeric order. It is not required that the numbering of the replaceable parameters be in numeric order as long as the variables/strings in the CORVID_DB command match correctly. However, it is easier to read the command if it is in numeric order, and clicking the “Fix Numbering” button will make any changes needed to put it in numeric order.

16.6.10 Batch Command Option

A single database command can perform several SQL commands by starting a list of commands with the word BATCH, and separating the commands with a semicolon.

1. When building the SQL command, type the word "BATCH" in front of the list of SQL commands.
2. Type 2 or more SQL commands into the SQL command edit box. Separate each command with a semicolon "**;**" character. The last entry does not need to end with a semicolon ";", however a final semicolon will be ignored if present.

   (e.g: BATCH UPDATE ...; INSERT ...; UPDATE ...; INSERT ...; DELETE ...)

Replaceable parameters can be added anywhere in any of the individual commands.

The only SQL commands allowed in the BATCH list are INSERT, UPDATE, and DELETE. However, some databases and drivers also allow ALTER, CREATE, DROP and other similar commands. The SELECT command is not allowed in a BATCH.

For example, suppose an expert system needs to add two rows of data to the table. For the SQL command enter a command structured like:

BATCH INSERT INTO table_name VALUES (123 , 'Bill Smith' , '10 Main Street');
INSERT INTO table_name VALUES (456 , 'Tom Green' , '86 Central Avenue')

However the actual names and values would normally be replaceable parameters from Corvid variables.

Be sure to assign the returned value to a String variable, such as [database_status]. If all commands were successful, then [database_status] will be "Success". If any error occurred, [database_status] will have a single error message.

If an error occurs in one of the SQL commands, the remaining SQL commands may or may not be performed. The behavior depends on your driver. Different driver manufacturers handle errors in different ways.

16.6.11 Using a Database Command for a MetaBlock Data “File”

MetaBlock systems analyze data in a tab delimited data file. Usually, this is a static file of data, but that can be replaced by a CORVID_DB command that returns data in the form of a tab delimited file. Since the Corvid Runtime reads the entire MetaBlock data file and stores it internally, only one database call is required to get all the data.

To build a MetaBlock system that gets its data from a database:

1. Create a static file of data that has the correct structure and column labels that the system will need. This can be created in a text editor such as Notepad, or built in a spreadsheet program like Excel and saved as a tab-delimited file. Add a few rows of test data to use in building the MetaBlock rules. Save this Sample File.

2. Create a new Logic Block and click the “MetaBlock” check box. Enter the name of the Sample File as the source for the MetaBlock data. Build the rules as described in the MetaBlock section of the manual. Test the system with the sample data in the file to make sure it runs correctly.

3. When everything is working correctly, click on the “MetaBlock” check box to open the MetaBlock window again. This time, select "Database Command" as the source. This will open the window for adding database commands.

4. The command to get the data in the form needed by the MetaBlock is a standard SQL command, but adding the header line requires using one of the command Options. This is done by creating a special line in the database command file with the header information and then adding that to a separate SQL command as on Option.

5. Add a MetaBlock header line to the Database Command File. To do this:
   - Click "Build New command".
   - Enter an identifier (e.g. MetablockHeader).
   - In "SQL Command" enter the tab delimited header line. Copy and paste this from the top line of the static Sample File. It can also be entered directly. (If entered directly, tab moves to the next edit field and Ctrl-tab must be used to enter a tab in the edit box).
   - Make note of the Identifier that was used and click OK to save the header in the DB command file.

6. Back in the database Command tab, "Build New command" Add a command to select the needed columns of data from the DB. These must EXACTLY match the columns in the static sample data header. If the header line defines 3 columns, the SQL command must return 3 items of data for each row. This can be done by a SELECT command listing the field names of the data needed. (e.g. SELECT Part, Price, Material FROM Table1). The SQL command can use WHERE.. with replaceable parameters to limit the data returned if needed. Click OK to add the new command to the command list.

