Net.Data

Administration and Programming Guide for OS/2, Windows NT, and UNIX

Version 6 Release 1

Net.Data

Administration and Programming Guide for OS/2, Windows NT, and UNIX

Version 6 Release 1

Note

Be sure to read the information in “Appendix G. Notices” on page 211 before using this information and the product it supports.

Sixth Edition (May 1999)

Contents

Preface

Thank you for selecting Net.Data® Version 6.1, the IBM™ development tool for creating dynamic Web pages! With Net.Data, you can rapidly develop Web pages with dynamic content by incorporating data from a variety of data sources and by using the power of programming languages you already know.

About Net.Data

With IBM's Net.Data product, you can create dynamic Web pages using data from both relational and non-relational database management systems (DBMSs), including DB2, IMS, and ODBC-enabled databases, and using applications written in programming languages such as Java, JavaScript, Perl, C, C++, and REXX.

Net.Data is a macro processor that executes as middleware on a Web server machine. You can write Net.Data application programs, called *macros*, that Net.Data interprets to create dynamic Web pages with customized content based on input from the user, the current state of your databases, other data sources, existing business logic, and other factors that you design into your macro.

A request, in the form of a URL (uniform resource locator), flows from a browser, such as Netscape Navigator or Internet Explorer, to a Web server that forwards the request to Net.Data for execution. Net.Data locates and executes the macro and builds a Web page that it customizes based on functions that you write. These functions can:

- Encapsulate business logic within Perl scripts, C and C++ applications, or REXX programs.
- Access databases such as DB2
- Access other data sources such as flat files.

Net.Data passes this Web page to the Web server, which in turn forwards the page over the network for display at the browser.

Net.Data can be used in server environments that are configured to use interfaces such as HyperText Transfer Protocol (HTTP) and Common Gateway Interface (CGI). HTTP is an industry-standard interface for interaction between a browser and Web server, and CGI is an industry-standard interface for Web server invocation of gateway applications like Net.Data. These interfaces allow you to select your favorite browser or Web server for use with Net.Data. Net.Data also supports a variety of Web server Application Programming Interfaces (APIs) for improved performance. The Net.Data family of products provide similar capablities on the OS/400, OS/390, Windows NT, AIX, OS/2, HP-UX, Sun Solaris, Linux, and Santa Cruz Operating System (SCO) operating systems. Net.Data also supports FastCGI and the major Web server Application Programming Interfaces (APIs) on multiple operating systems.

A graphical administration tool helps you administer Net.Data configuration settings for the AIX, Windows NT, and OS/2 operating systems. The administration tool also assists you in specifying security for your connections to databases that use Live Connection.

To help you easily access data from your database, Net.Data provides a variety of tools, including NetObjects Fusion plug-ins and smartguides for Java-based development. These tools work with the Net.Data Java servlets in the Java

environment, allowing you to create applications that are portable across operating systems. NetObjects Fusion plug-ins allow you to use the NetObjects Fusion Web development tool to build sophisticated applications with dynamic data from relational data sources. Net.Data smartguides provide a graphical tool to guide you through creating basic Net.Data macros.

What's New?

The following sections describe the new enhancements for Net.Data.

What's New in Version 2.0.7 Fixpack?

Net.Data Version 2.0.7 provides the following enchancements:

- Ability to use DB2 File Manager and the DATALINK data type
- Support for new Net.Data table processing functions
- Configuration variables for security: DTW_DIRECT_REQUEST and DTW_SHOWSQL and new guidelines for improving the security of your applications

What's New in Version 6 Release 1?

Net.Data for OS/2, Windows NT, and UNIX provides the following features in Version 6 Release 1:

- Performance enhancements include:
  - Support for FastCGI on the Solaris operating system
- Language environment enhancements include:
  - Support for the Datalink data type in the SQL language environment on Windows NT
  - Support for the IMS Web language environment on the Solaris operating system
  - Support for large objects (LOBS) in the ODBC language environment
  - Support for new LOB types: Musical Instruments Digital Interface audio file (.mid), Audio Interchange File Format audio file (.aif), Basic Audio files (.au), Real Audio files (.ra), Portable Document Format files (.pdf), and Windows Audio Video files (.wav)
  - Simplified maintenance of the temporary large objects directory (`tmplobs`) with the `cleanup` daemon
- Macro language enhancements include:
  - Support for milliseconds in the DTW_TIME function.
  - Support for REPORT blocks in MACRO_FUNCTION blocks
  - Support for multiple REPORT blocks in FUNCTION and MACRO_FUNCTION blocks
  - Support for dynamically building variable references
  - Ability to set the VALUE attribute in the OPTION element of DTW_SELECT()
  - Support for the hash (#) character in variable names.
- Ability to log Live Connection errors and activities
- Support Net.Data on the Linux operating system
- Support for Unicode characters in macro files and DB2 databases with the DTW_UNICODE configuration variable

About This Book

This book discusses administration and programming concepts for Net.Data, as well as how to configure Net.Data and its components, plan for security, and improve performance.

Building on your knowledge of programming languages and database, you learn how to use the Net.Data macro language or Java servlets to develop macros. You learn how to use Net.Data-provided language environments that access DB2 databases, and IMS transactions using IMS Web, as well as use Java, REXX, Perl, and other programming languages to access your data.

This book may refer to products or features that are announced, but not yet available.

More information including sample Net.Data macros, demos, and the latest copy of this book, is available from the following World Wide Web site:

http://www.software.ibm.com/data/net.data

Who Should Read This Book

This book is intended for people involved in planning and writing Net.Data applications. To understand the concepts discussed in this book, you should be familiar with how a Web server works, understand simple SQL statements, and know HTML tags, including HTML form tags.

The Net.Data macro language, variables, and built-in functions, as well as operating system differences are described in *Net.Data Reference*.

About Examples in This Book

Examples used in this book are kept simple to illustrate specific concepts and do not show every way Net.Data constructs can be used. Some examples are fragments that require additional code to work.

Chapter 1. Introduction

Most Web pages on the Internet are static Web pages; in other words, pages that do not change unless you edit them. To put “live” data and applications on the Web (such as current sales statistics), Web site developers usually write programs that execute as middleware at the Web server to dynamically build Web pages. Writing these types of programs is not easy.

Net.Data simplifies the writing of interactive Web applications through *macros*.

This chapter describes Net.Data and the reasons why you might want to use it for your Web applications.

What is Net.Data?

Using Net.Data macros, you can execute programming logic, access and manipulate variables, call functions, and use report-generating tools. A macro is a text file containing Net.Data macro language constructs, HTML tags, Javascript, and language environment statements, such as SQL and Perl. Net.Data processes the macro to produce output that can be displayed by a Web browser. Macros combine the simplicity of HTML with the dynamic functionality of Web server programs, making it easy to add live data to static Web pages. The live data can be extracted from local or remote databases and from flat files, or be generated by applications and system services.

Figure 1 on page 2 illustrates the relationship between Net.Data, the Web server, and supported data and programming language environments.

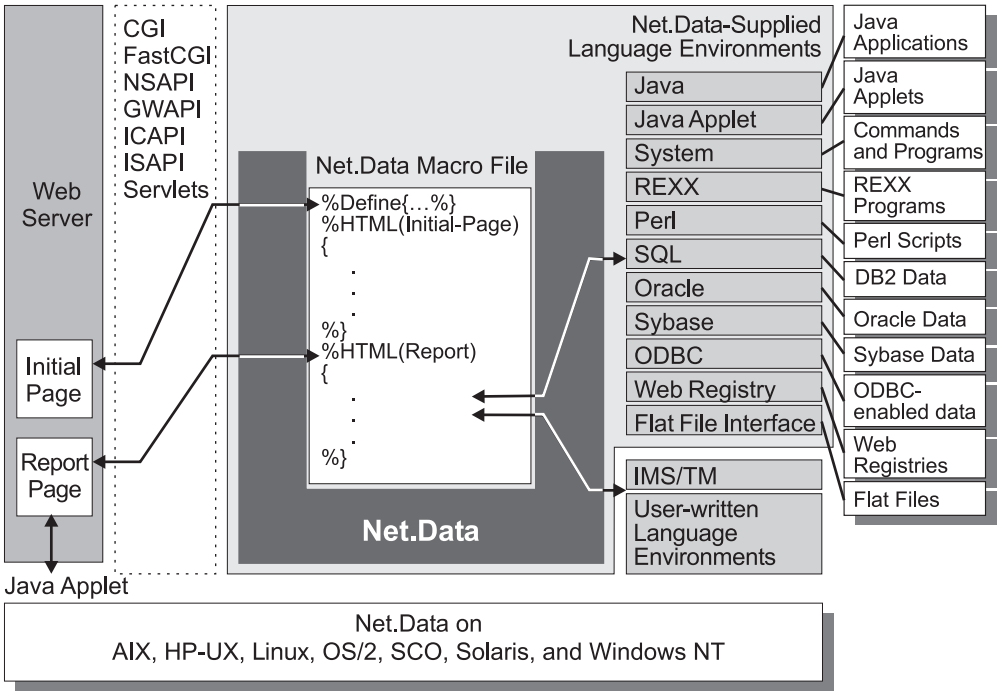


Figure 1. The Relationship between Net.Data, the Web Server, and Supported Data and Program Sources

The Web server invokes Net.Data as a CGI, FastCGI, or Web server application programming interface (API) by calling Net.Data as a DLL or shared library when it receives a URL that requests Net.Data services. The URL includes Net.Data-specific information, including either the macro that is to be processed or the SQL statement or program that is to be directly invoked. When Net.Data finishes processing the request, it sends the resulting Web page to the Web server. The server passes it on to the Web client, where it is displayed by the browser.

Why Use Net.Data?

Net.Data is a good choice for creating dynamic Web pages because using the macro language is simpler than writing your own Web server applications and because Net.Data lets you use languages that you already know, such as HTML, SQL, Perl, REXX, and JavaScript. Net.Data also provides language environments that access DB2 databases, execute IMS transactions using IMS Web, or use REXX, Perl, and other languages for your applications. In addition, changes to a macro can be seen instantaneously on a browser.

Net.Data complements data management capabilities that already exist on your operating system by enabling both data and related business logic for the Web. More specifically, Net.Data:

- Provides a simple, yet powerful macro language that allows for rapid development of Internet and Intranet applications. The Net.Data Web application environment provides the following features:
- Permits the separation of data generation logic from presentation logic within your Web applications. Net.Data does not impose any restrictions on the method

with which the data is presented (such as HTML or Javascript). This separation allows users to easily change the presentation of data using the latest presentation techniques.

- Allows you to use existing skills and business logic to generate Web pages by providing the ability to interface with programs written in C, C++, REXX, Java or other languages.
- Provides the ability to develop complex Internet applications quickly, using a simple macro language.
- Provides high-performance access to data that is stored in DB2 and in any remote DRDA-enabled database.
- Provides easy migration of macros between all operating systems supported by the Net.Data family of products.

Interpreted Macro Language

The Net.Data macro language is an interpreted language. When Net.Data is invoked to process a macro, Net.Data directly interprets each language statement in a sequential fashion, starting from the top of the file. Using this approach, any changes you make to a macro can be immediately seen when you next specify the URL that executes the macro. No recompilation is required.

Direct Requests

Simple requests that require the execution of a single SQL statement, DB2 stored procedure, REXX program, C or C++ program, or Perl script do not require the creation of a macro. These requests can be specified directly within the URL that flows from the browser to the Web server.

Free Format

The Net.Data macro language has only a few rules about programming format. This simplicity provides programmers with freedom and flexibility. A single instruction can span many lines, or multiple instructions can be entered on a single line. Instructions can begin in any column. Spaces or entire lines can be skipped. Comments can be used anywhere.

Variables Without Type

Net.Data regards all data as character strings. Net.Data uses built-in functions to perform arithmetic operations on a string that represents a valid number, including those in exponential formats. Macro language variables are discussed in detail in “Net.Data Macro Variables” on page 86.

Built-in Functions

Net.Data supplies built-in functions that perform various processing, searching, and comparison operations for both text and numbers. Other built-in functions provide formatting capabilities and arithmetic calculations.

Error Handling

When Net.Data detects an error, messages with explanations are returned to the client. You can customize the error messages before they are returned to a user at a browser. See *Net.Data Reference* for more information.

Chapter 2. Configuring Net.Data

You can install Net.Data for your operating system by using the instructions in the README file that accompanied the product. Most configuration steps are completed during installation; this varies by operating system.

After installing Net.Data for your operating system, you must configure Net.Data and modify your configuration for the Web server. Configuration tasks include:

- Customizing the Net.Data initialization (INI) file
- Configuring Net.Data for use with FastCGI or one of the support Web server APIs (optional)
- Customizing the Web server configuration and environment variable files
- Configuring the Cache Manager (optional)
- Configuring Live Connection (optional)
- Setting up the Net.Data language environments
- Specifying access rights

You use the following tools to configure Net.Data:

- A text editor

  Use a text editor to edit the initialization file and the Live Connection configuration file on all operating systems. You also use a text editor to update any Web server configuration files. It is a good idea to back up the files before you make changes.
- The Net.Data administration tool

  The administration tool provides a graphical interface for customizing the initialization file and the Live Connection configuration file. You can use the administration tool to configure Net.Data on the OS/2, Windows NT, and AIX operating systems.

The method you use depends on which components need to be configured and the operating system Net.Data is running on, as described in Table 1. If you start using one particular method for a configuration task, you should continue to use that method for the best results.

Table 1. Comparison of configuration methods with tasks and operating systems. **A** *- Can be configured with the administration tool or manually.* **M** *- Can be configured manually, only.*

Task	**Operating Systems:**			
	AIX	**NT**	**OS/2**	**HP|SUN|SCO**
Configure the Net.Data INI file	A	A	A	M
Define cliettes ports	A	A	A	M
Define cliettes	A	A	A	M
Turn on cliette password encryption	A	N/A	N/A	N/A
Turn on error logging	A	A	A	M
Configure Web Server for FastCGI, CGI, and APIs*	M	M	M	M
Define Cache Manager Ports	M	M	N/A	N/A
Configure Cache Manager	M	M	N/A	N/A

Table 1. Comparison of configuration methods with tasks and operating systems. **A** *- Can be configured with the administration tool or manually.* **M** *- Can be configured manually, only. (continued)*

Task	Operating Systems:			
	AIX	NT	OS/2	HP|SUN|SCO

***Tip:** Many Web servers have administration tools that you can use to configure the Web server.

This chapter describes how to configure Net.Data and how to modify your configuration of the Web server for use with Net.Data. Additionally, it describes how to configure optional components.

About the Net.Data Initialization File

Net.Data uses its initialization file to establish the settings of various configuration variables and to configure language environments and search paths. The settings of configuration variables control various aspects of Net.Data operation, such as the following:

- The encoding of character data as Unicode
- Whether string and word functions are DBCS enabled
- The name of the DB2 instance for access to database data
- How Net.Data connects and communicates with the language environments, databases, connection management, and caching
- Whether error logging is activated

The language environment statements define the Net.Data language environments that are available and identify special input and output parameter values that flow to and from the language environments. The language environments enable Net.Data to access different data sources, such as DB2 databases and system services. The path statements specify the directory paths to files that Net.Data uses, such as macros, REXX programs, and Perl scripts.

The Net.Data initialization file, db2www.ini, is located in the Web server's document directory. See the README file for you operating system for more information.

Authorization Tip: Ensure that the Web server has access rights to this file. See "Granting Access Rights to Files Accessed by Net.Data" on page 51 for more information.

About the Net.Data Configuration Files for Optional Components

The following sections discuss the configuration files for optional components of Net.Data.

The Live Connection Configuration File

Live Connection provides connection management on Windows NT, OS/2, AIX, and Sun Solaris operating systems to improve performance by eliminating start-up overhead. The Net.Data Live Connection configuration file contains information about one or more named cliettes. A cliette is a long-running process that maintains a connection to a database or a Java application that endures over Net.Data macro invocations from multiple users. After a cliette is started, it continues to exist until Net.Data Live Connection terminates. Multiple cliettes can connect to a single database.

As part of the cliette information in the configuration file, you specify a cliette name, unique ports, and the minimum and maximum number of processes. For database cliettes, you can also specify the database name, login, and password for each cliette entry. On AIX, you can encrypt the password.

Authorization Tip: Ensure that the user ID that starts Connection Manager has access rights to this file. See "Granting Access Rights to Files Accessed by Net.Data" on page 51 for more information.

The Cache Manager Configuration File

The Cache Manager configuration file contains the definitions for the Cache Manager and each of the caches. Net.Data caching is described in "Net.Data Caching" on page 161. Configuring the Cache Manager is described in "Configuring the Cache Manager and Net.Data Caches" on page 166. The structure of the file is a series of sections, or stanzas:

Cache Manager stanza
: This stanza defines the parameters of the Cache Manager itself and includes network information, logging status, and tracing status. The stanza is required and must be labeled `cache-manager`.

Cache definition stanzas
: These stanzas define the parameters for each cache; one cache definition stanza in the configuration file exists for each cache that is managed by the Cache Manager; this section contains network information, memory and space requirements, logging status, and statistics status. The cache definition stanza is required for each cache that is managed by the Cache Manager.

The Cache Manager configuration file is not managed by the administration tool and can be updated with any text editor. See "Net.Data Caching" on page 161 to learn how to define this file.

Authorization Tip: Ensure that the user ID that starts the Cache Manager has access rights to this file. See “Granting Access Rights to Files Accessed by Net.Data” on page 51 for more information.

Common Sections of the Net.Data Initialization, Control, and Macro Files

Certain portions of the Net.Data initialization, configuration and macro files must be consistent for all components of Net.Data to work as a whole. The following table summarizes the areas of each of these files that must match.

Table 2. Consistency Requirements for the Net.Data Configuration Files and the Macro

File	Common Sections	Notes
Net.Data INI File	Environment Statement	The language environments that use Live Connection must specify the database cliette name in their environment statement
	Live Connection configuration variables	When using Net.Data Live Connection, specify the Live Connection port, DTW_CM_PORT. This variable value must match the MAIN_PORT value in the Live Connection configuration file.
	Cache configuration variables	When using Net.Data caching, optionally include port number and machine name variables. These values must match those used in the Cache Manager configuration file, if used.
Live Connection Configuration File	Cliette Definitions	Each cliette definition must match a corresponding definition in the INI file. Additionally, the MAIN_PORT value must match the DTW_CM_PORT variable value in the INI file.
Cache Manager Configuration File	Cache Manager Configuration Variables	When using Net.Data caching, you can optionally include port number and machine name variables. These values must match those used in the INI file, if used.

The following fragments illustrate the relationship between a macro, a Net.Data initialization file, and a Live Connection configuration file. Two cliettes are used by the macro (`DTW_SQL:SAMPLE, DTW_SQL:CELDIAL`) and access two DB2 databases, called `SAMPLE` and `CELDIAL`. The Live Connection configuration file contains the cliette names and definitions. The ENVIRONMENT statement in the Net.Data initialization file refers to the cliette name. The LOGIN and PASSWORD values are specified in the Live Connection configuration file.

Figure 2 on page 9 shows a fragment of the macro that contains the @DTW_ASSIGN statement that defines which cliette is to be used to access a database.

```
<3*********************************************>
<3** This is an HTML comment                    **>
<3** Access the SAMPLE database using       **>
<3** cliette DTW_SQL:SAMPLE                  **>
<3*********************************************>
@DTW_ASSIGN (DATABASE, ″ SAMPLE ″)
@insert_customer
(customer_name, customer_street, customer_city, customer_state,
customer_country, customer_zip, customer_credit, customer_expiry)


<3*********************************************>
<3** This is an HTML comment                    **>
<3** Process the CELDIAL database using       **>
<3** the cliette DTW_SQL:CELDIAL **>
<3*********************************************>
@DTW_ASSIGN (DATABASE, ″ CELDIAL ″)
@insert_customer
(customer_name, customer_street, customer_city, customer_state,
customer_country, customer_zip, customer_credit, customer_expiry)
```

Figure 2. Net.Data Macro Fragment

Note that the DATABASE configuration variable will substituted into the ENVIRONMENT statement of the initialization file to generate the cliette name. This allows you to access multiple databases from the same macro.

Figure 3 shows a fragment of the Net.Data initialization file that contains the ENVIRONMENT statement and the associated cliette type. There is one ENVIRONMENT statement for each cliette type in the initialization file. For each database cliette type, the ENVIRONMENT statement specifies a cliette name. The name is made up of the cliette type and a variable reference, $(DATABASE), which is resolved at run time. Each language environment that uses Live Connection must have a cliette definition in the ENVIRONMENT statement.

```
ENVIRONMENT (DTW_SQL)
    (IN DATABASE, LOGIN, PASSWORD, TRANSACTION_SCOPE, SHOWSQL,
    ALIGN, START_ROW_NUM, DTW_SET_TOTAL_ROWS)
    CLIETTE ″DTW_SQL:$(DATABASE)″
```

Figure 3. Net.Data Initialization File Fragment

Figure 4 on page 10 shows a fragment of the Live Connection configuration file, which contains the cliette definitions for DTW_SQL:SAMPLE and DTW_SQL:CELDIAL.

```
CONNECTION_MANAGER{
        MAIN_PORT=7128
        ADMIN_PORT1=7131
        ADMIN_PORT2=7133
        ENCRYPTION=OFF
        }

        ######################################################################
        # This is a comment in a Live Connection configuration file.
        # Comments start with a pound (hash) character.
        # Comments terminate at the end of the line and do not continue to
        # the next line unless another pound (hash) character is specified.
        # You can include comments at the end of lines containing Live
        # Connection keywords except on password lines.
        # You cannot include comments anywhere on lines containing the
        # password keyword.
        # You cannot include spaces and pound (hash) characters within any
        # name, such as cliette name or in database cliette passwords.
       ######################################################################
        CLIETTE DTW_SQL:SAMPLE{
        MIN_PROCESS=1
        MAX_PROCESS=3
        START_PRIVATE_PORT=7100
        START_PUBLIC_PORT=7300
        EXEC_NAME=dtwcdb2.exe
        DATABASE=SAMPLE
        LOGIN=USER1
        PASSWORD=HAMLET
        }

        CLIETTE DTW_SQL:CELDIAL{
        MIN_PROCESS=1
        MAX_PROCESS=5
        START_PRIVATE_PORT=7500
        START_PUBLIC_PORT=7700
        EXEC_NAME=dtwcdb2.exe
        DATABASE=CELDIAL
        LOGIN=USER2
        PASSWORD=OPHELIA
        }
```

Figure 4. Live Connection configuration file fragment

Customizing the Net.Data Initialization File

The information contained in the initialization file is specified using three types of configuration statements, described in the following sections:

- “Configuration Variable Statements” on page 11
- “Path Configuration Statements” on page 17
- “Environment Configuration Statements” on page 21

The sample initialization file shown in Figure 5 on page 11 contains examples of these statements and is valid for for OS/2 and Windows NT.

The text of each individual configuration statement must all be on one line. (An ENVIRONMENT statement is shown on several lines for readability.) Ensure that the initialization file contains an ENVIRONMENT statement for each language environment that you call from your macros. If you fully qualify all references to files within the macro, you do not need to specify any of the path configuration statements.

```
1  DTW_CM_PORT 7128
2  DTW_INST_DIR c:\db2www
3  DTW_LOG_DIR c:\db2www\logs
4  DB2INSTANCE DB2
5  DTW_DIRECT_REQUEST NO
6  DTW_SHOWSQL NO
7  MACRO_PATH c:\DB2WWW\Macro
8  HTML_PATH  c:\www\html
9  INCLUDE_PATH c:\db2www\Macro
10 EXEC_PATH c:\db2www\Macro
11 FFI_PATH c:\pub\ffi;pub\ffi\data
12 ENVIRONMENT (DTW_SQL)          [DLL path]   [Parameter list]
13 ENVIRONMENT (DTW_SYB)          [DLL path]   [Parameter list]
14 ENVIRONMENT (DTW_ORA)          [DLL path]   [Parameter list]
15 ENVIRONMENT (DTW_ODBC)         [DLL path]   [Parameter list]
16 ENVIRONMENT (DTW_DEFAULT)      [DLL path]   [Parameter list]
17 ENVIRONMENT (DTW_APPLET)       [DLL path]   [Parameter list]
18 ENVIRONMENT (DTW_REXX)         [DLL path]   [Parameter list]
19 ENVIRONMENT (DTW_PERL)         [DLL path]   [Parameter list]
20 ENVIRONMENT (DTW_SYSTEM)       [DLL path]   [Parameter list]
21 ENVIRONMENT (DTW_FILE)         [DLL path]   [Parameter list]
22 ENVIRONMENT (DTW_WEBREG)       [DLL path]   [Parameter list]
23 ENVIRONMENT (DTW_JAVAPPS)      [DLL PATH]   [Parameter list]
24 ENVIRONMENT (HWS_LE)           [DLL path]   [Parameter list]
```

- Lines 1 - 6 define configuration variables
- Lines 7 - 11 define paths to files required to process the macro
- Lines 12 - 24 define the environment statements that are available.

Figure 5. The Net.Data initialization file

The following sections describe how to customize the configuration statements in the initialization file.

Configuration Variable Statements

Net.Data configuration variable statements set the values of configuration variables. Configuration variables are used for various purposes. Some variables are required by a language environment to work properly or to operate in an alternate mode. Other variables control the character encoding or content of the Web page being constructed. Additionally, you can use configuration variable statements to define application-specific variables.

The configuration variables you use depend on the language environments, and databases, you are using, as well as other factors that are specific to the application.

To update the configuration variable statements:

Customize the initialization file with the configuration variables that are required for your application. A configuration variable has the following syntax:

NAME[=]*value-string*

The equal sign is optional, as denoted by the brackets.

The following sub-sections describe the configuration variables statements that you can specify in the initialization file:

Cache Manager Configuration Variables

Two optional configuration variables are used if the Cache Manager runs on a machine other than where the Net.Data macro runs:

- DTW_CACHE_PORT specifies which port number Net.Data uses to connect to the Cache Manager.
- DTW_CACHE_HOST specifies the TCP/IP host name of the local or remote machine.

If the Cache Manager runs on the local machine, UNIX-domain sockets or named pipes are used for communication and no configuration is necessary.

The Cache Manager runs on AIX and Windows NT machines, only. See "Net.Data Caching" on page 161 to learn about Net.Data caching.

DTW_CACHE_PORT: Cache Manager Port Variable

Specifies the TCP/IP port the Cache Manager is listening on. This port number must match the port number specified in the Cache Manager configuration file, so Net.Data can communicate with the Cache Manager. If not specified, Cache Manager uses the default port 7175.

Syntax:

```
DTW_CACHE_PORT=port_number
```

Parameter:

port_number

A unique port number assigned to the Cache Manager to service cache requests. The default value is 7175.

Table 3 describes the options for specifying machine IDs and port numbers for these variables.

Table 3. Cache Manager Configuration Variables: Configuration Options

Default Connection Manager Values	If the cache machine is specified ...	If the cache machine is not specified ...

Table 3. Cache Manager Configuration Variables: Configuration Options (continued)

If the cache port is specified ...	Net.Data connects to the Cache Manager on the specified machine using the specified port.	Net.Data connects to the Cache Manager on the local machine using the specified port.
If the cache port is not specified ...	Net.Data connects to the Cache Manager on the specified machine using the default port of 7175.	Net.Data connects to the Cache Manager on the local machine using the default port of 7175.

DTW_CACHE_HOST: Cache Manager Machine ID Variable

Specifies the machine where the Cache Manager resides. If not specified, Net.Data assumes the correct machine is the local machine.

Syntax:

```
DTW_CACHE_HOST=host_name
```

Parameter:

host_name

The qualified TCP/IP host name of the local or remote machine where the Cache Manager runs. The default value is the local machine's host name.

DB2INSTANCE: DB2 Instance Variable

Specifies the instance of DB2 used by the SQL language environment. This variable value is required when Net.Data connects to DB2 running on the Windows NT, OS/2, and UNIX operating systems.

DB2 on the OS/2, Windows NT, and UNIX operating systems needs DB2INSTANCE to be defined as an environment variable. If Net.Data detects that DB2INSTANCE is not defined as an environment variable, it will set the DB2INSTANCE environment variable to the value of DB2INSTANCE found in the initialization file before attempting to connect to DB2.

Syntax:

```
DB2INSTANCE instance_name
```

DTW_CM_PORT: Live Connection Port Number Variable

Specifies a unique port number that Net.Data uses for Live Connection.

Syntax:

```
DTW_CM_PORT port_number
```

Where *port_number* specifies the unique port number used for Live Connection.

DTW_DIRECT_REQUEST: Enable Direct Request Variable

Enables or disables Net.Data direct request invocation. By default, direct request is disabled.

The direct request method of invoking Net.Data allows a user to specify the execution of an SQL statement or Perl, REXX, or C program directly within a URL. When direct request is disabled, the user must invoke Net.Data using the macro request method, allows users to execute only those SQL statements and functions

defined or called in a macro. See “Using Net.Data Mechanisms” on page 56 for security-related recommendations when using DTW_DIRECT_REQUEST.

Syntax:

```
DTW_DIRECT_REQUEST YES|NO
```

Where:

YES Enables Net.Data direct request.

NO Disables Net.Data direct request. NO is the default.

DTW_INST_DIR: Net.Data Installation Directory Variable

Locates certain files during Net.Data execution. You set this variable at installation time to specify the home directory, *<inst_dir>*, where Net.Data is installed. Do not change this value after installation.

DTW_LOG_DIR: Error Log Location Variable

Specifies the directory where the error logs are stored. When logging is enabled with the DTW_LOG_LEVEL variable in the macro, the log files are stored in the directory specified in the path statement of the DTW_LOG_DIR variable. The default is `\inst_dir\logs\netdata.logs`. See “Logging Net.Data Error Messages” on page 183 to learn about logging error messages with Net.Data and about the DTW_LOG_LEVEL variable.

Requirement: The DTW_LOG_DIR variable must be defined for Net.Data to log files. If not defined, logging does not occur even if DTW_LOG_LEVEL is set to ERROR or WARNING in the macro.

Syntax:

```
DTW_LOG_DIR \inst_dir\path
```

Example: Initialization file configuration

```
DTW_LOG_DIR \inst_dir\mylogfiles\
```

DTW_MBMODE: Native Language Support Variable

Activates national language support for word and string functions. When the value of this variable is YES, all string and word functions correctly process DBCS characters within strings by treating strings as mixed data (that is, as strings that potentially contain characters from both single-byte character sets and double byte character sets). The default value is NO. You can override the value set in the initialization file by setting the DTW_MBMODE variable in a Net.Data macro.

This configuration variable works with the DTW_UNICODE configuration variable. If DTW_UNICODE uses the default value of NO, the value of DTW_MBMODE is used. If DTW_UNICODE is set to a value other than NO, its value is used. Table 4 illustrates how the settings of these two variables determine how built-in functions process strings:

Table 4. Relationship Between the Settings of DTW_UNICODE and DTW_MBMODE

If DTW_UNICODE is set to	If DTW_MBMODE=YES	If DTW_MBMODE=NO
NO	Supports DBCS mixed with SBCS	Supports SBCS only

Table 4. Relationship Between the Settings of DTW_UNICODE and DTW_MBMODE (continued)

If DTW_UNICODE is set to	If DTW_MBMODE=YES	If DTW_MBMODE=NO
UTF8	Supports UTF-8	Supports UTF-8

Syntax:

```
DTW_MBMODE [=] NO|YES
```

DTW_SHOWSQL: Enable or Disable SHOWSQL Configuration Variable

Overrides the effect of setting SHOWSQL within your Net.Data macros.

Syntax:

```
DTW_SHOWSQL YES|NO
```

Where:

YES Enables SHOWSQL in any macro that sets the value of SHOWSQL to YES.

NO Disables SHOWSQL in your macros, even if the variable SHOWSQL is set to YES. NO is the default.

Table 5 describes how the settings in the Net.Data initialization file and the macro determine whether the SHOWSQL variable is enabled or disabled for a particular macro.

Table 5. The Relationship Between Settings in the Net.Data Initialization File and the Macro for SHOWSQL

Setting of DTW_SHOWSQL	Setting SHOWSQL	SQL statement is displayed
NO	NO	NO
NO	YES	NO
YES	NO	NO
YES	YES	YES

DTW_SMTP_SERVER: E-mail SMTP Server Variable

Specifies the SMTP server to use for sending out e-mail messages using the DTW_SENDMAIL built-in function. The value of this variable can either be a host name or an IP address. If this variable is not set, Net.Data uses the local host as the SMTP server.

Syntax:

```
DTW_SMTP_SERVER server_name
```

Where `server_name` is the host name or IP address of the the SMTP server that is to be used for sending e-mail messages.

Performance tip: Specify an IP address for this value to prevent Net.Data from connecting to a domain name server when retrieving the IP address of the specified SMTP server.

Example:

```
DTW_SMTP_SERVER us.ibm.com
```

DTW_UNICODE: Unicode Variable

Specifies whether Net.Data supports Unicode in:

- Macros
- Form data
- Data retrieved from a DB2 database
- Strings processed by Net.Data built-in functions

Net.Data supports UTF-8 Unicode format in macros, form data, and built-in functions and the output is always in UTF-8. Net.Data can access a database that contains UTF-16 data and converts it to UTF-8.

When set to UTF8, DTW_UNICODE tells Net.Data to run in a Unicode environment, which means that it expects the macro file data, the form data input from the browser, and the data coming from the DB2 database to be in UTF-8. Setting this variable tells Net.Data that macro input and output are in UTF-8; as a result, Net.Data generates Web pages in UTF-8.

The DTW_UNICODE configuration variable works with the DTW_MBMODE configuration variable. The value of the DTW_UNICODE configuration variable overrides the setting of the DTW_MBMODE variable, when processing word and string built-in functions. If DTW_UNICODE uses the default value of NO, the value of DTW_MBMODE is used. If DTW_UNICODE is set to a value other than NO, its value is used. Table 6 illustrates how the settings of these two variables determine how built-in functions process strings:

Table 6. Relationship Between the Settings of DTW_UNICODE and DTW_MBMODE

If DTW_UNICODE is set to	If DTW_MBMODE=YES	If DTW_MBMODE=NO
NO	Supports DBCS mixed with SBCS	Supports SBCS only
UTF8	Supports UTF-8	Supports UTF-8

Syntax:

```
DTW_UNICODE NO|UTF8|UTF16
```

Where:

NO Specifies to defer to the value of the DTW_MBMODE variable. Table 6 describes Net.Data support based on the value of DTW_MBMODE.

UTF-8 Specifies to support UTF-8 code page and ignore the value of the DTW_MBMODE configuration variable. UTF-8 represents characters by a variable number of bytes and is ASCII safe.

Required DB2 environment variable: In addition to the DTW_UNICODE flag, a DB2–specific environment variable must be set before Net.Data can connect to a DB2 Unicode database. The DB2CODEPAGE environment variable must be set to `1208` in the environment in which Net.Data runs. For example, for the Apache Web server, add the following line to the HTTPD.CONF file:

```
SetEnv DB2CODEPAGE 1208
```

See your Web server documentation to determine how to set environment variables for CGI scripts, Web server APIs, Fast-CGI programs, or servlets. Net.Data uses the English message catalogue when running in a Unicode environment.

DTW_VARIABLE_SCOPE: Variable Scope Variable

Specifies how Net.Data treats local variable scope: whether local variables remain local or whether local variables can be used outside the function block in which they were created. This variable is provided for backward compatibility with previous versions of Net.Data.

Syntax:

```
DTW_VARIABLE_SCOPE = LOCAL|GLOBAL
```

Where:

LOCAL
Specifies that local variables remain local. This behavior was introduced with Net.Data Version 2.0 and is the default.

GLOBAL
Specifies that local variables can be used outside the function block they were created in. This behavior was the default prior to Net.Data Version 2 and specifying this value prevents you from having to change macro files.

Path Configuration Statements

Net.Data determines the location of files and executable programs used by Net.Data macros from the settings of path configuration statements. The path statements are:

- "MACRO_PATH" on page 18
- "EXEC_PATH" on page 18
- "INCLUDE_PATH" on page 19
- "FFI_PATH" on page 20
- "HTML_PATH" on page 21

These path statements identify one or more directories that Net.Data searches when attempting to locate macros, executable files, text files, LOB files, and include files. The path statements that you need depend on the Net.Data capabilities that your macros use.

Update guidelines:

Several general guidelines apply to the path statements. Exceptions are noted in the description of each path statement.

- Each specified directory ends with a semicolon (;).
- Forward slashes (/) and back slashes (\) are treated the same.
- Each path statement can specify multiple paths, except for the HTML_PATH, which can have only one path statement. Paths are searched from left to right in the order specified. This multiple-path capability lets you organize your files within multiple directories. For example, you can place each of your Web applications in its own directory.
- It is recommended to use absolute path statements.

The following sections describe the purpose and syntax of each path statement and provide examples of valid path statements. The examples can differ from your application, depending on your operating system and configuration.

MACRO_PATH

The MACRO_PATH configuration statement identifies the directories that Net.Data searches for Net.Data macros. For example, specifying the following URL requests the Net.Data macro with the path and file name /macro/sqlm.d2w:

```
http://server/cgi-bin/db2www/macro/sqlm.d2w/report
```

Syntax:

```
MACRO_PATH [=] path1;path2;...;pathn
```

The equal sign (=) is optional, as indicated by brackets.

Net.Data appends the path /macro/sqlm.d2w to the paths in the MACRO_PATH configuration statement, from left to right until Net.Data finds the macro or searches all paths. See “Chapter 4. Invoking Net.Data” on page 61 for information on invoking Net.Data macros.

Example: The following example shows the MACRO PATH statement in the initialization file and the related link that invokes Net.Data.

Net.Data initialization file:

```
MACRO_PATH /u/user1/macros;/usr/lpp/netdata/macros;
```

HTML link:

```
<A HREF="http://server/cgi-bin/db2www/query.d2w/input">Submit another query.</A>
```

If the file *query.d2w* is found in the directory /u/user1/macros, then the fully-qualified path is /u/user1/macros/query.d2w.

If the file is not found in the directories specified in the MACRO_PATH statement:

- If the specified path is absolute, Net.Data searches for the file in the specified path. For example, if the following URL is submitted:

  ```
  http://server/cgi-bin/db2www/u/user1/macros/myfile.txt/report
  ```

  Net.Data searches for the file in the /u/user1/macros/myfile.txt directory path.
- If the specified path is relative, Net.Data searches for the file in all directories, starting with the root (/) directory. For example, if the following URL is submitted:

  ```
  http://server/cgi-bin/db2www/myfile.txt/report
  ```

  and the file myfile.txt was not found in any of the directories specified in MACRO_PATH, then Net.Data attempts to find the file in the root (/) directory: /myfile.txt

EXEC_PATH

The EXEC_PATH configuration statement identifies one or more directories that Net.Data searches for an external program that is invoked by the EXEC statement or an executable variable. The order of the directories in the path statement determines the order Net.Data searches for the directories. If the program is found, the external program name is appended to the path specification, resulting in a fully qualified file name that is passed to the language environment for execution.

Syntax:

```
EXEC_PATH [=] path1;path2;...;pathn
```

Example: The following example shows the EXEC PATH statement in the initialization file and the EXEC statement in the macro that invokes an external program.

Net.Data initialization file:

```
EXEC_PATH /u/user1/prgms;/usr/lpp/netdata/prgms;
```

Net.Data macro:

```
%FUNCTION(DTW_REXX) myFunction() {
  %EXEC{ myFunction.cmd %}
%}
```

If the file `myFunction.cmd` is found in the `/usr/lpp/netdata/prgms` directory, the qualified name of the program is `/usr/lpp/netdata/prgms/myFunction.cmd`.

If the file is not found in the directories specified in the EXEC_PATH statement:

- If the specified path is absolute, Net.Data searches for the file in the specified path. For example, if the following URL is submitted:

  ```
  http://myserver/cgi-bin/db2www/usr/user1/prgms/myFunction.cmd
  ```

  Net.Data searches for the file in the `/u/user1/prgms/myFunction.cmd` directory path.
- If the specified path is relative, Net.Data searches the current working directory. For example, if the following URL is submitted:

  ```
  http://myserver/cgi-bin/db2www/myFunction.cmd/report
  ```

  and the file `myFunction.cmd` was not found in any of the directories specified in EXEC_PATH, then Net.Data attempts to find the file in the current working directory.

INCLUDE_PATH

The INCLUDE_PATH configuration statement identifies one or more directories that Net.Data searches, in the order in which they are specified, to find a file specified on an INCLUDE statement in a Net.Data macro. When it finds the file, Net.Data appends the include file name to the path specification to produce the qualified include file name.

Syntax:

```
INCLUDE_PATH [=] path1;path2;...;pathn
```

Tip: If you are including HTML files from a local Web server, use the INCLUDE_URL construct as shown in the local Web server example for INCLUDE_URL in *Net.Data Reference*. By using the demonstrated syntax, you do not need to update the INCLUDE_PATH to specify directories that are already known to the Web server.

Example 1: The following example shows both the INCLUDE_PATH statement in the initialization file and the INCLUDE statement that specifies the include file.

Net.Data initialization file:

```
INCLUDE_PATH /u/user1/includes;/usr/lpp/netdata/includes;
```

Net.Data macro:

```
%INCLUDE "myInclude.txt"
```

If the file *myInclude.txt* is found in the /u/user1/includes directory, the fully-qualified name of the include file is /u/user1/includes/myInclude.txt.

Example 2: The following example shows the INCLUDE_PATH statement and an INCLUDE file with a subdirectory name.

Net.Data initialization file:

```
INCLUDE_PATH /u/user1/includes;/usr/lpp/netdata/includes;
```

Net.Data macro:

```
%INCLUDE "OE/oeheader.inc"
```

The include file is searched for in the directories /u/user1/includes/OE and /usr/lpp/netdata/includes/OE. If the file is found in /usr/lpp/netdata/includes/OE, the fully qualified name of the include file is /usr/lpp/netdata/includes/OE/oeheader.inc.

If the file is not found in the directories specified in the INCLUDE_PATH statement:

- If the specified path is absolute, Net.Data searches for the file in the specified path. For example, if the following URL is submitted:

  ```
  http://myserver/cgi-bin/db2www/u/user1/includes/oeheader.inc
  ```

  Net.Data searches for the file in the /u/user1/includes/oeheader.inc directory path.
- If the specified path is relative, Net.Data searches the current working directory. For example, if the following URL is submitted:

  ```
  http://myserver/cgi-bin/db2www/my.cmd/report
  ```

  and the file myFunction.cmd was not found in any of the directories specified in INCLUDE_PATH, then Net.Data attempts to find the file in the current working directory.

FFI_PATH

The FFI_PATH configuration statement identifies one or more directories that Net.Data searches, in the order in which they are specified, for a flat file that is referenced by a flat file interface (FFI) function.

Syntax:

```
FFI_PATH [=] path1;path2;...;pathn
```

Example: The following example shows an FFI_PATH statement in the initialization file.

Net.Data initialization file:

```
FFI_PATH /u/user1/ffi;/usr/lpp/netdata/ffi;
```

When the FFI language environment is called, Net.Data looks in the path specified in the FFI_PATH statement.

Because the FFI_PATH statement is used to provide security to those files not in directories in the path statement, there are special provisions for FFI files that are not found. See the FFI built-in functions section in *Net.Data Reference*.

HTML_PATH

The HTML_PATH configuration statement specifies into which directory Net.Data writes large objects (LOBs). This path statement accepts only one directory path.

During installation, Net.Data creates a directory called `tmplobs`, under the directory specified in the HTML_PATH path configuration variable. Net.Data stores all LOB files in this directory. If you change the value of HTML_PATH, create a new subdirectory under the new directory.

Syntax:

```
HTML_PATH [=] path
```

Example: The following example shows the HTML PATH statement in the initialization file.

Net.Data initialization file:

```
HTML_PATH /db2/lobs
```

When a query returns a LOB, Net.Data saves it in the directory specified in the HTML_PATH configuration statement.

Performance tip: Consider system limitations when using LOBs because they can quickly consume resources. See “Using Large Objects” on page 123 for more information.

Environment Configuration Statements

An ENVIRONMENT statement configures a language environment. A language environment is a component of Net.Data that Net.Data uses to access a data source such as a DB2 database or to execute a program written in a language such as REXX. Net.Data provides a set of language environments, as well as an interface that allows you to create your own language environments. These language environments are described in “Chapter 6. Using Language Environments” on page 117 and the language environment interface is described in *Net.Data Language Environment Interface Reference*.

Net.Data requires that an ENVIRONMENT statement for a particular language environment exist before you can invoke that language environment.

You can associate variables with a language environment by specifying the variables as parameters in the ENVIRONMENT statement. Net.Data implicitly passes the parameters that are specified on an ENVIRONMENT statement to the language environment as macro variables. To change the value of a parameter that is specified on an ENVIRONMENT statement in the macro, either assign a value to the variable using the DTW_ASSIGN() function or define the variable in a DEFINE section. **Important:** If a macro variable is defined in a macro but is not specified on the ENVIRONMENT statement, the macro variable will not be passed to the language environment.

For example, a macro can define a DATABASE variable to specify the name of a database at which an SQL statement within a DTW_SQL function is to be executed. The value of DATABASE must be passed to the SQL language environment (DTW_SQL) so that the SQL language environment can connect to the designated

database. To pass the variable to the language environment, you must add the DATABASE variable to the parameter list of the environment statement for DTW_SQL.

The sample Net.Data initialization file makes several assumptions about customizing the setting of Net.Data environment configuration statements. These assumptions may not be correct for your environment. Modify the statements appropriately for your environment.

To add or update an ENVIRONMENT statement:

ENVIRONMENT statements have the following syntax:

```
ENVIRONMENT(type) library_name (parameter_list, ...) [CLIETTE "cliette_name"]
```

Parameters:

- *type*

  The name by which Net.Data associates this language environment with a FUNCTION block that is defined in a Net.Data macro. You must specify the type of the language environment on a FUNCTION block definition to identify the language environment that Net.Data should use to execute the function.

- *library_name*

  The name of the DLL or shared library containing the language environment interfaces that Net.Data calls.

  – In AIX, the name of the shared library is specified with the *.o* extension.
  – In HP-UX, the name of the shared library is specified with the *.sl* extension
  – In SUN, SCO, and LINUX the name of the shared library is specified with the *.so* extension
  – In OS/2 and Windows NT the DLL name is specified without the *.dll* extension.

- *parameter_list*

  The list of parameters that are passed to the language environment on each function call, in addition to the parameters that are specified in the FUNCTION block definition.

  To set and pass the variables in the parameters list, define the variable in the macro.

  You must define these parameters as configuration variables or as variables in your macro before executing a function that will be processed by the language environment. If a function modifies any of its output parameters, the parameters keep their modified value after the function completes. The following list specifies which variables the ENVIRONMENT statements can pass:

  DTW_SQL: TRANSACTION_SCOPE, LOCATION, DB2SSID, DB2PLAN

  DTW_ODBC: TRANSACTION_SCOPE, LOCATION

- *cliette_name*

  The name of the cliette. The *cliette_ name* can refer to the Java Application language environment cliette, or it can be a database cliette. The *cliette_name* parameter is used with the CLIETTE keyword, both of which are only used with Live Connection. `CLIETTE` and *`cliette_name`* are optional and can be specified only for database and Java application language environments, and for user-defined language environments that have cliettes written for them.

  Java Application cliette
  This cliette name specifies the Java Application language environment.

Syntax:

```
CLIETTE "DTW_JAVAPPS"
```

Database cliette

This cliette name specifies a cliette that is associated with a database.

Syntax:

```
CLIETTE "type:db_name"
```

Parameters:

type The database language environment associated with the cliette. See page 49 for a list of valid types.

db_name
The database cliette name. This name is often the same as the database with which the cliette is associated, such as MYDBASE, but can also be another name. *db_name* is optional when using the Oracle language environment.

When Net.Data processes the initialization file, it does not load the language environment DLLs or shared libraries. Net.Data loads a language environment DLL or shared library when it first executes a function that identifies that language environment. The DLL or shared library then remains loaded for as long as Net.Data is loaded.

Example: ENVIRONMENT statements for Net.Data-provided language environments

When customizing the ENVIRONMENT statements for your application, add the variables on the ENVIRONMENT statements that need to be passed from your initialization file to a language environment or that Net.Data macro writers need to set or override in their macros.

```
ENVIRONMENT (DTW_SQL)      DTWSQL    ( IN DATABASE, LOGIN, PASSWORD,
  TRANSACTION_SCOPE, SHOWSQL, ALIGN, START_ROW_NUM, DTW_SET_TOTAL_ROWS)
  CLIETTE "DTW_SQL:MYDBASE"
ENVIRONMENT (DTW_SYB)      DTWSYB    ( IN DATABASE, LOGIN, PASSWORD,
  TRANSACTION_SCOPE, SHOWSQL, ALIGN, START_ROW_NUM, DTW_SET_TOTAL_ROWS)
ENVIRONMENT (DTW_ORA)      DTWORA    ( IN DATABASE, LOGIN, PASSWORD,
  TRANSACTION_SCOPE, SHOWSQL, ALIGN, START_ROW_NUM, DTW_SET_TOTAL_ROWS)
ENVIRONMENT (DTW_ODBC)     DTWODBC   ( IN DATABASE, LOGIN, PASSWORD,
  TRANSACTION_SCOPE, SHOWSQL, ALIGN, DTW_SET_TOTAL_ROWS)
ENVIRONMENT (DTW_APPLET)   DTWJAVA   ( )
ENVIRONMENT (DTW_JAVAPPS)  DTWJAVAPPS ( OUT RETURN_CODE ) CLIETTE "DTW_JAVAPPS"
ENVIRONMENT (DTW_PERL)     DTWPERL   ( OUT RETURN_CODE )
ENVIRONMENT (DTW_REXX)     DTWREXX   ( OUT RETURN_CODE )
ENVIRONMENT (DTW_SYSTEM)   DTWSYS    ( OUT RETURN_CODE )
ENVIRONMENT (HWS_LE)       DTWHWS    ( OUT RETURN_CODE )
```

Required: Each ENVIRONMENT statement must be on a single line.

Setting Up the Language Environments

After you modify configuration variables and ENVIRONMENT configuration statements for the Net.Data language environments, some additional setup is required before the following language environments can function properly. The following sections describe the steps necessary to set up the language environments:

- "Setting up the IMS Web Language Environment"
- "Setting up the Java Language Environment"
- "Setting up the Oracle Language Environment" on page 25
- "Setting up the Sybase Language Environment" on page 27

Setting up the IMS Web Language Environment

To use the IMS Web language environment, you must complete the following steps:

1. Install the IMS Web Runtime component on the Web server running Net.Data. For information about installing and using the IMS Web Runtime component, see *IMS Web User's Guide*:

   `http://www.software.ibm.com/data/ims/about/imsweb/document/`

2. Create the transaction DLL.
   a. Generate the C++ code, makefile (DTW*proj*.mak), and Net.Data macros (DTW*proj*.d2w) files for your transaction with the IMS Web Studio tool.
   b. If you are running Net.Data on an operating system that is different than the operating system on which the IMS Web Studio tool is run, move the DLL source files to an IMS Web development machine for the target operating system. For example, if you run the IMS Web Studio tool on Windows NT and the target platform is AIX or OS/390, move the source for the transaction DLL to an IMS Web development machine running under AIX or OS/390, respectively.
   c. Build the executable form of the transaction DLL using the generated make file.
3. Copy the transaction DLL file (DTW*proj*.dll) and Net.Data macro file (DTW*proj*.d2w) to your Web server.
   a. Place the macro in a directory from which Net.Data retrieves macros. (See "MACRO_PATH" on page 18 for more information.)
   b. Place the transaction DLL or shared library in a directory from which the Web server retrieves DLLs or shared libraries.
4. Use the link in the sample file that is generated by the IMS Web Studio tool, DTW*proj*.htm, to modify an HTML file in your Web server's HTML tree. You can then use the link to invoke Net.Data and display the input HTML form to invoke the IMS Web language environment. The IMS Web language environment then calls the IMS transaction DLL, which uses the services proved by the IMS Web Runtime DLLs to run the transaction and return its output to the Web browser.

   The IMS Web Runtime DLLs formulate and send a request message through IMS TOC to OTMA, which in turn causes the appropriate transaction to be queued. The output of the transaction is then returned by OTMA through IMS TOC to IMS Web. The transaction os then passed back through the IMS Web language environment to Net.Data for display on the Web browser.

Setting up the Java Language Environment

The Java language environment requires some additional set up before you can call functions from a macro:

1. Create a batch file to launch the Java application because Net.Data cannot directly start a Java application. Net.Data uses this file to launch the Java Virtual Machine, which runs your Java function. The batch file must include the java-classpath statement to ensure the required Java packages (the standard

and application-specific packages) can be found. For example, the batch file, `launchjv.bat`, contains the following java-classpath:

```
java -classpath %CLASSPATH%;C:\DB2WWW\Javaclas dtw_samp %1 %2 %3 %4 %5 %6
```

2. Define a cliette to work with the Java language environment in the Live Connection configuration file, `dtwcm.cnf`. Specify unique port numbers for the cliette and the related batch file name with the EXEC_NAME configuration variable. In the following example, the Java cliette name is defined as DTW_JAVAPPS and the EXEC_NAME configuration variable is set to the name of the batch file, `launchjv.bat`:

```
CLIETTE DTW_JAVAPPS{
MIN_PROCESS=1               <= Required: this value must be 1 because
                               the JAVAPPS cliette is multi-threaded.
MAX_PROCESS=1               <= Required: this value must be 1 because
                               the JAVAPPS cliette is multi-threaded.
START_PRIVATE_PORT=5100     <= Must be a unique port number
START_PUBLIC_PORT=5300      <= Must be a unique port number
EXEC_NAME=launchjv.bat      <= The name of the batch file that includes the
                               classpath statements
}
```

   When you start the Net.Data Connection Manager, Net.Data starts the Java cliette specified in the configuration file. The cliette becomes available to process Java language environment requests from your Net.Data macro applications.

3. Update the DTW_JAVAPPS ENVIRONMENT path statement in the Net.Data initialization file, `db2www.ini`, by adding each
4. cliette name to the statement. For example:

```
ENVIRONMENT DTW_JAVAPPS  ( OUT RETURN_CODE ) CLIETTE "DTW_JAVAPPS"
```

Setting up the Oracle Language Environment

Use the following steps to access Oracle databases from a Net.Data macro:

1. Ensure the appropriate components of Oracle are installed and working as follows:
   a. Install SQL*Net on the machine where Net.Data is installed, if it is not already installed. For more information, see the following URL:

      `http://www.oracle.com/products/networking/html/stnd_sqlnet.html`

   b. Verify that the Oracle *tnsping* function can be used with the same security authorization that your Web server uses. To verify, log on with your Web server's user ID and type:

      `tnsping` *`oracle-instance-name`*

      Where *oracle-instance-name* is the name of the Oracle system that your Net.Data macros access.

      You might not be able to verify the *tnsping* function on Windows NT if your Web server runs under system authority. If so, skip this step.

   c. Verify that the Oracle tables can be accessed with the same security authorization that your Web server uses. To verify, enter an SQL SELECT statement, using the SQL*Plus line command tool, to access an Oracle table with an SQL SELECT statement with the authority of your Web server. For example:

      `SELECT * FROM` *`tablename`*

You might not be able to verify table access on Windows NT if your Web server runs under system authority. If so, skip this step.

Troubleshooting: Do not proceed if the above steps fail. If any of the steps fail, check your Oracle configuration.

2. Ensure that the Oracle environment variables are set correctly in your Web server process.
   - For AIX, put the following lines in the `/etc/environment` file:

     ```
     ORACLE_SID=oracle-instance-name
     ORACLE_HOME=oracle-runtime-library-directory
     ```

   - For Windows NT, use the System Properties Control panel to add the following environment variables:

     ```
     ORACLE_SID=oracle-instance-name
     ORACLE_HOME=oracle-runtime-library-directory
     ```

   Hint: You might require additional lines for other Oracle environment variables, depending on the Oracle facilities you plan to use, such as national language support and two phase commit. Consult the Oracle administration documentation for more information on these environment variables.
3. Test the connection to Oracle from Net.Data. In your Net.Data macro, specify the appropriate values in the LOGIN and PASSWORD variables. *Do not* define the Net.Data DATABASE variable when accessing Oracle databases. The following is an example of connect statement in a macro:

   ```
   %DEFINE LOGIN=user_ID@remote-oracle-instance-name
   %DEFINE PASSWORD=password
   ```

   Local Oracle instances:

   If you access the local Oracle instance only, do not specify the remote-oracle-instance name as part of the login user ID, as in the following example:

   ```
   %DEFINE LOGIN=user_ID
   %DEFINE PASSWORD=password
   ```

   Live Connection:

   If you use Live Connection, then you can specify the LOGIN and PASSWORD in the Live Connection configuration file, although it is not recommended for security purposes. For example:

   ```
   LOGIN=user_ID
   PASSWORD=password
   ```

   Hint: Do not specify the DATABASE variable for Oracle.
4. Test your configuration by running a CGI shell script to ensure that the Oracle instance can be accessed from your Web server, as in the following example:

   ```
   #! /bin/sh
   echo "content-type; text/html
   echo
   echo "< html>< pre>"
   set
   echo "</pre>< p>< pre>"
   tnsping oracle-instance-name
   echo
   ```

   Alternatively, you can execute *tnsping* directly from a Net.Data macro, as in the following example:

```
%DEFINE testora = %exec "tnsping oracle-instance-name"
%HTML (report){
< P>About to test Oracle access with tnsping.
< hr>
$(testora)
< hr>
< P>The Oracle test is complete.
%}
```

Troubleshooting:

If the verification step fails, check that all the preceding steps were successful by verifying the following items:

- Check your Oracle configuration.
- Verify that the Oracle environment variable syntax is correct and that no variables are missing.
- Check the Oracle connection, ensuring that you have entered the correct user ID and password.

If the verification step still fails, contact IBM Service.

Example:

After you have completed the accessing verification steps, you can make calls to the Oracle language environment with functions in the macro, as in the following example:

```
%FUNCTION(DTW_ORA) STL1() {
insert into $(tablename) (int1,int2) values (111,NULL)
%}
```

Setting up the Sybase Language Environment

To access Sybase from Net.Data

1. Ensure the appropriate components of Sybase are installed and working as follows:
   a. Install Sybase's Open Client on the machine where Net.Data is installed, if it is not already installed. For more information, see the Sybase Open Client documentation for more information.
   b. Verify that the Sybase *ping* function can be used with the same security authorization that your Web server uses. To verify, log on with your Web server's user ID and type:

      ```
      ping sybase-instance-name
      ```

      Where *sybase-instance-name* is the name of the Sybase system that your Net.Data macros access.

      You might not be able to verify the *ping* function on Windows NT if your Web server runs as an Windows NT service. If so, skip this step.
   c. Verify that the Sybase tables can be accessed with the same security authorization that your Web server uses. To verify, enter an SQL SELECT statement, using the ISQL line command tool, to access an Sybase table with the authority of your Web server. For example:

      ```
      SELECT * FROM tablename
      ```

You might not be able to verify table access on Windows NT if your Web server runs as a Windows NT service. If so, skip this step.

Troubleshooting: Do not proceed if the above steps fail. If any of the steps fail, check your Sybase configuration.

2. Ensure that the Sybase environment variables are set correctly in your Web server process.
   - For AIX, put the following lines in the `/etc/environment` file:

     ```
     DSQUERY=sybase-instance-name
     SYBASE=sybase-runtime-library-directory
     ```

   - For Windows NT, use the System Properties Control panel to add the following environment variables:

     ```
     DSQUERY=sybase-instance-name
     SYBASE=sybase-runtime-library-directory
     ```

   Hint: You might require additional lines for other Sybase environment variables, depending on the Sybase facilities you plan to use, such as national language support and two-phase commit. Consult the Sybase administration documentation for more information on these environment variables.

3. Test the connection to Sybase from Net.Data. In your Net.Data macro, specify the appropriate values in the LOGIN, PASSWORD, and DATABASE variables. The following is an example of connect statement in a macro:

   ```
   %DEFINE DATABASE=database-name
   %DEFINE LOGIN=user_ID@remote-sybase-instance-name
   %DEFINE PASSWORD=password
   ```

   Live Connection: If you use Live Connection, then you can specify the LOGIN and PASSWORD in the Live Connection configuration file, although it is not recommend for security purposes. For example:

   ```
   DATABASE=database-name
   LOGIN=user_ID
   PASSWORD=password
   ```

4. Test your configuration by running a CGI shell script to ensure that the Sybase instance can be accessed from your Web server, as in the following example:

   ```
   #! /bin/sh
   echo "content-type; text/html
   echo
   echo "< html>< pre>"
   set
   echo "</pre>< p>< pre>"
   isql -u user_ID -p password << EOFF
   SELECT * FROM tablename
   EOFF
   echo
   ```

   Troubleshooting:

   If the verification step fails, check that all the preceding steps were successful by verifying the following items:

   - Check your Sybase configuration.
   - Verify that the Sybase environment variable syntax is correct and that no variables are missing.
   - Check the Sybase connection, ensuring that you have entered the correct user ID and password.

   If the verification step still fails, contact IBM Service.

Example:

Once you have completed the accessing verification steps, you can make calls to the Sybase language environment with functions in the macro, as in the following example:

```
%function(DTW_SYB) STL1() {
insert into $(tablename) (int1,int2) values (111,NULL)
%}
```

Configuring Live Connection

Live Connection manages database and Java application connections to improve performance for Net.Data on the Windows NT, OS/2, AIX, and Sun Solaris operating systems. Through the use of a Connection Manager and cliettes, processes that maintain open connections, Live Connection eliminates the start up overhead of connecting to a database or starting a Java Virtual Machine.

Live Connection uses a configuration file, `dtwcm.cnf`, to determine what cliettes need to be started. It contains administration information and definitions for each of the cliettes used with Live Connection. See “Managing Connections” on page 158 to learn more about Live Connection.

The sample configuration file shown in Figure 6 contains the following types of information:

- Connection Manager port information
- SQL cliette information for a DB2 connection
- Java application cliette information

```
 1  CONNECTION_MANAGER{
 2  MAIN_PORT=7100
 3  ADMIN_PORT1=7101
 4  ADMIN_PORT2=7102
 5  }
 6
 7  CLIETTE DTW_SQL:CELDIAL{
 8  MIN_PROCESS=1
 9  MAX_PROCESS=5
10  START_PRIVATE_PORT=7200
11  START_PUBLIC_PORT=7210
12  EXEC_NAME=./dtwcdb2
13  DATABASE=CELDIAL
14  LOGIN=marshall
15  PASSWORD=stlpwd
16  }
17
18  CLIETTE DTW_JAVAPPS{
19  MIN_PROCESS=1
20  MAX_PROCESS=5
21  START_PRIVATE_PORT=7300
22  START_PUBLIC_PORT=7310
23  EXEC_NAME=./javaapp
24  }
```

- Lines 1 - 5 are required for the configuration file and define unique port numbers used with Live Connection.
- Lines 7 - 16 define all database cliettes, identifying the cliette name, the number of processes to be run, database name, port numbers, and the cliette exec file. You can include additional information, such as a user ID and password for connecting to a DB2 database. These additional values are shown in lines 13 - 15.
- Lines 19 - 25 define all cliettes for Java applications, identifying the cliette name, the number of processes to be run, unique port numbers, and the cliette exec file.

Figure 6. The Live Connection configuration file

Before you begin: Read the hints and tips section following these steps before customizing the Live Connection configuration file.

To configure Live Connection ports:

1. Open the configuration file, `dtwcm.cnf`, with an editor.
2. Configure the three Live Connection port numbers:
   - MAIN_PORT
   - ADMIN_PORT1
   - ADMIN_PORT2

   Figure 6 on page 29 displays the default port numbers. If these numbers are not unique, you must change them to unique port numbers.
3. **Important:** Ensure that the value of MAIN_PORT matches the value of DTW_CM_PORT in the Net.Data initialization file.

To configure the database cliettes:

1. Type the cliette environment statement.

   `CLIETTE` *`type`*`:`*`db_name`*

   Parameters:

   type The name that associates a language environment with a cliette. See on page 49 for a list of valid types.

   db_name
   The database cliette name, which is often the same as the database with which the cliette is associated, such as MYDBASE; however the *db_name* can also be another name. *db_name* is optional when using the Oracle language environment.
2. Determine values for MIN_PROCESS and MAX_PROCESS. MIN_PROCESS specifies the number of processes to be started when the Connection Manager is started. Afterwards, if additional simultaneous requests arrive, the Connection Manager starts more cliettes, adding one as needed, until the value specified for MAX_PROCESS is reached. The values you use can affect performance, but you can change them later.

   Type the MIN_PROCESS and MAX_PROCESS statements:

   `MIN_PROCESS=`*`min_num`*
   `MAX_PROCESS=`*`max_num`*

   Parameters:

   min_num
   The number of cliette processes to be started when the Connection Manager is started. You must have enough available unique port numbers for this number of cliettes.

   max_num
   The maximum number of cliettes that can be run simultaneously. You must have enough available unique port numbers for this number of cliettes.
3. Determine which port numbers to use on your system for the database cliette. These numbers must be unique to avoid conflicting with port numbers used for the Cache Manager or other applications. Each cliette uses two ports. When you specify a set of ports, you must specify the range of port numbers to be used. The first two values are START_PUBLIC_PORT and START_PRIVATE_PORT. The other is MAX_PROCESS, indicating the maximum number of cliettes. The following example shows which port numbers are to be used.

```
START_PUBLIC_PORT=1000
START_PRIVATE_PORT=1010
MAX_PROCESS=5
```

The example uses the following ports:

1000	1010
1001	1011
1002	1012
1003	1013
1004	1014

A common error is to have two sets of cliettes overlap the port numbers they use, or overlap with the Cache Manager port numbers. Check with your system administrator to ensure that the port numbers you plan to use are available. The README file for your operating system has general guidelines on what port numbers are valid for your operating system.

4. Specify the name of the cliette executable file. This file name is specified as:

   `EXEC_NAME=./dtwc`*`xxx`*

   Where *xxx* is the database type identifier. Refer to Table 7 for valid executable file names:

Table 7. Cliette exec file names

Cliette Description	Cliette Type	Names		Platform Availability					
		UNIX	Windows NT or OS/2	AIX	NT	OS/2	HP	SUN	SCO
DB2 process cliette	DTW_SQL	dtwcdb2	dtwcdb2.exe	Y	Y	Y	Y	Y	N
ODBC process cliette	DTW_ODBC	dtwcodbc	dtwcodbc.exe	Y	Y	N	N	N	N
Sybase process cliette	DTW_SYB	dtwcsyb	dtwcsyb.exe	Y	Y	N	N	N	N
Oracle process cliette	DTW_ORA	dtwcora	dtwcora.exe	Y	Y	N	N	N	N

5. Specify the name of the database with which the cliette is associated:

   `DATABASE=`*`db_name`*

   Where *db_name* is the name of the database with which the cliette is associated; for example, MYDBASE.

6. Optional: Change the default values for the LOGIN and PASSWORD variables so that Net.Data uses the same user ID that started the Connection Manager to connect to the DB2 database. By specifying these default values, you avoid placing this information in the configuration file. For example, replace lines 14 and 15, in the sample configuration file in Figure 6 on page 29 with these lines:

   ```
   LOGIN=*USE_DEFAULT
   PASSWORD=*USE_DEFAULT
   ```

   Tip: If you define multiple cliette entries in the configuration file, you can specify various database login and passwords for a particular database.

To configure the Java application cliettes:

1. Type the cliette environment statement:

   ```
   CLIETTE DTW_JAVAPPS
   ```

2. Determine values for MIN_PROCESS and MAX_PROCESS. MIN_PROCESS specifies the number of processes that are to be started when the Connection Manager is started. Afterwards, if simultaneous processes arrive, the Connection Manager starts more cliettes, adding one as needed, until the value specified for MAX_PROCESS is reached. The values you use can affect performance, but you can change them later.

   Type the MIN_PROCESS and MAX_PROCESS statements.

   ```
   MIN_PROCESS=min_num
   MAX_PROCESS=max_num
   ```

   Parameters:

   min_num
   : The number of cliette processes started when the Connection Manager is started. You must have enough available unique port numbers for this number of cliettes.

   max_num
   : The maximum number of additional cliettes that can be run simultaneously. You must have enough available unique port numbers for this number of cliettes.

3. Determine which port numbers to use on your system for the database cliette. These numbers must be unique to avoid conflicting with port numbers used for the Cache Manager or other applications. Each cliette uses two ports. When you specify a set of ports, you must specify the range of port numbers to be used. The first two values are START_PUBLIC_PORT and START_PRIVATE_PORT. The other is MAX_PROCESS, indicating the maximum number of cliettes. The following example shows which port numbers are to be used.

   ```
   START_PUBLIC_PORT=1000
   START_PRIVATE_PORT=1010
   MAX_PROCESS=5
   ```

   The example uses the following ports:

1000	1010
1001	1011
1002	1012
1003	1013
1004	1014

   A common error is to have two sets of cliettes overlap the port numbers they use, or overlap with the Cache Manager port numbers. Check with your system administrator to ensure that the port numbers you plan to use are available. The README file for your operating system has general guidelines on what port numbers are valid for your operating system.

Hints and tips for configuring Live Connection:

- Cliette names are used by the Connection Manager to uniquely identify a set of cliettes.
- For database cliettes, you must have one named set of cliettes for each database you plan to access. For databases that are rarely accessed, you can

set the MIN and MAX number of cliettes to 1. Alternatively, you can also set MIN to 0, which means processes are not started until a Net.Data request is made for the cliette.

- The NAME of the cliette must be consistent with the cliette name referenced in the ENVIRONMENT statement for the cliettes type in initialization file. The cliette name can contain variables, and in the case of DB2 cliettes, it should include the variable reference $(DATABASE). The default value for the cliette name in the ENVIRONMENT statement is DTW_SQL:$(DATABASE). You can use a variable reference in the initialization file, but not the Live Connection configuration file.

  The DATABASE variable is defined in the Net.Data macro. When an SQL statement in the macro is encountered, the $(DATABASE) variable reference in the Net.Data initialization file is replaced with the current value of DATABASE.

  You can use this method to access multiple databases. If you have three databases that you wanted to access in your Net.Data macro (for example, D1, D2, and D3), and your initialization file has the standard CLIETTE ″DTW_SQL:$(DATABASE)″ line, then you need three sections in the configuration file such as:

  ```
  CLIETTE DTW_SQL:D1{ ...}
  CLIETTE DTW_SQL:D2{....}
  CLIETTE DTW_SQL:D3{....}
  ```

- Processes are started but not stopped. If you set the maximum number of processes to M and at any time M processes are used simultaneously, they stay active until you shut down the Connection Manager, therefore you do not want the value of MAX_PROCESS to be so high that you use up all of your system resources starting processes that are rarely used.

  Recommendation: Try using different values for MIN_PROCESS and MAX_PROCESS to see what works best for your system. If the Connection Manager receives more requests than the specified maximum value, the last request is queued until a cliette finishes processing. When a cliette becomes available, the queued request is then processed. This process of queuing requests is transparent to the application user.

- You can use the same type of cliette for different named sections. For example, all DB2 database sections of the configuration file use the same cliette type. You cannot have two sections with the same name.

If you are using CGI, and want only some databases to use Live Connection, simply list the databases you want in the configuration file. When Net.Data is processing a Net.Data macro and encounters an SQL section, it asks the Connection Manager for a specific cliette. If the Connection Manager does not have that type of cliette, it responds with a NO_CLIETTE_AVAIL message. Net.Data processes the request with a DLL version instead.

To automatically start Connection Manager as a Windows NT service:

On Windows NT, you can specify to have Connection Manager start as an Windows NT service, instead of from the command line. Running Connection Manager as an Windows NT service allows Connection Manager to be automatically started each time the machine is started.

Tip: Start Connection Manager from the command line before setting it up to start automatically to insure that the Live Connection configuration file is correct.

- From the Windows NT task bar, select **Start->Settings->Control Panel ->Services**.
- Select **Net.Data Connection Manager**, then click the **Startup** button.

- Select **Automatic startup type**, then click on **OK**.

Configuring the Web Server for Use with CGI

The Common Gateway Interface (CGI) is an industry-standard interface that enables a Web server to invoke an application program such as Net.Data. Net.Data's support for CGI lets you use Net.Data with your favorite Web server.

Configure the Web server to invoke Net.Data by adding Map, Exec, and Pass directives to the HTTP configuration file so that Net.Data gets invoked.

Recommendation: Organize the directives in the following order within the HTTP configuration file to prevent directives from being ignored: Map, Exec, Pass. For example, if the following Pass directive precedes a Map or Exec directive, the Map and Exec directives are ignored:

```
Pass /*
```

Map directives

The Map directives map entries using the format `/cgi-bin/db2www/*` to the library where the Net.Data program resides on your system. (The asterisk (*) at the end of the string refers to anything that follows the string.) Both upper- and lower-case map statements are included, because the directives are case sensitive.

Exec directives

The Exec directive enables the Web server to execute any CGI programs in the CGI library. Specify the library where the program resides (not the program itself) on the directive.

Pass directives

Pass directives are not used by Net.Data. If you want to simplify your URL, then use the MACRO_PATH statement in a Net.Data initialization file, discussed in "MACRO_PATH" on page 18.

Configuring Net.Data for FastCGI

Net.Data can execute as a FastCGI process on Apache Web Server and Domino Go Webserver. FastCGI provides similar performance to the other Web API programs with the reliability of CGI. FastCGI is supported on the AIX and Sun Solaris operating systems.

Before You Begin:

Before you use FastCGI, ensure that you have installed the prerequisite products:

- **For Apache:** Download and install the Apache Web Server 1.2.0 or higher.
- **For Domino Go Webserver:** Download and install the Domino Go Webserver for AIX or Sun from:

  `http://www.ics.raleigh.ibm.com/dominowebserver`

To configure Net.Data for FastCGI:

1. Configure the Web server and FastCGI configuration file for your operating system:

For Apache Web server:

Update the `http.conf` file.

- Declare the new application:

```
AppClass inst_dir
-processes proc_num
-initial-env LIBPATH=libpath
-initial-env ORACLE_HOME=oracle_path
-initial-env ORACLE_SID=oracle_instance
-initial-env SYBASE=sybase_path
-initial-env DSQUERY=sybase_instance
-initial-env DB2INSTANCE=db2_instance
-initial-env RXQUEUE_OWNER_PID=REXX_perf_var
-initial-env LANG=locale
```

- Declare the FastCGI module:

```
<location /fcgi-bin>
SetHandler fastcgi-script
</location>
```

For Domino Go Webserver:

Update the `httpd.conf` and `fcgi.conf` files:

- In the `httpd.conf` file, declare the service section:

```
ServerInit /u/mydir/http/fcgi-bin/fcgi.o:FCGIInit
/u/mydir/http/fcgi.conf service/fcgi-bin/*
/u/mydir/http/fcgi-bin/fcgi.o:FCGIDispatcher*ServerTerm
/u/mydir/http/fcgi-bin/fcgi.o:FCGIStop
```

- In the `fcgi.conf` file, declare the application:

```
Local {
Exec inst_dir
Role Responder
URL /fcgi-bin/db2www
BindPath /tmp/db2www.ibm
NumProcesses proc_num
Environ LIBPATH=libpath
Environ ORACLE_HOME=oracle_path
Environ ORACLE_SID=oracle_instance
Environ SYBASE=sybase_path
Environ DSQUERY=sybase_instance
Environ DB2INSTANCE=db2_instance
Environ RXQUEUE_OWNER_PID=REXX_perf_var
Environ LANG=locale
}
```

Parameters:

inst_dir

The path and directory name for Net.Data's executable files.

For Apache:

```
AppClass /u/mydir/apache/fcgi-bin/db2www
```

For Domino Go Webserver:

```
Exec /u/mydir/http/fcgi-bin/db2www
```

Role Responder

Required keyword for Domino Go Webserver, only.

URL Required keyword and URL address for Domino Go Webserver, only. The URL points to the path specified for the EXEC_PATH statement.

BindPath

Required keyword and path statement for Domino Go Webserver on AIX only. The path of the unique UNIX socket used by Net.Data and FastCGI.

proc_num

The number of requests that can be handled simultaneously. The default is 1, but should be increased to improved performance, based on your application requirements. See “Using FastCGI” on page 157 for tuning information.

For Apache:

```
-processes 7
```

For ICS or Domino Go Webserver:

```
NumProcesses 7
```

libpath The LIBPATH (shared library or DLL) statements declared in each ENVIRONMENT statement in the Net.Data initialization file. When accessing DB2, the LIBPATH statement should contain the path to the DB2 UDB library directory. For example:

```
/usr/lpp/db2_05_00/lib
```

For Apache:

```
-initial-env LIBPATH=/u/mydir/apache/lib:/u/mydir/apache:/usr/lib
```

For Domino Go Webserver:

```
Environ LIBPATH=/u/mydir/http/lib:/u/mydir/http:/usr/lib
```

oracle_path

Required when using Oracle. The path and directory of the Oracle database executable files.

For Apache:

```
-initial-env ORACLE_HOME=/home.native/oracle/product/7.2
```

For Domino Go Webserver:

```
Environ ORACLE_HOME=/home.native/oracle/product/7.2
```

oracle_instance

Required when using Oracle. The instance of the Oracle database. You must use Live Connection for Oracle.

For Apache:

```
-initial-env ORACLE_SID=mvpdb2
```

For Domino Go Webserver:

```
Environ ORACLE_SID=mvpdb2
```

sybase_path

Required when using Sybase. The path and directory of the Sybase database executable files.

For Apache:

```
-initial-env SYBASE=/home.native/sybase/product
```

For Domino Go Webserver:

```
Environ SYBASE=/home.native/sybase/product
```

sybase_instance
Required when using Sybase. The instance of the Sybase database. You must use Live Connection for Sybase

For Apache:

```
-initial-env DSQUERY=SybaseAIX
```

For Domino Go Webserver:

```
Environ DSQUERY=SybaseAIX
```

db2_instance
Required when using DB2. The instance of the DB2 database.

For Apache:

```
-initial-env DB2INSTANCE=wwwinst
```

For Domino Go Webserver:

```
Environ DB2INSTANCE=wwwinst
```

REXX_perf_var
Required when using REXX on AIX. The performance variable is used with FastCGI and REXX on the AIX operating system. The default is 0. For other products and operating systems, declare this variable in the Net.Data macro. See "Appendix B. Net.Data for AIX" on page 193 for more information about this variable.

For Apache:

```
-initial-env RXQUEUE_OWNER_PID=0
```

For Domino Go Webserver:

```
Environ RXQUEUE_OWNER_PID=0
```

locale The UNIX locale variable. Use En_US for U.S. English.

For Apache:

```
-initial-env LANG=En_US
```

For Domino Go Webserver:

```
Environ LANG=En_US
```

2. **For Apache:** Add the `fgi-bin` directory as a new script alias in the srm.conf file:
   `ScripAlias /fcgi-bin/ /u/mydir/apache/fci-bin`
3. Migrate any hyperlinks in static or dynamically generated Web pages from CGI-BIN to FCGI-BIN. For example:
   `<A HREF="http://server/fcgi-bin/db2www/filename.ext/block/[?name=val&...]">any text</A>`
4. Modify end-user documentation for URL invocations of Net.Data with FCGI-BIN instead of CGI-BIN. For example:
   `http://server/fcgi-bin/db2www/filename.ext/block/[?name=val&...]`

Configuring Net.Data for use with Java Servlets and Java Beans

See your Web server documentation for instructions on registering and using servlets. The Net.Data servlets are contained in the NetDataServlets.jar file. Your Web server will require that you add *inst_dir*`/servlet-lib/NetDataServlets.jar` and *inst_dir*`/servlet-lib` to your CLASSPATH.

For more information on installing the Web server and on Web server configuration file directives, refer to your Web server documentation.

Configuring Net.Data for Use with the Web Server APIs

Using a Web server application programming interface (API) rather than CGI can improve the performance of Net.Data considerably. Net.Data supports the following server APIs:

- IBM Internet Connection Server API (ICAPI)
- Lotus Domino Go Webserver API (GWAPI)
- Microsoft Internet Server API (ISAPI)
- Netscape API (NSAPI)

For more information about each API, see "Using the Web Server APIs" on page 157 and the README file for your version of Net.Data.

Requirement: To run Net.Data in ICAPI, GWAPI, ISAPI, or NSAPI mode, you must reconfigure your Web server to use Net.Data DLLs or shared libraries as its service directives. After reconfiguring, you must restart your Web server so that any changes you make to the Net.Data initialization file take effect. By default, Net.Data runs in CGI mode.

The following sections describe how to configure Net.Data and the Web server to run Web server API mode. General steps and examples are provided, but they might differ for your operating system. Refer to the Net.Data README file for your operating system for specific instructions.

To configure ICAPI and GWAPI:

The Domino Go Webserver is the follow-on product to IBM Internet Connection Secure Server. If you are upgrading, you might wish to use the new Domino Go Webserver. Note that GWAPI and ICAPI are the same product, just renamed to identify which Web server is being used.

1. Stop the Web server.
2. Ensure that the ICAPI or GWAPI DLL or shared library is in the server's `CGI-BIN` or `ICAPI-LIB` directory.

   See the Net.Data README file or program directory for your operating system for specific file and directory names.
3. Add a service statement to your Web server's configuration file (`httpd.conf` or `httpd.cnf`) to call the API.

   For example:

   ```
   Service /cgi-bin/db2www*   /usr/lpp/internet/server_root/cgi-bin/dtwicapi.o:dtw_icapi*
   ```

   See the Net.Data README file for your operating system for specific file and directory names.
4. Restart the Web server.

ICAPI and GWAPI have the full compatibility to support the existing applications. Use the same methods as you use for CGI to invoke a URL, form, or link with ICAPI or GWAPI. Any macro that executes successfully using CGI will execute successfully using ICAPI or and GWAPI. No modifications need to be made to these macros.

To configure ISAPI:

1. Stop the Web server.
2. Copy the DLL for ISAPI that comes with Net.Data into the server's subdirectory. For example:

   `/inetsrv/scripts/dtwisapi.`*`filetype`*

   Where *filetype* is `.dll` for Window NT and OS/2 and `.o` for UNIX operating systems.

   See the Net.Data README file for your operating system for specific file and directory names.
3. Because ISAPI bypasses CGI processing, you do not need to have the `cgi-bin/db2www/` part of the URL in forms and links. Instead, use `dtwisapi.`*`filetype`*. For example, if the following URL invokes Net.Data as the CGI program:

   `http://server1.stl.ibm.com/cgi-bin/db2www/test1.d2w/report`

   Then you should invoke Net.Data as the ISAPI plug-in with the following URL:

   `http://server1.stl.ibm.com/scripts/dtwisapi.dll/test1.d2w/report`
4. If you stored your macro `test1.d2w` in the subdirectory `/order/` under one of the directories specified in MACRO_PATH or current directory of the Web server, invoke Net.Data in CGI mode using the following URL:

   `http://server1.stl.ibm.com/cgi-bin/db2www/orders/test1.d2w/report`

   Then the equivalent URL to invoke Net.Data in ISAPI mode is:

   `http://server1.stl.ibm.com/scripts/dtwisapi.dll/orders/test1.d2w/report`
5. Restart the Web server.

To configure NSAPI:

1. Stop the Web server.
2. Copy the DLL for NSAPI that comes with Net.Data into the server directory. For example:

   `/netscape/server/bin/httpd/dtwnsapi.`*`filetype`*

   Where *filetype* is `.dll` for Window NT and OS/2 and `.o` for UNIX operating systems.

   See the Net.Data README file for your operating system for specific file and directory names.
3. Modify your server configuration file with the changes listed below. See the Net.Data README file or program directory for your operating system for operating system differences.

obj.conf — Add to the top of the file:

```
Init fn="load-modules" shlib="<path>dtwnsapi.dll" funcs=dtw_nsapi
```

obj.conf — Add to the Services directive:

```
Service fn="dtw_nsapi" method=(GET|HEAD|POST)
  type="magnus-internal/d2w"
```

mime.types — Add this type, where *d2w* is the default extension of the macro. You can specify any three-character combination.

```
type=magnum-internal/d2w exts=d2w
```

4. Move the Net.Data macro files from the `netdata/macro` directory to the server's root document directory:

   `/netscape/server/docs/`
5. Add the server's root document directory to the MACRO_PATH statement, in the initialization file. This change tells Net.Data where to look for the macro files.
6. Because NSAPI bypasses CGI processing, you do not need to have the `cgi-bin/db2www/` part of the URL in forms and links. The server knows files with a `d2w` file type are Net.Data macros because you defined it when you changed the Netscape configuration files. For example, the following URL invokes Net.Data as the CGI program:

   `http://server1.stl.ibm.com/cgi-bin/db2www/test1.d2w/report`

   While the following URL invokes Net.Data as the NSAPI plug-in:

   `http://server1.stl.ibm.com/test1.d2w/report`
7. Restart the Web server.

If you keep your Net.Data macros in several directories, the last three steps change:

1. Move the directories with the Net.Data macros they contain to the server's root document directory.
2. Update the MACRO_PATH variable in the initialization file to include all of the directories and subdirectories where you macro files are located.
3. Modify the links and forms that point to these Net.Data macros, keeping their directory names. For example, when running in CGI mode, the following URL calls a Net.Data macro that is stored in the `/orders/` directory:

   `http://server1.stl.ibm.com/cgi-bin/db2www/orders/test1.d2w/report`

   The updated URL used to invoke Net.Data in NSAPI mode is shorter, but keeps the directory name:

   `http://server1.stl.ibm.com/orders/test1.d2w/report`

Configuring Net.Data with the Net.Data Administration Tool

The Net.Data administration tool helps you to configure and manage the Net.Data initialization file (DB2WWW.INI) and the configuration file for Live Connection (dtwcm.cnf) on the Windows NT, AIX, and OS/2 operating systems. Using this tool, you can complete the following tasks:

- "Starting the Administration Tool" on page 41
- "Configuring Path Statements" on page 41
- "Configuring Ports" on page 43
- "Configuring Cliettes" on page 43
- "Configuring Language Environments" on page 47
- "Defining Configuration Variables" on page 50

See "Before You Begin" to learn about setting up the administration tool and ensuring your have the correct software prerequisites.

Before You Begin

1. Plan the configuration of Net.Data language environments, databases, cliettes, ports, and configuration variables.
2. Install Net.Data from CD-ROM.

3. Install the Java run-time libraries (JDK 1.1 and subsequent versions for each operating systems). Check the Net.Data README file for your operating system for more information.

   Make sure you have `classes.zip` in your CLASSPATH after installing JDK.
4. If you have installed the IBM JDBC driver that is packaged with DB2 Universal Database, add the driver directory to your Java CLASSPATH statement to enable the DB2 login test.
5. Change to the directory where the Net.Data administration tool program is stored:

   For OS/2 and Windows NT:

   The *inst_dir*\connect*admin_directory*, where *inst_dir* is the directory you specified for Net.Data during installation and *admin_directory* is the directory where the administration tool files exist.

   For AIX:

   The /usr/lpp/internet/db2www/db2.v2/*admin directory*, where *admin_directory* is the directory where the administration tool files exist

Starting the Administration Tool

The operating system that you use determines how you start the administration tool.

For OS/2 and Windows NT:

From the IBM Net.Data Version 2 folder, select the **Net.Data Admin Tool** icon.

For AIX:

Change to the Net.Data installation directory (inst_dir). From the command line, enter `ndadmin` to start the tool.

The administration tool is launched and the Net.Data Administration notebook is displayed.

Configuring Path Statements

Use the **Path** page to add, modify, or delete the path statements for locating the files the Net.Data needs to process Net.Data macros. These statements are described in “Path Configuration Statements” on page 17. Figure 7 on page 42 shows the **Path** page.

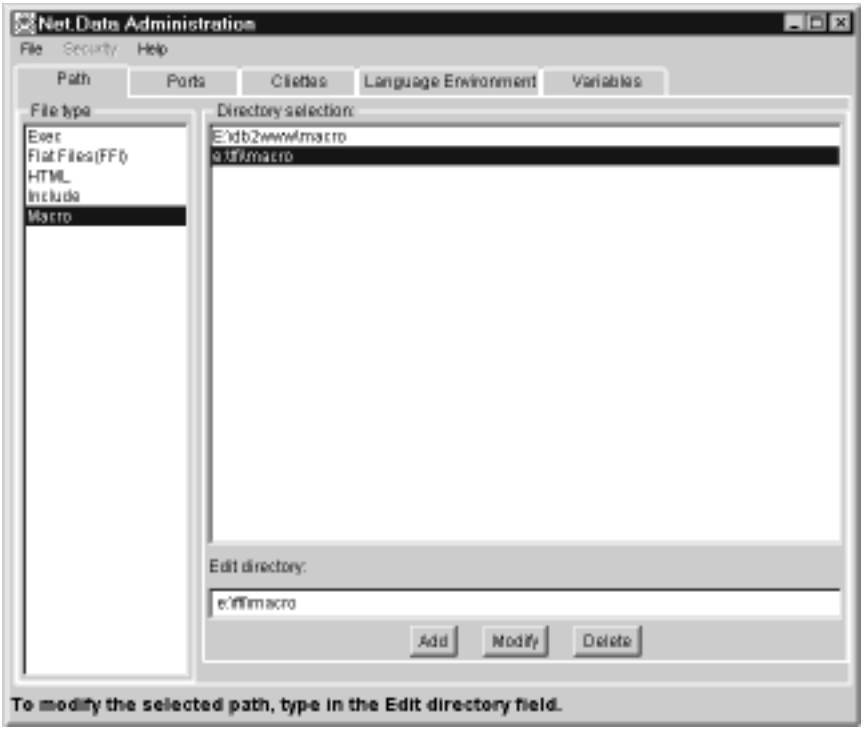


Figure 7. The Path Page of the Net.Data Administration Tool. Use this page to add, modify or delete path statements.

Configuration tip: The HTML file type can have one path, only.

To add a path statement:

1. Start the administration tool.
2. From the **Path** page, select a file type from the **File type**, for example, select `Exec`.
3. In the **Edit directory** field, type the new path and click on the **Add** button.

   If the path you specified does not exist, a warning window opens. If no directory is selected, the new directory is added as the last item in the list.
4. Close the administration tool, or click on another tab to complete additional configuration tasks.

To modify a path statement:

1. Start the administration tool.
2. From the **Path** page, select the file type you want to change from the **File type** list.
3. Select the path you want to modify in the **Directory selection** list. The selected path opens in the **Edit directory** field.
4. Modify the path in the **Edit directory** field and click on the **Modify** button. If the path you entered does not exist, a warning window opens.
5. Close the administration tool, or click on another tab to complete additional configuration tasks.

To delete a path statement:

1. Start the administration tool.

2. From the **Path** page, select the file type that you want to delete from the **File type** list.
3. In the **Directory selection** field, select the path you want to delete. The selected path opens in the **Edit directory** field.
4. Click on the **Delete** button.
5. Close the administration tool, or click on another tab to complete additional configuration tasks.

Configuring Ports

Use the **Port** page to specify the TCP/IP port numbers used by Net.Data. Figure 8 shows the **Port** page.

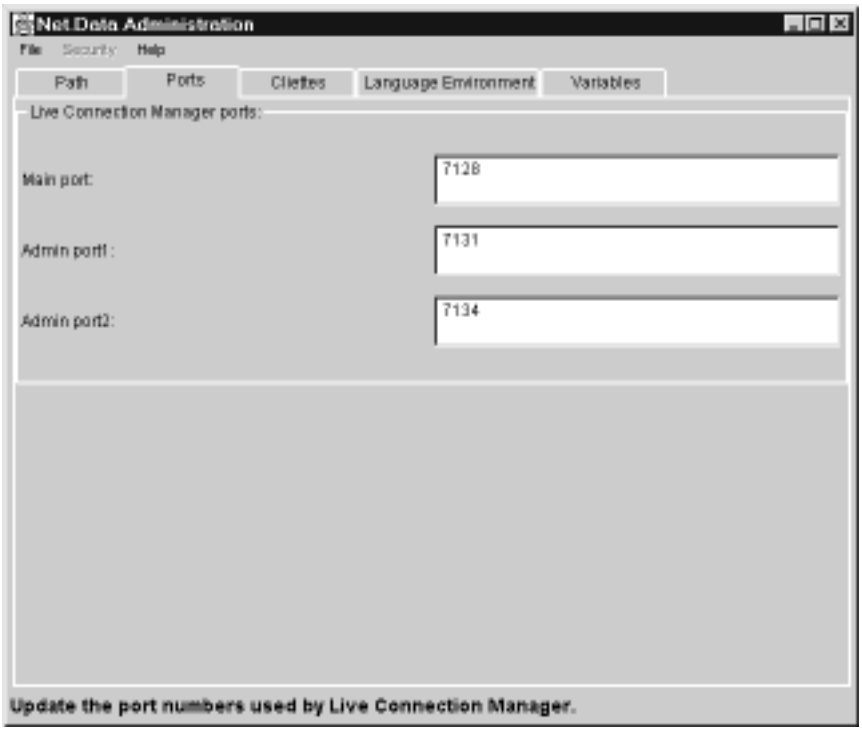


Figure 8. The Port Page of the Net.Data Administration Tool. Use this page to specify ports.