7. With the command just added selected.
   - Click the Options tab. Click "Spreadsheet header line identifier" and enter the header identifier. This is the identifier name from step #5.
   - Still on the Options tab, also click "Use tabs to separate items"
   - Click OK to save the command

The source for the MetaBlock data will be "database" and the edit field will have the CORVID_DB... command.

When the command is executed, the Corvid Runtime will call the SQL command, which will return the specified fields of data for each applicable record in the database. Corvid will automatically convert the data returned into the form needed by the MetaBlock. The header row will be added and the values will be chopped into rows that match the column headers. The number of header columns defines how the data will be broken into rows, so it is very important that the “Select” identifiers match the column headers.

16.6.12 Installing the CorvidDB Servlet

If your system is run with the Corvid **Applet** Runtime and uses database calls, it requires the installing the CorvidDB servlet provided with Exsys Corvid. If your system will be run with the Corvid Servlet Runtime, the Corvid DB servlet is NOT needed and should not be installed.

1. Your server must support running Java Servlets. This requires a “Servlet Container” such as Apache Tomcat, Glass fish, IBM Webshere or similar program. Check with your server administrator if you are unsure if this support is provided.

2. The CorvidDB.war file is distributed with Exsys Corvid and can be found in the “Program Files/Exsys/Corvid/ServerPrograms” folder.

3. Put the 'CorvidDB.war' file in the appropriate folder for your servlet engine and activate it.
   For example, in Tomcat you usually put the .war file in the 'webapps' folder before you start or restart Tomcat. (Check with your system administrator for details on installing servlets on your system.)

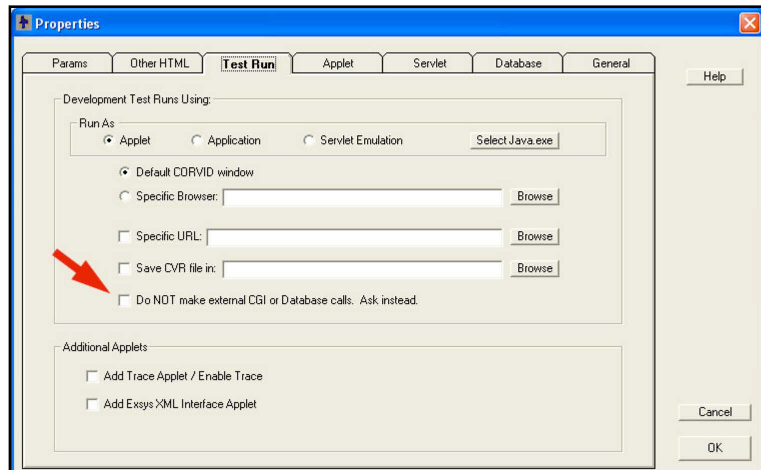
4. Be sure the database is installed on the server and a DSN is created on the server as a "System DSN". If the expert system is to write to the database, make sure the DSN does not have 'read only' selected.
5. Put the '.cdb' file in the 'CorvidDB' folder created when the CorvidDB.war file deploys. The .cdb file was created in the folder you specified when you built the database command.
6. Test from your browser. Go to the URL **http://your_host:8080/CorvidDB/corviddbservlet**. (Where "your_host:8080" is the address to run the installed servlet. Check with your system administrator for the naming conventions on your server.) This will display a form page where you can test your database commands.
7. In your system, make sure the URL to CorvidDB is specified when building CORVID_DB commands.

16.6.13 Turning Off Database Calls During Development

Most external interface sources are easy to implement in the development environment since they apply to both applet and servlet runtimes, or only require having a file in the system folder. However, database interfaces (especially ones using the Corvid Servlet Runtime) can require changing the commands in a system during development and then restoring them when the system is fielded. Often it is easier to simply disable the database commands in the development environment.