To specify TCP/IP port numbers:

1. Start the administration tool.
2. From the **Port** page, type a unique port number in each of the port fields. The administration tool verifies the port number you type in each field when you tab to the next field.
3. Close the administration tool, or click on another tab to complete additional configuration tasks.

Configuring Cliettes

Use the **Cliette** page to add, modify, or delete Live Connection database cliettes, and you can also manage database and administrator user IDs and passwords for database cliettes. More information about cliettes is provided in “Managing Connections” on page 158. Figure 9 on page 44 shows the **Cliette** page.

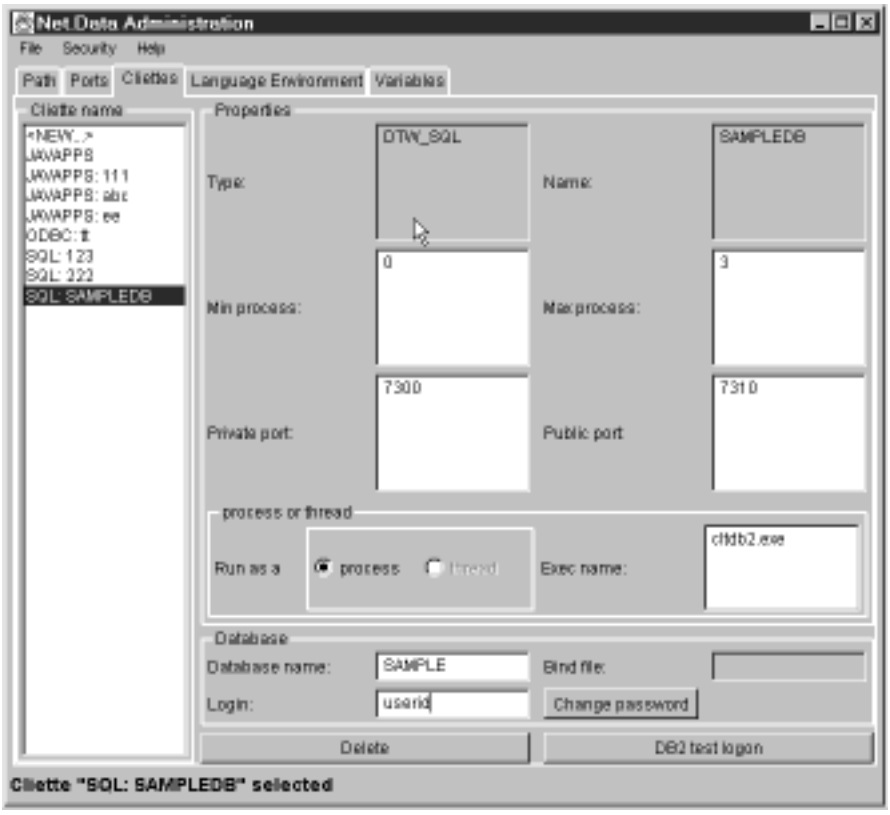


Figure 9. The Cliette Page of the Net.Data Administration Tool. Use this page to add, modify, and delete cliettes.

To add a cliette:

1. Start the administration tool.
2. From the **Cliette** page, select **<NEW...>** from the **Cliette name** list. The **Add a cliette** window opens.

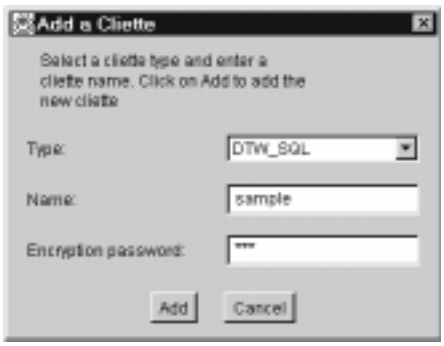


Figure 10. The Add a Cliette Window of the Net.Data Administration Tool. Use this page to add cliettes.

   If you have enabled encryption, you are prompted for the encryption password the first time you create or modify a cliette. This password is saved and you will not ever have to enter it again.
3. Select a cliette type from the **Type** list.
4. Type a name for the new cliette in the **Name** field. The name can be the name of the database or another unique cliette name. For example: MYCLIETTE.

5. Type the encryption password if the **Encryption password** field is enabled. You will not need to type the password again, as the administration tool saves the password for you.
6. Click on the **Add** button.

   The new cliette is created and is added to the bottom of the cliette list. Additionally, the new name is highlighted and the default properties for the cliette are displayed in the **Properties** group box. You can change these values to fit your configuration.
7. Close the administration tool, or click on another tab to complete additional configuration tasks.

To modify a cliette:

1. Start the administration tool.
2. From the **Cliette** page, select the name of the cliette that you want to change from the **Cliette name** list. The properties of the cliette are displayed in the **Properties** group box.
3. Modify the properties from the **Properties** group box, as needed.
   a. The **Type** field displays the type of cliette that is being defined and corresponds to a language environment type name. Net.Data populates this field when you add a new cliette, and the choices are defined in the **Cliette type** list in the Add a Cliette window.
   b. The **Name** field displays the name of the cliette, which is usually the name of the database. Net.Data populates this field when you add a new cliette.
   c. Type the number of cliette process that can be started when Connection Manager is started in the **Min process** field. You need one unique port address for each process. See “Configuring Live Connection” on page 29 for more information about MIN Process values.
   d. In the **Max process** field, type the number of cliette processes that can be run at the same time, in addition to the processes started when Connection Manager is started. You need one unique port address for each process. See “Configuring Live Connection” on page 29 for more information about MAX Process values.
   e. Type a unique port number in the **Private port** field to specify the starting port number for use with the cliette processes that are started with the Connection Manager. An additional port number is used for each of the processes specified by the **Min Process** value. For example, if you specify the port number 7012 for **Private port** and the value 5 for **Min process**, port numbers 7012-7016 are used and must not conflict with other port assignments in the system.
   f. Type a unique port number in the **Public port** field to specify the starting port number used with the cliette processes that are started when additional processes are started, up to the number specified in the **Max process** field. An additional port number is used for each of the processes For example, if you specify the port number 7020 for **Public port** and the value 5 for **Max process**, port numbers 7020-7024 are used and must not conflict with other port assignments in the system.
   g. The **Exec name** field displays the name of the cliette executable file.
4. If the cliette is being used with a database, modify the values for the **Database** group box, as needed:
   a. Specify the database name of the database with which the cliette is associated **Database name** field, for example, MYDBASE.

b. The **Bind file** field contains the name and path of the bind file for the type of cliette that you are using.

c. The **Login** field specifies the login user ID used to connect to the database.

d. The **Change password** push button opens the Change Database Password window. Type the encryption password and the new password, twice. You can encrypt the database password by using the encryption functions specified in the **Security** pull-down menu.

5. Select **File** and then **Save** to save your changes.
6. Close the administration tool, or click on another tab to complete additional configuration tasks.

To test the DB2 database logon and connection:

1. From the **Cliette** page of the administration tool, click on the **DB2 test logon** push button. When the test is complete a confirmation window opens, displaying the status of the connection test.
2. Close the window to continue configuring or close the administration tool.

To delete a cliette:

1. Start the administration tool.
2. From the **Cliette** page, select the name of the cliette that you want to delete from the **Cliette name** list.
3. Click on the **Delete** button.
4. Close the administration tool, or click on another tab to complete additional configuration tasks.

To turn on encryption of cliette user IDs and passwords:

Encryption provides security for database connections with cliettes. When encryption is turned on, all database passwords in the Live Connection configuration file are encrypted and require an encryption password for access and decryption.

Requirement: You must use a Net.Data Version 2 Live Connection configuration file to use encryption.

1. **Important:** Back up a copy of your Live Connection configuration file, `<path>dtwcm.cnf`. You need this file if you lose the encryption password, or want to decrypt database passwords and need to restore the passwords.
2. From the **Cliette** page of the administration tool, select the **Security -> Turn encryption on** pull-down menu option. The Turn Encryption On confirmation window opens.
3. Click on **Yes** to continue. The Encryption Password window opens.
4. Type the password twice for authorization to work with cliettes that have encrypted passwords.
5. Click on **OK** to define the new password and encrypt all of the database passwords for your cliettes.

To turn off encryption of cliette user IDs and passwords:

1. From the **Cliette** page of the administration tool, select the **Security -> Turn encryption off** pull-down menu option. The Turn Encryption Off confirmation window opens.

2. Click on **Yes** to continue. All passwords are set to *USE_DEFAULT for security reasons. You can restore your passwords from the backup copy of the Live Connection file, <path>dtwcm.cnf.

To change the password for encryption:

1. From the **Cliette** page of the administration tool, select the **Security -> Change Encryption Password** pull-down menu option. The Change Encryption Password confirmation window opens.
2. Click on **Yes** to continue. The Change Encryption Password window opens.
3. Type the old encryption password once, and the new password twice.
4. Click on **OK** to change the encryption password.

To change the database password:

1. From the **Cliette** page of the administration tool, click on the **Change Password** push button. The Change Database Password window opens.
2. Type the encryption password once and the new database password twice.
3. Click on **OK** to change the password and close the window. The changed database password is encrypted if you have turned encryption on.

Configuring Language Environments

Use the **Language Environment** page to add, modify, or delete Net.Data language environments. Language environments are discussed in “Environment Configuration Statements” on page 21. Figure 11 shows the **Language Environment** page.

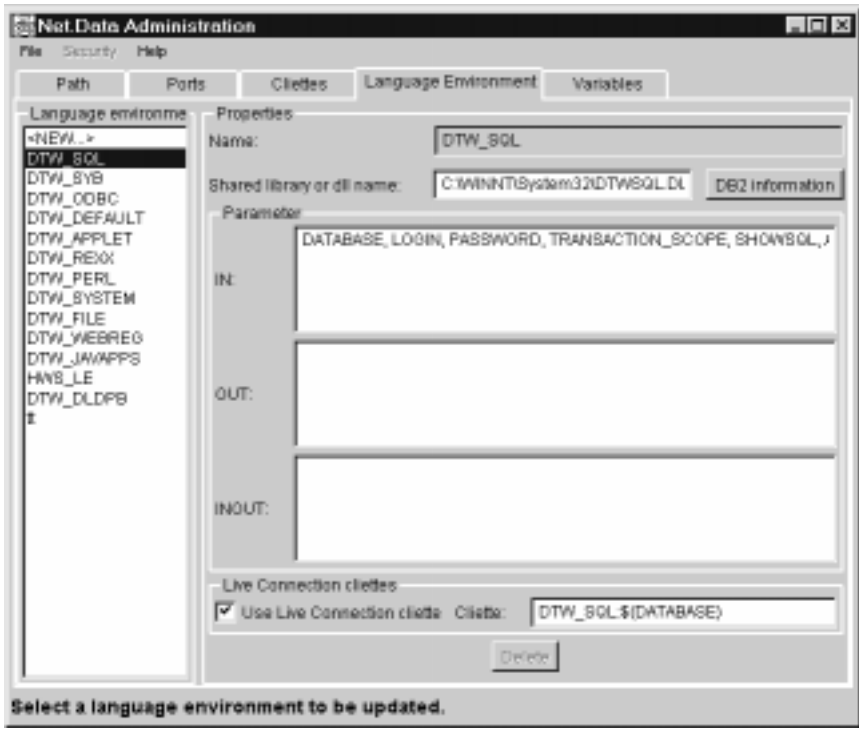


Figure 11. The Language Environment Page of the Net.Data Administration Tool. Use this page to specify language environments.

To add a language environment:

1. Start the administration tool.
2. From the **Language Environment** page, select **<NEW...>** from the **Language environment** list. The **Add a new language environment** window opens.
3. Type a name of the language environment in the field and click on the **Add** button. The Add a Language Environment window opens.

Figure 12. The Add a Language Environment window of the Net.Data Administration Tool. Use this page to specify a new language environment.

   The new language environment is created and its name is added to the bottom of the language environment list. Additionally, the new name is highlighted, and the default properties for the language environment are displayed in the **Properties** group box. You can change these values to fit your configuration.
4. Close the administration tool, or click on another tab to complete additional configuration tasks.

To modify a language environment:

1. Start the administration tool.
2. From the **Language Environment** page, select the name of the language environment that you want to change from the **Language environment** list. The properties of the cliette are displayed in the **Properties** group box.
3. Modify the properties in the **Properties** group box, shown in Figure 12 as needed:
   a. Specify the name of the language environment in the **Name** field; this name corresponds to the language environment type used to define a cliette. To change this value, double click on a different name from the **Language environment** list. See “Environment Configuration Statements” on page 21 for more information about language environment types.
   b. Specify the shared library or DLL program name and path for the language environment in the **Shared library or dll name** field.
   c. Select the **DB2 information** push button to display the DB2 Information window as shown in Figure 13 on page 49.

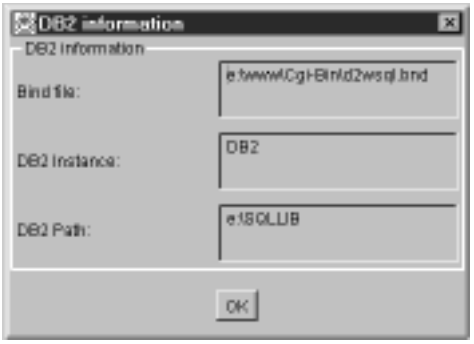


Figure 13. The DB2 Information window of the Net.Data Administration Tool. Use this page to specify information specifically for DB2 databases.

Specify the values for the DB2 environment variables:

1) Type the path and file name of the bind file in the **Bind file** field.
2) Specify the DB2INSTANCE value for the database associated when you use the SQL language environment in the **DB2 Instance** field.
3) Specify the path directory name for the DB2 product executable files, usually \SQLLIB, in the **DB2 Path** field.
4) Click on **OK** to save your changes and close the window.

d. Specify the input and output parameters that are passed to or from a language environment each time the language environment is called in the **Parameters** group box.

   Tip: Do not update these fields unless you are defining your own language environment.

e. Specify whether to use cliettes and which cliette should be associated with the language environment in the **Live Connection cliettes** group box.

   1) Specify whether the cliette for the language environment is active by checking the **Use Live Connection cliette** check box. Select this check box if you want to use the cliette specified in the **Cliette** field when calling the language environment.
   2) Specify the name of the cliette that is to be run with the language environment being defined in the **Cliette** field. The syntax of the name depends on whether you are configuring a database or the Java Application language environment. The default is `DTW_SQL:$(DATABASE)`.

      Syntax for Databases:

      type`:`*name*

      Where:

      type The language environment type for the cliette. It can be one of the following values:

      For Windows NT:
      DTW_ODBC, DTW_ORA, DTW_SYB, DTW_SQL, DTW_JAVAPPS

      For OS/2:
      DTW_SQL, DTW_JAVAPPS

      For AIX:
      DTW_ODBC, DTW_ORA, DTW_SYB, DTW_SQL, DTW_JAVAPPS

name The name of the cliette as defined on the **Cliette** page. The default is $(DATABASE).

Syntax for Java Applications:

```
DTW_JAVAPPS
```

4. Select **File** and then **Save** to save your changes
5. Close the administration tool, or click on another tab to complete additional configuration tasks.

To delete a language environment:

Restriction: You can delete only the language environments created by users, not the default language environments that come with Net.Data.

1. Start the administration tool.
2. From the **Language Environment** page, select the name of the language environment that you want to delete from the **Language environment** list.
3. Click on the **Delete** button.
4. Close the administration tool, or click on another tab to complete additional configuration tasks.

Defining Configuration Variables

Use the **Variables** page to specify the home directory for Net.Data and to select the level of error messages logging. Figure 14 shows the **Variables** page.

Figure 14. The Variables Page the Net.Data Administration Tool. Use this page to specify initialization variables.

To specify the home directory for Net.Data:

This variable is also known as the installation directory variable.

1. Start the administration tool.
2. From the **Variables** page, type the path for the directory where the log file is to be stored in the **Installation directory** field. The default is `\inst_dir\logs\`. For example: `e:\db2www`.
3. Close the administration tool, or click on another tab to complete additional configuration tasks.

To specify the error message logging level for Net.Data:

1. Start the administration tool.
2. From the **Variables** page, select a level of error logging from the **Error logging** group box:
   - **off**
   - **errors only**
   - **both warnings and errors**
3. Close the administration tool, or click on another tab to complete additional configuration tasks.

Granting Access Rights to Files Accessed by Net.Data

Before using Net.Data, you need to ensure that the user IDs under which Net.Data executes have the appropriate access rights to files that are referenced in a Net.Data macro and to the macro that a URL references. This means that these files must be in directories or libraries to which the Web server can connect, or to which these user IDs have explicit access rights.

More specifically, ensure that the user IDs under which Net.Data executes have the following authorizations:

- To read the Net.Data initialization file, `db2www.ini`
- To execute the Net.Data executable files and DLLs, and to search the directories in the paths to the executable files and DLLs
- To read the appropriate Net.Data macro files and search the appropriate directories identified by the MACRO_PATH path configuration statement
- To execute the appropriate files and to search the appropriate directories identified by the EXEC_PATH path configuration statement
- To read the appropriate files and to search the appropriate directories identified by the INCLUDE_PATH path configuration statement
- To read and write the appropriate files, and to search the appropriate directories identified by the FFI_PATH path configuration statement
- To read the Live Connection configuration file, dtwcm.cnf
- To read the Cache Manager configuration file, CACHEMGR.CNF
- To read external Perl and REXX executable files referenced by the language environments

The methods for granting access to these files depend on the operating system on which Net.Data is running.

Chapter 3. Keeping Your Assets Secure

Internet security is provided through a combination of firewall technology, operating systems features, Web server features, Net.Data mechanisms, and the access control mechanisms that are part of your data sources.

You must decide on what level of security is appropriate for your assets. This chapter describes methods you can use for keeping your assets secure and also provides references to additional resources you can use to plan for the security of your Web site.

The following sections contain guidelines for protecting your assets. The security mechanisms described include:

- "Using Firewalls"
- "Encrypting Your Data on the Network" on page 55
- "Using Authentication" on page 55
- "Using Authorization" on page 56
- "Using Net.Data Mechanisms" on page 56

Additionally, Net.Data provides database cliette password encryption; see "Configuring Cliettes" on page 43 for more information.

Using Firewalls

Firewalls are collections of hardware, software, and policies that are designed to limit access to resources in a networked environment.

Firewalls:

- Protect the internal network from infiltration or intrusion
- Protect the internal network from data and programs that are brought in by internal users
- Limit internal user access to external data
- Limit the damage that can be done if the firewall is breached

Net.Data can be used with firewall products that execute in your environment.

The following possible configurations provide recommendations for managing the security of your Net.Data application. These configurations provide high-level information and assume that you have configured a firewall that isolates your secure intranet from the public Internet. Carefully consider these configurations with your organization's security policies:

- **High security configuration**

  This configuration creates a subnetwork that isolates Net.Data and the Web server from both the secure intranet and the public Internet. The firewall software is used to create a firewall between the Web server and the public Internet, and another firewall between the Web server and the secured intranet, which contains DB2 Server. This configuration is shown by Figure 15 on page 54.

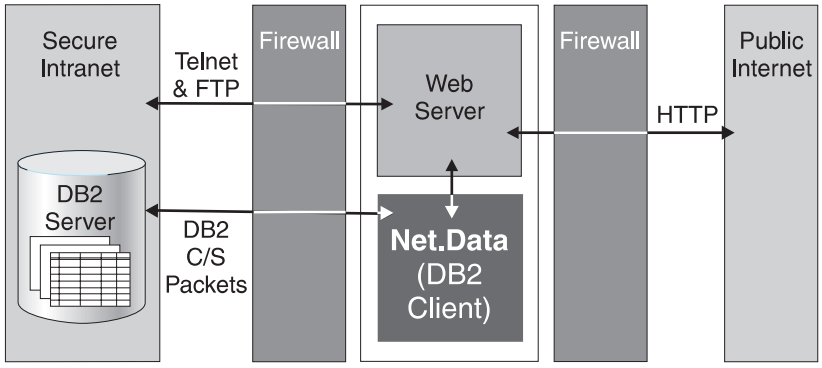


Figure 15. High Security Configuration

To set up this configuration:

– Install Net.Data on the Web server machine and ensure that Net.Data can access DB2 Server inside the intranet by:
  - Installing Client Application Enabler (CAE) on the Web server machine.
  - Configuring the firewall to allow DB2 traffic through the firewall. One method is to add a packet filtering rule to allow DB2 client requests from Net.Data and acknowledge packets from DB2 Server to Net.Data.

– Allow FTP and Telnet access between the Web server and the secure intranet. One method is to install a socks server on the Web server machine.

– In the packet filtering configuration file of the firewall software, specify that incoming TCP packets from the standard HTTP port can access the Web server. Also, specify that outgoing TCP acknowledge packets can go to any hosts on the public Internet from the Web server.

- **Intermediate security configuration**

  In this configuration, firewall software isolates the secured intranet with DB2 server from the public Internet. Net.Data and the Web server are outside the firewall on a workstation platform. This configuration is simpler than the first, but still offers database protection. Figure 16 shows this configuration.

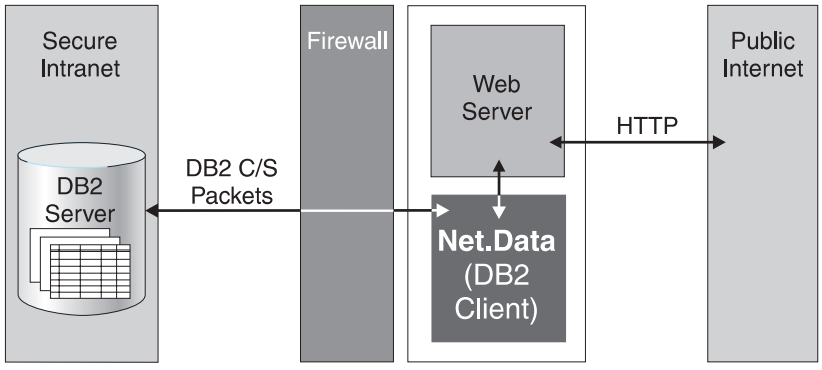


Figure 16. Intermediate Security Configuration:

You must install CAE on the Web server to allow Net.Data to communicate with DB2 server.The firewall must be configured to allow DB2 client requests to flow from Net.Data to DB2 and to allow acknowledge packets to flow from DB2 to Net.Data.

- **Low security configuration**

In this configuration, DB2 server and Net.Data are installed outside of the firewall and the secured intranet. They are not protected from external attacks. The firewall needs no packet filtering rules for this configuration. Figure 17 shows this configuration.

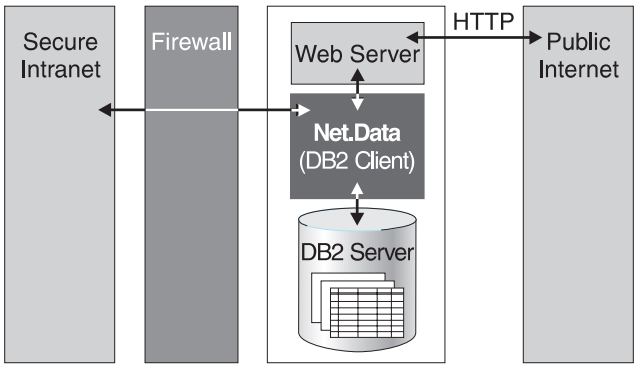


Figure 17. Low Security Configuration:

Encrypting Your Data on the Network

You can encrypt all data that is sent between a client system and your Web server when you use a Web server that supports Secured Sockets Layer (SSL). This security measure supports the encryption of login IDs, passwords, and all data that is transmitted through HTML forms from the client system to the Web server and all data that is sent from the Web server to the client system. Most Web servers support SSL, such as Internet Connection Secure Server, Version 2 Release 2 or higher and Lotus Domino Go Webserver, 4.6.1 or higher.

Using Authentication

Authentication is used to ensure that a user ID making a Net.Data request is authorized to access and update data within the application. Authentication is the process of matching the user ID with a password to validate that the request comes from a valid user ID. The Web server associates a user ID with each Net.Data request that it processes. The process or thread that is handling the request can then access any resource to which that user ID is authorized.

You can use two types of authentication: one protecting certain directories on your server and one protecting your database.

- Most Web servers allow you to specify directories on the server to protect. You can also have your system require a user ID and password for people accessing files in directories you specify. See the *Administrator's Guide* for your Web server to determine your system's capabilities.
- DB2 has an authentication system for database access that can restrict access to tables and columns to certain users. You can use Net.Data's special variables, such as LOGIN and PASSWORD, to link to the DB2 authentication routine.

Tip: To protect Net.Data macros do the following:

1. Add protection directives in the Web server configuration file for the Net.Data program object.

2. Ensure the user ID that Net.Data will be running under has access rights to the macro files. For more information on granting access rights, see “Granting Access Rights to Files Accessed by Net.Data” on page 51.

Using Authorization

Authorization provides a user with complete or restricted access to an object, resource, or function. Data sources such as DB2 provide their own authorization mechanisms to protect the information that they manage. These authorization mechanisms assume that the user ID associated with the process that is executing the Net.Data request has been properly authenticated, as explained in “Using Authentication” on page 55. The existing access control mechanisms for these data sources then either permit or deny access based on the authorizations that are held by the authenticated user ID.

Using Net.Data Mechanisms

In addition to the methods described above, you can use Net.Data configuration variables or macro development techniques to limit the activities of end users, to conceal corporate assets such as the design of your database, and to validate user-provided input values within production environments.

Net.Data Configuration Variables

Net.Data provides several configuration variables that can be used to limit the activities of end users or conceal the design of your database.

Control file access with path statements

Net.Data evaluates the settings of path configuration statements to determine the location of files and executable programs that are used by Net.Data macros. These path statements identify one or more directories that Net.Data searches when attempting to locate macro files, executable files, include files, or other flat files. By selectively including directories on these path statements, you can explicitly control the files that are accessible by users at browsers. Refer to “Chapter 2. Configuring Net.Data” on page 5 for additional detail about path statements.

You should also use authorization checking as described in “Using Authorization” and verify that file names cannot be changed in INCLUDE statements as described in “Macro Development Techniques” on page 57.

Disable SHOWSQL for production systems

The SHOWSQL variable allows the user to specify that Net.Data displays the SQL statements specified within Net.Data functions at a Web browser. This variable is used primarily for developing and testing the SQL within an application and is not intended for use in production systems.

You can disable the display of SQL statements in production environments using one of the following methods:

- When using Net.Data Version 2.0.7 or higher, use the DTW_SHOWSQL configuration variable in the Net.Data initialization file to override the effect of setting SHOWSQL within your Net.Data macros. See “DTW_SHOWSQL: Enable or Disable SHOWSQL Configuration Variable” on page 15 for syntax and additional information.

- Users of Net.Data Version 2.0.5 and earlier can use the DTW_ASSIGN() function as described in “Macro Development Techniques”.

See SHOWSQL in the variables chapter of *Net.Data Reference* for syntax and examples for the SHOWSQL Net.Data variable.

Consider whether it is appropriate to enable direct request for production environments

The direct request method of invoking Net.Data allows a user to specify the execution of an SQL statement or Perl, REXX, or C program directly from a URL. The macro request method allows users to execute only those SQL statements and functions defined or called in a macro.

You should carefully consider whether to allow the use of direct request because it might give your users the ability to execute a very broad set of functions. When enabling this method of invocation, ensure that user ID under which the Net.Data request is processed has the appropriate level of authorization.

You can use the DTW_DIRECT_REQUEST configuration variable to disable direct request. See “DTW_DIRECT_REQUEST: Enable Direct Request Variable” on page 13 for syntax and additional information.

Macro Development Techniques

Net.Data provides several mechanisms that allow users to assign values to input variables. To ensure that macros execute in the manner intended, these input variables should be validated by the macro. Your database and application should also be designed to limit a user’s access to the data that the user is authorized to see.

Use the following development techniques when writing your Net.Data macros. These techniques will help you ensure that your applications execute as intended and that access to data is limited to properly authorized users.

Ensure that Net.Data variables cannot be overridden in a URL

The setting of Net.Data variables by a user within a URL overrides the effect of DEFINE statements used to initialize variables in a macro. This might alter the manner in which your macro executes. To safeguard against this possibility, initialize your Net.Data variables using the DTW_ASSIGN() function.

Example: Instead of using `%DEFINE SHOWSQL="NO"` to set the Net.Data SHOWSQL variable, use `@DTW_ASSIGN(SHOWSQL, "NO")`. Then, a query string assignment such as `SHOWSQL=YES` does not override the macro setting.

You can disable the display of SQL statements in production environments using one of the following methods:

- When using versions of Net.Data that support the DTW_SHOWSQL configuration variable, use this variable in the Net.Data initialization file to override the effect of setting SHOWSQL within your Net.Data macros. See “DTW_SHOWSQL: Enable or Disable SHOWSQL Configuration Variable” on page 15 for syntax and additional information.
- Use the DTW_ASSIGN() function as described in the above example, to assign the value of SHOWSQL to prevent it from being overridden.

See SHOWSQL in the variables chapter of *Net.Data Reference* for syntax and examples for the SHOWSQL Net.Data variable.

You can also use DTW_ASSIGN to ensure that other Net.Data variables, such as RPT_MAX_ROWS or START_ROW_NUM, are not overridden. See the variables chapter of *Net.Data Reference* for more information about these variables.

Validate that your SQL statements cannot be modified in ways that alter the intended behavior of your application

Adding a Net.Data variable to an SQL statement within a macro allows users to dynamically alter the SQL statement before executing it. It is the responsibility of the macro writer to validate user-provided input values and ensure that an SQL statement containing a variable reference is not being modified in an unexpected manner. Your Net.Data application should validate user-provided input values from the URL so the Net.Data application can reject invalid input. Your validation design process should include for the following steps:

1. Identify the syntax of valid input; for example, a customer ID must start with a letter and can contain only alphanumeric characters.
2. Determine what potential harm can be caused by allowing incorrect input, intentionally harmful input, or input entered to gain access to internal assets of the Net.Data application.
3. Include input verification statements in the macro that meet the needs of the application. Such verification depends on the syntax of the input and how it is used. In simpler cases it can be enough to check for invalid content in the input or to invoke Net.Data to verify the type of the input. If the syntax of the input is more complex, the macro developer might have to parse the input partially or completely to verify whether it is valid.

Example 1: Using the DTW_POS() string function to verify SQL statements

```
%FUNCTION(DTW_SQL) query1() {
   select * from shopper where shlogid = '$(shlogid)'
%}
```

The value of the `shlogid` variable is intended to be a shopper ID. Its purpose is to limit the rows returned by the SELECT statement to rows that contain information about the shopper identified by the shopper ID. However, if the string "`smith' or shlogid<>'smith`" is passed as the value of the variable `shlogid`, the query becomes:

```
select * from shopper where shlogid = 'smith' or shlogid<>'smith'
```

This user-modified version of the original SQL SELECT statement returns the entire shopper table.

The Net.Data string functions can be used to verify that the SQL statement is not modified by the user in inappropriate ways. For example, the following logic can be used to ensure that the input value associated with the `shlogid` variable consists of a single shopper ID:

```
@DTW_POS(" ", $(shlogid), result)
%IF (result == "0")
  @query1()
%ELSE
  %{ perform some sort of error processing %}
%ENDIF
```

Example 2: Using DTW_TRANSLATE()

Suppose that your application needs to validate that the value provided in the input variable `number_of_orders` is an integer. One way of accomplishing this is to create a translation table `input_translation_table` that contains all keyboard characters except the numeric characters 0-9 and to use the DTW_TRANSLATE and DTW_POS string functions to validate the input:

```
@DTW_TRANSLATE(number_of_orders, "x", input_translation_table, "x", string_out)

 @DTW_POS("x", string_out, result)

     %IF (result = "0")

       %{ continue with normal processing %}

     %ELSE

       %{ perform some sort of error processing %}

     %ENDIF
```

Note that SQL statements within stored procedures cannot be modified by users at Web browsers and that user-provided input parameter values are constrained by the SQL data types associated with the input parameters. In situations where it is impractical to validate user input values using the Net.Data string functions, you can use stored procedures.

Ensure that a file name in an INCLUDE statement is not modified in ways that alter the intended behavior of your application

If you specify the value for the file name with an INCLUDE statement using a Net.Data variable, then the file to be included is not determined until the INCLUDE file is executed. If your intent is to set the value of this variable within your macro, but to not allow a user at the browser to override the macro-provided value, then you should set the value of the variable using DTW_ASSIGN instead of DEFINE. If you do intend to permit the user at a browser to provide a value for the file name, then your macro should validate the value provided.

Example: A query string assignment such as `filename="../../x"` can result in the inclusion of a file from a directory not normally specified in the INCLUDE_PATH configuration statement. Suppose that your Net.Data initialization file contains the following path configuration statement:

```
INCLUDE_PATH /usr/lpp/netdata/include
```

and that your Net.Data macro contains the following INCLUDE statement:

```
%INCLUDE "$(filename)"
```

A query string assignment of `filename="../../x"` would include the file `/usr/lpp/x` , which was not intended by the INCLUDE_PATH configuration statement specification.

The Net.Data string functions can be used to verify that the file name provided is appropriate for the application. For example, the following logic can be used to ensure that the input value associated with the file name variable does not contain the string ″..″:

```
@DTW_POS("..", $(filename), result)
%IF (result > "0")
  %{ perform some sort of error processing %}
%ELSE
%{ continue with normal processing %}
%ENDIF
```

Design your database and queries so that user requests do not have access to sensitive data about other users

Some database designs collect sensitive user data in a single table. Unless SQL SELECT requests are qualified in some fashion, this approach may make all of the sensitive data available to any user at a web browser.

Example: The following SQL statement returns order information for an order identified by the variable `order_rn`:

```
select setsstatcode, setsfailtype, mestname
from   merchant, setstatus
where  merfnbr   = setsmenbr
and    setsornbr = $(order_rn)
```

This method permits users at a browser to specify random order numbers and possibly obtain sensitive information about the orders of other customers. One way to safeguard against this type of exposure is to make the following changes:

- Add a column to the order information table that identifies the customer associated with the order information within a specific row.
- Modify the SQL SELECT statement to ensure that the SELECT is qualified by an authenticated customer ID provided by the user at the browser.

For example, if `shlogid` is the column containing the customer ID associated with the order, and SESSION_ID is a Net.Data variable that contains the authenticated ID of the user at the browser, then you can replace the previous SELECT statement with the following statement:

```
select setsstatcode, setsfailtype, mestname
  from   merchant, setstatus
  where  merfnbr   = setsmenbr
  and    setsornbr = $(order_rn)
  and    shlogid   = $(SESSION_ID)
```

Use Net.Data hidden variables

You can use Net.Data hidden variables to conceal various characteristics of your Net.Data macro from users that view your HTML source with their Web browser. For example, you can hide the internal structure of your database. See “Hidden Variables” on page 93 for more information about hidden variables.

Request validation information from a user

You can create your own protection scheme based on user-provided input. For example, you can request validation information from a user through an HTML form and validate it using data that your Net.Data macro retrieves from a database or by calling an external program from a function defined in your Net.Data macro.

For more information on protecting your assets, see the Internet security list of frequently asked questions (FAQ) at this Web site:

`http://www.w3.org/Security/Faq`

Chapter 4. Invoking Net.Data

This chapter describes how you invoke Net.Data using the various Web server interfaces. Before you can use one of the methods of invocation, Net.Data must first be configured for the specified interface. You can configure Net.Data to use the following Web server interfaces:

- Common Gateway Interface (CGI)
- FastCGI
- Lotus Domino Go Web server (GWAPI)
- Internet Connection Server (ICAPI)
- Netscape Server (NSAPI)
- Microsoft Internet Server (ISAPI)
- Java Servlets

See “Chapter 2. Configuring Net.Data” on page 5 to learn more about configuring Net.Data for these interfaces. By default, the Web server invokes Net.Data as a CGI program, with each Net.Data request running in a new and separate process. You determine how Net.Data is invoked when you configure the Web server.

The following sections describe the types of requests Net.Data accepts and the methods you can use to invoke Net.Data using the various APIs and Servlets.

- “Types of Invocation Requests”
- “Invoking Net.Data through the Web Server APIs” on page 70
- “Invoking Net.Data with Java Servlets and JavaBeans” on page 72

Types of Invocation Requests

Regardless of the method with which you invoke Net.Data, there are two types of requests that can be specified, depending on whether you want to execute a macro, or whether you want to execute a single SQL statement, stored procedure, or function.

Macro Request
Specifies that Net.Data execute the macro specified.

Direct Request
Specifies that Net.Data execute an SQL statement, stored procedure, or function. The request specifies:

- The name of a language environment
- An SQL statement or the name of a function, along with any parameter values that are required for the invocation of the function
- Form data that is required for invocation of the SQL statement or function

Web developers who want to write a single SQL query or call a single function such as a DB2 stored procedure, REXX program, or Perl function can issue a direct request to the database. A direct request does not have any complex Net.Data application logic that requires a Net.Data macro, and therefore bypasses the Net.Data macro processor. Direct request parameters are passed to the appropriate language environment for processing for improved performance.

Figure 18 on page 62 illustrates the differences between a macro request and a direct request. A macro request always specifies a macro within the URL for the

request and can also use form data. A direct request never specifies a macro within the URL, but can still use form data.

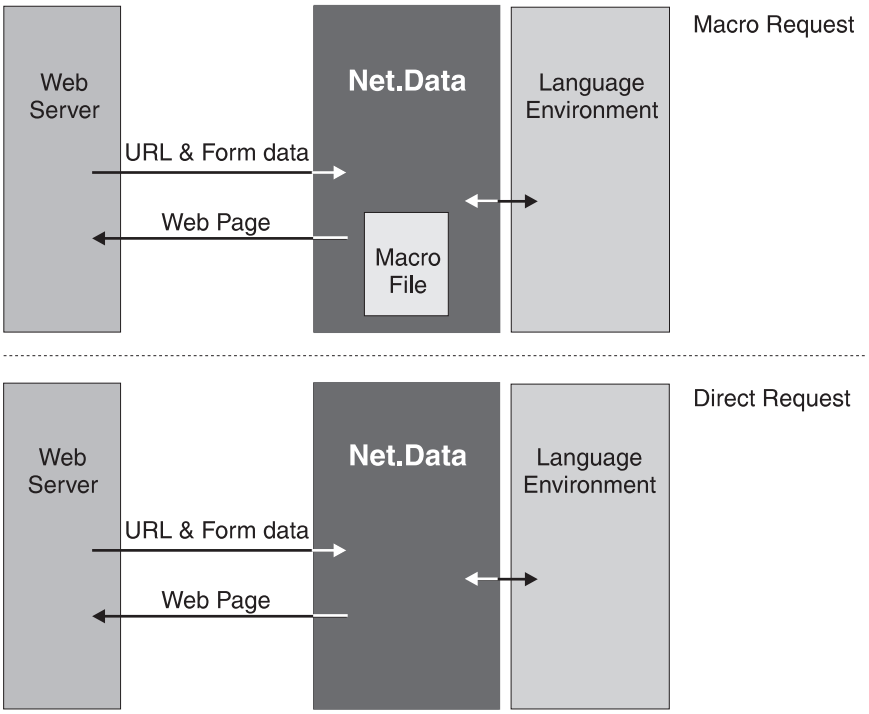


Figure 18. Macro Request Versus Direct Request

The syntax for invoking Net.Data depends on how Net.Data is configured and the type of request that you make. For both macro and direct requests, Net.Data is invoked using a URL. The URL can be entered directly by the user, or it can be coded into the HTML page as an HTML link or an HTML form. The Web server invokes Net.Data using CGI, FastCGI, or one of the Web server APIs. Additionally, you can invoke Net.Data using Net.Data servlets.

For macro requests, specify within the URL the name of the Net.Data macro and the name of the HTML block that is to be executed within the Net.Data macro. For direct requests, specify within the URL the name of the Net.Data language environment, the SQL statement or the name of the function, and any additional required parameter values. You specify these values using a syntax defined by Net.Data.

The following sections describe these invocation requests in more detail:

- “Invoking Net.Data with a Macro (Macro Request)” on page 63
- “Invoking Net.Data without a Macro (Direct Request)” on page 65

Although the examples specify the syntax to use when invoking Net.Data using CGI, the concepts apply to all interfaces that are used to invoke Net.Data. For the exact syntax required for each type of interface, refer to the section specific to each.

- “Invoking Net.Data through the Web Server APIs” on page 70
- “Invoking Net.Data with Java Servlets and JavaBeans” on page 72

Invoking Net.Data with a Macro (Macro Request)

This section shows you how to invoke Net.Data by specifying a macro.

The following syntax statements show how to invoke Net.Data.

- URL:

  `http://server/Net.Data_invocation_path/filename/block[?name=val&...]`

Parameters:

server Specifies the name and pathof the Web server. If the server is the local server, you can omit the server name and use a relative URL.

Net.Data_invocation_path
The path and filename of the Net.Data executable file, servlet class, DLL, or shared library. For example, `/cgi-bin/db2www/`.

filename
Specifies the name of the Net.Data macro file. Net.Data searches for and tries to match this file name with the path statements defined in the MACRO_PATH initialization path variable. See "MACRO_PATH" on page 18 for more information.

block Specifies the name of the HTML block in the referenced Net.Data macro.

method
Specifies the HTML method used with the form.

?name=val&...
Specifies one or more optional parameters passed to Net.Data.

You can then specify the URL directly in your browser, or you can use it in an HTML link or form as follows:

- HTML link:

  `<A HREF="URL">any text</A>`

- HTML form:

  `<FORM METHOD=method ACTION="URL">any text</FORM>`

Parameters:

method
Specifies the HTML method used with the form.

URL Specifies the URL used to run the Net.Data macro, the parameters of which are described above.

Examples

The following examples demonstrate the different methods of invoking Net.Data.

Example 1: Invoking Net.Data using an HTML link:

```
<A HREF="http://server/cgi-bin/db2www/myMacro.d2w/report">
.
.
.
</A>
```

Example 2: Invoking Net.Data using a form

```
<FORM METHOD=POST
 ACTION="http://server/cgi-bin/db2www/myMacro.d2w/report">
.
.
.
</FORM>
```

The following sections describe HTML links and forms and more about how to invoke Net.Data with them:

- "HTML Links"
- "HTML Forms"

HTML Links

If you are authoring a Web page, you can create an HTML link that results in the execution of an HTML block. When a user at a browser clicks on a text or image that is defined as an HTML link, Net.Data executes the HTML block within the macro.

To create an HTML link, use the HTML <a> tag. Decide which text or graphic you want to use as your hyperlink to the Net.Data macro, then surround it by the <a> and tags. In the HREF attribute of the <a> tag, specify the macro and the HTML block.

The following example shows a link that results in the execution of an SQL query when a user selects the text ″List all monitors″ on a Web page.

```
<a href="http://server/cgi-bin/db2www/listA.d2w/report">
List all monitors</a>
```

Clicking on the link calls a macro named listA.d2w, which has an HTML block named ″report″, as in the following example:

```
%DEFINE DATABASE="MNS97"


%FUNCTION(DTW_SQL) myQuery(){
SELECT MODNO, COST, DESCRIP FROM EQPTABLE
WHERE TYPE='MONITOR'
%}

%HTML(report){
@myQuery()
%}
```

The query returns a table that contains model number, cost, and description information for each monitor that is described within the EQPTABLE table. This example displays the results of the query by generating a default report. See "Report Blocks" on page 107 for information on how you can customize your reports using a REPORT block.

HTML Forms

You can dynamically customize the execution of your Net.Data macros using HTML forms. Forms allow users to provide input values that can affect the execution of the macro and the contents of the Web page that Net.Data builds.

The following example builds on the monitor list example in “HTML Links” on page 64 by letting users at a browser use a simple HTML form to select the type of product for which information will be displayed.

```
<H1>Hardware Query Form</H1>
<HR>
<FORM METHOD=POST ACTION="/cgi-bin/db2www/equiplst.d2w/report">
<P>What type of hardware do you want to see?
<MENU>
<LI><INPUT TYPE="RADIO" NAME="hdware" VALUE="MON" checked> Monitors
<LI><INPUT TYPE="RADIO" NAME="hdware" VALUE="PNT"> Pointing devices
<LI><INPUT TYPE="RADIO" NAME="hdware" VALUE="PRT"> Printers
<LI><INPUT TYPE="RADIO" NAME="hdware" VALUE="SCN"> Scanners
</MENU>

<INPUT TYPE="SUBMIT" VALUE="Submit">
</FORM>
```

After the user at the browser makes a selection and clicks on the Submit button, the Web server processes the ACTION parameter of the FORM tag, which invokes Net.Data. Net.Data then executes the HTML report block in the `equiplst.d2w` macro:

```
%DEFINE DATABASE="MNS97"

 %FUNCTION(DTW_SQL) myQuery(){
SELECT MODNO, COST, DESCRIP FROM EQPTABLE
WHERE TYPE='$(hdware)'
%REPORT{
<H3>Here is the list you requested</H3>
%ROW{
<HR>
$(N1): $(V1), $(N2): $(V2)
<P>$(N3): $(V3)
%}
%}
%}

%HTML(report){
@myQuery()
%}
```

In the above example, the value of `TYPE=$(hdware)` in the SQL statement is taken from the HTML form input.

See *Net.Data Reference* for a detailed description of the variables that are used in the ROW block.

Invoking Net.Data without a Macro (Direct Request)

This section shows you how to invoke Net.Data using *direct request*. When you use direct request, you do not specify the name of a macro in the URL. Instead, you specify the Net.Data language environment, the SQL statement or a program to be executed, and any additional required parameter values within the URL, using a syntax defined by Net.Data. See “DTW_DIRECT_REQUEST: Enable Direct Request Variable” on page 13 to learn how to enable and disable direct request.

The SQL statement or program and any other specified parameters are passed directly to the designated language environment for processing. Direct request improves performance because Net.Data does not need to read and process a macro. The SQL, ODBC, Oracle, Sybase, Java, System, Perl, and REXX

Net.Data-supplied language environments support direct request, and you can call Net.Data using a URL, an HTML form, or a link.

A direct request invokes Net.Data by passing parameters in the query string of the URL or the form data. The following example illustrates the context in which you specify a direct request.

```
<A HREF="http://server/cgi-bin/db2www/?direct_request">any text</A>
```

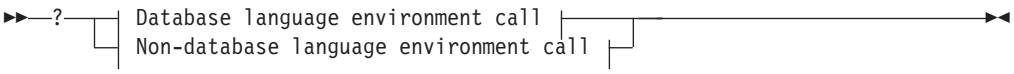
Where *direct_request* represents the direct request syntax. For example, the following HTML link contains the direct request:

```
<A HREF="http://server/cgi-bin/db2www/?LANGENV=DTW_PERL&FUNC=my_perl(hi)">
  any text</A>
```

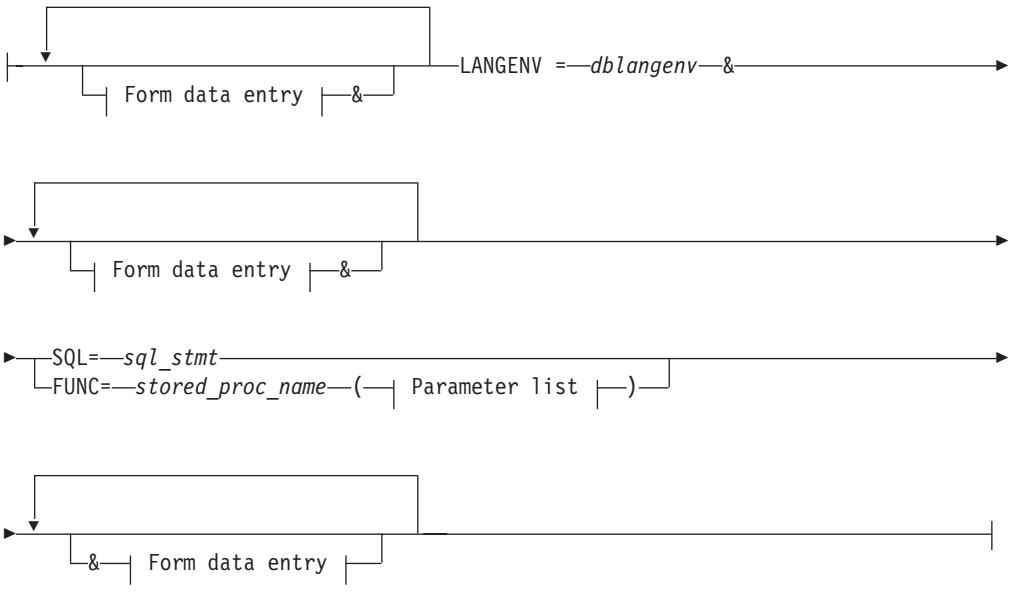
Direct Request Syntax

The syntax for invoking Net.Data with direct request can contain a call to either a database or a non-database language environment.

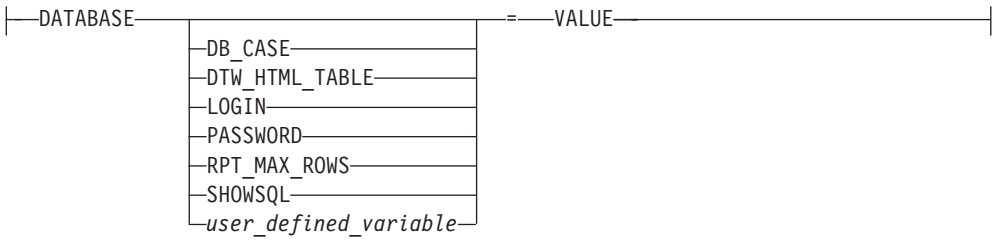
Syntax



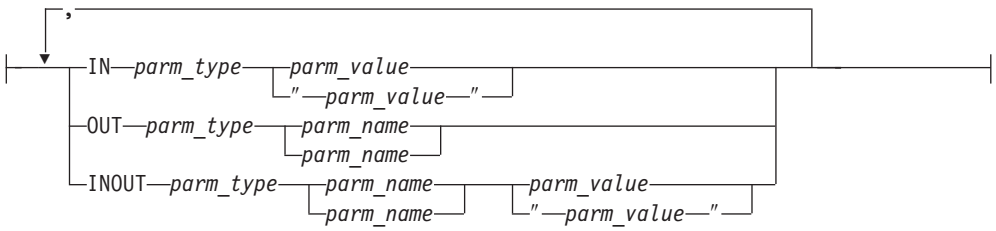
Database language environment call:



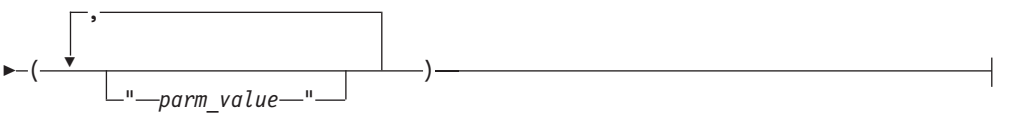
Form data entry:



Parameter list:



Non-database language environment call:



Parameters

Database language environment call
Specifies a direct request to Net.Data that invokes a database language environment.

Form data entry
Parameters that allow you to specify the settings of SQL variables or to request simple HTML formatting. See the variables chapter of *Net.Data Reference* to learn more about these variables.

DATABASE
Specifies the database to which Net.Data should pass the SQL request. This parameter is required.

DB_CASE
Specifies the case (upper or lower) for SQL statements.

DTW_HTML_TABLE
Specifies whether Net.Data should return an HTML table or a pre-formatted text table.

LOGIN
Specifies the database user ID.

PASSWORD
Specifies the database password.

RPT_MAX_ROWS
Specifies the maximum number of rows within a table that a function will return in a report.

SHOWSQL
Specifies whether Net.Data should hide or display the SQL statement being executed.

START_ROW_NUM
Specifies the row number in a table for a function to use as the start of its report.

user_defined_variable
Variables that are passed to Net.Data and provide required information or effect Net.Data behavior. User-defined variables are variables that you define for your application.

VALUE
Specifies the value of the Net.Data variable.

LANGENV
Specifies the target language environment for the SQL statement or stored procedure call. If the language environment is one of the database language environments, the database name must also be specified.

dblangenv
The name of the database language environment:

- DTW_SQL
- DTW_ODBC
- DTW_ORA
- DTW_SYB

SQL
Indicates that the direct request specifies the execution of an in-line SQL statement.

sql_stmt
Specifies a string that contains any valid SQL statement that can be executed using dynamic SQL.

FUNC
Indicates that the direct request specifies the execution of a stored procedure.

stored_proc_name
Specifies any valid DB2 stored procedure name.

parm_type
Specifies any valid parameter type for a DB2 stored procedure.

parm_name
Specifies any valid parameter name.

parm_value
Specifies any valid parameter value for a DB2 stored procedure.

IN Specifies that Net.Data should use the parameter to pass input data to the stored procedure.

INOUT
Specifies that Net.Data should use the parameter to both pass input data to the stored procedure and return output data from the language environment.

OUT
Specifies that the language environment should use the parameter to return output data from the stored procedure.

Non-database language environment call
Specifies a direct request to Net.Data that invokes a non-database language environment.

LANGENV
Specifies the target language environment for the execution of the function.

lang_env
Specifies the name of the non-database language environment:

- DTW_PERL
- DTW_REXX
- DTW_SYSTEM

FUNC
Indicates that the direct request specifies the execution of a program.

program_name
Specifies the program containing the function to be executed.

parm_value
Specifies any valid parameter value for the function.

Direct Request Examples

The following examples show the different ways you can invoke Net.Data while using the direct request method.

HTML Links: The following examples use direct request to invoke Net.Data through links.

Example 1: A link that invokes the Perl language environment and calls a Perl script that is in the EXEC path statement of the Net.Data initialization file

```
<A HREF="http://server/cgi-bin/db2www/?LANGENV=DTW_PERL&FUNC=my_perl(hi)">
  any text</A>
```

Example 2: A link that invokes the Perl language environment, as in the previous example, but passes a string with URL-encoded values for the double quote and the space characters

```
<A HREF="http://server/cgi-bin/db2www/?LANGENV=DTW_PERL&FUNC=my_perl
     (%22Hello+World%22)">any text</A>
```

Tip: You must encode certain characters, such as spaces and double quotes, within URLs. In this example, the double quotes characters and spaces within the parameter value must be encoded as %22 or the + character, respectively. You can use the built-in function DTW_URLESCSEQ to encode any text that must be encoded witin a URL. For more information on the DTW_URLESCSEQ function, see its description in *Net.Data Reference*.

HTML Forms: The following examples use direct request to invoke Net.Data through forms.

Example 1: An HTML form that results in the execution of an SQL query using the SQL language environment, connects to the CELDIAL database, and queries a table

```
<FORM METHOD="POST"
 ACTION="http://server/cgi-bin/db2www/">
<INPUT TYPE=hidden NAME="LANGENV" VALUE="DTW_SQL">
<INPUT TYPE=hidden NAME="DATABASE" VALUE="CELDIAL"
 <INPUT TYPE=hidden NAME="SQL" VALUE="select * from Table1 where col1=$(InputName)">
Enter Customer name:
<INPUT TYPE=text NAME="InputName" VALUE="John">
<INPUT TYPE=SUBMIT>
</FORM>
```

This example contains a variable substitution in the SQL statement to make the WHERE clause dynamic.

URL: The following examples use direct request to invoke Net.Data through URLs.

Example 1: A URL that results in the execution of an SQL query using the SQL language environment

```
http://server/cgi-bin/db2www/?LANGENV=DTW_SQL&DATABASE=CELDIAL
 &SQL=select+*+from+customer
```

Example 2: A URL that invokes the Perl language environment and calls an executable file that is not in the EXEC path statement of the Net.Data initialization file

```
http://server/cgi-bin/db2www/?LANGENV=DTW_PERL&FUNC=/u/MYDIR/macros/myexec.pl
```

Example 3: A URL that invokes the System language environment and calls an external Perl script

```
http://server/cgi-bin/db2www/?LANGENV=DTW_SYSTEM&FUNC=perl+/u/MYDIR/macros/myexec.pl
```

Example 4: A URL that invokes the REXX language environment, calls a REXX program, and passes parameters to the program

```
http://server/cgi-bin/db2www/?LANGENV=DTW_REXX&FUNC=myexec.cmd(parm1,parm2)
```

Example 5: A URL that calls a stored procedure and passes parameters to the SQL language environment

```
http://server/cgi-bin/db2www/?LANGENV=DTW_SQL&FUNC=MY_STORED_PROC
   (IN+CHAR(30)+Salaries)&DATABASE=CELDIAL
```

Invoking Net.Data through the Web Server APIs

Net.Data supports the Web APIs in the following list, depending on your operating system:

GWAPI plug-in and ICAPI plug-in
: The Lotus Domino Go Webserver API plug-in as the follow-on to the IBM Internet Connection Secure Sever plug-in

ISAPI plug-in
: Microsoft Internet Server API plug-in

NSAPI plug-in
: Netscape Server API plug-in

See the operating system reference appendix in *Net.Data Reference* to determine which Web server APIs are supported for your operating system. See “Configuring

Net.Data for Use with the Web Server APIs" on page 38 to learn how to configure Net.Data and the Web server for use with APIs.

Requirements:

- If running Net.Data in GWAPI, ICAPI, ISAPI or NSAPI mode, restart your Web server so that the Web server can reload Net.Data and run it as a process.
- If you make changes to the initialization file after the Web server invokes Net.Data in API mode, you must restart the Web server. Any changes to the Net.Data initialization file (`db2www.ini`) have no effect. In API mode, Net.Data reads the initialization file only once to reduce the performance overhead.
- When running in API mode, the Oracle and Sybase language environments require Live Connection.

To invoke the Web server APIs:

For ICAPI and GWAPI:

Syntax:

`http://`*`server_name`*`/CGI-BIN/db2www/`*`macro_name`*`/`*`html_block`*

Parameters:

server_name
The name of the server.

macro_name
The relative path name of your macro under the directory specified by MACRO_PATH.

html_block
The name of the HTML block in the macro to be processed.

Example:

```
http://myserver/CGI-BIN/db2www/mymacro.d2w/report
```

For ISAPI:

Syntax:

`http://`*`server_name/server_HTML_root_directory/dll_name/macro_name/`*
`html_block`

Parameters:

server_name
The name of the server.

server_HTML_root_directory
The Web server HTML root directory name.

dll_name
Net.Data's ISAPI .dll file name, `dtwisapi.dll`.

macro_name
The relative path name of your macro under the directory specified by MACRO_PATH.

html_block
The name of the HTML block in the macro to be processed.

Example:

```
http://myserver/scripts/dtwisapi.dll/mymacro.d2w/report
```

For NSAPI:

Syntax:

```
http://server_name/macro_name/html_block
```

Parameters:

server_name
The name of the server.

macro_name
The relative path name of your macro under the directory specified by MACRO_PATH. The extension of the macro file, for example, `.d2w`, must be defined in the Web server configuration file. See "Configuring Net.Data for Use with the Web Server APIs" on page 38 for more information.

html_block
The name of the HTML block in the macro to be processed.

Example:

```
http://myserver/mymacro.d2w/report
```

Invoking Net.Data with Java Servlets and JavaBeans

Servlets are Java classes that perform a role similar to that of CGI programs or Web server API plug-ins. A Web server can use servlets to dynamically build HTML pages. Servlets do not have their own graphical user interface, but their classes can be dynamically loaded locally or from across the network. They can be called using a URL address (remotely) or by a class name (locally).

Net.Data provides servlets that you can use to invoke Net.Data macros, run Net.Data functions, or execute SQL statements through Net.Data within any Java-enabled operating system supported by Net.Data.

This chapter describes the concepts and tasks for:

Net.Data Servlets

Net.Data provides servlets and NetObjects Fusion plug-ins with Net.Data functions that can be used in a Java environment. With servlets and plug-ins you can:

- Run Net.Data macros from both a URL and as a Server-Side-Include (SSI)
- Run Net.Data functions from both a URL and as an SSI
- Use NetObjects Fusion (NOF) to manage your macros, providing better integration with your Web site management and an easy-to-use graphical user interface to develop simple macros. See "Appendix E. Using NetObjects Fusion NOF Plug-ins with Net.Data Servlets" on page 201 to learn how to use NOF plug-ins.

This section describes the following servlet topics:

About Net.Data Servlets

Net.Data provides servlets to help you develop and manage macros using the Java environment. Servlets are Java classes that perform a role similar to that of CGI programs or Web server API plug-ins. Servlets are used by a Java servlet-enabled Web server to build HTML pages. Servlets do not have their own graphical user interface, but their classes can be dynamically loaded locally, or from across the network, and can be called using a URL address (remotely) or by a class name (locally). Servlets are available for Windows NT and AIX operating systems.

The Net.Data servlets are a Java-based wrapper that run a Net.Data Version 2 macro or direct request using a native DLL file. These servlets allow you to run an existing macro (MacroServlet) or a single function (FunctionServlet). Net.Data Version 2 is required to use these servlets. Net.Data servlets can run the database or non-database language environments and can run with Live Connection.

The servlets come with an API for use with your applications. The servlet APIs are documented with Net.Data. See the <*inst_dir*>/servlets/NetDataServlets.jar file for API documentation.

Net.Data provides two servlets:

Macro Servlet (com.ibm.netdata.servlets.MacroServlet)
Executes an existing Net.Data macro.

Using the macro servlet is similar to running a Net.Data macro through the CGI-BIN interface, except you run the macro through a Java servlet. The macro servlet requires that Net.Data Version 2 or higher be installed.

The advantages of using the macro servlet include:

- SQL queries are run through ODBC using Live Connection Manager for increased performance.
- You can run macros through Server-Side-Includes (SSI) to embed multiple macros in your HTML file.

The macro servlet also allows native access to heterogeneous databases, such as DB2, Oracle, and Sybase, as well as various language environments, such as Perl, REXX, and Java.

Function Servlet (com.ibm.netdata.servlets.FunctionServlet)
Executes the Net.Data function or SQL statement through a servlet interface, such as %FUNCTION DTW_SQL(). See “Invoking Net.Data without a Macro (Direct Request)” on page 65 for more information.

The function servlet requires that Net.Data Version 2 be installed.

Running Net.Data Servlets

The Net.Data servlets can be run either from a URL or as an SSI within an HTML file. You can use the NetObjects Fusion plug-ins to incorporate the Net.Data servlets into your NOF site. The following sections discuss how to modify and run the servlets by typing in the syntax for the servlet. See “Appendix E. Using NetObjects Fusion NOF Plug-ins with Net.Data Servlets” on page 201 to learn how to modify and run the servlets with NetObjects Fusion.

- “Running the Macro Servlet” on page 74
- “Running the Function Servlet” on page 75

Running the Macro Servlet: From within an HTML file, enter the servlet parameters using one of the following syntax options:

1. URL:

```
http://myserver/servlet/com.ibm.netdata.servlets.MacroServlet
  ?MACRO=macro_value&BLOCK=block_value&parmnn=valuenn
```

   For example:

```
http://myserver/servlet/com.ibm.netdata.servlets.MacroServlet?MACRO=my_macro
  &BLOCK=my_block&field1=custno
```

2. SSI:

```
<servlet code="com.ibm.netdata.servlets.MacroServlet">
     <param name="MACRO" value="macro_value">
     <param name="BLOCK" value="block_value">
     <param name="parmnn" value="valuenn">
 </servlet>
```

   For example:

```
<servlet code="com.ibm.netdata.servlets.MacroServlet">
     <param name="MACRO" value="my_macro.d2w">
     <param name="BLOCK" value="report">
     <param name="field1" value="custno">
 </servlet>
```

Parameters:

macro_value
: The fully qualified path to an existing Net.Data macro

block_value
: The name of the HTML block in the specified Net.Data macro to execute; the default is `report` (optional).

parmnn
: Any additional parameters that your macro requires, such as

```
<param name="field1" ...
```

valuenn
: Any additional values that your macro requires, such as

```
... value="custnum"
```

HTMLPATH parameter: If you get an error message referring to a missing HTMLPATH parameter, add the HTMLPATH parameter to your servlet invocation command:

- URL:

```
http://myserver/servlet/com.ibm.netdata.servlets.MacroServlet
?MACRO=macro_name&BLOCK=block_value&htmlpath=html_path&parmnn=valuenn
```

  For example:

```
http://myserver/servlet/com.ibm.netdata.servlets.MacroServlet?MACRO=my_macro
  &BLOCK=my_blockhtmlpath=e:\html&field1=custno
```

- SSI:

```
<servlet code="com.ibm.netdata.servlets.MacroServlet">
     <param name="MACRO" value="macro_value">
      <param name="BLOCK" value="block_value">
      <param name="htmlpath" value="html_path">
     <param name="parmnn" value="valuenn">
 </servlet>
```

For example:

```
<servlet code="com.ibm.netdata.servlets.MacroServlet">
<param name="MACRO" value="my_macro">
<param name="BLOCK" value="my_block">
<param name="htmlpath" value="e:\html">
<param name="field1" value="custno">
</servlet>
```

OUTBUFLEN parameter: If your macro results are larger than 32 KB, specify the OUTBUFLEN parameter. Failure to specify these parameters when required can result in unpredictable results.

- URL:

```
http://myserver/servlet/com.ibm.netdata.servlets.MacroServlet
?MACRO=macro_name&BLOCK=block_value
&OUTBUFLEN=output_buffer_size&parmnn=valuenn
```

For example:

```
http://myserver/servlet/com.ibm.netdata.servlets.MacroServlet?MACRO=my_macro
  &BLOCK=my_block&OUTBUFLEN=48&field1=custno
```

- SSI:

```
<servlet code="com.ibm.netdata.servlets.MacroServlet">
     <param name="MACRO" value="macro_value">
      <param name="BLOCK" value="block_value">
<param name="OUTBUFLEN" value="output_buffer_size">
<param name="parmnn" value="valuenn">
 </servlet>
```

For example:

```
<servlet code="com.ibm.netdata.servlets.MacroServlet">
<param name="MACRO" value="my_macro">
<param name="BLOCK" value="my_block">
<param name="OUTBUFLEN" value="48">
<param name="field1" value="custno">
</servlet>
```

Running the Function Servlet: The function servlet can invoke Net.Data using direct request to execute either a function (such as a REXX function) or an SQL statement. The parameters you specify for the servlet depend on whether you are executing a function or an SQL statement. From within an HTML file, enter the servlet parameters using one of the following syntax options:

1. For a URL:
   - ***To invoke a function:***

     ```
     http://myserver/servlet/com.ibm.netdata.servlets.FunctionServlet
       ?LANGENV=lang_env_name&FUNC=program_name&parmnn=valuenn
     ```

     For example:

     ```
     http://myserver/servlet/com.ibm.netdata.servlets.FunctionServlet
       ?LANGENV=DTW_REXX&FUNC=my_rexx&field1=custno
     ```

   - ***To invoke an SQL statement:***

     ```
     http://myserver/servlet/com.ibm.netdata.servlets.FunctionServlet
        ?LANGENV=database_lang_env_name&SQL=SQL_statement
        &DATABASE=database_name&parmnn=valuenn
     ```

     For example:

     ```
     http://myserver/servlet/com.ibm.netdata.servlets.FunctionServlet
     ?LANGENV=DTW_SQL&SQL=select+*+from+myTable&DATABASE=CELDIAL
     ```

2. SSI:

- ***To invoke a function:***

```
<servlet code="com.ibm.netdata.servlets.FunctionServlet">
     <param name="LANGENV" value="lang_env_name">
      <param name="FUNC" value="program_name">
      <param name="parmnn" value="valuenn">
 </servlet>
```

  For example:

```
<servlet code="com.ibm.netdata.servlets.FunctionServlet">
<param name="LANGENV" value="DTW_REXX">
<param name="FUNC" value="myREXX">
<param name="field1" value="custno">
</servlet>
```

- ***To invoke an SQL statement:***

```
<servlet code="com.ibm.netdata.servlets.FunctionServlet">
     <param name="LANGENV" value="lang_env_name">
      <param name="SQL" value="SQL_stmt_name">
      <param name="DATABASE" value="database_name">
     <param name="parmnn" value="valuenn">
 </servlet>
```

  For example:

```
<servlet code="com.ibm.netdata.servlets.FunctionServlet">
<param name="LANGENV" value="DTW_SQL">
<param name="SQL" value="select * from employee">
<param name="DATABASE" value="CELDIAL">
</servlet>
```

Parameters:

lang_env_name
: The Net.Data language environment (such as DTW_SQL, DTW_REXX) being called to process the function, SQL statement, or stored procedure. This parameter requires that FUNC or SQL be used.

program_name
: The name of the program which contains the function to be executed. For example, `my_rexx`, where `my_rexx` is the name of a REXX executable. Specify input parameters for the function with the *parmnn* and *valuenn* parameters.

database_name
: The name of the database associated with the DATABASE parameter. The specified

SQL_stmt_name
: An SQL statement or stored procedure name that accesses a database. For example, `"select * from employee"`. Specify input parameters for the SQL statement or stored procedure with the *parmnn* and *valuenn* parameters.

parmnn
: Any additional parameters that your macro requires, such as `<param name="field1" ...`.

valuenn
: Any additional values that your macro requires, such as `... value="custnum"`.

HTMLPATH parameter: If you get an error message referring to a missing HTMLPATH parameter, add the HTMLPATH parameter to your servlet invocation command:

- URL:

  ```
  http://myserver/servlet/com.ibm.netdata.servlets.FunctionServlet
  ?LANGENV=lang_env_name&FUNC=program_name&htmlpath=html_path
  &parmnn=valuenn
  ```

  For example:

  ```
  http://myserver/servlet/com.ibm.netdata.servlets.FunctionServlet
  ?LANGENV=DTW_REXX&FUNC=my_rexx&htmlpath=e:\html&field1=custno
  ```

- SSI:

  ```
  <servlet code="com.ibm.netdata.servlets.FunctionServlet">
       <param name="LANGENV" value="lang_env_name">
        <param name="SQL" value="SQL_stmt_name">
        <param name="htmlpath" value="html_path">
       <param name="parmnn" value="valuenn">
   </servlet>
  ```

  For example:

  ```
  <servlet code="com.ibm.netdata.servlets.FunctionServlet">
  <param name="LANGENV" value="DTW_SQL">
  <param name="SQL" value="select * from employee">
  <param name="htmlpath" value="e:\html">
  <param name="field1" value="custno">
  <param name="DATABASE" value="SAMPLE">
  </servlet>
  ```

Where *html_path* specifies the path to the Web server root HTML directory; for example: `htmlpath=e:\html`.

OUTBUFLEN parameters: If your macro results are larger than 32 KB, you must specify the OUTBUFLEN parameter. Failure to specify these parameters when required can result in unpredictable results.

- URL:

  ```
  http://myserver/servlet/com.ibm.netdata.servlets.FunctionServlet
  ?LANGENV=lang_env_name&FUNC=program_name&OUTBUFLEN=output_buffer_size&parmnn=valuenn
  ```

  For example:

  ```
  http://myserver/servlet/com.ibm.netdata.servlets.FunctionServlet?LANGENV=DTW_REXX
    &FUNC=my_rexx&OUTBUFLEN=48K&field1=custno
  ```

- SSI:

  ```
  <servlet code="com.ibm.netdata.servlets.FunctionServlet">
       <param name="LANGENV" value="lang_env_name">
        <param name="FUNC" value="program_name">
  <param name="OUTBUFLEN" value="output_buffer_size">
  <param name="parmnn" value="valuenn">
   </servlet>
  ```

  For example:

  ```
  <servlet code="com.ibm.netdata.servlets.FunctionServlet">
  <param name="LANGENV" value="DTW_REXX">
  <param name="FUNC" value="my_rexx">
  <param name="OUTBUFLEN" value="48K">
  <param name="field1" value="custno">
  </servlet>
  ```

Net.Data JavaBeans

Net.Data provides JavaBeans that can be used in a Java environment without having a Web server running. A JavaBean is an object-oriented programming

interface that lets you build reusable applications or program building blocks. These objects can be used in a network on Java-enabled operating system.

Using a native Net.Data DLL, the JavaBean invokes Net.Data, populating the return code and a string containing the Net.Data output (results). Because JavaBeans use a native DLL, you do not have to have a Web server running to use Net.Data functions.

Design tip: The results returned by the Net.Data JavaBeans are whatever your macro or function returns; in general, this is HTML. Consider passing the results to an HTML-like JavaBean which understands HTML and displays the results.

With JavaBeans you can:

- Run Net.Data macros
- Run SQL statements through Net.Data

This section describes the following JavaBean topics:

- "About Net.Data JavaBeans"
- "Setting Up and Running the Net.Data JavaBeans"

About Net.Data JavaBeans

Net.Data provides JavaBeans to help you develop and manage macros using the Java environment. JavaBeans are Java objects that provide the following interface:

- When used in a JavaBean development environment (such as Lotus BeanMachine), you use the provided customizer to hook together the desired components to process and display the results of a macro or SQL statement and produce a Java applet.
- When using the API, you can use the JavaBeans to provide Net.Data functionality to your own Java applet or application. The API documentation is in `<inst_dir>/beans/NetDataBeans.jar`.

Net.Data provides two types of JavaBeans:

Net.Data Macro JavaBean
: Provides a Java-based interface for executing an existing Net.Data macro through Net.Data.

Net.Data SQL JavaBean
: Provides a Java based interface for executing an SQL statement through Net.Data.

The Net.Data JavaBeans are Java-based wrappers that run through Net.Data using a native DLL file. Both require Net.Data Version 2 or higher and JDK Version 1.1 or higher to be installed.

Setting Up and Running the Net.Data JavaBeans

This section describes how to set up and run Net.Data JavaBeans using a JavaBean development tool, such as Bean Machine. Steps for using development tools are generic so that you can use the tool of your choice.

- "The Macro Bean" on page 79
- "The SQL Bean" on page 79

The Macro Bean: The Net.Data Macro bean, com.ibm.netdata.beans.NetDataMacro, lets you use Java to run an existing macro. To use this bean, you need to specify Net.Data properties for the bean so that it can work with the macro.

To set up the Net.Data macro JavaBean with a JavaBean development tool:

1. Add or import the <*inst_dir*>/beans/NetDataBeans.jar file into your JavaBean development tool.
2. Using the development tool's customizer interface, set the following input properties:

   Macro Specifies the name of the existing macro to execute. For example: MyMacro.mac

   Block Specifies the name of the HTML block section to execute; the default is report.

   HTML path
   Specifies the path to the Net.Data db2www.ini file.

   Parameters
   Specifies the parameter name and values to use when running macro.

   Syntax:

   ```
   name1=value1&nameN=valueN
   ```

To run the Net.Data macro JavaBean with a JavaBean development tool:

1. Select the run or execute action provided by your JavaBean development tool to run the macro.
2. After the macro has run, you can reference the following output properties:

   RC Specifies the return code returned from Net.Data.

   Results
   Specifies the data returned from the execution of the Net.Data macro.

The SQL Bean: The Net.Data SQL bean, com.ibm.netdata.beans.NetDataSQL, lets you use Java to run an SQL statement through Net.Data. To use this bean, you need to specify Net.Data properties for the bean so that it can work with the macro.

To set up the Net.Data SQL JavaBean with a JavaBean development tool:

1. Add or import the NetDataBeans.jar file into your JavaBean development tool.
2. Using the development tool's customizer interface, set the following input properties:

   Language environment
   Specifies the language environment to use; the default is DTW_SQL.

   SQL Specifies the SQL statement to run; the default is select * from employee.

   DATABASE
   Specifies the database to use; the default is SAMPLE.

   HTML path
   Specifies the path to the Net.Data db2www.ini file.

   Parameters
   Specifies the parameter name and values to use when running the SQL statement.

Syntax:

```
name1=value1&nameN=valueN
```

To run the Net.Data SQL JavaBean with a JavaBean development tool:

1. Select the run or execute action provided by your JavaBean development tool to run the macro.
2. After the SQL statement has run, you can reference the following output properties:

   RC Specifies the return code returned from Net.Data.

   Results
   Specifies the data returned from the SQL statement.

Chapter 5. Developing Net.Data Macros

A Net.Data macro is a text file consisting of a series of Net.Data macro language constructs that:

- Specify the layout of Web pages
- Define variables and functions
- Call functions that are built-in to Net.Data or defined in the macro
- Format the processing output and return it to the Web browser for display

The Net.Data macro contains two organizational parts: the declaration part and the presentation part, as shown in Figure 19.

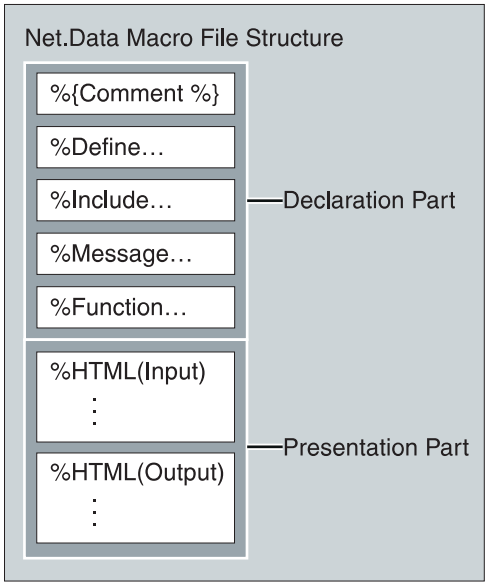


Figure 19. Macro Structure

- The *declaration part* contains the definitions of variables and functions in the macro.
- The *presentation part* contains HTML blocks that specify the layout of the Web page. The HTML blocks are made up of text presentation statements that are supported by your Web browser, such as HTML and JavaScript.

You can use these parts multiple times and in any order. See *Net.Data Reference* for syntax of the macro parts and constructs.

Authorization Tip: Ensure that the Web server has access rights to this file. See “Granting Access Rights to Files Accessed by Net.Data” on page 51 for more information.

This chapter examines the different blocks that make up a Net.Data macro and methods you can use for writing the macro.

- “Conditional Logic and Looping in a Macro” on page 112

Anatomy of a Net.Data Macro

The macro consists of two parts:

- The declaration part, that contains definitions used in the presentation part. The declaration part uses two major optional blocks:
  - DEFINE block
  - FUNCTION block

  The declaration part can also contain other language constructs and statements, such as EXEC statements, IF blocks, INCLUDE statements, and MESSAGE blocks. For more information about the language constructs, see the chapter about language constructs in *Net.Data Reference*.

  Authorization Tip: Ensure that the Web server has access rights to files referenced by the EXEC and INCLUDE statements. See “Granting Access Rights to Files Accessed by Net.Data” on page 51 for more information.
- The presentation part defines the layout of the Web page, references variables, and calls functions using HTML blocks that are used as entry and exit points from the macro. When you invoke Net.Data, you specify an HTML block name as an entry point for processing the macro. The HTML blocks are described in “HTML Blocks” on page 84.

In this section, a simple Net.Data macro illustrates the elements of the macro language. This example macro presents a form that prompts for information to pass to a REXX program. The macro passes this information to an external REXX program called `ompsamp.cmd`, which echoes the data that the user enters. The results are then displayed on a second HTML page.

First, look at the entire macro, and then each block in detail:

```
%{ **********************       DEFINE block        ************************%}
%DEFINE {
   page_title="Net.Data Macro Template"
%}

%{ ********************** FUNCTION Definition block ************************%}
%FUNCTION(DTW_REXX) rexx1 (IN input) returns(result)
{
  %EXEC{ompsamp.cmd %}
%}

%FUNCTION(DTW_REXX) today () RETURNS(result)
{
      result = date()
%}

%{ **********************      HTML Block: Input    ************************%}
%HTML (INPUT) {
<html>
<head>
<title>$(page_title)</title>
</head><body>
<h1>Input Form</h1>
Today is @today()

<FORM METHOD="post" ACTION="OUTPUT">
```

```
Type some data to pass to a REXX program:
<INPUT NAME="input_data" TYPE="text" SIZE="30">
<p>
<INPUT TYPE="submit" VALUE="Enter">

</form>

<hr>
<p>[<a href="/">Home page</a>]
</body></html>
%}

%{ **********************    HTML Block: Output     ************************%}
%HTML (OUTPUT) {
<html>
<head>
<title>$(page_title)</title>
</head><body>
<h1>Output Page</h1>
<p>@rexx1(input_data)
<p><hr>
<p>[<a href="/">Home page</a> |
<a href="input">Previous page</a>]
</body></html>
%}
```

The sample macro consists of four major blocks: the DEFINE, the FUNCTION, and the two HTML blocks. You can have multiple DEFINE, FUNCTION, and HTML blocks in one Net.Data macro.

The two HTML blocks contain text presentation statements such as HTML, which make writing Web macros easy. If you are familiar with HTML, building a macro simply involves adding macro statements to be processed dynamically at the server and SQL statements to send to the database.

Although the macro looks similar to an HTML document, the Web server accesses it through Net.Data using CGI, a Web server API, or a Java Servlet. To invoke a macro, Net.Data requires two parameters: the name of the macro to process, and the HTML block in that macro to display.

When the macro is invoked, Net.Data processes it from the beginning. The following sections look at what happens as Net.Data processes the file.

The DEFINE Block

The DEFINE block contains the DEFINE language construct and variable definitions used later in the HTML blocks. The following example shows a DEFINE block with one variable definition:

```
%{ **********************       DEFINE Block        ************************%}
%DEFINE {
   page_title="Net.Data Macro Template"
%}
```

The first line is a comment. A comment is any text inside `%{` and `%}`. Comments can be anywhere in the macro. The next statement starts a DEFINE block. You can define multiple variables in one define block. In this example, only one variable, `page_title`, is defined. After it is defined, this variable can be referenced anywhere in the macro using the syntax, `$(page_title)`. Using variables makes it easy to make global changes to your macro later. The last line of this block, `%}`, identifies the end of the DEFINE block.

The FUNCTION Block

The FUNCTION block contains declarations for functions invoked by the HTML blocks. Functions are processed by language environments and can execute programs, SQL queries, or stored procedures.

The following example shows two FUNCTION blocks. One defines a call to an external REXX program and the other contains inline REXX statements.

```
%{ ********************** FUNCTION Block ********************************%}
%FUNCTION(DTW_REXX) rexx1 (IN input) returns(result) { <-- This function accepts
                                                          one parameter and returns the
                                                          variable 'result', which is
                                                          assigned by the external REXX
                                                          program
     %EXEC{ompsamp.cmd %}  <-- The function executes an external REXX program
                               called "ompsamp.cmd"
%}

%FUNCTION(DTW_REXX) today () RETURNS(result) {
      result = date()  <-- The single source statement for this function is
                           contained inline.
%}
```

The first function block, `rexx1`, is a REXX function declaration that in turn, runs an external REXX program called `ompsamp.cmd`. One input variable, `input`, is accepted by this function and automatically passed to the external REXX command. The REXX command also returns one variable called `result`. The contents of the `result` variable in the REXX command replaces the invoking `@rexx1()` function call contained in the OUTPUT block. The variables `input` and `result` are directly accessible by the REXX program, as shown in the source code for `ompsamp.cmd`:

```
/* REXX */
result = 'The REXX program received "'input'" from the macro.'
```

The code in this function echoes the data that was passed to it. You can format the resulting text any way you want by enclosing the requesting `@rexx1()` function call in normal mark-up style tags (like <b> or). Rather than using the `result` variable, the REXX program could have written HTML tags to standard output using REXX SAY statements.

The second function block, also refers to a REXX program, `today`. However, the entire REXX program in this case is contained in the function declaration itself. An external program is not needed. Inline programs are allowed for both REXX and Perl functions because they are interpreted languages that can be parsed and executed dynamically. Inline programs have the advantage of simplicity by not requiring a separate program file to manage. The first REXX function could also have been handled inline.

HTML Blocks

HTML blocks define the layout of the Web page, reference variables, and call functions. HTML blocks are used as entry and exit points from the macro. An HTML block is always specified in the Net.Data macro request and every macro must have at least one HTML block.

The first HTML block in the example macro is named `INPUT`. The `HTML(INPUT)` contains the HTML for a simple form with one input field.

```
%{ **********************      HTML Block: Input    ************************%}
%HTML (INPUT) {               <--- Identifies the name of this HTML block.
<html>
<head>
<title>$(page_title)</title>  <--- Note the variable substitution.
</head><body>
<h1>Input Form</h1>
Today is @today()             <--- This line contains a call to a function.

<FORM METHOD="post" ACTION="OUTPUT">  <--- When this form is submitted,
                                           the "OUTPUT" HTML block is called.
Type some data to pass to a REXX program:
<INPUT NAME="input_data"      <--- "input_data" is defined when the form
TYPE="text" SIZE="30">             is submitted and can be referenced elsewhere in
                                   this macro. It is initialized to whatever the
                                   user types into the input field.
<p>
<INPUT TYPE="submit" VALUE="Enter">

<hr>
<p>
[
<a href="/">Home page</a>]
 </body><html>
%}                            <--- Closes the HTML block.
```

The entire block is surrounded by the HTML block identifier, `%HTML (INPUT) {...%}`. `INPUT` identifies the name of this block. The name can contain any alphanumeric character, underscores, or periods. The HTML `<title>` tag contains an example of variable substitution. The value of the variable `page_title` is substituted into the title of the form.

This block also has a function call. The expression `@today()` is a call to the function `today`. This function is defined in the FUNCTION block that is described above. Net.Data inserts the result of the `today` function, the current date, into the HTML text in the same location that the `@today()` expression is located.

The ACTION parameter of the FORM statement provides an example of navigation between HTML blocks or between macros. Referencing the name of another block in an ACTION parameter accesses that block when the form is submitted. Any input data from an HTML form is passed to the block as implicit variables. This is true of the single input field defined on this form. When the form is submitted, data entered in this form is passed to the `HTML(OUTPUT)` block in the variable *input_data*.

You can access HTML blocks in other macros with a relative reference if the macros are on the same Web server. For example, the ACTION parameter `ACTION="../othermacro.d2w/main"` accesses the HTML block called main in the macro `othermacro.d2w`. Again, any data entered into the form is passed to this macro in the variable *input_data*.

When you invoke Net.Data, you pass the variable as part of the URL. For example:

```
<a href="/cgi-bin/db2www/othermacro.d2w/main?input_data=value">Next macro</a>
```

You can access or manipulate form data in the macro by referencing the variable name specified in the form.

The next HTML block in the example is the `HTML(OUTPUT)` block. It contains the HTML tagging and Net.Data macro statements that define the output processed from the `HTML(INPUT)` request.

```
%{ **********************    HTML Block: Output     ************************%}
%HTML (OUTPUT) {
<html>
<head>
<title>$(page_title)</title>  <--- More substitution.

</head><body>
<h1>Output Page</h1>
<p>@rexx1(input_data)  <--- This line contains a call to function rexx1
                           passing the argument "input_data".
<p>
<hr>
<p>
[
<a href="/">Home page</a> |
<a href="input">Previous page</a>]
%}
```

Like the HTML(INPUT) block, this block is standard HTML with Net.Data macro statements to substitute variables and a function call. Again the page_title variable is substituted into the title statement. And, as before, this block contains a function call. In this case, it calls the function rexx1 and passes to it the contents of the variable input_data, which it received from the form defined in the Input block. You can pass any number of variables to and from a function. The function definition specifies the number and the usage of variables that are passed.

Net.Data Macro Variables

Net.Data lets you define and reference variables in a Net.Data macro. In addition, you can pass these variables from the macro to language environments and back.

Net.Data variables can be defined depending on the type of variable and whether it has a predefined value. These variables can be categorized into the following types, based on how they are defined:

- Explicitly defined variables using the DEFINE statement in the DEFINE block
- Predefined variables, which are variables that are made available by Net.Data and are set to a value. This value usually cannot be changed.
- Implicitly defined variables, which are of four types:
  - Variables that are not explicitly defined but are instantiated when first assigned a value.
  - Parameter variables that are part of a FUNCTION block definition and that can only be referenced within a FUNCTION block.
  - Variables that are instantiated by Net.Data and correspond to form data or query string data.
  - Variables that are associated with a Net.Data table and that can only be referenced within a ROW block or REPORT block.

The following sections describe:

Identifier Scope

An identifier, which is a variable or a function call, becomes *visible*, meaning that it can be referenced when it is declared or instantiated. The region where an identifier is visible is called its *scope*. The five types of scope are:

- Global

  An identifier has global scope if you can reference it anywhere within a macro. Identifiers that have global scope are:
  - Net.Data built-in functions
  - Form data
  - Query string data
  - Variables instantiated from within an HTML block
- Macro

  An identifier has this scope if its declaration appears outside of any block. A block starts with an opening bracket ({) and ends with a percent sign and bracket (%}). (Note that DEFINE blocks are excluded from this definition and should be treated as separate DEFINE statements.) Unlike an identifier with global scope, one with macro scope can only be referred to by items in the macro that follow the idenfier's declaration.
- FUNCTION block or MACRO_FUNCTION block

  An identifier has function block scope if:
  - The identifier is declared in the parameter list of the function definition.

    If an identifier with the same name already exists outside the function definition, then Net.Data uses the identifier from the function parameter list within the function block.
  - The identifier is instantiated in the function block and is not declared or instantiated prior to the function call.

  An identifier does not have function block scope if it has been declared or initialized outside of the function and is not declared in the function parameter list. The value of the identifier within the function block remains unchanged unless updated by the function.
- REPORT block

  An identifier has report block scope if it can be referenced only from within a REPORT block (for example, table column names N1, N2, ..., N*n*). Only those variables that Net.Data implicitly defines as part of its table processing can have a report block scope. Any other variables that are instantiated have function block scope.
- ROW block

  An identifier has row block scope if it can only be referenced from within a ROW block (for example, table value names V1, V2, ..., Vn). Only those variables that Net.Data implicitly defines as part of its table processing can have a row block scope. Any other variables that are instantiated have function block scope.

Defining Variables

There are three ways to define variables in a Net.Data macro:

- Define statement or block
- HTML form tags
- Query string data

A variable value received from form or query string data overrides a variable value set by a DEFINE statement in a Net.Data macro.

- **DEFINE statement or block**

  The simplest way to define a variable for use in a Net.Data macro is to use the DEFINE statement. The syntax is as follows:

```
%DEFINE variable_name="variable value"

%DEFINE variable_name={ variable value on multiple
                        lines of text  %}

%DEFINE {
    variable_name1="variable value 1"
    variable_name2="variable value 2"
%}
```

  The *variable_name* is the name you give the variable. Variable names must begin with a letter or underscore and can contain any alphanumeric character, an underscore, a period, or a hash (#). All variable names are case-sensitive, except N_*columnName* and V_*columnName*, which are table variables.

  For example:

```
%DEFINE reply="hello"
```

  The variable `reply` has the value `hello`.

  Two consecutive quotes alone is equal to an empty string. For example:

```
%DEFINE empty=""
```

  The variable `empty` has an empty string.

  If your variable contains special characters, such as an end-of-line, use block braces around the value:

```
%DEFINE introduction={
Hello,
My name is John.
%}
```

  To include quotes in a string, you can use two quotes consecutively.

```
%DEFINE HI="say ""hello"""
```

  You can also use block braces to escape the quotes:

```
%DEFINE HI={ say "hello" %}
```

  To define several variables with one DEFINE statement, use a DEFINE block:

```
%DEFINE {
         variable1="value1"
         variable2="value2"
         variable3="value3"
         variable4="value4"
%}
```

- **HTML form tags: SELECT, INPUT, and TEXTAREA**

  You can use HTML FORM tags to assign values to variables, namely the SELECT, INPUT, and TEXTAREA tags. The following example uses standard HTML form tags to define Net.Data variables:

```
<INPUT NAME="variable_name" TYPE=...>
```

  or

```
<SELECT NAME="variable_name">
  <OPTION>value one
  <OPTION>value two
</SELECT>
```

To assign a variable that spans multiple lines or contains special characters, such as quotes, the TEXTAREA tag can be used:

```
<TEXTAREA NAME="variable_name" ROWS="4">
Please type the multi-line value
of your variable here.
</TEXTAREA>
```

The *variable_name* is the name you give the variable, and the value of the variable is determined from the input received in the form. See "HTML Forms" on page 64 for an example of how this type of variable definition is used in a Net.Data macro.