If the system only uses the database to set the values of variables, this will just result in these values being asked of the end user. To disable the database commands during development and testing:

- Go to the Properties window
- Select the “Test Run” tab
- Check the “Do NOT make external CGI or Database calls. Ask instead”



When ready to deploy the system to an environment that provides the database support, uncheck this box and be sure to test in the full database environment.

Some systems that have complex database interactions (such as Blackboards), will not function without full database support and can not be run with this option.

16.7 PARAM Command Details

16.7.1 PARAM Data

When using the Corvid Applet Runtime, an applet tag is automatically added to the HTML page used to run the system. This tag looks similar to:

```
<APPLET
CODEBASE = "./"
CODE = "Corvid.Runtime.class"
NAME = "CorvidRuntime"
ARCHIVE = "ExsysCorvid.jar"
WIDTH = 700
HEIGHT = 400
HSPACE = 0
VSPACE = 0
ALIGN = middle
>
<PARAM NAME = "KBBASE" VALUE = "" >
<PARAM NAME = "KBNAME" VALUE = "MySystem.cvR">
<PARAM NAME = "KBWIDTH" VALUE = "700">
</APPLET>
```

Within the tag are “PARAM” name/value pairs that are used to pass the name of the Corvid system and applet window width into the runtime. PARAMs are a standard way to set values that can be accessed from within the applet.

PARAMs have a name and a value. Both the name and value are in quotes.

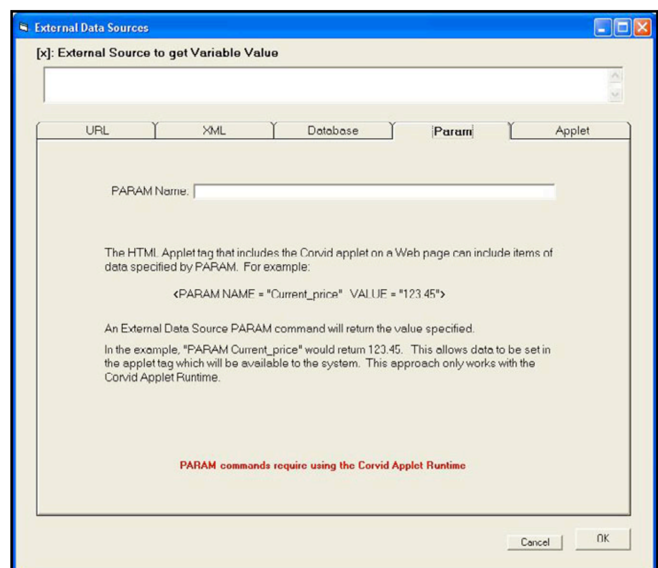
In addition to the required PARAM name/value pairs that Corvid automatically adds, other PARAM name/value pairs can be added to the applet tag. The value of any PARAM is available in the system by referencing the name in a Corvid PARAM command.

Additional PARAM name/value pairs should be added after the ones Corvid automatically adds. For example, adding:

```
<PARAM NAME = "TODAYS_SPECIAL" VALUE = "X123">
```

to the PARAM list would make this data available in the system.

PARAM commands can be used in the same places any of the other external data source commands can be used. To add a PARAM command, go to the External Data Source window and select the “Param” tab. Enter the PARAM Name - this must match exactly in both spelling and case.



The PARAM command will set the associated value.

PARAM commands can only be used with the Applet Runtime and have no meaning with the Corvid Servlet Runtime. They provide an easy way to modify data in a system just by changing the text of the HTML page used to run the system. There can be any number of PARAM name/value pairs. If the HTML page is dynamically generated, such as with Cold Fusion, these PARAM values can be set when the page is created.

If PARAM data is used, the HTML page used to run the system must be custom modified. This can be done by running the system once, and then editing the KBName.HTML page Corvid generates. Edit this to add the custom PARAM data and save it as a different name. Open the “Properties” window and on the “Test Run” tab, enter the new HTML page as the “Specific URL”. Corvid will then use that page when test running rather than the page it generates which would not have the custom PARAM data.