- **Query string data**

  You can pass variables to Net.Data through the query string. For example:

  ```
  http://www.ibm.com/cgi-bin/db2www/stdqry1.d2w/input?field=custno
  ```

  In the above example, the variable name, `field`, and the variable value, `custno`, specify additional data that Net.Data receives from the query string. Net.Data receives and processes the data as it would from form data.

Referencing Variables

You can reference a previously defined variable to return its value. To reference a variable in Net.Data macros, specify the variable name inside `$(` and `)`. For example:

```
$(variableName)
$(homeURL)
```

When Net.Data finds a variable reference, it substitutes the variable reference with the value of the variable. Variable references can contain strings, variable references, and function calls.

You can dynamically generate variable names. With this technique, you can use loops to process variably-sized tables or input data for lists that are built at run time, when the number in the list cannot be determined in advanced. For example, you can generate lists of HTML form elements that are generated based on records returned from an SQL query.

To use variables as part of your text presentation statements, reference them in the HTML blocks of your macro.

Invalid variable references: Invalid variable references are resolved to the empty string. For example, if a variable reference contains invalid characters such as an exclamation point (!), the reference is resolved to the empty string.

Valid variable names must begin with an alphanumeric character or an underscore, and they can consist of alphanumeric characters, including a period, underscore, and hash mark.

Example 1: Variable reference in a link

If you have defined the variable *homeURL*:

```
%DEFINE homeURL="http://www.ibm.com/"
```

You can refer to the home page as $(*homeURL*) and create a link:

```
<A href="$(homeURL)">Home page</A>
```

You can reference variables in many parts of the Net.Data macro; check the language constructs in this chapter to determine in which parts of the macro variable references are allowed. If the variable has not yet been defined at the time it is referenced, Net.Data returns an empty string. A variable reference alone does not define the variable.

Example 2: Dynamically generated variable references

Assume that you run an SQL SELECT statement with any number of elements. You can create an HTML form with input fields using the following ROW blocks:

```
...
%ROW {
<input type=text name=@dtw_rconcat("I", ROW_NUM) size=10 maxlength=10>
%}
...
```

Because you created INPUT fields, you would probably want to access the values that the user entered when the form is submitted to your macro for processing. You can code a loop to retrieve the values in a variable length list:

```
<PRE>
...
@dtw_assign(rowIndex, "1")
%while (rowIndex <= rowCount) {
The value entered for row $(rowIndex) is: $(I$(rowIndex))
@dtw_add(rowIndex, "1", rowIndex) %}
...
</PRE>
```

Net.Data first generates the variable name using the I$(rowIndex) reference. For example, the first variable name would be I1. Net.Data then uses that value and resolves to the value of the variable.

Example 3: A variable reference with nested variable references and a function call

```
%define my = "my"
%define u = "lower"
%define myLOWERvar = "hey"


$($(my)@dtw_ruppercase(u)var)
```

The variable reference returns the value of `hey`.

Variable Types

You can use the following types of variable in your macros.

If you assign strings to variables that are defined a certain way by Net.Data, such as ENVVAR, LIST, condition list variables, the variable no longer behaves in the defined way. In other words, the variable becomes a simple variable, containing a string.

See *Net.Data Reference* for syntax and examples of each variable.

Conditional Variables

Conditional variables let you define a conditional value for a variable by using a method similar to an IF, THEN construct. When defining the conditional variable, you can specify two possible variable values. If the first variable you reference exists, the conditional variable gets the first value; otherwise the conditional variable gets the second value. The syntax for a conditional variable is:

```
varA = varB ? "value_1" : "value_2"
```

If `varB` is defined, `varA="value_1"`, otherwise `varA="value_2"`. This is equivalent to using an IF block, as in the following example:

```
%IF ($(varB))
    varA = "value_1"
%ELSE
    varA = "value_2"
%ENDIF
```

See “List Variables” on page 94 for an example of using conditional variables with list variables.

Environment Variables

You can reference environment variables that the Web server makes available to the process or thread that is processing your Net.Data request. When the ENVVAR variable is referenced, Net.Data returns the current value of the environment variable by the same name.

The syntax for defining environment variables is:

```
%DEFINE var=%ENVVAR
```

Where *var* is the name of the environment variable being defined.

For example, the variable SERVER_NAME can be defined as environment variable:

```
%DEFINE SERVER_NAME=%ENVVAR
```

And then referenced:

```
The server is $(SERVER_NAME)
```

The output looks like this:

```
The server is www.software.ibm.com
```

See *Net.Data Reference* for more information about the ENVVAR statement.

Executable Variables

You can invoke other programs from a variable reference using executable variables.

Define executable variables in a Net.Data macro using the EXEC language construct in the DEFINE block. For more information about the EXEC language element, see the language constructs chapter in the *Net.Data Reference*. In the following example, the variable `runit` is defined to execute the executable program `testProg`:

```
%DEFINE runit=%EXEC "testProg"
```

`runit` becomes an executable variable.

Net.Data runs the executable program when it encounters a valid variable reference in a Net.Data macro. For example, the program `testProg` is executed when a valid variable reference is made to the variable `runit` in a Net.Data macro.

A simple method is to reference an executable variable from another variable definition. The following example demonstrates this method. The variable `date` is defined as an executable variable and `dateRpt` contains a reference to the executable variable.

```
%DEFINE date=%EXEC "date"
%DEFINE dateRpt="Today is $(date)"
```

Wherever `$(dateRpt)` appears in the Net.Data macro, Net.Data searches for the executable program `date`, and when it locates it, returns:

```
Today is Tue 11-07-1999
```

When Net.Data encounters an executable variable in a macro, it looks for the referenced executable program using the following method:

1. It searches the directories specified by the EXEC_PATH in the Net.Data initialization file. See “EXEC_PATH” on page 18 for details.
2. If Net.Data does not locate the program, the system searches the directories defined by the system PATH environment variable or the library list. If it locates the executable program, Net.Data runs the program.

Restriction: Do not set an executable variable to the value of the output of the executable program it calls. In the previous example, the value of the variable `date` is NULL. If you use this variable in a DTW_ASSIGN function call to assign its value to another variable, the value of the new variable after the assignment is NULL also. The only purpose of an executable variable is to invoke the program it defines.

You can also pass parameters to the program to be executed by specifying them with the program name on the variable definition. In this example, the values of distance and time are passed to the program *calcMPH*.

```
%DEFINE mph=%EXEC "calcMPH $(distance) $(time)"
```

This next example returns the system date as part of the report:

```
%DEFINE database="celdial"
 %DEFINE tstamp=%EXEC "date"

%FUNCTION(DTW_SQL) myQuery() {
SELECT CUSTNO, CUSTNAME from dist1.customer
%REPORT{
%ROW{
```

```
<A HREF="/cgi-bin/db2www/exmp.d2w/report?value1=$(V1)&value2=$(V2)">
$(V1) $(V2) </A> <BR>
%}
%}
%}

%HTML(report){
<H1>Report made: $(tstamp) </H1>
@myQuery()
%}
```

Each report displays the date for easy tracking. This example also puts the customer number and name in a link for another Net.Data macro. Clicking on any customer in the report calls the `exmp.d2w` Net.Data macro, passing the customer number and name to the Net.Data macro.

Hidden Variables

You can use hidden variables to conceal the actual name of a variable from application users who view your Web page source with their Web browser. To define a hidden variable:

1. Define a variable for each string you want to hide, after the variable's last reference in the HTML block. Variables are always defined with the DEFINE language construct after they are used in the HTML block, as in the following example. The $$(*variable*) variables are referenced and then defined.
2. In the HTML block where the variables are referenced, use double dollar signs instead of a single dollar sign to reference the variables. For example, `$$(X)` instead of `$(X)`.

   ```
   %HTML(INPUT) {
   <FORM ...>
   <P>Select fields to view:
   shanghai<SELECT NAME="Field">
   <OPTION VALUE="$$(name)"> Name
   <OPTION VALUE="$$(addr)"> Address
   ...
   </FORM>
   %}

   %DEFINE {
   name="customer.name"
   addr="customer.address"
   %}

   %FUNCTION(DTW_SQL) mySelect() {
     SELECT $(Field) FROM customer
   %}

   ...
   ```

   When a Web browser displays the HTML form, `$$(name)` and `$$(addr)` are replaced with `$(name)` and `$(addr)` respectively, so the actual table and column names never appear on the HTML form. Application users cannot tell that the true variable names are hidden. When the user submits the form, the HTML(REPORT) block is called. When `@mySelect()` calls the FUNCTION block, `$(Field)` is substituted in the SQL statement with `customer.name` or `customer.addr` in the SQL query.

List Variables

Use list variables to build a delimited string of values. They are particularly useful in helping you construct an SQL query with multiple items like those found in some WHERE or HAVING clauses. The syntax for a list variable is:

```
%LIST " value_separator " variable_name
```

Recommendation: The blanks are significant. Insert a space before and after the value separator for most cases. Most queries use Boolean or mathematical operators (for example, AND, OR, or >) for the value separator. The following example illustrates the use of conditional, hidden, and list variables:

```
%HTML(INPUT) {
<FORM METHOD="POST" ACTION="/cgi-bin/db2www/example2.d2w/report">
<H2>Select one or more cities:</H2>
<INPUT TYPE="checkbox" NAME="conditions" VALUE="$$(cond1)">Sao Paolo<BR>
<INPUT TYPE="checkbox" NAME="conditions" VALUE="$$(cond2)">Seattle<BR>
<INPUT TYPE="checkbox" NAME="conditions" VALUE="$$(cond3)">Shanghai<BR>
<INPUT TYPE="submit" VALUE="Submit Query">
</FORM>
%}

%DEFINE{
DATABASE="custcity"
 %LIST " OR " conditions
cond1="cond1='Sao Paolo'"
cond2="cond2='Seattle'"
cond3="cond3='Shanghai'"
whereClause= ? "WHERE $(conditions)" : ""
%}

%FUNCTION(DTW_SQL) mySelect(){
SELECT name, city FROM citylist
$(whereClause)
%}

%HTML(REPORT){
@mySelect()
%}
```

In the HTML form, if no boxes are checked, `conditions` is NULL, so `whereClause` is also NULL in the query. Otherwise, `whereClause` has the selected values separated by OR. For example, if all three cities are selected, the SQL query is:

```
SELECT name, city FROM citylist
WHERE cond1='Sao Paolo' OR cond2='Seattle' OR cond3='Shanghai'
```

This example shows that `Seattle` is selected, which results in this SQL query:

```
SELECT name, city FROM citylist
WHERE cond1='Seattle'
```

Table Variables

The table variable defines a collection of related data. It contains a set of rows and columns including a row of column headers. A table is defined in the Net.Data macro as in the following statement:

```
%DEFINE myTable=%TABLE(30)
```

The number following `%TABLE` is the limit on the number of rows that this table variable can contain. To specify a table with no limit on the number of rows, use the default or specify ALL, as shown in these examples:

```
%DEFINE myTable2=%TABLE
%DEFINE myTable3=%TABLE(ALL)
```

When you define a table, it has zero rows and zero columns. The only way you can populate a table with values is by passing it as an OUT or INOUT parameter to a function or by using the built-in table functions provided by Net.Data. The DTW_SQL language environment automatically puts the results of a SELECT statement into a table.

For non-database language environments, such as DTW_REXX or DTW_PERL, the language environment is also responsible for setting table values. However, the language environment script or program defines the table values cell-by-cell. See "Chapter 6. Using Language Environments" on page 117 for more information about how language environments use table variables.

You can pass a table between functions by referring to the table variable name. The individual elements of the table can be referred to in a REPORT block of a function or by using the Net.Data table functions. See "Table Processing Variables" on page 96 for accessing individual elements in a table within a REPORT block, and see "Table Functions" on page 105 for accessing individual elements of a table using a table function. Table variables are usually populated with values in an SQL function, and then used as input to a report, either in the SQL function or in another function after being passed to that function as a parameter. You can pass table variables as IN, OUT, or INOUT parameters to any non-SQL function. Tables can be passed to SQL functions only as OUT parameters.

If you reference a table variable, the contents of the table are displayed and formatted based on the setting of the DTW_HTML_TABLE variable. In the following example, the contents of `myTable` would be displayed:

```
%HTML (output) {
  $(myTable)
}
```

The column names and field values in a table are addressed as array elements with an origin of 1.

Miscellaneous Variables

These variables are Net.Data-defined variables that you can use to:

- Affect Net.Data processing
- Find out the status of a function call
- Obtain information about the result set of a database query
- Determine information about file locations and dates

Miscellaneous variables can either have a predefined value that Net.Data determines or have values that you set. For example, Net.Data determines the DTW_CURRENT_FILENAME variable value based on the current file that it is processing, whereas you can specify whether Net.Data removes extra white space caused by tabulators and new-line characters.

Predefined variables are used as variable references within the macro and provide information about the current status of files, dates, or the status of a function call. For example, to retrieve the name of the current file, you could use:

```
%REPORT {
  <p>This file is <i>$(DTW_CURRENT_FILENAME)</i>.</P>
}
```

Modifiable variable values are generally set using a DEFINE statement or with the @DTW_ASSIGN() function and let you affect how Net.Data processes the macro. For example, to specify whether white space is removed, you could use the following DEFINE statement:

```
%DEFINE DTW_REMOVE_WS="YES"
```

Table Processing Variables

Net.Data defines table processing variables for use in the REPORT and ROW blocks. Use these variables to reference values from SQL queries and function calls.

Table processing variables have a predefined value that Net.Data determines. These variables allow you to reference values from the result sets of SQL queries or function calls by the column, row, or field that is being processed. You can also access information about the number of rows being processed or a list of all the column names.

For example, as Net.Data processes a result set from an SQL query, it assigns the value of the variable Nn for each current column name, such that N1 is assigned to the first column, N2 is assigned to the second column, and so on. You can reference the current column name for your Web page output.

Use table processing variables as variable references within the macro. For example, to retrieve the name of the current column being processed, you could use:

```
%REPORT {
  <p>Column 1 is <i>$(N1)</i>.</P>
}
```

Table processing variables also provide information about the results of a query. You can reference the variable TOTAL_ROWS in the macro to show how many rows are returned from an SQL query, as in the following example:

```
Names found: $(TOTAL_ROWS)
```

Some of the table processing variables are affected by other variables or built-in functions. For example, TOTAL_ROWS requires that the DTW_SET_TOTAL_ROWS SQL language environment variable be activated so that Net.Data assigns the value of TOTAL_ROWS when processing the results from a SQL query or function call as in the following example:

```
%DEFINE DTW_SET_TOTAL_ROWS="YES"
...

Names found: $(TOTAL_ROWS)
```

Report Variables

Net.Data displays Web page output generated from the macro in a default report format. The default report format displays in a table format using <PRE> </PRE> tags. You can override the default report by defining a REPORT block with instructions for displaying the output or by using one of the report variables to prevent the default report from being generated.

Report variables help you customize how your Web page output is displayed and used with default reports and Net.Data tables. You must define these variables before using them with a DEFINE statement or with the @DTW_ASSIGN() function.

The report variables specify spacing, override default report formats, specify HTML table output versus default table output, and specify other display features. For example, you can use the ALIGN variable to control leading and trailing spaces for table processing variables. The following example uses the ALIGN variable to separate by a space each column name in a list that is returned by a query.

```
%DEFINE ALIGN="YES"
...
<p>Your query was on these columns: $(NLIST)
```

The START_ROW_NUM report variable lets you determine at which row to begin displaying the results of a query. For example, the following variable value specifies that Net.Data will begin displaying the results of a query at the third row.

```
%DEFINE START_ROW_NUM = "3"
```

You can also determine whether Net.Data uses HTML tags for default formatting. With DTW_HTML_TABLE set to YES, an HTML table is generated rather than a text-formatted table.

```
%DEFINE DTW_HTML_TABLE="YES"

%FUNCTION(DTW_SQL){
SELECT NAME, ADDRESS FROM $(qTable)
%}
```

Language Environment Variables

These variables are used with language environments and affect how the language environment processes a request.

With these variables, you can perform tasks such as establishing connections to databases,supplying alternate text for Java applets,enabling NLS support, and determining whether the execution of an SQL statement is successful.

For example, you can use the SQL_STATE variable to access or display the SQL state value returned from the database.

```
%FUNCTION (DTW_SQL) val1() {
 select * from customer
%REPORT {
 ...
%ROW {
 ...
%}
 SQLSTATE=$(SQL_STATE)
%}
```

This next example shows how to define which database is to accessed.

```
%DEFINE DATABASE="CELDIAL"
```

Net.Data Functions

Net.Data provides built-in functions for use in your applications, such as word and string manipulation functions or functions that retrieve and set table variable functions. You can also define functions for use with your application, for example to call an external program or a stored procedure.

User-defined functions
Those functions that you define for use with your application, for example to call an external program or a stored procedure.

Net.Data built-in functions
Those functions that Net.Data provides for use in your applications, such as functions for manipulating words and strings and functions that get and set table variables.

These sections describe the following topics:

Defining Functions

To define your own functions in the macro, use a FUNCTION block or MACRO_FUNCTION block:

FUNCTION block
Defines a subroutine that is invoked from a Net.Data macro and is processed by a language environment. FUNCTION blocks must contain language statements or calls to an external program.

MACRO_FUNCTION block
Defines a subroutine that is invoked from a Net.Data macro and is processed by Net.Data rather than a language environment. MACRO_FUNCTION blocks can contain any statement that is allowed in an HTML block.

Syntax: Use the following syntax to define functions:

FUNCTION block:

```
%FUNCTION(type) function-name([usage] [datatype] parameter, ...) [RETURNS(return-var)] {
  executable-statements
  [report-block]
  ...
  [report-block]
  [message-block]
%}
```

MACRO_ FUNCTION block:

```
%MACRO_FUNCTION function-name([usage] parameter, ...)  {
  executable-statements
  report-block
  ...
  report-block
%}
```

Where:

type Identifies a language environment that is configured in the initialization file. The language environment invokes a specific language processor (which processes the executable statements) and provides a standard interface between Net.Data and the language processor.

function-name
Specifies the name of the FUNCTION or MACRO_FUNCTION block. A

function call specifies the *function-name*, preceded by an at (@) sign. See “Calling Functions” on page 102 for details.

You can define multiple FUNCTION or MACRO_FUNCTION blocks with the same name so that they are processed at the same time. Each of the blocks must all have identical parameter lists. When Net.Data calls the function, all FUNCTION blocks with the same name or MACRO_FUNCTION blocks with the same name are executed in the order they are defined in the Net.Data macro.

usage Specifies whether a parameter is an input (IN) parameter, an output (OUT) parameter, or both types (INOUT). This designation indicates whether the parameter is passed into or received back from the FUNCTION block, MACRO_FUNCTION block, or both. The usage type applies to all of the subsequent parameters in the parameter list until changed by another usage type. The default type is IN.

datatype
The data type of the parameter. Some language environments expect data types for the parameters that are passed. For example, the SQL language environment expects them when calling stored procedures. See “Chapter 6. Using Language Environments” on page 117 to learn more about the supported data types for the language environment you are using.

parameter
The name of a variable with local scope that is replaced with the value of a corresponding argument specified on a function call. Parameter references, for example `$(parm1)`, in the executable statements or REPORT block are replaced with the actual value of the parameter. In addition, parameters are passed to the language environment and are accessible to the executable statements using the natural syntax of that language or as environment variables. Parameter variable references are not valid outside the FUNCTION or MACRO_FUNCTION blocks.

return-var
Specify this parameter after the RETURNS keyword to identify a special OUT parameter. The value of the return variable is assigned in the function block, and its value is returned to the place in the macro from which the function was called. For example, in the following sentence, `<p>My name is @my_name().`, `@my_name()` gets replaced by the value of the return variable. If you do not specify the RETURNS clause, the value of the function call is:

- NULL if the return code from the call to the language environment is zero
- The value of the return code, when the return code is non-zero.

executable-statements
The set of language statements that is passed to the specified language environment for processing after the variables are substituted and the functions are processed. *executable-statements* can contain Net.Data variable references and Net.Data function calls. *executable-statements* includes those executable statements that are allowed in an HTML block.

For FUNCTION blocks, Net.Data replaces all variable references with the variable values, executes all function calls, and replaces the function calls with their resulting values before the executable statements are passed to the language environment. Each language environment processes the statements differently. For more information about specifying executable statements or calling executable programs, see “Executable Variables” on page 92.

For MACRO_FUNCTION blocks, the executable statements are a combination of text and Net.Data macro language constructs. In this case, no language environment is involved because Net.Data acts as the language processor and processes the executable statements.

report-block
Defines one or more REPORT blocks for handling the output of the FUNCTION or MACRO_FUNCTION block. See "Report Blocks" on page 107.

message-block
Defines the MESSAGE block, which handles any messages returned by the FUNCTION block. See "Message Blocks" on page 101.

Define functions outside of any other block and before they are called in the Net.Data macro.

Using Special Characters in Functions

When characters that match Net.Data language constructs syntax are used in the language statements section of a function block as part of syntactically valid embedded program code (such as REXX or Perl), they can be misinterpreted as Net.Data language constructs, causing errors or unpredictable results in a macro.

For example, a Perl function might use the COMMENT block delimiter characters, %{. When the macro is run, the %{ characters are interpreted as the beginning of a COMMENT block. Net.Data then looks for the end of the COMMENT block, which it thinks it finds when it reads the end of the function block. Net.Data then proceeds to look for the end of the function block, and when it can't be found, issues an error.

Use one of the following methods to use COMMENT block delimiter characters, or any other Net.Data special characters as part of your embedded program code, without having them interpreted by Net.Data as special characters:

- Use the EXEC statement to call the program code, rather than putting the code inline.
- Use a variable reference to specify the special characters.

For example, the following Perl function contains characters representing a COMMENT block delimiter, %{, as part of its Perl language statements:

```
%FUNCTION(DTW_PERL) func() {
  ...
  for $num_words (sort bynumber keys %{ $Rtitles{$num} }) {
    &make_links($Rtitles{$num}{$num_words});
  }
  ...
%}
```

To ensure that Net.Data interprets the %{ characters as Perl source code rather than as a Net.Data COMMENT block delimiter, rewrite the function in either of the following ways:

- Use the %EXEC statement:

```
%FUNCTION(DTW_PERL) func() {
  %EXEC{ func.prl %}
%}
```

- Use a variable reference to specify the %{ characters:

```
%define percent_openbrace = "%{"

%FUNCTION(DTW_PERL) func() {
  ...
  for $num_words (sort by number keys $(percent_openbrace) $Rtitles{$num} } {
    &make_links($Rtitles{$num}{$num_words});
  }
  ...
%}
```

Message Blocks

The MESSAGE block lets you determine how to proceed after a function call, based on the success or failure of the function call, and lets you display information to the caller of the function. When processing a message, Net.Data sets the language environment variable RETURN_CODE for each function call to a FUNCTION block. RETURN_CODE is not set on a function call to a MACRO_FUNCTION block.

A MESSAGE block consists of a series of message statements, each of which specifies a return code value, message text, and an action to take. The syntax of a MESSAGE block is shown in the language constructs chapter of *Net.Data Reference*.

A MESSAGE block can have a global or a local scope. If the MESSAGE block is defined in a FUNCTION block, its scope is local to that FUNCTION block. If it is specified at the outermost macro layer, the MESSAGE block has global scope and is active for all function calls executed in the Net.Data macro. If you define more than one global MESSAGE block, the last one defined is active.

Net.Data uses these rules to process the value of the RETURN_CODE variable from a function call:

1. Check local MESSAGE block for an exact match; exit or continue as specified.
2. If RETURN_CODE is not 0, check local MESSAGE block for `+default` or `-default`; depending on the sign of RETURN_CODE, exit or continue as specified.
3. If RETURN_CODE is not 0, check local MESSAGE block for `default`; exit or continue as specified.
4. Check global MESSAGE block for an exact match; exit or continue as specified.
5. If RETURN_CODE is not 0, check global MESSAGE block for `+default` or `-default`; depending on the sign of RETURN_CODE, exit or continue as specified.
6. If RETURN_CODE is not 0, check global MESSAGE block for `default`; exit or continue as specified.
7. If RETURN_CODE is not 0, issue Net.Data internal default message and exit.

The following example shows part of a Net.Data macro with a global MESSAGE block and a MESSAGE block for a function.

```
%{ global message block %}
%MESSAGE {
   -100     : "Return code -100 message"   : exit
    100     : "Return code 100 message"    : continue
   +default : {
This is a long message that spans more
than one line. You can use HTML tags, including
links and forms, in this message. %}   : continue
%}
```

```
%{ local message block inside a FUNCTION block %}
%FUNCTION(DTW_REXX) my_function() {
  %EXEC { my_command.cmd %}
  %MESSAGE {
     -100     : "Return code -100 message"   : exit
      100     : "Return code 100 message"    : continue
     -default : {
This is a long message that spans more
than one line. You can use HTML tags, including
links and forms, in this message. %}   : exit
  %}
```

If *my_function()* returns with a RETURN_CODE value of 50, Net.Data processes the error in this order:

1. Check for an exact match in the local MESSAGE block.
2. Check for `+default` in the local MESSAGE block.
3. Check for `default` in the local MESSAGE block.
4. Check for an exact match in the global MESSAGE block.
5. Check for `+default` in the global MESSAGE block.

When Net.Data finds a match, it sends the message text to the Web browser and checks the requested action.

When you specify `continue`, Net.Data continues to process the Net.Data macro after printing the message text. For example, if a macro calls `my_functions()` five times and error 100 is found during processing with the MESSAGE block in the example, output from a program can look like this:

```
.
.
.
11 May 1997                 $245.45
13 May 1997                 $623.23
19 May 1997                 $ 83.02
return code 100 message
22 May 1997                 $ 42.67

Total:                      $994.37
```

Calling Functions

Use a Net.Data function call statement to call both user-defined functions and built-in functions. Use the at (@) character followed by a function name or a macro function name:

`@`*`function_name`*`([ `*`argument`*`,... ])`

function_name
: This is the name of the function or macro function to invoke. The function must already be defined in the Net.Data macro, unless this is a built-in function.

argument
: This is the name of a variable, a quoted string, a variable reference, or a function call. Arguments on a function call are matched up with the parameters on a function or macro function parameter list. And, each parameter is assigned the value of its corresponding argument while the function or macro function is being processed. The arguments must be the same number and type as the corresponding parameters.

Quoted strings as arguments can contain variable references and functions calls.

Example 1: Function call with a text string argument

```
@myFunction("abc")
```

Example 2: Function call with a variable and a function call arguments

```
@myFunction(myvar, @DTW_rADD("2","3"))
```

Example 3: Function call with a text string argument that contains a variable reference and a function call

```
@myFunction("abc$(myvar)def@DTW_rADD("2","3")ghi")
```

Calling Net.Data Built-in Functions

Net.Data provides a large set of built-in functions to simplify Web page development. These functions are already defined by Net.Data, so you do not need to define them. You can call these functions as you would call other functions.

Figure 20 shows how the Net.Data built-in functions and the macro interact.

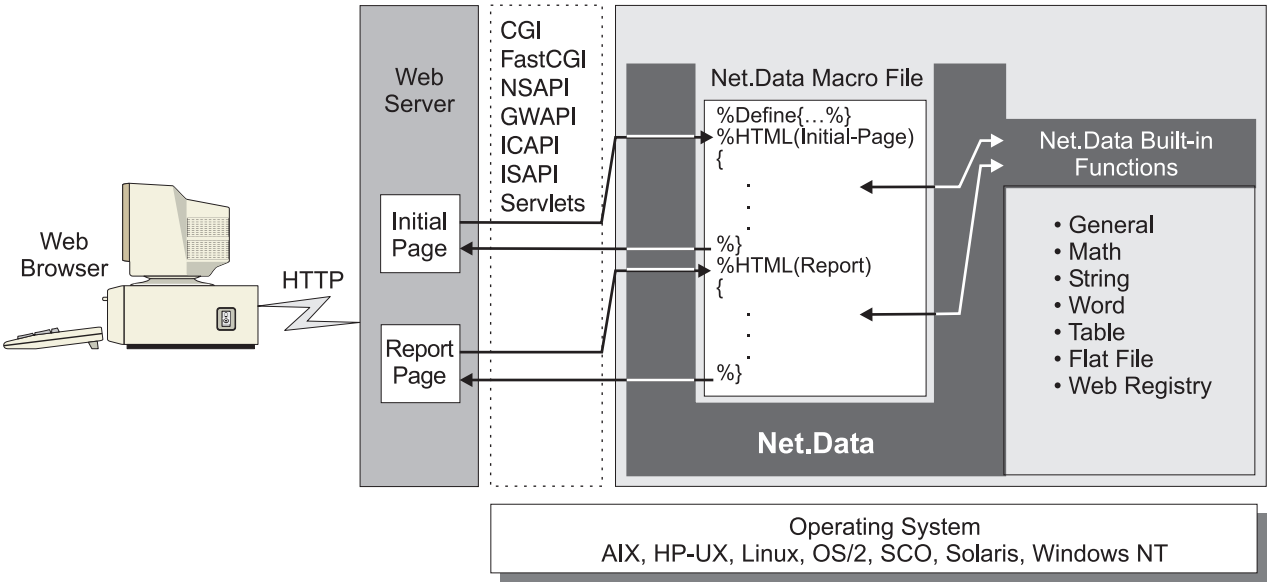


Figure 20. Net.Data Built-in Functions

Built-in functions can return their results in three ways, depending on its prefix:

- **DTW_, DTWF_, and DTWR_:** The results of the call are returned in an output parameter or no result is returned. (**DTWF**_ is the prefix for flat file functions. **DTWR**_ is the prefix for Web registry functions.)
- **DTW_r, DTWF_r, and DTWR_r:** The results of the function call replace the function call in the macro, in the same way the value of the RETURNS keyword replaces the function call for a user-defined function which has specified a RETURNS keyword.
- **DTW_m:** Multiple results are returned in each of the parameters passed to the function.

Some built-in functions do not have each type. To determine which type a particular built-in function has, see the Net.Data built-in functions chapter in *Net.Data Reference*.

The following sections provide a high-level overview of the Net.Data built-in functions. Use these functions to perform general purpose, math, string, word, or table manipulation functions. See *Net.Data Reference* for descriptions of each function with syntax and examples. Some of these functions required variables to be set prior to their use, or must be used in a specific context. Not all operating systems support each built-in function. See *Net.Data Reference* to determine which functions are supported for your operating system.

General Purpose Functions

This set of functions help you develop Web pages by altering data or accessing system services. You can use them to send mail, process HTTP cookies, generate HTML escape codes, and get other useful information from the system.

For example, to specify that Net.Data should exit a macro if a specific condition occurs, without processing the rest of the macro, you use the DTW_EXIT function:

```
%HTML(cache_example) {

<html>
 <head>
 <title>This is the page title</title>
 </head>
 <body>
 <center>
 <h3>This is the Main Heading</h3>
 <!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!>
 <! Joe Smith sees a very short page                    !>
 <!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!>
 %IF (customer == "Joe Smith")
</body>
</html>

@DTW_EXIT()

%ENDIF

...

</body>
 </html>
 %}
```

Another useful function is the DTW_URLESCSEQ function, which replaces characters that are not allowed in a URL with their escape values. For example, if the input variable `string1` equals `"Guys & Dolls"`, DTW_URLESCSEQ assigns the output variable to the value `"Guys%20%26%20Dolls"`.

Math Functions

These functions perform mathematical operations, letting you calculate or alter numeric data. Besides standard mathematical operations, you can also perform modulus division, specify a result precision, and use scientific notation.

For example, the function DTW_POWER raises the value of its first parameter to the power of its second parameter and returns the result, as shown in the following example:

```
@DTW_POWER("2", "-3", result)
```

DTW_POWER returns ".125" in the variable result

String Functions

These functions let you manipulate characters within strings. You can change a string's case, insert or delete characters, assign a string value to another variable, plus other useful functions.

For example, you can use DTW_ASSIGN to assign the value of an input variable to an output variable. You can also use this function to change a variable in a macro. In the following example, the variable RC is assigned to zero.

```
@DTW_ASSIGN(RC, "0")
```

Other string functions include DTW_CONCAT, which concatenates strings, and DTW_INSERT, which inserts strings at a specific position, as well many other string manipulations functions.

Word Functions

These functions let you manipulate words in character strings. Most of these functions work similar to string functions, but on entire words. For example, they let you count the number of words in a string, remove words, search a string for a word.

For example, use DTW_DELW0RD to delete a specified number of words from a string:

```
@DTW_DELWORD("Now is the time", "2", "2", result)
```

DTW_DELWORD returns the string "Now time".

Other word functions include DTW_WORDLENGTH, which returns the number of characters in a word, and DTW_WORDPOS, which returns the position of a word within a string.

Table Functions

You can use these functions to generate reports or forms using the data in a Net.Data table variable. You can also use these functions to create Net.Data tables, and to manipulate and retrieve values in those tables. Table variables contain a set of values and their associated column names. They provide a convenient way to pass groups of values to a function.

For example, DTW_TB_APPENDROW appends a row to the table. In the following example, Net.Data appends ten rows to the table, myTable:

```
@DTW_TB_APPENDROW(myTable, "10")
```

Additionally, DTW_TB_DUMPH, returns the contents of a macro table variable, enclosed in <PRE></PRE> tags, with each row of the table is displayed on a different line. And DTW_TB_CHECKBOX returns one or more HTML check box input tags from a macro table variable.

Flat File Functions

Use the flat file interface (FFI) functions to open, read, and manipulate data from flat file sources (text files), as well as store data in flat files.

For example, DTWF_APPEND, writes the contents of a table variable to the end of a file, and DTWF_DELETE deletes records from a file.

Additionally, the FFI functions allow file locking with DTWF_CLOSE and DTWF_OPEN. DTWF_OPEN locks a file that so that another request cannot read or update the file. DTWF_CLOSE releases the file when Net.Data is done with it, allowing other requests to access the file.

Web Registry Functions

Use the Web registry functions to maintain registries and the entries they contain. A Web registry is a file with a key maintained by Net.Data to allow you to add, retrieve, and delete entries easily.

For example, DTWR_ADDENTRY adds entries, while DTWR_DELENTRY deletes entries. DTWR_LISTSUB returns information about the registry entries in an OUT table parameter, and DTWR_UPDATEENTRY replaces the existing values for a specified registry entry with a new value.

Generating Web Pages in a Macro

Net.Data lets you easily present standard Web pages on the application user's browser. The following sections describe the HTML and REPORT blocks of the macro and show you how to format Web pages in Net.Data macros. See the language constructs chapter in *Net.Data Reference* for syntax information for these blocks.

HTML Blocks

A Net.Data macro contains HTML blocks that generate text presentation statements, such as HTML, to a Web browser. In a macro, you must specify at least one HTML block, but can specify as many as you want. Each HTML block generates a single Web page at the browser. Net.Data processes only one HTML block each time it is invoked. To create an application consisting of many Web pages, you can invoke Net.Data multiple times to process HTML blocks using standard navigation techniques, such as links and forms.

Any valid text presentation statements, such as HTML or JavaScript, can appear in an HTML block. In addition, you can use INCLUDE statements, function calls, and variable references in an HTML block. The following example shows a common use of HTML blocks in a Net.Data macro:

```
%DEFINE DATABASE="MNS96"

%HTML(INPUT){
<H1>Hardware Query Form</H1>
<HR>
```

```
<FORM METHOD="POST" ACTION="/cgi-bin/db2www/equiplst.d2w/report">
<dl>
<dt>What hardware do you want to list?
<dd><input type="radio" name="hdware" value="MON" checked>Monitors
<dd><input type="radio" name="hdware" value="PNT">Pointing devices
<dd><input type="radio" name="hdware" value="PRT">Printers
<dd><input type="radio" name="hdware" value="SCN">Scanners
</dl>
<HR>
<input type="submit" value="Submit">
</FORM>
%}

%FUNCTION(DTW_SQL) myQuery() {
SELECT MODNO, COST, DESCRIP FROM EQPTABLE WHERE TYPE=$(hdware)
%REPORT{
<B>Here is the list you requested:</B><BR>
%ROW{
<HR>
$(N1): $(V1)    $(N2): $(V2)
<P>
$(V3)
%}
%}
%}

%HTML(REPORT){
@myQuery()
%}
```

You can invoke the Net.Data macro from an HTML link like the one in the following example:

```
<a href="http://www.ibm.com/cgi-bin/db2www/equiplst.d2w/input">
  List of hardware</a>
```

When the application user clicks on this link, the Web browser invokes Net.Data, and Net.Data parses the macro. When Net.Data begins processing the HTML block specified on the invocation, in this case the HTML(INPUT) block, it begins to process the text inside the block. Anything that Net.Data does not recognize as a Net.Data macro language construct, it sends to the browser for display.

After the user makes a selection and presses the Submit button, Net.Data runs the ACTION part of the HTML FORM element, which specifies a call to the Net.Data macro's HTML(OUTPUT) block. Net.Data then processes the HTML(OUTPUT) block just as the HTML(INPUT) block was.

Net.Data then processes the `myQuery()` function call, which in turn invokes the SQL FUNCTION block. After replacing the `$(hdware)` variable reference in the SQL statement with the value returned from the input form, Net.Data runs the query. At this point, Net.Data resumes processing the report, displaying the results of the query according to the text presentation statements specified in the REPORT block.

After Net.Data completes the REPORT block processing, it returns to the HTML(OUTPUT) block, and finishes processing.

Report Blocks

Use the REPORT block language construct to format and display data output from a FUNCTION block. This output is typically table data, although any valid combination of text, macro variable references, and function calls can be specified.

A table name can optionally be specified on the REPORT block. Except for SQL and ODBC language environments, if you do not specify a table name, Net.Data uses the table data from the first output table in the FUNCTION parameter list.

The REPORT block has three parts, each of which is optional:

- Header information, which contains text that is displayed once before the table row data.
- A ROW block, which contains text and table variables that are displayed once for each row of the result table.
- Footer information, which contains text that is displayed once after the table row data.

Example:

```
%REPORT{
<H2>Query Results</H2>
<P>Select a name for details.
<TABLE BORDER=1>
  <TR>
    <TD>Name</TD>
    <TD>Location</TD></TR>
  %ROW{
  <TR>
    <TD>
    <a href="/cgi-bin/db2www/name.d2w/details?name=$(V1)&location;=$(V2)">$(V1)</a>
    </TD>
    <TD>$(V2)</TD>
  </TR>
  %}
</TABLE>
%}
```

REPORT Block Guidelines

Use the following guidelines when creating REPORT blocks:

- To avoid displaying any table output from the ROW block, leave the ROW block empty or omit it entirely.
- Use Net.Data-provided variables inside the REPORT block to access the data in the Net.Data macro results table. These variables are described in “Table Processing Variables” on page 96. For additional detail, see the Report Variables section in the *Net.Data Reference*.
- To provide header and footer information, provide the text before and after the ROW block. Net.Data processes everything it finds before a ROW block as header information. Net.Data processes everything it finds after the ROW block as footer information. As with the HTML block, Net.Data treats everything in the header, ROW, and footer blocks that is not recognized as macro language constructs as text presentation statements and sends these statements to the browser.
- You can call functions and reference variables in a REPORT block.
- To have Net.Data print a default report using pre-formatted text, do not include the REPORT block in the macro. The following example shows the default report format:

```
 SHIPDATE   | RECDATE    | SHIPNO   |
------------------------------------
 25/05/1997 | 30/05/1997 | 1495194B |
------------------------------------
 25/05/1997 | 28/05/1997 | 2942821G |
------------------------------------
```

- To use the HTML tags instead of the pre-formatted text, set DTW_HTML_TABLE to YES.
- To disable the printing of the a default report, set DTW_DEFAULT_REPORT to NO or by specifying an empty REPORT block. For example:

```
%REPORT{%}
```

Example: Customizing a Report

The following example shows how you can customize report formats using special variables and HTML tags. It displays the names, phone numbers, and FAX numbers from the table `CustomerTbl`:

```
%DEFINE SET_TOTAL_ROWS="YES"
...
  %FUNCTION(DTW_SQL) custlist() {
    SELECT Name, Phone, Fax FROM CustomerTbl
    %REPORT{
<I>Phone Query Results:</I>
<BR>
=====================
<BR>
    %ROW{
Name: <B>$(V1)</B>
<BR>
Phone: $(V2)
<BR>
Fax: $(V3)
<BR>
------------------------------
<BR>
    %}
    Total records retrieved: $(TOTAL_ROWS)
  %}
%}
```

The resulting report looks like this in the Web browser:

```
Phone Query Results:
====================
Name: Doen, David
Phone: 422-245-1293
Fax: 422-245-7383
------------------------------
Name: Ramirez, Paolo
Phone: 955-768-3489
Fax: 955-768-3974
------------------------------
Name: Wu, Jianli
Phone: 525-472-1234
Fax: 525-472-1234
------------------------------
Total records retrieved: 3
```

Net.Data generated the report by:

1. Printing *Phone Query Results:* once at the beginning of the report. This text, along with the separator line, is the header part of the REPORT block.
2. Replacing the variables `V1`, `V2`, and `V3` with their values for Name, Phone, and Fax respectively for each row as it is retrieved.
3. Printing the string *Total records retrieved:* and the value for `TOTAL_ROWS` once at the end of the report. (This text is the footer part of the REPORT block.)

Multiple REPORT Blocks

You can specify multiple REPORT blocks within a single FUNCTION or MACRO FUNCTION block to generate multiple reports with one function call.

Typically, you would use multiple REPORT blocks with the DTW_SQL language environment with a function that calls a stored procedure, which returns multiple result sets (see “Stored Procedures” on page 125). However, multiple REPORT blocks can be used with any language environment to generate multiple reports.

To use multiple REPORT blocks, place a result set name on the stored procedure CALL for each result set. If more result sets are returned from the stored procedure than the number of REPORT blocks you have specified, and if the Net.Data built-in function DTW_DEFAULT_REPORT = ″MULTIPLE″, then default reports are generated for each table that is not associated with a report block. If no report blocks are specified, and if DTW_DEFAULT_REPORT = ″YES″, then only one default report will be generated. Note that for the SQL language environment only, a DTW_DEFAULT_REPORT value of ″YES″ is equivalent to a value of ″MULTIPLE″.

Examples: The following examples demonstrate ways in which you can use multiple report blocks.

To display multiple reports using default report formatting:

Example 1: DTW_SQL language environment

```
%DEFINE DTW_DEFAULT_REPORT = "MULTIPLE"
%FUNCTION (dtw_sql) myStoredProc () {
     CALL myproc (table1, table2)  %}
```

In this example, the stored procedure `myproc` returns two result sets, which are placed in `table1` and `table2`. Because no REPORT blocks are specified, default reports are displayed for both tables, `table1` first, then `table2`.

Example 2: MACRO_FUNCTION block. In this example, two tables are passed into the MACRO_FUNCTION block. When DTW_DEFAULT_REPORT=″MULTIPLE″ is specified, Net.Data generates reports for both tables.

```
%DEFINE DTW_DEFAULT_REPORT = "MULTIPLE"
%MACRO_FUNCTION multReport (INOUT tablename1, tablename2) {
%}
```

In this example, two tables are passed into the MACRO_FUNCTION `multReport`. Again, Net.Data displays default reports for the two tables in the order in which they appear in the MACRO FUNCTION block parameter list, `table1` first, then `table2`.

Example 3: DTW_REXX language environment

```
%DEFINE DTW_DEFAULT_REPORT = "YES"
%FUNCTION (dtw_rexx) multReport (INOUT table1, table2) {
     SAY 'Generating multiple default reports...<BR>'
%}
```

In this example, two tables are passed into the REXX function `multReport`. Because DTW_DEFAULT_REPORT=″YES″ is specified, Net.Data displays a default report for the first table only.

To display multiple reports by specifying REPORT blocks for display processing:

Example 1: Named REPORT blocks

```
%FUNCTION(dtw_sql) myStoredProc () {
   CALL myproc (table1, table2)

   %REPORT(table2) {
     ...
     %ROW {  ....   %}
     ...
   %}

   %REPORT(table1) {
     ...
     %row {  ....   %}
     ...
   %}
%}
```

In this example, REPORT blocks have been specified for both of the tables passed in the FUNCTION block parameter list. The tables are displayed in the order they are specified on the REPORT blocks, `table2` first, then `table1`. By specifying a table name on the REPORT block, you can control the order in which the reports are displayed.

Example 2: Unnamed REPORT blocks

```
%FUNCTION(dtw_sql) myStoredProc  () {
   CALL myproc

   %REPORT {
     ...
     %ROW {  ....   %}
     ...
   %}
   %REPORT {
     ...
     %ROW {  ....   %}
     ...
   %}
%}
```

In this example, REPORT blocks have been specified for both of the tables passed in the FUNCTION block parameter list. Because there are no table names specified on the REPORT blocks, reports are displayed for the two tables in the order in which they are returned from the stored procedure.

To display multiple reports using a combination of default reports and REPORT blocks:

Example: A combination of default reports and REPORT blocks

```
%DEFINE DTW_DEFAULT_REPORT = "MULTIPLE"
%FUNCTION(dtw_system) editTables (INOUT table1, table2, table3) {
    %REPORT(table2) {
    ...
    %ROW {  ....   %}
    ...
  %}
%}
```

In this example, only one REPORT block is specified. Because the block specifies `table2`, and `table2` is the second result set listed on the CALL statement, the second result set is used to display the report. Because there are fewer REPORT blocks specified than the number of result sets returned from the stored procedure,

default reports are then displayed for the remaining result sets: first, a default report for the first result set, `table1`; then a default report for the third result set, `table3`. One output table is specified, `table1`, which can be used for processing later in the macro file.

Guidelines and Restrictions for Multiple REPORT Blocks: Use the following guidelines and restrictions when specifying multiple REPORT blocks in a FUNCTION or MACRO_FUNCTION block.

Guidelines:

- You can specify one or more REPORT block per result set or table name. The name specified for the REPORT block must match a corresponding result set name or table name parameter in the CALL statement or the FUNCTION block parameter list.
- Specify REPORT blocks for multiple tables in the order in which you want them to be processed.
- To specify default processing when there is not a REPORT block specified for a table, define DTW_DEFAULT_REPORT = ″MULTIPLE″. When Net.Data builds the Web page, it displays default reports for tables after it displays the reports for tables having REPORT blocks.
- To prevent Net.Data from displaying tables that do not have REPORT blocks, set DTW_DEFAULT_REPORT = ″NO″.
- When using the DTW_SAVE_TABLE_IN variable with a function that returns more than one table, the first table returned from the function is assigned to the DTW_SAVE_TABLE_IN table.
- Multiple report blocks can be used with any language environment.

Restrictions:

- The values of all report variables in a function apply to all the REPORT blocks in that function. You cannot modify the value of a report variable for individual REPORT blocks.
- The MESSAGE block must be located either before or after a list of REPORT blocks, and not between REPORT blocks.
- Table variables must be defined within the TABLE statement before being passed to the function.
- If the first report block specifies a table name, then all report blocks must specify table names.
- If the first report block does not specify a table name, then none of the report blocks can specify table names.
- The maximum number of tables for a single stored procedure is 32.

Conditional Logic and Looping in a Macro

Net.Data lets you incorporate conditional logic and looping in your Net.Data macros using the IF and WHILE blocks.

IF and WHILE blocks use a condition list that helps you test one or more conditions, and then to perform a block of statements based on the outcome of the condition test. The condition list contains logical operators, such as = and <+, and terms, which are made up of quoted strings, variables, variable references, and function calls. Quoted strings can contain variable references and functions calls, as well. You can nest the condition list.

The following sections describe conditional logic and looping:

Conditional Logic: IF Blocks

Use the IF block for conditional processing in a Net.Data macro. The IF block is similar to IF statements in most high-level languages because it provides the ability to test one or more conditions, and then to perform a block of statements based on the outcome of the condition test.

You can specify IF blocks almost anywhere in a macro and can nest them. The syntax of an IF block is shown in the language constructs chapter in *Net.Data Reference*.

IF Block Rules: The rules for IF block syntax are determined by the block’s position in the macro. The elements allowed in the executable block of statements of an IF block depend on the location of the IF block itself.

- Any element that is valid in the block containing the IF block is valid within that IF block. For example, if you specify an IF block inside an HTML block, any element that is allowed in the HTML block is allowed in the IF block, such as INCLUDE statements and WHILE blocks.

```
%HTML block
...
  %IF block
...
    %INCLUDE
...
    %WHILE
...
  %ENDIF
%}
```

- Similarly, if you specify the IF block outside of any other block in the declaration part of the Net.Data macro, only those elements allowed outside of any other block (such as a DEFINE block or FUNCTION block) are allowed in the IF block.

```
%IF
...
  %DEFINE
...
  %FUNCTION
...
%ENDIF
```

- When a IF block is nested within an IF block that is outside of any other block in the declaration part, it can use any element that the outside block can use. When an IF block is nested within another block that is in an IF block, it takes on the syntax rules for the block it is inside.

  For example, a nested IF block must follow the rules used when it is inside an HTML block.

```
%IF
...
  %HTML {
...
     %IF
...
     %ENDIF
  %}
...
%ENDIF
```

Exception: Do not specify a ROW block in an IF block.

IF Block String Comparison

Net.Data processes the IF block condition list in one of two ways based on the contents of the terms making up the conditions. The default action is to treat all terms as strings, and to perform string comparisons as specified in the conditions. However, if the comparison is between two strings representing integers, then the comparison is numeric. Net.Data assumes a string is numeric if it contains only digits, optionally preceded by a '+' or '-' character. The string cannot contain any non-digit characters other than the '+' or '-'. Net.Data does not support numerical comparison of non-integer numbers.

Examples of valid integer strings:

```
+1234567890
-47
000812
92000
```

Examples of invalid integer strings:

```
-  20     (contains blank characters)
234,000   (contains a comma)
57.987    (contains a decimal point)
```

Net.Data evaluates the IF condition at the time it executes the block, which can be different than the time it is originally read by Net.Data. For example, if you specify an IF block in a REPORT block, Net.Data does not evaluate the condition list associated with the IF block when it reads the FUNCTION block definition containing the REPORT block, but rather when it calls the function and executes it. This is true for both the condition list part of the IF block and the block of statements to be executed.

IF Block Example: A macro containing IF blocks inside other blocks

```
%{ This macro is called from another macro, passing the operating system
   and version variables in the form data.
%}

%IF (platform == "AS400")
  %IF (version == "V3R2")
     %INCLUDE "as400v3r2_def.hti"
  %ELIF (version == "V3R7")
     %INCLUDE "as400v3r7_def.hti"
  %ELIF (version == "V4R1")
     %INCLUDE "as400v4r1_def.hti"
  %ENDIF
%ELSE
     %INCLUDE "default_def.hti"
%ENDIF

%MACRO_FUNCTION numericCompare(IN term1, term2, OUT result) {
  %IF (term1 < term2)
    @dtw_assign(result, "-1")
  %ELIF (term1 > term2)
    @dtw_assign(result, "1")
  %ELSE
    @dtw_assign(result, "0")
  %ENDIF
%}

%HTML(report){
  %WHILE (a < "10") {
    outer while loop #$(a)<BR>
```

```
    %IF (@dtw_rdivrem(a,"2") == "0")
      this is an even number loop<BR>
    %ENDIF
    @DTW_ADD(a, "1", a)
  %}
%}
```

Looping Constructs: WHILE Blocks

Use the WHILE block to perform looping in a Net.Data macro. Like the IF block, the WHILE block provides the ability to test one or more conditions, and then to perform a block of statements based on the outcome of the condition test. Unlike the IF block, the block of statements can be executed any number of times based on the outcome of the condition test.

You can specify WHILE blocks inside HTML blocks, REPORT blocks, ROW blocks, MACRO_FUNCTION blocks, and IF blocks, and you can nest them. The syntax of a WHILE block is shown in the language constructs chapter of *Net.Data Reference*.

Net.Data processes the WHILE block exactly the same way it processes the IF block, but re-evaluates the condition after each execution of the block. And, like any conditional looping construct, it is possible for processing to go into an infinite loop if the condition is coded incorrectly.

Example: A macro with a WHILE block

```
%DEFINE loopCounter = "1"

%HTML(build_table) {
  %WHILE (loopCounter <= "100") {
    %{ generate table tag and column headings %}
    %IF (loopCounter == "1")
      <TABLE BORDER>
      <TR>
      <TH>Item #
      <TH>Description
    %ENDIF

    %{ generate individual rows %}
    <TR>
    <TD>$(loopCounter)
    <TD>@getDescription(loopCounter)

    %{ generate end table tag %}
    %IF (loopCounter == "100")
    %ENDIF

    %{ increment loop counter %}
    @DTW_ADD(loopCounter, "1", loopCounter)
  %}
%}
```

Chapter 6. Using Language Environments

Net.Data supplies language environments that you use to access data sources and to execute application programs containing business logic. For example, the SQL language environment lets you pass SQL statements to a DB2 database, and the REXX language environment lets you invoke REXX programs. You can also use the SYSTEM language environment to execute a program or issue a command.

With Net.Data, you can add user-written language environments in a pluggable fashion. Each user-written language environment must support a standard set of interfaces that are defined by Net.Data and must be implemented as a dynamic link library (DLL) or a shared library. For complete details on Net.Data-supplied language environments and on how to create a user-written language environment, see the *Net.Data Language Environment Interface Reference*.

Figure 21 shows the relationship between the Web server, Net.Data, and the Net.Data language environments.

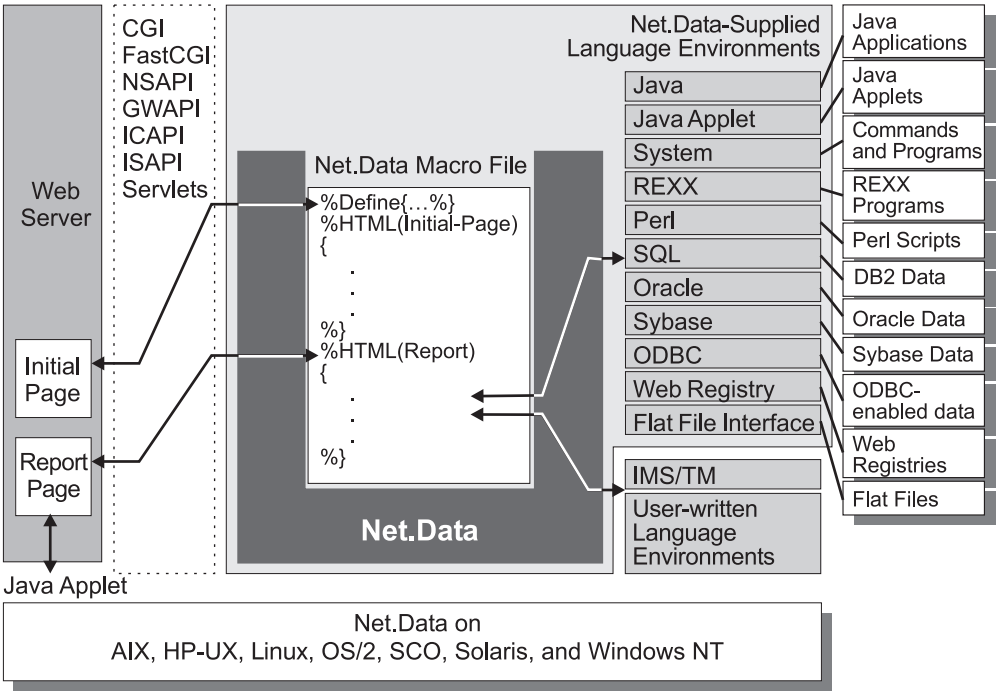


Figure 21. The Net.Data Language Environments

The following sections describe the Net.Data language environments and how to use them in your macros:

- "Overview of Net.Data-Supplied Language Environments" on page 118
- "Calling a Language Environment" on page 119
- "Data Language Environments" on page 119
- "Programming Language Environments" on page 138

For configuration information about the Net.Data-provided language environments, see "Setting Up the Language Environments" on page 23.

For information about improving performance when using the language environments, see “Optimizing the Language Environments” on page 181.

Overview of Net.Data-Supplied Language Environments

Net.Data provides language environments that let you access data and programming resources for your application.

Net.Data provides two types of language environments:

- “Data Language Environments” on page 119
- “Programming Language Environments” on page 138

Table 8 provides a brief description of each language environment. See the operating system appendix of *Net.Data Reference* to learn which language environments are support on what operating system.

Table 8. Net.Data Language Environments

Language Environment	Environment Name	Description
Flat File Interface	DTW_FILE	The flat file interface (FFI) provides functions that support text files as data sources.
IMS Web	HWS_LE	The IMS Web language environment lets you submit an IMS transaction using IMS Web and receive the output of the transaction at your Web browser.
Java Applet	DTW_APPLET	The Java applet language environment lets you use Java applets in your Net.Data applications. To generate an applet tag, you must provide the applet tag’s qualifiers and the applet’s parameter list.
Java Application	DTW_JAVAPPS	Net.Data supports your existing Java applications with the Java language environment.
ODBC	DTW_ODBC	The ODBC language environment executes SQL statements through an ODBC interface for access to multiple database management systems.
Oracle	DTW_ORA	The Oracle language environment lets you directly access your Oracle data.
Perl	DTW_PERL	The Perl language environment interprets internal Perl scripts that are specified in a FUNCTION block of the Net.Data macro, or it executes external Perl scripts stored in separate files.
REXX	DTW_REXX	The REXX language environment interprets internal REXX programs that are specified in a FUNCTION block of the Net.Data macro, or it can execute external REXX programs stored in a separate file.
SQL	DTW_SQL	The SQL language environment executes SQL statements through DB2. The results of the SQL statement can be returned in a table variable.
Sybase	DTW_SYB	The Sybase language environment lets you directly access your Sybase data.
System	DTW_SYSTEM	The System language environment supports executing commands and calling external programs.

Table 8. Net.Data Language Environments (continued)

Language Environment	Environment Name	Description
Web Registry	DTW_WEBREG	The Web Registry language environment provides functions for the persistent storage of application-related data.

Calling a Language Environment

To call a language environment:

- Use a FUNCTION statement to define a function that calls the language environment.
- Use a function call to the language environment.

For example:

```
%FUNCTION(DTW_SQL) custinfo() {
 select CUSTNAME, CUSTNO from ibmuser.customer
 %}
 ...
 %HTML(REPORT) {
 @custinfo()
 %}
```

Handling Error Conditions

When an error is detected in a language environment function, the language environment sets the Net.Data RETURN_CODE variable with an error code.

You can use the following resources to handle error conditions:

- The Net.Data-supplied language environments return error codes that are documented in *Net.Data Messages and Codes Reference*.
- The database language environments, such as the SQL language environment, set the RETURN_CODE to error codes that are returned by the database management system (DBMS), called SQLCODEs. See the messages and codes documentation for your DBMS to learn more about the SQLCODEs used by your DBMS.

Security

Ensure that the user ID that Net.Data is running under has the proper authority to access any object that may be referenced by the target of a language environment statement. For example, SQL language environment runs SQL statements, and SQL statements access database files, so the user ID that Net.Data is running under must have authority to the database files.

Data Language Environments

The data language environments provided by Net.Data enable you to access data from relational and hierarchical databases, and other data sources from a Net.Data macro. The following sections discuss the Net.Data-provided data language environments and how to use them in your Net.Data macros.

- "Relational Database Language Environments" on page 120
- "Flat File Interface Language Environment" on page 134

- "Web Registry Language Environment" on page 135
- "IMS Web Language Environment" on page 137

Relational Database Language Environments

Net.Data provides relational database language environments to help you access your relational data sources. Net.Data provides the following relational database language environments:

ODBC Language Environment

The Open Database Connectivity (ODBC) language environment executes SQL statements through an ODBC interface. ODBC is based on the X/Open SQL CAE specification, which lets a single application access many database management systems.

To use the ODBC language environment:

Have an ODBC driver and a driver manager. Your ODBC driver documentation describes how to install and configure your ODBC environment.

Verify that the following configuration statement is in the initialization file, on one line.

```
ENVIRONMENT (DTW_ODBC) DTWODBC ( IN DATABASE, LOGIN, PASSWORD,
  TRANSACTION_SCOPE, SHOWSQL, ALIGN, START_ROW_NUM, DTW_SET_TOTAL_ROWS)
```

Restrictions:

- When specifying the DATABASE variable, you must specify the same database as the data source in the ODBC initialization file.
- SQL statements in the inline statement block can be up to 64 KB. DB2 Universal Database has the following restrictions:
  - Version 6 or higher: 64 KB
  - Version 5 Release 2 or lower: 32 KB

  Your database might have different limits; refer to your database documentation to determine if your database has a different limit.

Oracle Language Environment

The Oracle language environment provides native access to your Oracle data. You can access Oracle databases from Net.Data when using CGI, FastCGI, NSAPI, ISAPI, ICAPI or GWAPI. This language environment supports Oracle 7.2, 7.3, and 8.0.

To use the Oracle language environment:

Verify that the following configuration statement is in the initialization file, on one line.

```
ENVIRONMENT (DTW_ORA) DTWORA (IN DATABASE, LOGIN, PASSWORD,
  TRANSACTION_SCOPE, SHOWSQL, ALIGN, START_ROW_NUM, DTW_SET_TOTAL_ROWS)
```

Important: See "Setting up the Oracle Language Environment" on page 25 to learn how to set up the Oracle language environment.

Restrictions:

The Oracle language environment has the following restrictions:

- The DATABASE variable is not used to access Oracle databases.

- The LOGIN variable must contain the Oracle database instance name. For example, *ora73* is the defined instance name in the following LOGIN variable:

  ```
  LOGON=admin@ora73
  ```

- You must use Live Connection when using an interface other than CGI.
- Stored procedures are not supported.

SQL Language Environment

The SQL language environment provides access to DB2 databases. Use this language environment for optimal performance when accessing DB2.

To use the SQL language environment:

Verify that the following configuration statement is in the initialization file, on one line.

```
ENVIRONMENT (DTW_SQL) DTWSQL (IN DATABASE, LOGIN, PASSWORD,
  TRANSACTION_SCOPE, SHOWSQL, ALIGN, START_ROW_NUM, DTW_SET_TOTAL_ROWS)
```

Restriction:

SQL statements in the inline statement block can be up to 64 KB. DB2 Universal Database has the following restrictions:

- Version 6 or higher: 64 KB
- Version 5 Release 2 or lower: 32 KB

Your database might have different limits; refer to your database documentation to determine if your DBMS has a different limit.

Sybase Language Environment

The Sybase language environment provides native access to your Sybase data. You can access Sybase databases from Net.Data when using CGI, FastCGI, NSAPI, ISAPI, ICAPI or GWAPI mode.

To use the Sybase language environment:

Verify that the following configuration statement is in the initialization file, on one line.

```
ENVIRONMENT (DTW_SYB)      DTWSYB    ( IN DATABASE, LOGIN, PASSWORD,
  TRANSACTION_SCOPE, SHOWSQL, ALIGN, START_ROW_NUM, DTW_SET_TOTAL_ROWS)
```

Important: See “Setting up the Sybase Language Environment” on page 27 to learn how to set up the Sybase language environment.

Restrictions:

The Sybase language environment has the following restrictions:

- The Sybase language environment does not support large objects, such as images and audio. Stored procedures are supported only for procedures without a SELECT statement.
- SQL statements in the inline statement block can be up to 64KB. Your database might have different limits; refer to your database documentation to determine if your database has a lower limit.
- You must use Live Connection when using an interface other than CGI.
- Stored procedures are not supported.

The following sections describe how to use these language environments:

- “Managing Transactions in a Net.Data Application” on page 122

Managing Transactions in a Net.Data Application

When you modify the content of a database using insert, delete, or update statements, the modifications do not become persistent until the database receives a commit statement from Net.Data. If an error occurs, Net.Data sends a rollback statement to the database, reversing all modifications since the last commit.

The way in which Net.Data sends the commit and possible rollback depends on how you set TRANSACTION_SCOPE and whether you specify the commit explicitly in the macro. The values for TRANSACTION_SCOPE are MULTIPLE and SINGLE.

MULTIPLE

Specifies that Net.Data will execute all SQL statements before a commit and possible rollback statement is issued. Net.Data sends the commit at the end of the request, and if each SQL statement is issued successfully, the commit makes all modifications in the database persistent. If any of the statements returns an error, Net.Data issues a rollback statement, which sets the database back to its original state. MULTIPLE is the default if TRANSACTION_SCOPE is not set.

To activate this commit method set TRANSACTION_SCOPE to MULTIPLE.

For example:

```
@DTW_ASSIGN(TRANSACTION_SCOPE,"MULTIPLE")
```

SINGLE

Specifies that Net.Data issues a commit statement after each successful SQL statement. If the SQL statement returns an error, a rollback statement is issued. Single transaction scope secures a database modification immediately; however, with this scope, it is not possible to undo a modification using a rollback statement later.

To activate this commit method, set TRANSACTION_SCOPE to SINGLE. For example:

```
@DTW_ASSIGN(TRANSACTION_SCOPE,"SINGLE")
```

You can issue a commit statement at the end of any SQL statement in your macro by using the COMMIT SQL statement. By leaving TRANSACTION_SCOPE set to MULTIPLE and issuing commit statements at the end of those groups of statements that you feel qualify as a transaction, you the application developer maintain full control over the commit and rollback behavior in your application. For example, issuing commit statements after each update in your macro can help ensure the integrity of your data.

To issue an SQL commit statement, you can define a function that you can call in at any point in your HTML block:

```
%FUNCTION(DTW_SQL) user_commit() {
  commit
%}

...
```

```
%HTML {
  ...
  @user_commit()
  ...
%}
```

Using Large Objects

You can store large object files (LOBs) in DB2 databases and access them using the SQL or ODBC language environment for your Web applications.

The SQL or ODBC language environment does not store large objects in Net.Data table processing variables (such as V1 or V2), or Net.Data table fields when a SQL query returns LOBs in a result set. Instead, when Net.Data encounters a LOB, it stores the LOB in a file that Net.Data creates. This file is in a directory specified by the HTML_PATH path configuration variable. The values of Net.Data table fields and table processing variables are set to the path of the file. Access to the file is limited to the user ID under which Net.Data is running.

The file name that the LOB is stored in is dynamically constructed, and has the following form:

name[.*extension*]

Where:

name Is a unique string identifying the large object

extension
Is a string that identifies the type of the object. For CLOBs and DBCLOBs, the extension is 'txt'. For BLOBs, the SQL language environment attempts to determine the extension by looking for a signature in the first few bytes of the LOB file that indicates what the LOB represents. The SQL language environment recognizes the following types of data (in parenthesis is the extension used):

- Bitmap image (.bmp)
- Graphical image format (.gif)
- JPEG image files (.jpg)
- Tagged image file format (.tif)
- Postscript (.ps)
- Musical instruments digital interface audio files (.mid)
- AIFF audio file (.aif)
- Audio visual interleave audio file (.avi)
- Basic audio files (.au)
- Real audio files (.ra)
- Windows audio visual files (.wav)
- Portable document format (.pdf)
- Midi sequence file (.rmi)

If the object type of the BLOB is not recognized, no extension is added to the file name.

When the LOB is referenced in the macro file, the SQL language environment returns the file name with the/`tmplobs/` string prepended to the LOB file name, using the following syntax:

/tmplobs/*name*.[*extension*]

Planning tip: Each query that returns LOBs results in files being created in the directory specified by the HTML_PATH path configuration variable. Consider system limitations when using LOBs because they can quickly consume resources. You might want to create a batch program that cleans the directory up periodically, or execute the `dtwclean` daemon. See "Managing Temporary LOBS" on page 125 for more information. It is recommended that you use DataLinks, which eliminate the need to store files in directories by the SQL language environment, resulting in better performance and the use of much less system resources.

Example: The application user must click on the file name to invoke the viewer because the application uses a MPEG audio (.MPA) file. The SQL language environment does not recognize this file type so an EXEC variable is used to append the extension to the file.

```
%DEFINE{
docroot="/usr/lpp/internet/server_root/html"
myFile=%EXEC "rename $(docroot)$(V3) $(docroot)$(V3).mpa"
%}

%{ where rename is the rename command on your operating system %}
%FUNCTION(DTW_SQL) queryData() {
SELECT Name, IDPhoto, Voice FROM RepProfile
%REPORT{
<P>Here is the information you selected:<P>
%ROW{
$(myFile)
$(V1) Voice sample <IMG SRC="$(V2)">
<A HREF="$(V3).mpa")>Voice sample</A><P>
%}
%}
%}

%HTML(REPORT){
@queryData()
%}
```

The `queryData` function returns the following HTML output:

```
<P>Here are the images you selected:<P>
Kinson Yamamoto
<IMG SRC="/tmplobs/p2345n1.gif">
<A HREF="/tmplobs/p2345n2.mpa">Voice sample</A><P>
Merilee Lau
<IMG SRC="/tmplobs/p2345n3.gif">
<A HREF="/tmplobs/p2345n4.mpa">Voice sample</A><P>
```

The REPORT block in the previous example uses the implicit table variables V1, V2, and V3.

- V1 is a person's name, which is plain text.
- V2 is a photo of the person in a .GIF file. The image is shown inline. The SQL language environment includes the prefix `/tmplobs/` and the .GIF extension automatically.
- V3 is a sample of the person's voice in a .MPA file. When Net.Data encounters an unrecognized file format, such as a .MPA file, it writes the file into the temporary LOB directory without a file extension. This example shows how to handle this file type by adding the extension using an EXEC variable. When the variable *$(V3)* is resolved, it has the path `/tmplobs/` added before the file name. For example, `/tmplobs/sound2a`. One way to deal with such files is to write a

REXX program to change the slashes to back slashes and rename the files. The voice sample is played when the application user clicks on Voice sample.

Access rights for LOBs: The default `tmplobs` directory for LOBs is under the directory specified by the HTML_PATH in the shipped Net.Data initialization file. It is accessible by any user ID. If the the HTML_PATH value is changed, ensure that the user ID that the Web server is running under has write access to the directory specified by HTML_PATH (see "HTML_PATH" on page 21 for more information).

Managing temporary LOBs:

Net.Data stores temporary LOBs in a subdirectory called `tmplobs`, under the directory specified in the HTML_PATH path configuration variable. These files can be large and should be cleaned out periodically to maintain acceptable performance.

Net.Data provides a daemon called `dtwclean` that helps you periodically manage the `tmplobs` directory. `dtwclean` uses port 7127.

To run the `dtwclean` daemon: Enter the following command from the command line window:

```
dtwclean [-t xx] [-d|-l]
```

Where:

-t Is a flag that specifies the interval in which `dtwclean` cleans the directory

xx Is the interval, in seconds, at which a file remains in the directory before `dtwclean` erases the file. This value does not have a limit. The default is 3600 seconds.

-d Is a flag that specifies debug mode; trace information is displayed in the command window.

-l Is a flag that specifies logging mode; trace information is printed to a log file.

Stored Procedures

A stored procedure is a compiled program stored in DB2 that can execute SQL statements. In Net.Data, stored procedures are called from Net.Data functions using a CALL statement. Stored procedure parameters are passed in from the Net.Data function parameter list. You can use stored procedures to improve performance and integrity by keeping compiled SQL statements with the database server. Net.Data supports the use of stored procedures with DB2 through the SQL and ODBC language environments.

This section describes following topics:

Stored Procedure Syntax: The syntax of the stored procedure uses the FUNCTION statement, the CALL statement, and optionally a REPORT block.

```
%FUNCTION (DTW_SQL) function_name ([IN datatype arg1, INOUT datatype arg2,
    OUT tablename, ...]) {
  CALL stored_procedure [(resultsetname, ...)]
[%REPORT [(resultsetname)] { %}]
...
[%REPORT [(resultsetname)] { %}]
[%MESSAGE %}]

%}
```

Where:

function_name
: Is the name of the Net.Data function that initiates the call of the stored procedure

stored_procedure
: Is the name of the stored procedure

datatype
: Is one of the database data types supported by Net.Data as shown in Table 9. The data types specified in the parameter list must match the data types in the stored procedure. See your database documentation for more information about these data types.

tablename
: Is the name of a Net.Data table in which the result set is to be stored (used only when the result set is to be stored in a Net.Data table). If specified, this parameter name must match the associated parameter name for *resultsetname*.

resultsetname
: Is the name that associates a result returned from a stored procedure with a REPORT block and a table name on the function parm list, or both. The *resultsetname* on a REPORT block must match a result set on the CALL statement.

Table 9. Stored Procedures Data Types

BIGINT	DOUBLEPRECISION	SMALLINT
CHAR	FLOAT	TIME
DATE	INTEGER	TIMESTAMP
DECIMAL	GRAPHIC	VARCHAR
DOUBLE	LONGVARCHAR	VARGRAPHIC
	LONGVARGRAPHIC	

Calling a Stored Procedure:

1. Define a function that initiates a call to the stored procedure.

   ```
   %FUNCTION (DTW_SQL) function_name()
   ```

2. Optionally, specify any IN, INOUT, or OUT parameters for the stored procedure, including a table variable name for storing a result set in a Net.Data table (you only need to specify a Net.Data table if you want the result set stored in a Net.Data table). You can also specify as the table names or result sets, as IN or INOUT parameters, from another stored procedure.

   ```
   %FUNCTION (DTW_SQL) function_name (IN datatype
   arg1, INOUT datatype arg2,
       OUT tablename...)
   ```

3. Use the CALL statement to identify the stored procedure name.

   ```
   CALL stored_procedure
   ```

4. If the stored procedure is going to generate one result set, optionally specify a REPORT block to define how Net.Data displays the result set.

```
%REPORT (resultsetname) {
...
%}
```

   Example:

```
%FUNCTION (DTW_SQL) mystoredproc (IN CHAR(30) arg1)  {
     CALL myproc
 %REPORT (mytable){
  ...
  %ROW {  ...   %}
  ...
 %}
%}
```

5. If the stored procedure is going to generate more than one result set:
   - Specify the result set names on the CALL statement.

     ```
     CALL stored_procedure (resultsetname1, resultsetname2, ...)
     ```

   - Optionally specify one or more REPORT blocks to define how Net.Data displays the result sets.

     ```
     %REPORT(resultsetname1) {
     ...
     %}
     ```

     Example:

     ```
     %FUNCTION (DTW_SQL) mystoredproc (IN CHAR(30) arg1, OUT table1) {
        CALL myproc (table1, table2)
      %REPORT (table2) {
           ...
           %ROW {  ...   %}
           ...
      %}
     %}
     ```

Passing Parameters: You can pass parameters to a stored procedure and you can have the stored procedure update the parameter values so that the new value is passed back to the Net.Data macro. The number and type of the parameters on the function parameter list must match the number and type defined for the stored procedure. For example, if a parameter on the parameter list defined for the stored procedure is INOUT, then the corresponding parameter on the function parameter list must be INOUT. If a parameter on the list defined for the stored procedure is of type CHAR(30), then the corresponding parameter on the function parameter list must also be CHAR(30).

Example 1: Passing a parameter value to the stored procedure

```
%FUNCTION (DTW_SQL) mystoredproc (IN CHAR(30) valuein)  {
    CALL myproc
...
```

Example 2: Returning a value from a stored procedure

```
%FUNCTION (DTW_SQL) mystoredproc (OUT VARCHAR(9) retvalue)  {
  CALL myproc
...
```

Processing Result Sets: You can return one or more result sets from a stored procedure using the SQL or ODBC language environments. The result sets can be stored in Net.Data tables for further processing within your macro or processed using a REPORT block. If a stored procedure generates multiple result sets, you

must associate a name with each result set generated by the stored procedure. This is done by specifying parameters on the CALL statement. The name you specify for a result set can then be associated with a REPORT block or a Net.Data table, enabling you to determine how each result set is processed by Net.Data. You can:

- Have the result processed in Net.Data's default report style by not defining a report block for the result set.
- Associate a result set with a REPORT block to apply your own report style. In the REPORT block, you can use Net.Data variables, text processing statements like HTML or JavaScript, or other functions to specify how the report data is displayed in the browser.
- Store the result sets in Net.Data tables when you want Net.Data to use the data later in the macro. For example, you can pass the Net.Data table to another function so that it can use the data for calculations and display the results based on those calculations.

See "Guidelines and Restrictions for Multiple REPORT Blocks" on page 112 for guidelines and restrictions when using multiple report blocks.

To return a single result set and use default reporting:

Use the following syntax:

```
%FUNCTION (DTW_SQL) function_name () {
    CALL stored_procedure
%}
```

For example:

```
 %FUNCTION (DTW_SQL) mystoredproc() {
    CALL myproc
%}
```

To return a single result set and specify a REPORT block:

Use the following syntax:

```
%FUNCTION (DTW_SQL) function_name () {
     CALL stored_procedure  [(resultsetname)]
 %REPORT [(resultsetname)] {
  ...
 %}
%}
```

For example:

```
%FUNCTION (DTW_SQL) mystoredproc () {
     CALL  myproc
 %REPORT {
      ...
      %ROW {  ...   %}
      ...
 %}
%}
```

Alternatively, the following syntax can be used:

```
%FUNCTION (DTW_SQL) function_name () {
    CALL stored_procedure (resultsetname)
```

```
 %REPORT (resultsetname) {
  ...
 %}
%}
```

For example:

```
%FUNCTION (DTW_SQL) mystoredproc () {
    CALL myproc (mytable1)
 %REPORT (mytable1) {
  ...
  %ROW {  ...   %}
  ...
 %}
%}
```

To store a single result set in a Net.Data table for further processing:

Use the following syntax:

```
%FUNCTION (DTW_SQL) function_name (OUT tablename) {
   CALL stored_procedure (resultsetname)
%}
```

For example:

```
%DEFINE DTW_DEFAULT_REPORT = "NO"

%FUNCTION (DTW_SQL) mystoredproc (OUT mytable1) {
   CALL myproc (mytable1)
%}
```

Note that DTW_DEFAULT_REPORT is set to NO so that a default report is not generated for the result set.

To return multiple result sets and display them using default report formatting:

Use the following syntax:

```
%FUNCTION (DTW_SQL) function_name () {
    CALL stored_procedure [(resultsetname1, resultsetname2, ...)]
%}
```

Where no report block is specified.

For example:

```
%DEFINE DTW_DEFAULT_REPORT = "YES"
%FUNCTION (DTW_SQL) mystoredproc () {
    CALL myproc
%}
```

To return multiple result sets and have the result sets stored in Net.Data tables for further processing:

Use the following syntax:

```
 %FUNCTION (DTW_SQL) function_name (OUT tablename1, tablename2, ...) {
    CALL stored_procedure (resultsetname1, resultsetname2, ...)
%}
```

For example:

```
%DEFINE DTW_DEFAULT_REPORT = "NO"

%FUNCTION (DTW_SQL) mystoredproc (OUT mytable1, mytable2) {
     CALL myproc (mytable1, mytable2)
%}
```

Note that DTW_DEFAULT_REPORT is set to NO so that a default report is not generated for the result sets.

To return multiple result sets and specify REPORT blocks for display processing:

Each result set is associated with its one or more REPORT blocks. Use the following syntax:

```
%FUNCTION (DTW_SQL) function_name (, ...) {
   CALL stored_procedure (resultsetname1, resultsetname2, ...)
 %REPORT (tablename1)
   ...
   %ROW {  ...   %}
   ...
 %}
 %REPORT (tablename2)
   ...
   %ROW {  ...   %}
   ...
 %}

...
%}
```

For example:

```
%FUNCTION (DTW_SQL) mystoredproc () {
   CALL myproc (mytable1, mytable2)

 %REPORT(mytable1) {
  ...
  %ROW {  ...   %}
  ...
 %}

 %REPORT(mytable2) {
  ...
  %ROW {  ...   %}
  ...
 %}
%}
```

To return multiple result sets and specify different display or processing options for each result set:

You can specify different processing options for each result set using unique parameter names. For example:

```
%FUNCTION (DTW_SQL) mystoredproc (OUT mytable2) {
    CALL myproc (mytable1, mytable2, mytable3)

 %REPORT(mytable1)
 ...
 %ROW {  ...   %}
 ...
 %}
%}
```

The result set `mytable1` is processed by the corresponding REPORT block and is displayed as specified by the macro writer. The result set `mytable2` is stored in the Net.Data table `mytable2` and can now be used for further processing, such as being passed to another function. The result set `mytable3` is displayed using Net.Data's default report format because no REPORT block was specified for it.

Encoding DataLink URLs in Result Sets

The DataLink data type is one of the basic building blocks for extending the types of data that can be stored in database files. With DataLink, the actual data stored in the column is only a pointer to the file. This file can be any type of file; an image file, a voice recording, or a text file. DataLinks store a URL to resolve the location of the file.

The DATALINK data type requires the use of DataLink File Manager. For more information about the DataLink File Manager, see the DataLinks documentation for your operating system. Before you use the DATALINK data type, you must ensure that the Web server has access to the file system managed by the DB2 File Manager Server.

When a SQL query returns a result set with DataLinks, and the DataLink column is created with FILE LINK CONTROL with READ PERMISSION DB DataLink options, the file paths in the DataLink column contains an access token. DB2 uses the access token to authenticate access to the file. Without this access token, all attempts to access the file fail with an authority violation. However, the access token might include characters that are not usable in a URL to be returned to a browser, such as the semi-colon (;) character. For example:

```
/datalink/pics/UN1B;0YPVKGG346KEBE;baibien.jpg
```

The URL is not a valid because it contains semi-colon (;) characters. To make the URL valid, the semi-colons must be encoded using the Net.Data built-in function DTW_URLESCSEQ. However, some string manipulation must be done before applying this function because this function encodes slashes (/), as well.

You can write a Net.Data MACRO_FUNCTION to automate the string manipulation and use the DTW_URLESCSEQ function. Use this technique in every macro that retrieves data from a DATALINK data type column.

Example 1: A MACRO_FUNCTION that automates the encoding of URLs returned from DB2 UDB

```
%{ TO DO: Apply DTW_URLESCSEQ to a DATALINK URL to make it a valid URL.
    IN: DATALINK URL from DB2 File Manager column.
    RETURN: The URL with token portion is URL encoded
%}
%MACRO_FUNCTION encodeDataLink(in DLURL) {
  @DTW_rCONCAT( @DTW_rDELSTR( DLURL,
  @DTW_rADD(@DTW_rLASTPOS("/", DLURL), "1" ) ),
  @DTW_rURLESCSEQ( @DTW_rSUBSTR(DLURL,
  @DTW_rADD( @DTW_rLASTPOS("/", DLURL), "1" ) ) ) )
%}
```

After using this MACRO_FUNCTION, the URL is encoded properly and the file specified in the DATALINK column can be referenced on any Web browser.

Example 2: A Net.Data macro specifying the SQL query that returns the DATALINK URL

```
%FUNCTION(DTW_SQL)myQuery(){
  select name, DLURLCOMPLETE(picture) from myTable where name like '%river%'
  %REPORT{
    %ROW{
      <p> $(V1) <br>
      Before Encoding: $(V2) <br>
      After Encoding: @encodeDataLInk($(V2)) <br>
      Make HREF: <a href="@encodeDataLink($(V2))"> click here </a> <br> <p>
    %}
  %}
%}
```

Note that a DataLink File Manager functions is used. The function `dlurlcomplete` returns a full URL.

Relational Database Language Environment Examples

The following examples show how you can call the relational database language environments from your macros:

ODBC

The following example defines and calls multiple function for the ODBC language environment.

```
%DEFINE {
   DATABASE="qesq1"
   SHOWSQL="YES"
   table="int_null"
   LOGIN="netdata1"
   PASSWORD="ibmdb2"%}

%function(dtw_odbc) sql1() {
create table int_null (int1 int, int2 int)
%}

%function(dtw_odbc) sql2() {
insert into $(table) (int1) values (111)
%}

%function(dtw_odbc) sql3() {
insert into $(table) (int2) values (222)
%}

%function(dtw_odbc) sql4() {
select * from $(table)
%}

%function(dtw_odbc) sql5() {
drop table $(table)
%}

%HTML(REPORT) {
@sql1()
@sql2()
@sql3()
@sql4()
%}
```

Oracle

The following example shows a macro with a DTW_ORA function definition that queries the Oracle database, `udatabase`, using a variable reference to determine the database table to be queried. The FUNCTION block also contains a MESSAGE block that handles error conditions. When Net.Data processes the macro, it displays a default report at the browser.

```
%DEFINE {
   LOGIN="ulogin"
   PASSWORD="upassword"
   DATABASE="udatabase"
   table= "utable"
%}


%FUNCTION(DTW_ORA) myQuery(){
select ename,job,empno,hiredate,sal,deptno from $(table) order by ename
%}
%MESSAGE{
100 : "<b>WARNING</b>: No employee were found that met your search criteria.<p>"
    : continue
%}

%HTML (REPORT) {
@myQuery()
%}
```

SQL

The following example shows a macro with a DTW_SQL function definition that calls an SQL stored procedure. It has three parameters of different data types. The DTW_SQL language environment passes each parameter to the stored procedure in accordance with the data type of the parameter. When the stored procedure completes processing, output parameters are returned and Net.Data updates the variables accordingly.

```
%{************************************************************
                    DEFINE BLOCK
************************************************************%}
%DEFINE {
 MACRO_NAME     = "TEST ALL TYPES"
 DTW_HTML_TABLE = "YES"
 Procedure      = "TESTTYPE"
 parm1          = "1"              %{SMALLINT      %}
 parm2          = "11"             %{INT         %}
 parm3          = "1.1"            %{DECIMAL (2,1)  %}
 %}

 %FUNCTION(DTW_SQL)   myProc
  (INOUT SMALLINT     parm1,
   INOUT INT            parm2,
   INOUT DECIMAL(2,1)    parm3){
CALL $(Procedure)
%}
%HTML(REPORT) {
<HEAD>
<TITLE>Net.Data : SQL Stored Procedure: Example '$(MACRO_NAME)'. </TITLE>
</HEAD>
<BODY BGCOLOR="#BBFFFF" TEXT="#000000" LINK="#000000">
<p><p>
Calling the function to create the stored procedure.
<p><p>
 @CRTPROC()
<hr>
<h2>
Values of the INOUT parameters
 prior to calling the stored procedure:<p>
</h2>
<b>parm1  (SMALLINT)</b><br>
$(parm1)<p>
<b>parm2  (INT)</b><br>
$(parm2)<p>
```

```
<b>parm3  (DECIMAL)</b><br>
$(parm3)<p>
<p>
<hr>
<h2>
Calling the function that executes the stored procedure.
</h2>
<p><p>
 @myProc(parm1,parm2,parm3)
<hr>
<h2>
Values of the INOUT parameters after
calling the stored procedure:<p>
</h2>
<b>parm1  (SMALLINT)</b><br>
$(parm1)<p>
<b>parm2  (INT)</b><br>
$(parm2)<p>
<b>parm3  (DECIMAL)</b><br>
$(parm3)<p>
</body>
%}
```

Sybase

The following example shows a macro with a DTW_SYB function definition that queries the Sybase database, `udatabase`, using a variable reference to determine the database table to be queried. The FUNCTION block also contains a MESSAGE block that handles error conditions. When Net.Data processes the macro, it displays a default report at the browser.

```
%DEFINE {
   LOGIN="ulogin"
   PASSWORD="upassword"
   DATABASE="udatabase"
   table= "utable"
%}


%FUNCTION(DTW_SYB) myQuery(){
select ename,job,empno,hiredate,sal,deptno from $(table) order by ename
%}
%MESSAGE{
100 : "<b>WARNING</b>: No employee were found that met your search criteria.<p>"
    : continue
%}

%HTML (REPORT) {
@myQuery()
%}
```

Flat File Interface Language Environment

If you choose to use flat files (or plain-text files) as your data source, use the flat file interface (FFI) and its associated functions to open, close, read, write, and delete files on the Web server. The file language support uses FFI functions to read from or write to files on the Web server at the Web client's request through the browser. FFI views the file as a record file, each record equivalent to a row in a Net.Data macro table variable, and each value in a record equivalent to a field value in a Net.Data macro table variable. FFI reads records from a file into rows of a Net.Data macro table, and writes rows from a table into records.

See *Net.Data Reference* for description and syntax of the FFI built-in functions.

Configuring the FFI Language Environment

Verify that the following configuration statement is in the initialization file, on one line:

```
ENVIRONMENT (DTW_FILE)   DTWFILE   ( OUT RETURN_CODE )
```

See “Environment Configuration Statements” on page 21 to learn more about the Net.Data initialization file and language environment ENVIRONMENT statements.

Calling FFI Built-in Functions

Call an FFI function as you would any other function. Use a DEFINE statement to define as variables any of the parameters that you want to pass; for example:

```
%DEFINE {
    myFile = "c:/private/myfile"
    myTable = %TABLE
    myWait = "1500"
    myRows = "2"
 %}
```

Then use a function call statement to invoke the function; for example:

```
@DTWF_UPDATE(myFile, "Delimited", "|", myTable, myWait, myRows)
```

Example

In this example, Net.Data reads the content of the `ffi001.dat` file into a Net.Data table and writes the content of this table into the `tmp.dat` file. Finally, Net.Data deletes the `tmp.dat` file.

```
%DEFINE {
mytable = %TABLE(ALL)
myfile  = "/usr/lpp/netdata/ffi//ffi001.dat"
tmpfile = "/usr/lpp/netdata/ffi/tmp.dat"
%}
%HTML(report) {
@DTWF_READ(myfile, "ASCIITEXT", " ", mytable)
@DTW_TB_TABLE(mytable)

@DTWF_WRITE(tmpfile, "ASCIITEXT", " ", mytable)
@DTW_TB_TABLE(mytable)

@DTWF_REMOVE(tmpfile)
%}
```

Web Registry Language Environment

The Net.Data Web registry provides persistent storage for application-related data. A Web registry can be used to store configuration information and other data that can be accessed dynamically at run time by Web-based applications. You can access Web registries only through Net.Data macros using Net.Data and the Web registry built-in support and from CGI programs written for this purpose. The Web registry is available on a subset of operating systems. See *Net.Data Reference* for description and syntax of the Web registry built-in function, as well as list of operating systems that support the language environment.

Standard Web page development requires that URLs be placed directly in the HTML source for the page. This makes changing links difficult. The static nature

also limits the type of links that can be easily placed on a Web page. Using a Web registry to store application-related data, for example URLs, can help in the creation of HTML pages with dynamically set links.

Information can be stored and maintained in a registry by application developers and Web administrators who have write access to the registry. Applications retrieve the information from their associated registries at run time. This facilitates the design of flexible applications and also allows movement of applications and servers. You can use Net.Data macros to create HTML pages using dynamically set links.

Information is stored in a Web registry in the form of registry entries. Each registry entry consists of a pair of character strings: a RegistryVariable string and a corresponding RegistryData string. Any information that can be represented by a pair of strings can be stored as a registry entry. Net.Data uses the variable string as a search key to locate and retrieve specific entries from a registry.

Table 10 displays a sample Web registry:

Table 10. Sample Web Registry

CompanyName	**WorldConnect**
Server	ftp.einet.net
JohnDoe/foreground	Green
CompanyURL/IBM Corp.	http://www.ibm.com
CompanyURL/Sun Microsystems Corp.	http://www.sun.com
CompanyURL/Digital Equipment Corp.	http://www.dec.com
JaneDoe/Home_page	http://jane.info.net

Reasons to consider using a Web registry:

- You can use a Web registry to store aliases for servers and URLs, facilitating the relocation of applications and servers.
- Application developers can ship their Web-based applications with data, such as URLs, predefined in the registry. The end user can modify the registry data to change the behavior of the application.
- A Web registry can be used to perform URL searches based on product name, national language, manufacturer, and so on.

Indexed entries in the Web Registry are entries whose RegistryVariable strings have an additional Index string appended to them, using the following syntax:

```
RegistryVariable/Index
```

The user provides the value of the index string in a separate parameter to a built-in function designed to work with indexed entries. Multiple indexed registry entries can have the same RegistryVariable string value, but they can maintain their uniqueness by having different Index string values.

Table 11. Sample Indexed Web Registry

Smith/Company_URL	http://www.ibmlink.ibm.com
Smith/Home_page	http://www.advantis.com

Even though the above two indexed entries have the same RegistryVariable string value `Smith`, the index string is different in each case. They are treated as two distinct entries by the Web registry functions.

Configuring the Web Registry Language Environment

Verify that the following configuration statement is in the initialization file, on one line:

```
ENVIRONMENT (DTW_WEBREG)   DTWWEB    ( OUT RETURN_CODE )
```

See “Environment Configuration Statements” on page 21 to learn more about the Net.Data initialization file and language environment ENVIRONMENT statements.