16.8 APPLET Command Details

16.8.1 Custom Java Applets

The APPLET command is used to communicate between the Corvid Applet Runtime and other Java Applets on the same HTML page. The Corvid Runtime Applet can send data to other applets, have those applets perform some action and send data back to the Corvid Runtime. This allows the other applets to add to the capabilities of the Corvid Runtime by providing other special interfaces to data sources, or even to provide custom interactive graphics or user interfaces.

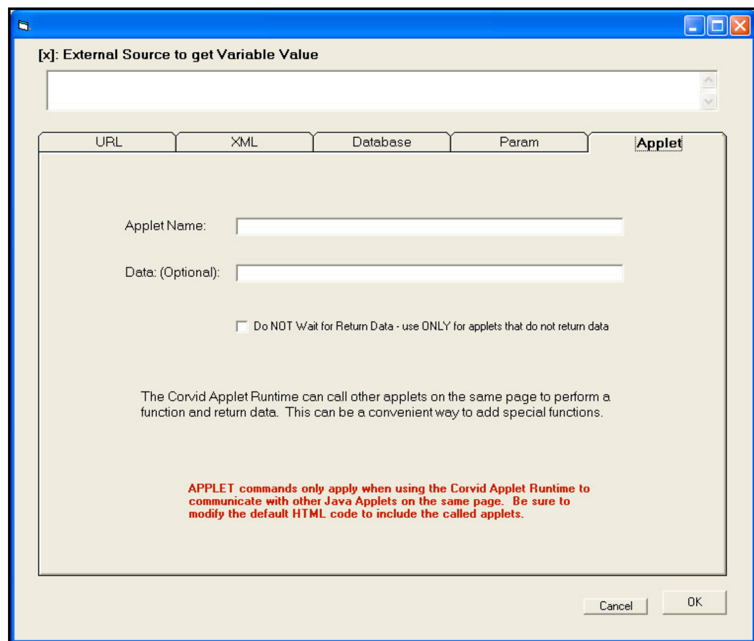
Corvid uses inter-applet communication for the XML interface in the Corvid Applet Runtime and the Corvid Trace applet. The same technique used in these can be adapted to add functionality with other custom applets.

16.8.2 APPLET Window

APPLET commands are built using the “Applet” tab in the external command builder window.

Applet Name: This is the name of the applet to call. This is set in the <APPLET> tag that includes the custom applet in the HTML page with the Corvid Runtime Applet. It will be something like:

```
<APPLET
CODEBASE = "./"
CODE = "MyApplet.MyClass.class"
NAME = "CustomApplet"
ARCHIVE = "MyApplet.jar"
WIDTH = 0
HEIGHT = 0
HSPACE = 0
VSPACE = 0
>
</APPLET>
```



The “Name” for the applet in the Applet tag can be any string. Use this name in the Corvid “Applet Name” field to call the applet. The case of the characters in the name must match exactly.

Data: This is an optional field, but is generally used. This is the data that will be passed to the applet. This data will be available in the called applet to tell it what to do. The data is passed as a string and can be any text data including spaces. Corvid variables in double square brackets can be used to pass Corvid system data to the called applet. If multiple values are to be passed, pass them in a way that will allow the called applet to parse out the various items.

For example, an applet to do a table lookup would need to be passed the name of the table and the value to look up. The name of the table might be static and the value to look up might be in the Corvid variable [X]. The data could be:

“MyDataFile.txt” [[X.VALUE]]

The quotes around the file name make it easy for the called applet to parse that out of the string (even if the filename contained spaces), and the value of the variable [X] would be embedded in the string.

There is no limit to the length of the string or number of variables passed in it.

Do Not Wait Option: Most called applets will perform some function and return data. In that case, be sure this option is NOT checked. However, applets can also be used to just display data or store it some custom way. Corvid does this with its built-in Trace applet which is sent trace data as Corvid runs, but which does not send anything back.