Calling Web Registry Built-in Functions

Call an Web Registry function as you would any other function. Use a DEFINE statement to define as variables any of the parameters that you want to pass. For example:

```
%DEFINE {
    name = "smith"
 %}
```

Then use a function call statement to invoke the function; for example:

```
@DTWR_ADDENTRY("URLLIST", name, "http://www.software.ibm.com/",
    "WORK_URL"
```

Example

The following example creates a Web registry and adds entries. It then displays a report containing the entries.

```
%DEFINE {
RegTable = %TABLE(ALL)
%}

%MESSAGE {
    default:"<p>Function Error: Return code: $(RETURN_CODE)." :continue
%}

%FUNCTION(DTW_WEBREG) ListTable(INOUT RegTable) {
%}

%HTML(report) {
@DTWR_CREATEREG("MYREG")
@DTWR_ADDENTRY("MYREG", "Dept. 1", "Payroll")
@DTWR_ADDENTRY("MYREG", "Dept. 2", "Technical Support")
@DTWR_ADDENTRY("MYREG", "Dept. 3", "Research")
@DTWR_LISTREG("MYREG", RegTable)

<p>Report:<br>
@ListTable(RegTable)

%}
```

IMS Web Language Environment

The IMS Web language environment is part of a complete end-to-end solution for running your IMS transactions in the World Wide Web environment using Net.Data. The IMS Web language environment provides:

- A Net.Data macro with:
  - The HTML used to enter the transaction input data
  - A Net.Data FUNCTION block that invokes the IMS Web language environment

  - The HTML that displays the output of the transaction
- The source for a transaction DLL that is invoked by the IMS Web language environment

The IMS Web Studio tool generates code for the DLL and the macro, as well as a make file for building the DLL executable or shared library, from the Message Format Service (MFS) source for the transaction and a sample HTML page for the IMS Web Net.Data application. After the executable form of the DLL has been built, the user moves theDLL and the macros to the Web server that is running Net.Data. The transaction is ready to run in the Web environment.

IMS Web uses the IMS TCP/IP Open Transaction Manager Access (OTMA) Connection to communicate between the Web server and IMS environments.

See the IMS Web home page for more information about using IMS Web.

```
http://www.software.ibm.com/data/ims/about/imsweb/document/
```

Configuring the IMS Web Language Environment

To use the IMS Web language environment, you need to verify the Net.Data initialization settings and set up the language environment.

Verify that the following configuration statement is in the initialization file, on one line.

```
ENVIRONMENT (HWS_LE)       DTWHWS    ( OUT RETURN_CODE )
```

See “Environment Configuration Statements” on page 21 to learn more about the Net.Data initialization file and language environment ENVIRONMENT statements.

Important: See “Setting up the IMS Web Language Environment” on page 24 to learn how to set up the IMS language environment.

Restrictions

The IMS Web language environment of Net.Data is only supported when Net.Data runs as a CGI application.

Programming Language Environments

Net.Data provides the following language environments for you to use when calling external programs:

- “Java Applet Language Environment” on page 139
- “Java Application Language Environment” on page 145
- “Perl Language Environment” on page 147
- “REXX Language Environment” on page 150
- “System Language Environment” on page 153

Access Rights: Ensure that the user ID under which Net.Data executes has access rights to execute programs, including any objects that the programs might access. See “Granting Access Rights to Files Accessed by Net.Data” on page 51 for more information.

Java Applet Language Environment

The Java applet language environment lets you easily generate HTML tags for Java applets in your Net.Data applications. When you call the Java applet language environment, you specify the name of your applet and pass any parameters that the applet needs. The language environment processes the macro and generates the HTML applet tags, which the Web browser uses to run the applet.

Additionally, Net.Data provides a set of interfaces your applet can use to access table parameters. These interfaces are contained in the class, DTW_Applet.class.

The following sections describe how to use the Java applet language environment to run your Java applets.

Configuring the Java Applet Language Environment

Verify that the following configuration statement is in the initialization file, on one line:

```
ENVIRONMENT (DTW_APPLET)   DTWJAVA    ( OUT RETURN_CODE )
```

See “Environment Configuration Statements” on page 21 to learn more about the Net.Data initialization file and language environment ENVIRONMENT statements.

Creating Java Applets

Before using the Net.Data Java applet language environment, you need to determine which applets you plan to use or which applets you need to write. See your Java documentation for more information on creating applets.

Generating the Applet Tags

You specify a call to the applet language environment with a Net.Data function call. No declaration is needed for the function call. The syntax for the function call is shown here:

```
@DTWA_AppletName(parm1, parm2, ..., parmN)
```

- DTWA_ identifies the function call to the applet language environment.
- AppletName is the name of the applet for which tags are generated.
- parm1 through parmN are parameters used to generate PARAM tags.

To write a macro that generates applet tags:

1. Define any parameters required by the applet in the DEFINE section of the macro. These parameters include any applet tag attributes, Net.Data variables, and Net.Data table parameters that you need as input for the applet. For example:

   ```
   %define{
   DATABASE = "celdial"                  <=Net.Data variable: name of the database
   MyGraph.codebase = "/netdata-java/"   <=Required applet attribute
   MyGraph.height = "200"                <=Required applet attribute
   MyGraph.width = "400"                 <=Required applet attribute
   MyTitle = "This is my Title"          <=Net.Data variable: name of the Web page
   MyTable = %TABLE(all)                 <=Table to store query results
   %}
   ```

2. Optional: Specify a query to the database to generate a result set as input for the applet. This is useful when you are using an applet that generates a chart or table. For example:

```
%FUNCTION(DTW_SQL) mySQL(OUT table){
select name, ages from ibmuser.guests
%}
```

3. Specify the function call in the Net.Data macro to call the Java applet language environment and invoke the applet. The function call specifies the name of the applet and the parameters you want to pass to the language environment. These parameters include any Net.Data variables, and Net.Data table or column parameters that you need as input for the applet.

   For example:

```
%HTML(report){                                    <=The start of the HTML block
@mySQL(MyTable)                                   <=A call to the SQL function
                                                          mySQL
@DTWA_MyGraph(MyTitle, DTW_COLUMN(ages) MyTable) <=Applet function call
%}
```

Applet Tag Attributes: You can specify attributes for applet tags anywhere in your Net.Data macro. Net.Data substitutes all variables that have the form *AppletName.attribute* into the applet tag as attributes. The syntax for defining an attribute on an applet tag is shown here:

```
%define AppletName.attribute = "value"
```

The following attributes are required for all applets:

- *codebase:* The location of the applet, which is identified by a URL.
- *height:* The height of the applet in pixels.
- *width:* The width of the applet in pixels.

The following attributes are optional:

- *align:* the alignment of the applet
- *alt:* any text that should be displayed if the browser understands the APPLET tag but can't run Java applets
- *archive:* an archive containing classes and other resources
- *hspace:* the number of pixels on each side of the applet
- *name:* a name for the applet instance
- *object:* the name of the file that contains a serialized representation of an applet
- *vspace:* the number of pixels above and below the applet

For example, if your applet is called MyGraph, you can define these required attributes as shown here:

```
%DEFINE{
MyGraph.codebase = "/netdata-java/"
MyGraph.height   = "200"
MyGraph.width    = "400"
%}
```

The actual assignment need not be in a DEFINE section. You can set the value with the DTW_ASSIGN function. If you do not define a variable for *AppletName.code* variable, Net.Data adds a default *code* parameter to the applet tag. The value of the *code*parameter is *AppletName.class*, where *AppletName* is the name of your applet.

Applet Tag Parameters: You define a list of parameters to pass to the Java applet language environment in the function call. You can pass parameters that include:

- Net.Data variables (including LIST variables)

- Net.Data tables
- Columns of Net.Data tables

When you pass a parameter, Net.Data creates a Java applet PARAM tag in the HTML output with the name and value that you assign to the parameter. You cannot pass string literals or results of function calls.

Net.Data Variable Parameters:

You can use Net.Data variables as parameters. If you define a variable in the DEFINE block of the macro and pass the variable value in the DTWA_*AppletName* function call, Net.Data generates a PARAM tag that has the same name and value as the variable. For example, given the following macro statement:

```
%define{

...

MyTitle = "This is my Title"
%}

%HTML(report){
@DTWA_MyGraph( MyTitle, ...)
%}
```

Net.Data produces the following applet PARAM tag:

```
<param name = 'MyTitle' value = "This is my Title" >
```

Net.Data Table Parameters:

Net.Data automatically generates a PARAM tag with the name DTW_NUMBER_OF_TABLES every time the Java applet language environment is called, specifying whether the function call has passed any table variables. The value is the number of table variables that Net.Data uses in the function. If no table variables are specified in the function call, the following tag is generated:

```
<param name = "DTW_NUMBER_OF_TABLES" value = "0" >
```

You can pass one or more Net.Data table variables as parameters on the function call. If you specify a Net.Data table variable on a DTWA_*AppletName* function call, Net.Data generates the following PARAM tags:

Table name parameter tag:

This tag specifies the names of the tables to pass. The tag has the following syntax:

```
<param name = 'DTW_TABLE_i_NAME' value = "tname" >
```

Where `i` is the number of the table based on the ordering of the function call, and *tname* is the name of the table.

Row and column specification parameter tags:

PARAM tags are generated to specify the number of rows and columns a particular table. This tag has the following syntax:

```
<param name = 'DTW_tname_NUMBER_OF_ROWS' value = "rows" >
<param name = 'DTW_tname_NUMBER_OF_COLUMNS' value = "cols" >
```

Where the name of the table is *tname*, *rows* is the number of rows in the table, and *cols* is the number of columns in the table. This pair of tags is generated for each unique table specified in the function call.

Column value parameter tags:

This PARAM tag specifies the column name of a particular column. This tag has the following syntax:

```
<param name = 'DTW_tname_COLUMN_NAME_j' value = "cname" >
```

Where the table name is *tname*, *j* is the column number, and *cname* is the name of the column in the table.

Row value parameter tags:

This PARAM tag specifies the values at a particular row and column. This tag has the following syntax:

```
<param name = 'DTW_tname_cname_VALUE_k' value = "val" >
```

Where the table name is *tname*, *cname* is the column name, *k* is the row number, and *val* is the value that matches the value in the corresponding row and column.

Table Column Parameters: You can pass a table column as a parameter on a function call to generate tags for a specific column. Net.Data generates the corresponding applet tags only for the specified column. A table column parameter uses the following syntax:

```
@DTWA_AppletName(DTW_COLUMN( x )Table)
```

Where *x* is the name or number of the column in the table.

Table column parameters use the same applet tags defined for the table parameters.

Alternate Text for the Applet Tag on Browsers that are not Java-Enabled: The variable DTW_APPLET_ALTTEXT specifies the text to display on browsers that do no support Java or have turned Java support off. For example, the following variable definition:

```
%define DTW_APPLET_ALTTEXT = "<P>Sorry, your browser is not Java-enabled."
```

produces the following HTML tag and text:

```
<P>Sorry, your browser is not Java-enabled.<BR>
```

If this variable is not defined, no alternate text is displayed.

Java Applet Example

The following example demonstrates a Net.Data macro that calls the Java applet language environment and the resulting applet tag that the language environment generates.

The Net.Data macro contains the following function calls to the Java applet language environment:

```
%define{
DATABASE = "celdial"
DTW_APPLET_ALTTEXT = "<P>Sorry, your browser is not Java-enabled."
DTW_DEFAULT_REPORT = "no"
MyGraph.codebase = "/netdata-java/"
```

```
MyGraph.height = "200"
MyGraph.width = "400"
MyTitle = "This is my Title"
%}
%FUNCTION(DTW_SQL) mySQL(OUT table){
select name, ages from ibmuser.guests
%}
%HTML(report){
@mySQL(MyTable)
@DTWA_MyGraph(MyTitle, DTW_COLUMN(ages) MyTable)
%}
```

The Net.Data macro lines in the DEFINE section specify the attributes of the applet tag:

```
MyGraph.codebase = "/netdata-java/"
MyGraph.height = "200"
MyGraph.width = "400"
```

The language environment generates an applet tag with the following qualifiers:

```
<applet code = 'MyGraph.class' codebase = '/netdata-java/' width = '400 'height = '200'>
```

Net.Data returns the SQL query results from the SQL section of the Net.Data macro in the output table, `MyTable`. This table is specified in the DEFINE section:

```
MyTable = %TABLE(all)
```

The call to the applet in the macro is specified in the HTML section:

```
@DTWA_MyGraph(MyTitle, DTW_COLUMN(ages) MyTable)
```

Based on the parameters in the function call, Net.Data generates the complete applet tag containing the information about the result table, such as the number of columns, the number of rows returned, and the result rows. Net.Data generates one parameter tag for each cell in the result table, as shown in the following example:

```
param name = 'DTW_MyTable_ages_VALUE_1' value = "35">
```

The parameter name, *DTW_MyTable_ages_VALUE_1*, specifies the table cell (row 1, column `ages`) in the table, `MyTable`, which has a value of 4. The keyword, DTW_COLUMN, in the function call to the applet, specifies that you are interested only in the column ages of the resulting table, MyTable, shown here:

```
@DTWA_MyGraph( MyTitle, DTW_COLUMN(ages) MyTable )
```

The following output shows the complete applet tag that Net.Data generates for the example:

```
<applet code = 'MyGraph.class' codebase = '/netdata-java/' width = '400' height = '200' >
<param name = 'MyTitle' value = "This is my Title" >
<param name = 'DTW_NUMBER_OF_TABLES' value = "1" >
<param name = 'DTW_TABLE_1_NAME' value = "MyTable" >
<param name = 'DTW_MyTable_NUMBER_OF_ROWS' value = "5" >
<param name = 'DTW_MyTable_NUMBER_OF_COLUMNS' value = "1" >
<param name = 'DTW_MyTable_COLUMN_NAME_1' value = "ages" >
<param name = 'DTW_MyTable_ages_VALUE_1' value = "35">
<param name = 'DTW_MyTable_ages_VALUE_2' value = "32">
<param name = 'DTW_MyTable_ages_VALUE_3' value = "31" >
<param name = 'DTW_MyTable_ages_VALUE_4' value = "28" >
<param name = 'DTW_MyTable_ages_VALUE_5' value = "40" >
<P>Sorry, your browser is not Java-enabled.<BR>
</applet>
```

Using the Net.Data Java Applet Interface

Net.Data provides a set of interfaces in a class called DTW_Applet.class, which you can use with your Java applets to help process the PARAM tags that are generated for table variables. You can create an applet that extends this interface to call the routines from your applet.

Net.Data provides these interfaces:

- **int GetNumberOfTables()** returns the number of tables found in the applet tag.
- **String [] GetTableNames()** returns a list of the table names found in the applet tag.
- **int GetNumberOfColumns(String table_name)** returns the number of columns in the table table_name.
- **int GetNumberOfRows(String table_name)** returns the number of rows in the table table_name.
- **String[] GetColumnNames(String table_name)** returns the names of the columns in the table table_name.
- **String[][] GetTable(String table_name)** returns a two-dimensional array of strings containing the values of the table's rows and columns.

To access the interfaces, use the EXTENDS keyword in your applet code to subclass your applet from the DTW_APPLET class, as shown in the following example:

```
import java.io.*;
import java.applet.Applet;

public class myDriver extends DTW_Applet
    {
    public void init()
        {
        super.init();

        if (GetNumberOfTables() > 0)
            {
            String [] tables = GetTableNames();
            printTables(tables);
            }
        }

    private void printTables(String[] tables)
        {
        String table_name;

        for (int i = 0; i < tables.length; i++)
            {
            table_name = tables[i];
            printTable(table_name);
            }
        }

    private void printTable(String table_name)
        {
        int nrows = GetNumberOfRows(table_name);
        int ncols = GetNumberOfColumns(table_name);

        System.out.println("Table: " + table_name + " has " + ncols + " columns and
                            " + nrows + " rows.");

        String [] col_names = GetColumnNames(table_name);
```

```
        System.out.println("-----------------------------------------------------");

        for (int i = 0; i < ncols; i++)
            System.out.print("   " + col_names[i] + "   ");
        System.out.println("\n-----------------------------------------------------");

        String [][] mytable = GetTable(table_name);

        for (int j = 0; j < nrows; j++)
            {
            for (int i = 0; i < ncols; i++)
                System.out.print("   " + mytable[i][j] + "   ");

            System.out.println("\n");
            }
        }
    }
```

Java Application Language Environment

Net.Data supports your existing Java applications with the Java language environment. With support for Java applets and Java methods (or applications), you can access DB2 through the Java Database Connectivity (JDBC**) API.

Details about JDBC are available from these Web sites:

- IBM Software has JDK 1.1 or higher, which is required to use JDBC with Net.Data:

  `http://www.software.ibm.com/data/db2/java/`
- JavaSoft has additional JDBC drivers, JDBC API documentation, and the latest updates of JDBC:

  `http://splash.javasoft.com/jdbc/`

Configuring the Java Language Environment

To use the Java language environment, you need to verify the Net.Data initialization settings and set up the language environment.

Verify that the following configuration statement is in the initialization file, on one line:

```
ENVIRONMENT (DTW_JAVAPPS)   ( OUT RETURN_CODE ) CLIETTE "DTW_JAVAPPS"
```

See “Environment Configuration Statements” on page 21 to learn more about the Net.Data initialization file and language environment ENVIRONMENT statements.

Important: See “Setting up the Java Language Environment” on page 24 to learn how to set up the Java language environment.

Calling Java Functions

The Java language environment provides a Remote Procedure Call (RPC)-like interface. You can issue Java function calls from your Net.Data macro with Net.Data strings as parameters and your invoked Java function can return a string. You must use the Net.Data Live Connection when you use the Java language environment (see “Managing Connections” on page 158) for more information about Live Connection).

To call Java functions:

1. Write your Java functions.
2. Create a Net.Data cliette for all your Java functions (Net.Data cliettes launch the Java Virtual Machine where your Java function runs).
3. Define a cliette on the Java ENVIRONMENT statement in Live Connection configuration file.

   Each time you introduce new Java functions, you must recreate the Java cliette.
4. Start Connection Manager.
5. Run the Net.Data macro that invokes the Java language environment.

Creating the Java Function: Modify the Java function sample file `UserFunctions.java`, or create a new file modeled on the following example file, called `myfile.java`:

```
====================myfile.java====================
import mypackage.*                                  <=contain your functions
 public String myfctcall(...parameters from macro...)
{
 return ( mypackage.mymethod(...parameters...));    <=high-level call to your functions
 }

public String lowlevelcall(...parameters...)
{
 string result;
  .......code using many functions of your package...
  return(result)
}
```

Java Language Environment File Structure: Net.Data creates several directories during the Net.Data installation. These directories include the files you need to create your Java functions, define the cliette, and run the macro with the Java language environment:

- A sample Java function called `UserFunctions.java`.
- A sample file called `makeClas`. When run, this file creates a Net.Data cliette class for your Java function.
- A sample file called `launchjv` used by the Net.Data cliette to launch the Java Virtual Machine and run your Java function.

Table 12 describes the directory and file names for the files on your operating system.

Table 12. The Files Used for Creating Java Functions

Operating System	File name	Directory
OS/2	UserFunctions.java	javaapps
	launchjv.com	connect
Windows NT	UserFunctions.java	javaclas
	makeClas.bat	javaclas
	launchjv.bat	connect
UNIX	UserFunctions.java	javaapps
	launchjv	javaapps

Defining the Java Language Environment Cliette: Modify the sample file, `makeClas.bat`, or create a new .bat file to generate a Net.Data cliette class, called `dtw_samp.class`, for all your Java functions. The following example shows how the batch file, `CreateServer`, processes three Java functions:

```
rem Batch file to create dtw_samp for Net.Data
java CreateServer dtw_samp.java UserFunctions.java myfile.java
javac dtw_samp.java
```

The batch file processes the following files, along with the Net.Data-supplied stub file called `Stub.java` to create `dtw_samp.class`.

- `dtw_samp.java`
- `UserFunctions.java`
- `myfile.java`

Writing a JDBC application or applet is very similar to writing a C application using DB2 CLI or ODBC to access a database. The primary difference between applications and applets is that an application might require special software to communicate with DB2, for example, DB2 Client Application Enabler. The applet depends on a Java-enabled Web browser, and does not require any DB2 code installed on the client.

Your system requires some configuration before using JDBC. These considerations are discussed at the DB2 JDBC Application and Applet Support Web site:

`http://www.software.ibm.com/data/db2/jdbc/db2java.html`

Java Language Environment Example

After you have created the Java function, defined the cliette class, and configured Net.Data, you can run the macro containing references to the Java function. **Important:** Start Connection Manager before invoking the Net.Data macro.

In the following example, the function call, *myfctcall* calls the sample function provided with Net.Data, using the cliette DTW_JAVAPPS.

```
%FUNCTION (DTW_JAVAPPS) myfctcall( ....parameters from macro ....)

%{ to call the sample provided with Net.Data %}
%FUNCTION (DTW_JAVAPPS) reverse_line(str);

%HTML(report){
you should see the string "Hello World" in reverse.
@reverse_line("Hello World")
You should have the result of your function call.
@myfctcall( ... ....)
%}
```

Perl Language Environment

The Perl language environment can interpret inline Perl scripts that you specify in a FUNCTION block of the Net.Data macro, or it can process external Perl scripts that are stored in separate files on the server.

Configuring the Perl Language Environment

Verify that the following configuration statement is in the Net.Data initialization file, on one line:

```
ENVIRONMENT (DTW_PERL)     DTWPERL   ( OUT RETURN_CODE )
```

See “Environment Configuration Statements” on page 21 to learn more about the Net.Data initialization file and language environment ENVIRONMENT statements.

Calling External Perl Scripts

Calls to external Perl scripts are identified in a FUNCTION block by an EXEC statement, using the following syntax:

```
%EXEC{ perl_script_name [optional parameters] %}
```

Required: Ensure that *perl_script_name*, the Perl script name, is listed in a path specified for the EXEC_PATH configuration variable in the Net.Data initialization file.

```
%FUNCTION(DTW_PERL) rexx1() {
%EXEC{MyPerl.pl %}
%}
```

Passing Parameters

There are two ways to pass information to a program that is invoked by the Perl (DTW_PERL language environment, directly and indirectly.

Directly

Pass parameters directly on the call to the Perl script. For example:

```
%DEFINE INPARM1 = "SWITCH1"

%FUNCTION(DTW_PERL) sys1() {
 %EXEC{
    MyPerl.pl $(INPARM1) "literal string"
 %}
%}
```

The Net.Data variable INPARM1 is referenced and passed to the Perl script. The parameters are passed to the Perl script in the same way the parameters are passed to the Perl script when the Perl script is called from the command line. The parameters that are passed to the Perl script using this method are considered input type parameters (the parameters passed to the Perl script can be used and manipulated by the Perl script, but changes to the parameters are not reflected back to Net.Data).

Indirectly

Pass parameters directly on the call to the Perl script using one of the following methods:

- Have Net.Data pass input parameters to the Perl script as environment variables. The Perl script can then retrieve the parameters through environment variables.
- Have the Perl script pass output parameters back to the language environment by writing to a named pipe whose name Net.Data passes in the environment variable, DTWPIPE. Use the following syntax to write data to the named pipe:

  ```
  name="value"
  ```

  For multiple data items, separate each item with a new-line or blank character.

  If a variable name has the same name as an output parameter and uses the above syntax, the new value replaces the current value. If a variable name does not correspond to an output parameter, Net.Data ignores it.

The following example shows how Net.Data passes variables from a macro.

```
%FUNCTION(DTW_PERL) today() RETURNS(result) {
  $date = 'date';
  chop $date;
  open(DTW, "> $ENV{DTWPIPE}") || die "Could not open: $!";
  print DTW "result = \"$date\"\n";
%}
%HTML(INPUT) {
  @today()
%}
```

If the Perl script is in an external file called `today.pl`, the same function can be written as in the next example:

```
%FUNCTION(DTW_PERL) today() RETURNS(result) {
  %EXEC { today.pl %}
%}
```

You can pass Net.Data tables to a Perl script called by the Perl language environment. The Perl script accesses the values of a Net.Data macro table parameter by their Net.Data name. The column headings and field values are contained in variables identified with the table name and column number. For example, in the table `myTable`, the column headings are `myTable_N_`*j*, and the field values are `myTable_V_i_`*j*, where *i* is the row number and *j* is the column number. The number of rows and columns for the table are `myTable_ROWS` and `myTable_COLS`.

REPORT and MESSAGE blocks in FUNCTION Sections

REPORT and MESSAGE blocks are permitted as in any FUNCTION section. They are processed by Net.Data, not by the language environment. A Perl script can, however, write text to the standard output stream to be included as part of the Web page.

Perl Language Environment Example

The following example shows how Net.Data generates a table by executing the external Perl script:

```
%define {
 c = %TABLE(20)
 rows = "5"
 columns = "5" %}
 %function(DTW_PERL) genTable(in rows, in columns, out table) {
 %exec{ perl.pl
 %}

 %message{
 default: "genTable: Unexpected Error"
 %}
 %}

 %HTML(REPORT) {
 @genTable(rows, columns, c)
 return code is $(RETURN_CODE)
 %}
 The Perl script (perl.pl):

 open(D2W,"> $ENV{DTWPIPE}");
 print "genTable begins ...

 ";
 $r = $ENV{ROWS};
 $c = $ENV{COLUMNS};
 print D2W "table_ROWS=\"$r\" ";
```

```
print D2W "table_COLS=\"$c\" ";
print "rows: $r
";
print "columns: $c";
for ($j=1; $j<=$c; $j++)
{
print D2W "table_N_$j=\"COL$j\" ";
}
for ($i=1; $i<=$r; $i++)
{
for ($j=1; $j<=$c; $j++)
{
print D2W "table_V_$i","_","$j=\"¦ $i $j ¿\" ";
}
}
close(D2W);
```

Results: `genTable` generates:

```
rows: 5 columns: 5
  COL1  |  COL2   |  COL3   |  COL4   |  COL5   |
-------------------------------------------------
[ 1 1 ] | [ 1 2 ] | [ 1 3 ] | [ 1 4 ] | [ 1 5 ] |
-------------------------------------------------
[ 2 1 ] | [ 2 2 ] | [ 2 3 ] | [ 2 4 ] | [ 2 5 ] |
-------------------------------------------------
[ 3 1 ] | [ 3 2 ] | [ 3 3 ] | [ 3 4 ] | [ 3 5 ] |
-------------------------------------------------
[ 4 1 ] | [ 4 2 ] | [ 4 3 ] | [ 4 4 ] | [ 4 5 ] |
-------------------------------------------------
[ 5 1 ] | [ 5 2 ] | [ 5 3 ] | [ 5 4 ] | [ 5 5 ] |
-------------------------------------------------
return code is 0
```

REXX Language Environment

The REXX language environment allows you to run REXX programs.

Configuring the REXX Language Environment

To use the REXX language environment, you need to verify the Net.Data initialization settings and set up the language environment.

Verify that the following configuration statement is in the initialization file, on one line:

```
ENVIRONMENT (DTW_REXX)     DTWREXX   ( OUT RETURN_CODE )
```

See “Environment Configuration Statements” on page 21 to learn more about the Net.Data initialization file and language environment ENVIRONMENT statements.

Executing REXX Programs

With the REXX language environment you can execute both in-line REXX programs or external REXX programs. An in-line REXX program is a REXX program that has the source of the REXX program in the macro. An external REXX program has the source of the REXX program in a external file.

To execute an in-line REXX program:

Define a function that uses the REXX (DTW_REXX) language environment and contains the REXX code in the language environment-executable section of the function.

Example: A function that contains a in-line REXX program

```
%function(DTW_REXX) helloWorld() {
        SAY 'Hello World'
%}
```

To run an external REXX program:

Define a function that uses the REXX (DTW_REXX) language environment and includes a path to the REXX program that is to be run in an EXEC statement.

Example: A function that contains an EXEC statement pointing to a the external program

```
%function(DTW_REXX) externalHelloWorld() {
%EXEC{ helloworld.exe%}
%}
```

Required: Ensure that the REXX file name is listed in a path specified for the EXEC_PATH configuration variable in the Net.Data initialization file. See "EXEC_PATH" on page 18 to learn how to define the EXEC_PATH configuration variable.

Passing Parameters to REXX programs

There are two ways to pass information to a REXX program that is invoked by the REXX (DTW_REXX) language environment, directly and indirectly.

Directly

Pass parameters directly to an external REXX program using the %EXEC statement. For example:

```
%FUNCTION(DTW_REXX) rexx1() {
  %EXEC{
    CALL1.CMD $(INPARM) "literal string"   %}
%}
```

The Net.Data variable INPARM1 is dereferenced and passed to the external REXX program. The REXX program can reference the variable by using REXX PARSE ARG instruction. The parameters that are passed to the program using this method are considered input type parameters (the parameters passed to the program can be used and manipulated by the program, but changes to the parameters are not reflected back to Net.Data).

Indirectly

Pass parameters indirectly, by way of the REXX program *variable pool*. When a REXX program is started, a space which contains information about all variables is created and maintained by the REXX interpreter. This space is called the variable pool.

When a REXX language environment (DTW_REXX) function is called, any function parameters that are input (IN) or input/output (INOUT) are stored in the variable pool by the REXX language environment prior to executing the REXX program. When the REXX program is invoked, it can access these variables directly. Upon the successful completion of the REXX program,

the DTW_REXX language environment determines whether there are any output (OUT) or INOUT function parameters. If so, the language environment retrieves the value corresponding to the function parameter from the variable pool and updates the function parameter value with the new value. When Net.Data receives control, it updates all OUT or INOUT parameters with the new values obtained from the REXX language environment. For example:

```
%DEFINE a = "3"
%DEFINE b = "0"
%FUNCTION(DTW_REXX) double_func(IN inp1, OUT outp1){
   outp1 = 2*inp1
%}

%HTML(REPORT) {
Value of b is $(b), @double_func(a, b) Value of b is $(b)
%}
```

In the above example, the call *@double_func* passes two parameters, *a* and *b*. The REXX function *double_func* doubles the first parameter and stores the result in the second parameter. When Net.Data invokes the macro, *b* has a value of 6.

You can pass Net.Data tables to a REXX program. A REXX program accesses the values of a Net.Data macro table parameter as REXX stem variables. To a REXX program, the column headings and field values are contained in variables identified with the table name and column number. For example, in the table `myTable`, the column headings are `myTable_N.`*`j`*, and the field values are `myTable_N.`*`i`*`.`*`j`*, where *i* is the row number and *j* is the column number. The number of rows in the table is `myTable_`*`ROWS`* and the number of columns in the table is `myTable_`*`COLS`*.

Improving Performance for the AIX operating system:

If you have many calls to the REXX language environment on your AIX system, consider setting the RXQUEUE_OWNER_PID environment variable to 0. Macros that make many calls to the REXX language environment can easily spawn many processes, swamping system resources.

You can set the environment variable in one of three ways:

- In the macro by using the DTW_SETENV built-in function:

  ```
  @DTW_rSETENV("RXQUEUE_OWNER_PID", "0")
  ```

- In the AIX system environment file by inserting the following statement:

  ```
  /etc/environment: RXQUEUE_OWNER_PID = 0
  ```

  This method affects the behavior of REXX for the whole machine.

- In the HTTP Web server environment file; for example, for the Domino Go Webserver, insert the following statement:

  ```
  InheritEnv RXQUEUE_OWNER_PID = 0
  ```

  This method affects the behavior of REXX for the Web server.

REXX Language Environment Example

The following example shows a macro that calls a REXX function to generate a Net.Data table that has two columns and three rows. Following the call to the REXX function, a built-in function, DTW_TB_TABLE(), is called to generate an HTML table that is sent back to the browser.

```
%DEFINE myTable = %TABLE
%DEFINE DTW_DEFAULT_REPORT = "NO"

%FUNCTION(DTW_REXX) genTable(out out_table) {
  out_table_ROWS = 3
  out_table_COLS = 2

  /* Set Column Headings */
  do j=1 to out_table_COLS
    out_table_N.j = 'COL'j
  end

  /* Set the fields in the row */
  do i = 1 to out_table_ROWS
    do j = 1 to out_table_COLS
      out_table_V.i.j = '[' i j ']'
    end
  end
%}

%HTML(REPORT)  {
 @genTable(myTable)
 @DTW_TB_TABLE(myTable)
%}
```

Results:

```
  COL1      COL2
[ 1 1 ]   [ 1 2 ]
[ 2 1 ]   [ 2 2 ]
[ 3 1 ]   [ 3 2 ]
```

System Language Environment

The System language environment supports executing commands and calling external programs.

Configuring the System Language Environment

Add the following configuration statement to the initialization file, on one line:

```
ENVIRONMENT (DTW_SYSTEM)   DTWSYS    ( OUT RETURN_CODE )
```

See “Environment Configuration Statements” on page 21 to learn more about the Net.Data initialization file and language environment ENVIRONMENT statements.

Issuing Commands and Calling Programs

To issue a command, define a function that uses the System (DTW_SYSTEM) language environment that includes a path to the command to be issued in an EXEC statement. For example:

```
%FUNCTION(DTW_SYSTEM) sys1() {
      %EXEC { ADDLIBLE.CMD %}
 %}
```

You can shorten the path to executable objects if you use the EXEC_PATH configuration variable to define paths to directories that contain the objects (such as, commands and programs). See “EXEC_PATH” on page 18 to learn how to define the EXEC_PATH configuration variable.

Example 1: Issues a command

```
%FUNCTION(DTW_SYSTEM) sys2() {
      %EXEC { MYPGM %}
 %}
```

Example 2: Calls a program

```
%FUNCTION(DTW_SYSTEM) sys3() {
      %EXEC {MYPGM.EXE %}
 %
```

Passing Parameters to Programs

There are two ways to pass information to a program that is invoked by the System (DTW_SYSTEM) language environment, directly and indirectly.

Directly

Pass parameters directly on the call to the program. For example:

```
%DEFINE INPARM1 = "SWITCH1"

%FUNCTION(DTW_SYSTEM) sys1() {
 %EXEC{
  CALL1.CMD  $(INPARM1) "literal string"
 %}
%}
```

The Net.Data variable INPARM1 is referenced and passed to the program. The parameters are passed to the program in the same way the parameters are passed to the program when the program is called from the command line. The parameters that are passed to the program using this method are considered input type parameters (the parameters passed to the program can be used and manipulated by the program, but changes to the parameters are not reflected back to Net.Data).

Indirectly

The System language environment cannot directly pass or retrieve Net.Data variables, so they are made available to programs in the following manner:

- Net.Data passes input parameters to the program as environment variables. The program can then retrieve the parameters through environment variables.
- The program passes output parameters back to the language environment by writing to a named pipe whose name Net.Data passes in the environment variable, DTWPIPE. Use the following syntax to write data to the named pipe:

  name="value"

  For multiple data items, separate each item with a new-line or blank character.

  If a variable name has the same name as an output parameter and uses the above syntax, the new value replaces the current value. If a variable name does not correspond to an output parameter, Net.Data ignores it.

The following example shows how Net.Data passes variables from a macro.

```
%FUNCTION(DTW_SYSTEM) sys1 (IN P1, OUT P2, P3) {
   %EXEC {
      UPDPGM
   %}
 %}
```

You can pass Net.Data tables to a program called by the System language environment. The program accesses the values of a Net.Data macro table parameter by their Net.Data name. The column headings and field values are contained in variables identified with the table name and column number. For example, in the table `myTable`, the column headings are `myTable_N_`*j*, and the field values are `myTable_V_i_`*j*, where *i* is the row number and *j* is the column number. The number of rows and columns for the table are `myTable_ROWS` and `myTable_COLS`.

It is not recommended that you pass tables with many rows because the number of environment variables for the process is limited.

System Language Environment Example

The following example shows a macro that contains a function definition with three parameters, `P1`, `P2`, and `P3`. `P1` is an input (IN) parameter and `P2` and `P3` are output (OUT) parameters. The function invokes a program, UPDPGM, which updates the parameter `P2` with the value of `P1` and sets `P3` to a character string. Prior to processing the statement in the %EXEC block, the DTW_SYSTEM language environment stores `P1` and the corresponding value in the environment space.

```
%DEFINE {
   MYPARM2  = "ValueOfParm2"
   MYPARM3  = "ValueOfParm3"
 %}
 %FUNCTION(DTW_SYSTEM) sys1 (IN P1, OUT P2, P3) {
   %EXEC {
      UPDPGM
   %}
 %}

 %HTML(upd1) {
 <P>
 Passing data to a program.  The current value
 of MYPARM2 is "$(MYPARM2)", and the current value of MYPARM3 is
 "$(MYPARM3)".  Now we invoke the Web macro function.

 @sys1("ValueOfParm1", MYPARM2, MYPARM3)

 <P>
 After the function call, the value of MYPARM2 is "$(MYPARM2)",
 and the value of MYPARM3 is "$(MYPARM3)".
 %}
```

Chapter 7. Improving Performance

Improving performance is an important part of tuning your system. This chapter discusses strategies for improving the performance of Net.Data. The following topics are discussed:

- "Using the Web Server APIs"
- "Using FastCGI"
- "Managing Connections" on page 158
- "Net.Data Caching" on page 161
- "Setting the Error Log Level" on page 180
- "Optimizing the Language Environments" on page 181

In addition, ensure that your Web server has been properly tuned. The performance of your Web server has a direct effect on response time, independently of how fast Net.Data processes a macro or direct request.

Using the Web Server APIs

You can improve performance by invoking Net.Data with a Web server API, such as ICAPI or GWAPI, instead of CGI. When Net.Data executes using a Web server API, Net.Data executes as a thread within the Web server's process. Because a Web server's process is multi-threaded, multiple Net.Data requests can be processed concurrently within the same address space, eliminating the overhead of invoking Net.Data as a CGI process.

Consideration: Using a Web server API provides improved performance, without application isolation. Because Net.Data runs in a multi-threaded environment, errors introduced within user-written language environments, improper invocations, or even database outages can cause problems with the Web server and potentially bring it down. When deciding whether to use one of the Web server APIs, determine whether the higher priority for your application is performance or application isolation.

Using FastCGI

FastCGI provides improved performance with the reliability of CGI-BIN. Using FastCGI allows you to run macros with the speed of the API servers, with the more reliable method of using separated memory space. Net.Data invocation default is to run with CGI.

You can use Net.Data with FastCGI on all servers that support FastCGI.

See "Configuring Net.Data for FastCGI" on page 34 to learn how to configure for FastCGI.

You can tune FastCGI to run the appropriate amount of processes to handle the number of incoming requests with the process configuration parameter. For example, one customer averaged 100 requests per second and each request took a half second to process. They set the process parameter to 50.

FastCGI is supported by the following language environments:

- SQL
- Oracle 7.2, 7.3, 8.0
- ODBC
- Sybase
- REXX
- Perl

Requirement: When running in FastCGI mode, the Oracle and Sybase language environments require Live Connection.

To tune the number of simultaneous processes:

1. Open the configuration file where the configuration parameter for processes is defined.
   - For Apache, this is the `httpd.conf` file.
   - For ICS or Lotus Domino Go Webserver, this is the `lgw_fcgi.conf` file.
2. Change configuration parameter value that specifies the number of processes:
   - For Apache: `Process=`*num*.
   - For ISC: `NumProcess=`*num*.

   Where *num* is the number of processes.

Managing Connections

Net.Data provides a component called Live Connection to manage database and Java virtual machine connections. Live Connection maintains persistent connections to improve performance. Some Net.Data actions require a large start-up time. For example, before a database query can be issued, the process must identify itself to the DBMS and connect to the database. This is often a significant portion of the processing time needed for Net.Data macros that access a database. Another example of an expensive start up is a Java virtual machine that is needed to run Java applications (but not Java applets). Because of the way CGI programs operate, these start-up costs are paid on every request to the Web server. Net.Data provides Live Connection on the OS/2, Windows NT, and UNIX operating systems to maintain persistent connections.

The following sections describe Live Connection.

About Live Connection

Live Connection can dramatically improve performance by eliminating start-up overhead. The savings come from continuously running one or more processes that perform the start up functions. These processes then wait to service requests. You can run Live Connection if you use Net.Data as a CGI or FastCGI program, or as a Web server API plug-in.

Live Connection consists of Connection Manager and cliettes. *Cliettes* are processes that the Connection Manager starts, and stay active while the server is running. Cliettes process data and communicate with Net.Data language environments that you specify in the initialization file with the keyword CLIETTE. Each type of cliette handles a specific language environment function, such as the DB2 cliette, which connects to the DB2 database and sets up operations to perform SQL calls before any Net.Data macros are processed by Net.Data. The executable file is named in the Live Connection configuration file, `dtwcm.cnf`. Figure 22 shows the interaction between Live Connection, the macro, and the language environments.

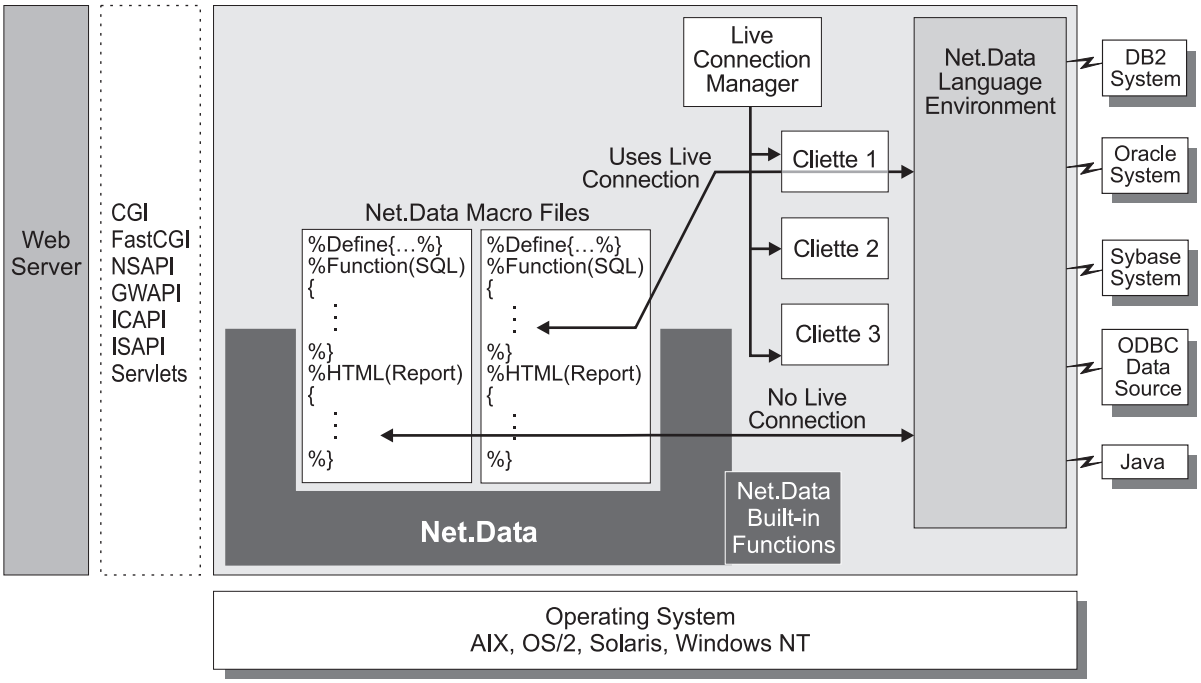


Figure 22. Live Connection with Cliettes

The following sections describe Live Connection in more detail. To learn how to configure Live Connection, see “Configuring Live Connection” on page 29.

Live Connection Advantages

The main advantages to using Live Connection are:

- **Improved performance**

  Reusing connections is more efficient than making new connections. Generally, if you request small SQL statements (for example, simple queries on a database with fewer than 100 000 rows), or if your database connection is difficult (for example, remote servers), the connect time is significant.

- **Multiple database access**

  Live Connection enables one Net.Data macro connection to multiple databases at the same time. This is possible because each database has unique cliettes, and therefore Net.Data simply communicates with multiple cliettes.

Should I Use Live Connection?

You can use Live Connection in CGI, FastCGI, or API mode to communicate with your database. In addition, you might benefit from Live Connection if your application requires data from multiple databases.

Live Connection, used together with an API plug-in, improves performance for many systems, depending on their load and configuration. You should experiment with your own system to determine the configuration that works best for you.

Many applications can improve performance without using Live Connection by using the ACTIVATE DATABASE or START DATABASE command to save time establishing database connections. See your database's documentation for details on the command your database uses. Also check your operating system's documentation to see if there are additional steps to help improve performance.

Requirement: The Oracle and Sybase language environments require Live Connection when running in FastCGI and API modes.

Starting the Connection Manager

The Connection Manager is a separate executable file that is shipped with Net.Data and named `dtwcm`. Start the Connection Manager when you start the Web server.

When you start Connection Manager, it reads a configuration file and starts a group of processes. In each process, the Connection Manager begins the execution of a particular cliette. To learn how to configure Live Connection, see "Configuring Live Connection" on page 29.

To start the Connection Manager with Windows NT and OS/2:

1. From the command line, change to the <*inst_dir*>\connect\ directory.
2. Enter `dtwcm`.

Where <*inst_dir*> is the Net.Data installation directory.

To start the Connection Manager with AIX:

1. From the command line, change to the `/usr/lpp/internet/db2www/db2/` directory.
2. Enter `dtwcm`.

To start the Connection Manager with the messages option:

By default, Connection Manager messages are suppressed. Use the -d option when starting Connection Manager if you want Connection Manager messages to be displayed.

From the command line, enter: `dtwcm -d`

After you use the -d option, you have to restart Connection Manager in order to suppress the messages again.

To automatically start Connection Manager as a Windows NT service:

On Windows NT, you can specify to have Connection Manager start as an Windows NT service, instead of from the command line. Running Connection Manager as an Windows NT service allows Connection Manager to be automatically started each time the machine is started.

Tip: Start Connection Manager from the command line before setting it up to start automatically to insure that the Live Connection configuration file is correct.

1. From the Windows NT task bar, select **Start->Settings->Control Panel ->Services**.
2. Select **Net.Data Connection Manager** and then click the **Startup** button.
3. Select **Automatic startup type** and then click on **OK**.