Normally, when Corvid calls an external applet, it waits for that applet to send data back. If this will not happen, be sure to check the “Do Not Wait” check box. In that case, Corvid will pass the data to the applet and continue running.

For example, a custom graphing applet could be called to display information from a Corvid system repeatedly monitoring a process. Each time the Corvid system would finish a check, it would write data to the graphing applet, which would display the data, but not send anything back to Corvid.

16.8.3 How to Create a Custom Applet

Building a custom applet requires a knowledge of Java and a development environment such as the free Oracle Sun Netbeans tool. This manual is not meant to explain the Java language or how to build applets, and only covers the special techniques used by Corvid to communicate with the applet to send and receive data.

Inter-applet communication can be done in various ways. Advanced techniques like pipes can be used when applets need to be in constant communication running in parallel, but Corvid applications call an applet, and must then wait for data before they can continue processing. This results in a more sequential process that does not require the complexity of setting up pipes. Corvid uses a VERY simple and reliable technique that can be added to an applet in just a few lines of code, and uses standard applet methods that are accepted by all compilers.

Corvid passes data to the called applet using the applet.getName() and applet.setName() methods. This may at first seem like an odd way to pass data, but it works extremely well and is very easy to implement.

In the custom applet:

- Create a boolean variable *running* that will indicate if the applet has finished its initialization.
- Create a string variable *origName* to hold the original name of the applet. This is set in the first time init() is called.

```
public class GetMyData extends Applet {
    boolean running = false;    // has the system done the first init?
    String origName;            // Original system name
```

The init() method for the applet gets called when the applet first starts on the page and again when it is called by Corvid to do something. To differentiate these calls, the *running* variable is used.

```
public void init() {
    if (running == false) {
        origName = this.getName();
        running = true;
        // any other code needed for init
    }
    else {
        processDataRequest(this.getName());
        this.setName(origName);
    }
```

When the applet first starts, *running* will be false. The original name of the applet is saved so it can be reset later, *running* is set to true and any other initialization code is executed. When Corvid calls the applet, init() will be called again (by Corvid) and this time since running is true, the processDataRequest() will be executed to do whatever the applet does to get and return data to Corvid, and the name is reset to the original name. This last step is important since without it Corvid will not be able to identify the applet for subsequent calls.

The processDataRequest() uses the getName() method to get the data sent from Corvid. Corvid will have already set this to the "data" string specified by the APPLET command in the Corvid system. When called by Corvid, the "name" of the applet will actually be the data sent from Corvid. This very simple technique allows large amounts of data to be sent in a very easy and reliable manner.

The processDataRequest() code is passed the data string from Corvid. This will be whatever string was specified in the Data field when the APPLET command was built. Any embedded Corvid variables will have been replaced by their values, and processDataRequest() must do whatever parsing is needed to separate out values. The data passed can then be used to do whatever the applet is designed to do, including obtaining data. Data may be obtained automatically from other sources invisibly, or an end user interface may be displayed for user interaction.

Once the applet has determined the data to send back to Corvid, it should use the returnDataToCorvid() method. This will pass the data back and tell Corvid to continue processing.

```
public void returnDataToCorvid(String s) {   // return data to Corvid
    Applet corvidRuntime;
    corvidRuntime = getAppletContext().getApplet("CorvidRuntime");
    corvidRuntime.setName(s);
    corvidRuntime.start();
}
```

This uses essentially the same technique to send data back as was used to pass data in. The corvidRuntime.setName() sets the data so that Corvid can recover it using a getName() command. The corvidRuntime.start() methods has a boolean flag that indicates it is waiting for an applet to return data, and calling it here tells Corvid to read the data and continue processing. Very large amount of data can be passed back if needed.

16.8.4 Returned Data

The syntax for the returned data is similar to all other external data sources. If the context of the external call indicates only a single string is expected (such as setting the Prompt text for a variable), the returned string will be used in that way.

If the external call is to set the value for a variable, the returned data must set the value for that variable but can also set the value for other variables at the same time. If the returned data is to set the value for the calling variable, it only needs to be the value to assign. For example, if the external call to the applet was to set the value for the numeric variable [X], the returned data could just be:

123

and [X] would be set to that value.

Note, the type of the variable determines what data is acceptable. If the string "abc" was returned for a numeric variable, it would produce an error. Static List variables can be set by either the number of the value, or the short text of the value. So a Static List variable [color] with value "red", "blue" and "green" could have return data or either "1" or "red" set the first value. For Static List variables, to set multiple values separate them with a comma - so "1,2" would set the values red and blue.

To return data for multiple variables, the form:

[varname] value

must be used, and the individual variable/value pairs must be separated by a tab. The return string:

[x] 123 *tab* [s] abc *tab* [color] red

would set [X] to 123, [s] to "abc" and [color] to red. In the Java code this would be:

returnDataToCorvid("[x] 123\t[s] abc\t[color] red")

16.8.5 The External Applet Tag

The applet being called must be added to the HTML page used to run the system. It must be on the same HTML page as the Corvid Runtime Applet.

The applet tag will be something like:

```
<APPLET
CODEBASE = "./"
CODE = "MyApplet.MyClass.class"
NAME = "CustomApplet"
ARCHIVE = "MyApplet.jar"
WIDTH = 0
HEIGHT = 0
HSPACE = 0
VSPACE = 0
>
</APPLET>
```

The name is the name used to call the applet from Corvid. The "CODE=" is the main class in the applet. The Archive is the .jar file for the applet.

The width and height values depend on if the applet will be used to display information, and how the applet should be displayed on the page. If the applet has no direct user interaction, these values can be "0" and the applet will be invisible, but will still run. If the applet displays data, or has some user interaction, set these values appropriately to whatever size is required. The applet can be anywhere on the page.

The HTML page that Corvid generates must be modified in a text editor or HTML editor to add the additional applet tag. Once a custom page has been created, you can have the system automatically run with this custom HTML page during development by:

- Open the “Properties” window
- Go to the “Test Run” tab
- In the “Specific URL” edit box, enter the name of the custom HTML page or browse to it

When the system is run, Corvid will use the custom HTML page that has the other applet included.

16.8.6 Uses for External Applets

There are many uses for external applets to enhance the capabilities of Corvid. The most common use is to have an applet perform some special function or interface not built in to the Corvid Runtime. Corvid uses this approach itself for the XML interface when using the Corvid Applet Runtime. The XML interface is built into the Corvid Servlet Runtime, but since it significantly increases the size and Java requirements of the Corvid Applet Runtime, and it is not needed in most systems, it makes more sense to have it as a separate applet that can be included and called as needed.

External applets can also be used to do complex mathematical operations not easily handled within Corvid. For example, if a Fourier transform was needed, it would be better to pass the data to an external applet that could implement the transform algorithm in Java rather than trying to do it in Corvid.

External applets can also be used to display data or even interactively ask questions of the end user. Data can be passed to an applet that could graph it or display it in ways not possible from within the Corvid Applet Runtime.

It is even possible to have the actual Corvid Applet Runtime invisible and to do all the user interaction through external applets. This is complicated and requires a good knowledge of Java, but can be done to create special interfaces not otherwise possible with the Corvid Applet Runtime.

Exsys, Inc.
6301 Indian School Rd. NE, Suite 700
Albuquerque, NM 87110 U.S.A.
Tel: (505) 888-9494, Fax: (505) 888-9509

info@exsys.com
www.exsys.com

For support questions contact: **support@exsys.com**

© 2010 Exsys, Inc. All rights reserved. Product names and trademarks may be trademarks and/or registered trademarks of their respective companies.