Net.Data and Live Connection Process Flow

After you've configured and started the database, Web server, and Connection Manager, Net.Data processing typically involves these steps when Live Connection is enabled:

1. The Web server receives a request and starts either a FastCGI, CGI or API process to run Net.Data.
2. Net.Data starts processing the Net.Data macro.
3. When Net.Data encounters a function call that uses Live Connection, it determines what type of cliette is needed from the initialization file. For DB2, the cliette type is often a name based on the DB2 database name, such as DTW_SQL:CELDIAL.
4. Net.Data asks the Connection Manager for a cliette of that type.
5. The Connection Manager looks for available cliettes of that type. If none is available, the Connection Manager puts the request on a queue and processes it when the right cliette type is available.
6. When a cliette becomes available, the Connection Manager tells Net.Data how to communicate with the cliette.
7. Net.Data asks the cliette to process the function.
8. This process is repeated from step 3 until the Net.Data macro processing is complete.
9. All cliettes are released.

If a cliette is specified in the initialization file but the Connection Manager is not running, Net.Data loads the DLL and processes the macro. If you use an API, you are likely to receive errors, and you should start the Connection Manager.

Net.Data Caching

Caching helps you to improve response times for the application user. Net.Data stores results from a request to the Web server locally for quick retrieval, until it is time to refresh the information. This chapter describes Net.Data caching concepts, tasks, and restrictions.

About Web Caching

Many software components perform caching for Web applications. Here are some examples of caching applications:

- A Web browser saves Web pages and related objects such as image and audio files and Java applets, locally, in memory or on disk to save network time when its user repeatedly accesses the same pages.
- A Web proxy server cache saves Web pages and related objects on a local server, near a group of users, to reduce network access time to remote Web servers, for example to reduce the number of times the Web servers retrieve requested items. A Web proxy server cache also enables efficient sharing of commonly accessed pages between multiple users.
- A Web server caches frequently retrieved pages and related objects in memory to save disk access time when users repeatedly retrieve the same pages.
- A database management system caches data items, which are usually held on disk, in memory to save disk access time when repeatedly retrieving the same data items.

All these components perform their caching independently but the overall result is improved response times for users. In order to determine when to refresh a cached item, the Web components (browser, proxy server, and Web server) usually take into consideration various options including:

- The browser and server configuration options
- The contents of the HTTP headers returned with the Web pages and related items from the Web server, in particular the expiry date information

About Net.Data Caching

Net.Data itself provides its own caching function for frequently accessed pages and related data items generated by Net.Data macros. By delivering a page from the Net.Data cache, you save the time required to run a Net.Data macro and to access a database in order to create the page.

You can use one Cache Manager per server. **Recommendation:** Use one Cache Manager for many instances of Net.Data, and multiple caches per Cache Manager.

Figure 23 on page 163 shows that Net.Data uses a Cache Manager to manage the caching of HTML output from a macro file. This output can include data from a database.

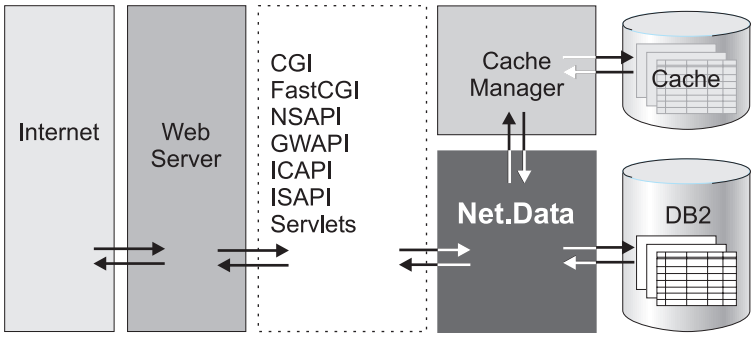


Figure 23. Net.Data Caching

Net.Data Caching Terminology

Net.Data documentation uses the following terms to describe Net.Data caching.

cache A type of memory that contains recently accessed data, designed to speed up subsequent access to the same data. The cache is often used to hold a local copy of frequently-used data that is accessible over a network. In Net.Data, local memory that contains Net.Data-generated HTML Web pages for reuse by the Net.Data macro. By having the pages stored in the cache, Net.Data does not have to regenerate the information in the cache. Each cache is managed by the Cache Manager, which can be responsible for multiple caches and can server multiple instances of Net.Data.

cache ID
A string that identifies a particular cache.

Cache Manager
The program that manages caching for one machine. It can manage multiple caches.

Cache Manager configuration file
The file containing the settings used by Net.Data to determine settings for logging, tracing, cache size, and other options. It contains settings for a Cache Manager and all the cache files managed by a particular Cache Manager. The file name is `cachemgr.cnf` when packaged with Net.Data.

Net.Data Caching Concepts

Depending on how many HTTP servers you have on your system, and whether each HTTP server runs its own copy of Net.Data (using separate Net.Data configuration files), you can have all the copies of Net.Data be associated with one Cache Manager or multiple Cache Managers. One Cache Manager can support a number of caches in memory, each cache has a cache identifier called a *cache ID*. Figure 24 on page 164 shows one Cache Manager working with multiple macro files and managing two caches.

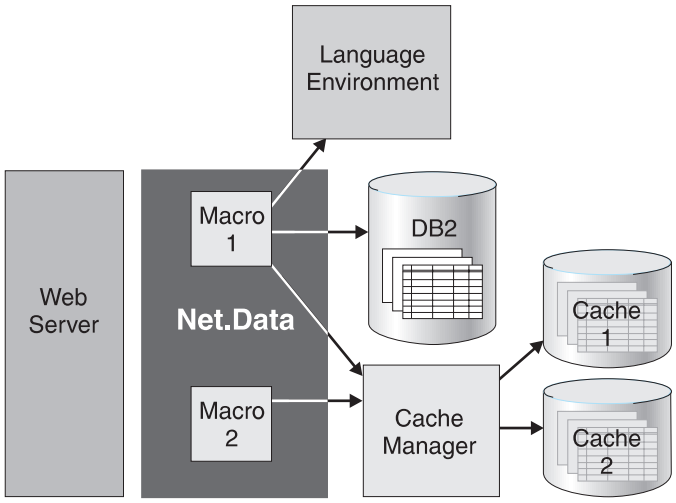


Figure 24. Cache Manager Works with Multiple Macro Files and Caches

Any number of items, known as *cached pages*, can be placed in a cache. Each cached page has a unique identifier, for example a Uniform Resource Locator (URL). A page is a segment of or a complete HTML page.

When Net.Data receives a request for cached data (for example, from the built-in function DTW_CACHE_PAGE), the following steps are taken:

1. Net.Data connects to the Cache Manager.
2. Net.Data checks to see if the data is cached.
   - If the data is cached and not expired, Net.Data requests the page from the Cache Manager, sends it to the browser, and stops executing the macro.
   - If the data is not cached, Net.Data continues processing the macro, then send the generated HTML page to the Web browser and to the Cache Manager, where it is cached.
3. Net.Data disconnects from the Cache Manager

The Cache Manager caches HTML output when the macro file successfully completes processing, ensuring that only successfully generated Web pages are cached. The data is not cached until after it has been sent to the browser, and the data that the user sees is the same data that is cached.

When Net.Data encounters an error or exits prematurely from the macro, the Cache Manager:

- Does not accept partial or defective pages
- Preserves existing pages in the cache

Net.Data Caching Restrictions

Net.Data caching has the following restrictions:

Security
: Security is not provided by the Cache Manager. For example, if one database user runs a macro and caches a pages of database results. Another database user can retrieve the cached page.

Direct request
: Direct request invocation of Net.Data cannot use Net.Data caching.

Net.Data Caching Interfaces

Net.Data provides a flexible set of interfaces for you to use when configuring and setting up caching for you application. Table 13 describes the various options for using the Net.Data caching features and where these features are described.

Table 13. Net.Data Cache Interfaces

Interface	Description	Go to ...
Cache Manager configuration options	You can specify a number of options for the Cache Manager such as logging and tracing, in the Cache Manager stanza of the Cache Manager configuration file.	"Defining the Cache Manager" on page 167
Cache configuration options	Within a single instance of Net.Data's Cache Manager, you can define a number of caches to hold the cached items. Each cache has its own set of characteristics, such as size and location, and the cache ID. These characteristics are defined in the cache stanza in the Cache Manager configuration file. Each stanza is identified by the cache ID.	"Defining a Cache" on page 168
Net.Data initialization options	If Net.Data and the corresponding Cache Manager run on separate systems, then you specify the Cache Manager system and port number in the Net.Data initialization file.	"Cache Manager Configuration Variables" on page 12
Net.Data cache built-in functions	You can manipulate the contents of a Net.Data cache using the Net.Data cache built-in functions. Specify the cache ID in the appropriate macro function to select the cache with the most appropriate characteristics.	See the built-in functions chapter of *Net.Data Reference*

Planning for the Cache Manager

When planning your use of the Net.Data cache functions you must consider:

- What pages would benefit from caching and the performance improvements that would be gained
- When to cache the items
- When to refresh the items in the cache and the refresh methods to use

To use Net.Data caching, you need to complete the following steps, which require knowing how you want to use caching.

Recommendation: Before embarking on a major application that uses caching, we strongly urge you to plan and prototype your application prior to putting it into production.

- Install Net.Data, which includes the caching function.
- Configure the Cache Manager. See “Configuring the Cache Manager and Net.Data Caches”.
- Determine how you are going to put the Net.Data application into production.
- Check the various Net.Data cache logs to determine whether improvements in the use of the cache and the way it is configured should be made.

Cache Errors

The Cache Manager does not cache Web pages when Net.Data encounters internal errors that cause it to exit the macro file before processing is complete. Cache Manager does not cache pages that are incomplete or contain Net.Data errors. These types of errors include macro syntax errors and SQL errors.

Pages with errors can be cached when:

- Net.Data encounters an error and Net.Data continues to execute the macro file because of a CONTINUE configuration variable in a message block and terminates normally.
- Errors occur outside of the Net.Data’s error determination scope, such as database rollbacks.

Cache Identifiers

You need to plan for two types of identifiers when designing caching for your application.

- **Identifier for a cache:** This identifier is the cache ID and specifies the name in the configuration file stanza that defines the cache. You can use many approaches for classifying and naming your caches. For example, you can name the cache by application. You could have a cache for each of your Net.Data applications, giving each cache a name that is derived from the Net.Data macro it serves.
- **Identifier for the cached page:** This identifier is the cached page ID and specifies the name of the page to be cached. The cached page ID can be any string, such as an URL address. You specify the identifier with the DTW_CACHE_PAGE() built-in function. See the built-in functions chapter of *Net.Data Reference* for syntax and examples.

Configuring the Cache Manager and Net.Data Caches

The Cache Manager manages one or more caches in your system. Each of these caches contains the contents of dynamically generated HTML pages. To configure the Cache Manager and each of the caches, you update the keyword values in the Cache Manager configuration file, `cachemgr.cnf`.

The Cache Manager configuration file contains two types of stanzas: the Cache Manager stanza and the Cache Definition stanza. The following steps describe how to customize these two types of stanzas for your application.

Defining the Cache Manager

Define the Cache Manager stanza by specifying values for the allowed keywords. All of the keywords are optional; you do not need to specify them unless you don't want to accept the default value.

To define the Cache Manager:

1. Specify the name of the Cache Manager log file. The log shows activity for all transactions for all caches and is provided for debugging and problem analysis.

   The default is to write messages to the console.

   Syntax:

   `log=`*`path`*

   Where *path* is the path and file name of the cache file.

   Tip: To specify a log file for each cache, use the **tran-log** keyword in the Cache Definition stanza.

2. Specify the TCP/IP port number used by the Cache Manager for incoming requests. This port number is used only for contacting the Cache Manager from a remote machine.

   This value must match the port number specified by the DTW_CACHE_PORT configuration variable in the Net.Data initialization file. The default value is determined in the following manner:

   a. The Cache Manager checks the path */etc/services* for the value associated with the name `ibm-cachemgrd`. If this value is found, the Cache Manager uses the value. If it is not found, it uses the next method.

   b. The Cache Manager uses the default port, 7175.

   Syntax:

   `port=`*`port_number`*

   Where *port_number* is a unique TCP/IP port number.

3. Specify the maximum length of time in seconds that the Cache Manager should allow a pending read to be left active. If this time is exceeded, the Cache Manager drops the connection.

   The default is 30 seconds.

   Syntax:

   `connection-timeout=`*`seconds`*

   Where *seconds* is the number of seconds used for the length of time a pending read should be active.

4. Specify whether to log messages.

   The default is **no** or **off**.

   Syntax:

   `logging=yes|on|no|off`

   Where:

   yes|on
   Indicates that logging is required

   no|off Indicates that logging should not be performed.

5. Specify whether to wrap the log.

The default is **no**. If specified as **yes**, the current log is closed when it reaches its maximum size (see **log-size**, below), the file has a file type of .old, and a new log is opened. Only one generation of the log file is maintained (existing .old files are overwritten).

Syntax:

```
wrap-log=yes|no
```

Where:

yes Specifies to wrap the log.

no Specifies not to wrap the log.

6. Specify the maximum size, in bytes, to which a log is allowed to grow, if wrap-log is specified.

   The default is 64000.

   Syntax:

   ```
   log-size=bytes
   ```

   Where *bytes* is the number of bytes of the maximum size.

7. Specify the level of messages to be written to the log. These values are set on when included in the *trace_flag_definitions* list and do not have any settings.

   The default is to log only the Cache Manager start up and shut down messages.

   Syntax:

   ```
   trace-flags=trace_flag_definitions
   ```

   Where:

   D_ALL
   Enables all trace flags.

   D_NONE
   Disables all trace flags

Example: Trace flag that specifies all trace flags are enabled:

```
trace=flags=D_ALL
```

Stanza example: A valid Configuration Manager stanza:

```
cache-manager {
log=/u/cached/logs/cached.log
port=7177
connection-timeout=0
logging=yes
wrap-log=yes
log-size=64000
trace-flags=D_ALL
}
```

Defining a Cache

Define the Cache Definition stanza by specifying values for the allowed keywords. Most keywords are optional and do not have to be specified unless you don't want to use the default value.

To define a cache:

1. Specify the path and directory name that are to hold the cache pages. On start up, the file system containing this directory must be at least as large as the

value of **fssize** (see below); otherwise the cache is not started. This value can be specified as an absolute path name or as a relative path name that corresponds to the path in which the Cache Manager was started.

Required.

Syntax:

```
root=path_name
```

Where:

path_name
: Is the absolute or relative name of the path and directory where the cache pages are stored.

2. Specify whether the current cache is active when the Cache Manager starts.

   Not required; the default is **yes**. If set to **no**, the cache is defined in the Cache Manager but not activated. You can activate it later with the **cacheadm** command.

   Syntax:

   ```
   caching=yes|no
   ```

   Where:

   yes Indicates that the cache is to be active when the Cache Manager starts.

   no Indicates that the cache is not to be active when the Cache Manager starts.

3. Specify the maximum space to be used in the file system by pages in the current cache. When the maximum amount of space is exceeded, the Cache Manager deletes enough pages, starting with the oldest, to bring the total space occupied by the cache within the limit. You can effectively disable automatic purging of entries by setting this value to a large number; however, if the physical file system space is exceeded, attempts to add new pages to cache fail.

   Not required; the default is 0 (no caching to disk).

   Syntax:

   ```
   fssize=nnB|nnKB|nnM
   ```

   Where:

   *nn***B** Is the number of bytes; for example 5000B.

   *nn***KB** Is the number of kilobytes; for example 640KB.

   *nn***MB** Is the number of megabytes; for example 30MB.

4. Specify the maximum amount of memory to be used by all of the pages in this cache. When the maximum amount of memory is exceeded, the Cache Manager deletes enough pages, starting with the oldest, to bring the total memory occupied by the cache within bounds. You can effectively disable automatic purging of pages by setting this to a large number; however, if the **cachemgrd** process consumes too much memory, the operating system may terminate it.

   Not required; the default is 1MB.

   Syntax:

   ```
   mem-size=nnB|nnKB|nnMB
   ```

   Where:

*nn***B** Is the number of bytes; for example 5000B.

*nn***KB** Is the number of kilobytes; for example 640KB.

*nn***MB** Is the number of megabytes; for example 30MB.

5. Specify the maximum length of time an page can be held in the cache. When this value is exceeded, the Cache Manager marks the page as expired but does not delete the page unless the **fssize** (if it is cached on disk) or **memsize** (if it is cached in memory) limits are reached. The Cache Manager deletes pages that are marked as expired before all other pages if **memsize** or **fssize** limits are reached. You can disable **lifetime** checking with the **check_expiration** keyword.

   Required: No; the default is 5 minutes.

   Syntax:

   `lifetime=`*`time_length`*

   Where:

   *nn***S** Is the number of seconds; for example, 600S.

   *nn***M** Is the number of minutes; for example, 20M.

   *nn***H** Is the number of hours; for example, 30H.

6. Specify whether to mark cache pages as expired and to perform lifetime checking.

   Not required; the default is **yes**, with a default lifetime length of 60 seconds. This value can also be set to a length of time, indicating a **yes** value and declaring a maximum length of time an item may be held in cache. When set to **no**, cache pages are never marked expired and lifetime checking is not performed.

   Syntax:

   `check-expiration=yes|`*`nn`*`S|`*`nn`*`M|`*`nn`*`H|no`

   Where:

   yes Indicates that the Cache Manager performs lifetime checking and cache pages are marked as expired.

   *nn***S** Is the number of seconds; for example, 600S.

   *nn***M** Is the number of minutes; for example, 20M.

   *nn***H** Is the number of hours; for example, 30H.

   no Indicates that the Cache Manager does not perform lifetime checking and cache pages are not marked as expired.

7. Specify the maximum amount of space a cached page can occupy within the memory cache. If an page is too large for memory, the file cache is checked. If adequate space exists, the Cache Manager stores the cache page in the file cache, instead. If the page does not fit in the file cache, the caching attempt fails. If the page is smaller than datum_memory_limit value (cacheobj-memory-limit), but if the cache doesn't have enough space, the oldest cache pages are deleted from the memory cache to accommodate the new page.

   Not required; the default is 1KB.

   Syntax:

   `datum-memory-limit (cacheobj-memory-limit)=`*`nn`*`B|`*`nn`*`KB|`*`nn`*`MB`

Where:

*nn***B** Is the number of bytes; for example 5000B.

*nn***KB** Is the number of kilobytes; for example 640KB.

*nn***MB** Is the number of megabytes; for example 30MB.

8. Specify the maximum amount of space a cache page can occupy within the file cache. If an page is smaller than datum_disk_limit, but no space remains in the file cache, the oldest cache pages are deleted from the file cache to accommodate the new page.

   Not required; the default is 1KB.

   Syntax:

   `datum-disk-limit (cacheobj-space-limit)=`*nn*`B|`*nn*`KB|`*nn*`MB`

   Where:

   *nn***B** Is the number of bytes; for example 5000B.

   *nn***KB** Is the number of kilobytes; for example 640KB.

   *nn***MB** Is the number of megabytes; for example 30MB.

9. Specify the time between creation of statistics records. If set to 0, no statistics records are written.

   Not required; the default is 0 (no statistics).

   Syntax:

   `stat-interval = `*nn*`S|`*nn*`M|`*nn*`H`

   Where:

   *nn***S** Is the number of seconds; for example, 600S.

   *nn***M** Is the number of minutes; for example, 1M.

   *nn***H** Is the number of hours; for example, 3H.

10. Specify the name of the path and file that are to be used for logging statistics for the current cache.

    Required when the value for **stat-interval** is greater than 0.

    Syntax:

    `stat-files=`*filename*

    Where *filename* is the path and name of the logging statistics file.

11. Specify whether statistics counters should be reset to 0 each time they are written to the log file.

    Not required; the default is **yes**.

    Syntax:

    `reset-stat-counters=yes|no`

    Where:

    yes Resets the statistics counters.

    no Does not reset the statistics counters.

12. Define the path and file name to hold the transaction log for each cache. Cache transaction log files are separate from Cache Manager log files, which are used to log overall Cache Manager activity.

    Required; if not specified, a transaction log for the cache is not created.

Syntax:

```
tran-log=filename
```

Where *filename* is the path and file name of the transaction logs for each cache.

13. Specify whether to turn on transaction logging for the cache when the Cache Manager first starts up. This parameter is ignored unless a valid transaction log file is specified via the tran-log parameter. You can activate transaction logging while the Cache Manager daemon is running using the **cacheadm** command if a valid tran-log value has been specified in the Cache Manager configuration file.

    Not required; the default is **no**.

    Syntax:

    ```
    tran-logging=yes|on|no|off
    ```

    Where:

    yes|on
    Indicates that logging is required.

    no|off Indicates that logging should not be performed.

14. Specify whether the transaction log should be wrapped.

    Not required; the default is **yes**. If specified as **yes**, the current log is closed when it reaches the maximum size (see **tran-log-size**), has file type of .old, and a new log is opened. Only one generation of the log is maintained (existing .old files are overwritten).

    Syntax:

    ```
    wrap-tran-log=yes|no
    ```

    Where:

    yes Indicates to wrap the log.

    no Indicates to not wrap the log.

15. Specify the maximum size in bytes to which a transaction log is allowed to grow, if **wrap-tran-log** is specified.

    Not required; the default is 64000.

    Syntax:

    ```
    tran-log-size=bytes
    ```

    Where *bytes* is the number of bytes of the maximum size.

Stanza example: A valid Cache Definition stanza for a cache:

```
test1
{
caching=on
fssize=5MB
mem-size=10MB
lifetime=6000000
check-expiration=150
datum-memory-limit=5KB
datum-disk-limit=500KB
stat-interval=60
reset-stat-counters=no
root=/u/cached/chaches/cache0
stat-files=/u/cached/logs/chache0.stats
tran-log=/u/cached/logs/cache0.log
```

```
tran-logging=yes
wrap-tran-log=yes
tran-log-size=100k
}
```

Starting and Stopping the Cache Manager

The following sections describe how to start and stop the Cache Manager.I

- "Starting the Cache Manager"
- "Stopping the Cache Manager"

Starting the Cache Manager

Use the `cachemgrd` command to start the Cache Manager daemon.

Syntax:

Parameters:

cachemgrd
The command keyword.

config_file
Specifies the name of the file where the Cache Manager and each of the caches managed by the Cache Manager are defined. The configuration file shipped with the Net.Data product is `cachemgr.cnf`.

Example:

```
cachemgrd -c myconfig.cfg
```

Stopping the Cache Manager

Use the `cacheadm` command to stop the Cache Manager.

Syntax:

Parameters:

cacheadm
The command keyword.

hostname
Specifies the name of the machine where the cache is running, if it is different from the machine where the `cacheadm` command is issued.

port_num
Specifies the cache port number, if the number is different from the default (7175).

terminate
Specifies to stop the Cache Manager.

Example:

```
cacheadm hostname host1 port 7178 terminate
```

Caching Web Pages

You can use the DTW_CACHE_PAGE built-in function to cache a Web page. When Net.Data sees the DTW_CACHE_PAGE function in the macro file, it contacts the Cache Manager and begins saving the HTML output for the macro file in memory. After Net.Data successfully processes a macro, the HTML output is sent to the browser and the Cache Manager caches the output in one transaction as shown in Figure 25.

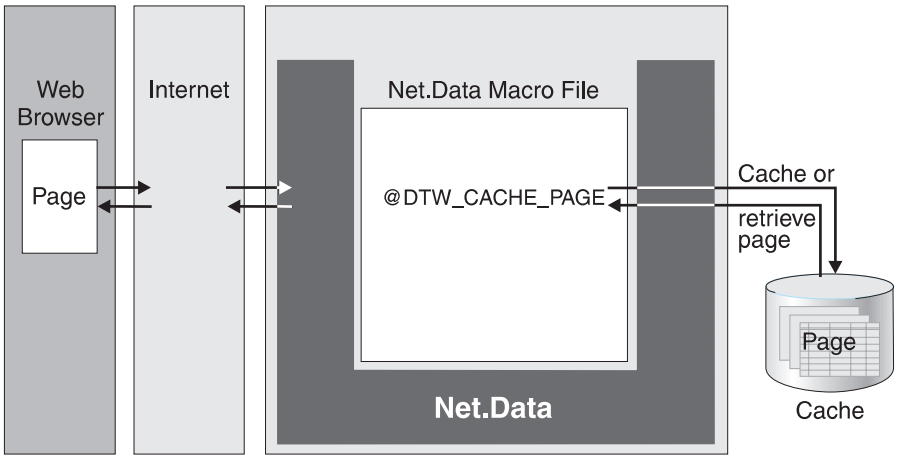


Figure 25. DTW_CACHE_PAGE Function Initiates Caching

Caching a Page

Specify Net.Data-generated pages to be written to the cache using the Net.Data DTW_CACHE_PAGE() built-in function.

The DTW_CACHE_PAGE() function caches all of the output from the macro file following the function statement, once it determines that the page does not already exist in the cache or has expired. If the page does not exist in the cache or is older than the specified age, Net.Data sends the output page back to the browser, generates new output pages from the macro execution, and stores the page in the cache.

If the Cache Manager finds the cached page and it is still current, it displays the cache contents and Net.Data exits out of the macro. This behavior ensures that no unnecessary processing is done after the Web page has been retrieved from the cache.

Performance tip: Place DTW_CACHE_PAGE() as the first, or one of the first statements in a macro file, to minimize the cost of executing the macro file.

To cache a page:

1. In the HTML block of a macro file, prior to the HTML coding, insert the following function statement:

   ```
   @DTW_CACHE_PAGE("cache_id", cached_page_id, "age", status)
   ```

   Use this function to specify that Net.Data is to cache all HTML output from the macro that follows this statement. Place this statement early in the macro file if you want to cache all HTML output.

   Parameters:

   cache_id
   : A string identifying the cache in which the page will be placed. You can associate cache IDs with macros or groups of macros.

   cached_page_id
   : A string containing an identifier used to identify the page in the cache in a subsequent @DTW_CACHE_PAGE cache request, for example the page's URL.

   age
   : A string variable containing a length of time in seconds that specifies when a page is considered out-of-date. If the requested page has been in the cache for longer than the value of *age*, Net.Data executes the macro, regenerates the page, and caches the generated page, replacing the out-of-date page. If the requested page has been in the cache for less than or the same as the value of age, Net.Data retrieves the page from the cache and sends it to the browser. In this case, Net.Data ends macro execution immediately.

   status
   : A string variable returned by Net.Data to indicate whether or not the page was cached successfully.

Example:

```
%HTML(cache_example) {
 %IF (customer == "Joe Smith")
 @DTW_CACHE_PAGE("mymacro.d2w", "http://www.mypage.org", "-1", status)
 %ENDIF
 ...
 <html>

<head>
 <:title>This is the page title</title>
 </head>

<body>
 <center>
 <h3>This is the Main Heading</h3>
 <p>It is $(time). Have a nice day!
 </body>

</html>
 %}
```

Advanced Caching: Determining Dynamically Whether to Cache

The DTW_CACHE_PAGE() function initiates caching from its location in the macro file. Normally, you place the function at the beginning of the macro file for better performance and to ensure that all HTML output is cached.

For advanced caching applications, you can place the DTW_CACHE_PAGE() function in the HTML output sections when you need to make the decision to cache at a specific point during processing, rather than at the beginning of the macro file.

For example, you might need to make the caching decision based on how many rows are returned from a query or function call.

Example: Places the function in the HTML block because the decision to cache depends on the expected size of the HTML output

```
% DEFINE { ...%}

...

%FUNCTION(DTW_SQL) count_rows(){
  select count(*) from customer
%REPORT{
 %ROW{
  @DTW_ASSIGN(ALL_ROWS, V1)
%}
%}
%}

%FUNCTION(DTW_SQL) all_customers(){
  select * from customer
%}

%HTML(OUTPUT) {
<html>
<head>
 <title>This is the customer list
 </head>
 <body>

@count_rows()

 %IF ($(ALL_ROWS) > "100")
 @DTW_CACHE_PAGE("mymacro.d2w", "http://www.mypage.org", "-1", status)
%ENDIF

@all_customers()

 </body>
 </html>
%}
```

In this example, the page is cached or retrieved based on the expected size of the HTML output. HTML output pages are considered cache-worthy only when the database table contains more than 100 rows. Net.Data always sends the text in the OUTPUT block, `This is the customer list`, to the browser after executing the macro; the text is never cached. The lines following the function call, `@count_rows()`, are cached or retrieved when the conditions of the IF block are satisfied. Together, both parts form a complete Net.Data output page.

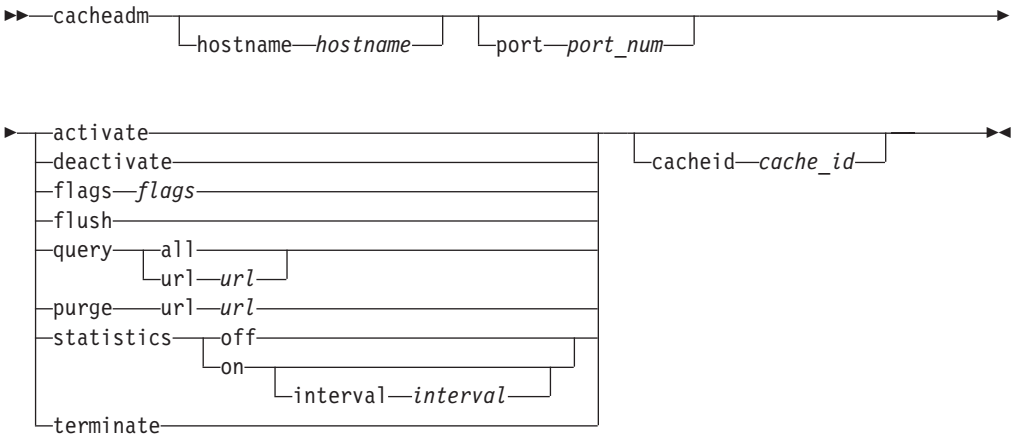
The CACHEADM Command

Use the CACHEADM command for the following tasks:

- Stopping the Cache Manager
- Flushing a specific cache
- Querying a specific cache
- Enabling or disabling logging
- Logging flags
- Starting and stopping statistics gathering

All parameters can be abbreviated to the minimum unique set of characters.

Syntax:



Parameters:

activate
Activates a specified cache. If the cache is already active, Cache Manager does nothing.

cache_id
A string variable identifying the cache in which the page is located. For example: `cache1`.

deactivate
Deactivates a specified cache. If the cache is already inactive, Cache Manager does nothing. All pending operations are completed and no new ones accepted. When the last operation is completed, the Cache Manager marks the cache as inactive.

flags
Specifies whether the listed flags should be toggled on or off.

D_ALL
Turn on all trace flags.

D_NONE
Turn off all trace flags.

flush
Flushes a cache, specified by the *cache_id* parameter, which is required by this parameter. This parameter unconditionally deletes all items from the specified cache.

hostname
Specifies the name of the machine where the cache is running, if it is different from the machine where the `cacheadm` command is issued. For example: `myhost`.

port_num
Specifies the cache port number, if the number is different from the default (7175). This number must be unique within the system.

purge
Specifies that a specific page to be purged from cache. If *url* is specified, Cache Manager purges the page with a key matching *url*. If dependency is specified,

Cache Manager purges all items with the associated dependency and writes their keys to `stdout`, the standard output stream.

query
Returns caching data, depending on the parameters you specify:

- Returns information about a cache, if only the cache ID is specified.
- Returns information about a specific cached page if *url* is specified.
- Returns information about all pages if `all` is specified.

  Other programs use `all` option to format or interpret the results. Each line contains the following information:

  – Page key
  – Page age
  – Page length
  – Page creation date
  – Page expiration date
  – Date the page was last referenced

  All dates are in standard UNIX integer time format.

  Performance tip: The option `cache query all` can impact performance and should be used sparingly.

statistics
Enables or disables logging of statistics gathering for a specific cache and requires *cache_id* parameter. If an interval is specified with the `statistics` parameter set to `on`, Net.Data sets or resets the interval between updates to the specified number of seconds.

terminate
Specifies to stop the Cache Manager.

tranlogging
Enables or disables transaction logging for a specific cache and requires the *cache_id* parameter. This parameter takes affect only if a valid transaction log for the cache is specified in the Cache Manager configuration file with the `tran-log` parameter.

url The Universal Relative Location (URL) address that specifies the location of the file on a Web server. For example: `http:www.ibm.com/mydir/page1`.

The Cache Log

Several types of statistics regarding internal operation are kept and optionally written to the cache log. A separate log for each cache can be maintained, or all statistics may be written to the same log. This section discusses the following cache log topics:

- "Configuring the Log"
- "Cache Log Format" on page 179

Configuring the Log

To log statistics, you must configure the Cache Manager configuration file.

To configure the log:

Specify the `stat-files` and `stat-interval` keywords in the cache stanza of the Cache Manager configuration file.

You can modify statistics settings without stopping, reconfiguring, and restarting the Cache Manager.

To modify statistics gathering settings:

Specify the `cacheadm statistics` command. Note however, that changes made with the `cacheadm statistics` command are not saved when the Cache Manager is restarted.

Cache Log Format

The statistics log is a plain ASCII file that can be processed or imported by spreadsheets or database programs. Three types of records are written:

- Initialization records document the startup of statistics gathering for a particular cache. These records have the following format:

  mm/*dd*/*yy hh*:*mm*:*ss id* `Initialization: interval` *n* `seconds`

  Where:

mm/dd/yy	Is the month, day, and year when statistics gathering starts
hh:mm:ss	Is the hour, minute, and second when statistics gathering starts
id	Is the name of the cache associated with the record
n	Is the collection interval

- Termination records document the termination of statistics gathering for a particular cache. These records have the following format:

  mm/*dd*/*yy hh*:*mm*:*ss id* `Termination`

  Where:

mm/dd/yy	Is the month, day, and year when statistics gathering stops
hh:mm:ss	Is the hour, minute, and second when statistics gathering stops
id	Is the name of the cache associated with the record

- Statistics records are a blank-delimited set of numbers showing activity within the cache. These records have the following format:

  mm/*dd*/*yy hh*:*mm*:*ss id statistics*

  Where:

mm/dd/yy	Is the month, day, and year when statistics gathering is created
hh:mm:ss	Is the hour, minute, and second when statistics gathering is created
id	Is the name of the cache associated with the record
<*statistics***>**	Is a blank-delimited list of statistics gathered for this cache as specified in Table 14 on page 180:

Table 14. List of Statistics

Field Number	Contents	Description	Counters Reinitialized to Zero
1	reads	Number of read operations performed against the cache	Yes
2	writes	Number of write operations performed against the cache	Yes
3	closes	Number of close operations performed on objects in the cache	Yes
4	open read	Number of open-read operations performed on objects in the cache	Yes
5	open write	Number of open-write operations performed on objects in the cache	Yes
6	open write query	Number of open-write-query operations performed on objects in the cache	Yes
7	read hits	Number of read hits on objects in the cache	Yes
8	write hits	Number of write hits on objects in the cache	Yes
9	write query hits	Number of write query hits on objects in the cache	Yes
10	initializations	Number of new sessions established with this cache	Yes
11	terminations	Number of sessions terminated with this cache	Yes
12	purges	Number of objects deleted from this cache	No
13	memory used	Amount of memory used by objects in the memory portion of the cache	No
14	disk used	Amount of disk space used by objects in the disk portion of the cache	No
15	memory available	Amount of memory still available for use by objects in the memory portion of the cache	No
16	disk available	Amount of disk space still available for use by objects in the disk portion of the cache	No
17	memory object count	Number of objects in the memory portion of the cache	No
18	file object count	Number of objects in the disk portion of the cache	No
19	session count	Number of sessions currently active against the cache	No

Setting the Error Log Level

Net.Data provides an error log so that you can monitor errors or performance problems in your Net.Data system.

When using the Net.Data error log, you might notice an impact on the performance of your system if many messages are being written to the error logs. For example, each time a user accesses a macro that Net.Data cannot find, Net.Data passes a message as output to the error log.

To reduce the impact on performance, check the logging level of the error log set in a Net.Data macro with the DTW_LOG_LEVEL keyword. If the level is set to WARNING, consider reducing the level to ERROR for a small performance gain, or to OFF for a larger performance gain.

Optimizing the Language Environments

The following sections describes techniques you can use to improve performance when using the Net.Data-provided language environments.

- "REXX Language Environment"
- "SQL Language Environment"
- "System and Perl Language Environments" on page 182

REXX Language Environment

Use the following tips to improve the performance of your Net.Data application:

- Combine your REXX programs where possible. Having fewer, larger programs provides better performance than more smaller programs because the REXX interpreter is initialized each time a REXX language environment function is called in the macro.
- Store the REXX program in an external file instead of including the REXX program inline in the Net.Data macro.
- For external REXX programs, reference the global variables on the command line in the %EXEC statement.
- Pass input-only parameters directly to a REXX program by defining global Net.Data variables and referencing the variables. For inline REXX programs, reference the global variables directly in your REXX source.

SQL Language Environment

In the sections that follow, some performance techniques about the database and SQL language environment are described. To learn about DB2 performance considerations, visit the web at:
http://review.software.ibm.com/data/db2/performance

Database Techniques

The following summary outlines some of the most simple database techniques that can improve database access:

- Activate the database. By issuing the command `db2 activate database`, connections to the database are made in significantly less time. See *DB2 Administration Guide*. When running Net.Data using CGI, this performance technique is a good alternative to using Live Connection, which is required for FastCGI or Web APIs.
- Avoid numeric conversion. When a column value and a literal value are being compared, try to specify the same data types and attributes. DB2 does not use an index for the named column if the literal value has a greater precision than the precision of the column. If the two items being compared have different data types, DB2 will have to convert one or the other of the values, which can result in inaccuracies (because of limited machine precision).

  For example, EDUCLVL is a halfword integer value (SMALLINT). Specify:

  ```
  ... WHERE EDUCLVL < 11 AND EDUCLVL >= 2
  ```

Instead of:

```
... WHERE EDUCLVL < 1.1E1 AND EDUCLVL > 1.3
```

- Avoid character string padding. Try to use the same data length when comparing a fixed-length character string column value to a literal value. DB2 does not use an index if the literal value is longer than the column length.

  For example, EMPNO is CHAR(6) and DEPTNO is CHAR(3). Specify:

  ```
  ... WHERE EMPNO > '000300' AND  DEPTNO < 'E20'
  ```

  Instead of:

  ```
  ... WHERE EMPNO > '000300 ' AND DEPTNO < 'E20 '
  ```

- Avoid the use of LIKE patterns beginning with % or _. The percent sign (%), and the underline (_), when used in the pattern of a LIKE predicate, specify a character string that is similar to the column value of rows you want to select. When used to denote characters in the middle or at the end of a character string, LIKE patterns can take advantage of indexes. For example:

  ```
  ... WHERE LASTNAME LIKE 'J%SON%'
  ```

  However, when used at the beginning of a character string, LIKE patterns can prevent DB2 from using any indexes that might be defined on the LASTNAME column to limit the number of rows scanned. For example:

  ```
  ... WHERE LASTNAME LIKE '%SON'
  ```

  Avoid using these symbols at the beginning of character strings, especially if you are accessing a particularly large table.

SQL Language Environment Techniques

Use the following SQL language environment techniques to improve performance.

- Use the START_ROW_NUM and RPT_MAX_ROWS Net.Data variables to reduce the size of returned tables. If a result set contains a large number of rows, you can specify a subset of the result set that is returned to the browser by using START_ROW_NUM and RPT_MAX_ROWS. START_ROW_NUM specifies the row number of the first row to return, and RPT_MAX_ROWS specifies the number of rows to return.

  Important: Net.Data reissues the query for every request because cursor position is not maintained across requests.

- Consider calling a stored procedure that uses static SQL. Dynamic SQL is prepared at run time, while static SQL is prepared at the precompile stage. The SQL language environment uses dynamic SQL, which allows it to run SQL statements at program run time. Because preparing statements requires additional processing time, static SQL might be more efficient.

System and Perl Language Environments

Pass input-only parameters directly to the program that the System or Perl language environment is invoking. Do this by defining global Net.Data variables and referencing them. For external programs and Perl scripts, reference the variables on the command line in the %EXEC statement. For inline Perl scripts, reference the variables directly in the Perl source.

Chapter 8. Net.Data Logging

Net.Data provides several logs for you to use when monitoring Net.Data performance and troubleshooting errors. Net.Data logs include:

Net.Data error message log
Log containing all Net.Data error messages.

Live Connection log
Log containing Live Connection error messages and communication between Net.Data, Live Connection and the DB2 database. Logging is available for the DTW_SQL and DTW_ODBC language environments for DB2 databases.

The following sections describe Net.Data logging:

- "Logging Net.Data Error Messages"
- "Logging Live Connection Cliette and Error Messages" on page 185

Logging Net.Data Error Messages

Net.Data writes error messages to the Net.Data error log file, `netdata.log`. The maximum size of the error log is fixed by Net.Data at 500 KB, approximately 3000 log entries.

You can browse the error log file, or archived copies, periodically to determine whether there are problems in your Net.Data system.

To activate the Net.Data error log:

- Set the Net.Data logging configuration variable, DTW_LOG_DIR:
  `DTW_LOG_DIR` *`path`*

  Where *path* is the directory where you want the error log file to be stored.
- Set the Net.Data logging variable in the macro file, DTW_LOG_LEVEL:
  `@DTW_ASSIGN(DTW_LOG_LEVEL, "`*`level`*`")`

  Where *level* is the level of logging. It can have the following values:

  off Net.Data does not log errors. This is the default.

  error Net.Data logs error messages.

  warning
  Net.Data logs warnings, as well as error messages.

This section discusses the following logging topics:

- "Planning for the Net.Data Error Log" on page 184
- "Controlling the Net.Data Logging Level" on page 184
- "Types of Net.Data Error Messages Not Logged" on page 184
- "Net.Data Error Log File Size and Rotation" on page 184
- "Net.Data Error Log Format" on page 185

Planning for the Net.Data Error Log

When logging errors, you need to plan for the following issues:

- Determining disk space:

  If you plan to use error logging, you must allow additional disk space for the error logs.
- Configuring Net.Data:

  If you plan to control the error logging for the whole Net.Data system, set one configuration variable in your Net.Data initialization file: DTW_LOG_DIR

  This variable is required for error logging, even if you have set the DTW_LOG_LEVEL variable in your macro to error or warning. See "DTW_LOG_DIR: Error Log Location Variable" on page 14 to learn how to update the initialization file.
- Writing Net.Data macros:

  Set the level of logging with the DTW_LOG_LEVEL keyword in your macro.
- Running Net.Data:

  If you are using error logging, then you can check the error logs and archive files for errors in your Net.Data system.
- Tuning:

  Be aware that logging can affect performance. See "Setting the Error Log Level" on page 180 for information about performance issues.

Controlling the Net.Data Logging Level

You can specify the level of logging with the DTW_LOG_LEVEL variable. Define this keyword in the Net.Data macro. The variable has three settings:

off Net.Data does not log errors. This is the default.

error Net.Data logs error messages.

warning
Net.Data logs warnings, as well as error messages.

Types of Net.Data Error Messages Not Logged

Net.Data does not log errors explicitly handled by a MESSAGE section in the macro.

Net.Data Error Log File Size and Rotation

The maximum size of the log file can be 500 KB. At this size, approximately 3000 log entries will fit.

When the log file reaches the maximum size, the file will be archived to netdata.log*MMMDDYYYY_nn*

Where:

MMM The month (Jan-Dec)

DD The date

YYYY The year

nn A number from 01 to 99, that uniquely identifies each archive file for a certain day.

Logging continues in the original file.

Net.Data Error Log Format

Log file entries have the following format:
`[DD/MMM/YYYY:HH:MM:SS] [MACRO] [BLOCK] [PID#] [TID#]error_message`

Parameters:

DD The date

MMM The month (Jan-Dec)

YYYY The year

HH The hour (00-23)

MM The minute (00-59)

SS The number of seconds (00-59)

MACRO
The macro that generated the error message

BLOCK
The name of the HTML block that generated the error message.

PID# The process ID number of the process that generated the error message. This ID is necessary because multiple Net.Data processes can be writing to the log file.

TID# The thread ID number of the thread that generated the error message. This ID is necessary because multiple threads from the same Net.Data process can be writing to the log file.

error_message
The text of the error message

Logging Live Connection Cliette and Error Messages

Live Connection records messages and communication between Live Connection, Net.Data, and the DB2 database in the Live Connection log file. The maximum size of the log is fixed by Net.Data at 1 MB, approximately 1200 log entries.

You can browse the log file, or archived copies, periodically to determine whether there are problems with your cliettes or the DB2 database.

To activate the Live Connection log:

Start the Connection Manager with the `-l` attribute:
`dtwcm -l [level]`

Where *level* is the level of logging. It can have the following values:

normal
Live Connection logs all cliette activities, related DB2 SQL statements and status messages, and Live Connection error messages

minimal
Live Connection logs only essential information, such as database queries and the number of rows in the result set.

This section discusses the following logging topics:

- "Planning for the Live Connection Log"
- "Controlling the Live Connection Logging Level"
- "Types of Live Connection Messages Not Logged"
- "Live Connection Log File Names" on page 187
- "Live Connection Log File Size and Rotation" on page 188
- "Live Connection Log Format" on page 188

Planning for the Live Connection Log

When logging messages, you need to plan for the following issues:

- Determining disk space:

  If you plan to use error logging, you must allow additional disk space for the log files.
- Running Connection Manager:

  You activate logging by entering an attribute on the `dtwcm` command. See "Logging Live Connection Cliette and Error Messages" on page 185 for syntax.

  If you are using logging, then you can check the logs and archive files for errors in your cliettes.
- Tuning:

  Be aware that logging can affect performance. See "Setting the Error Log Level" on page 180 for information about performance issues and "Controlling the Live Connection Logging Level"

Controlling the Live Connection Logging Level

You can specify the level of logging on the `dtwcm` command, while invoking Connection Manager. The `-l` attribute of the `dtwcm` command has two settings:

normal
Live Connection logs all cliette activities, related DB2 SQL statements and status messages, and Live Connection error messages

minimal
Live Connection logs only essential messages. This options provides fewer messages in the log.

Types of Live Connection Messages Not Logged

Live Connection does not log Net.Data errors or errors explicitly handled by a MESSAGE section in the macro.

Live Connection Log File Names

Live Connection creates a log file for the Connection Manager and for each cliette. The following list describes the name formats:

Connection Manager file

Format:

conman-*process_id*-*DDMMMYYYYHHMMSS*.log

Parameters:

process_id
The identifier of the Connection Manager process

DD
The date

MMM
The month (Jan-Dec)

YYYY
The year

HH
The hour, 24–hour clock

MM
The minutes

SS
The seconds

Example:

conman-513-01Feb1999095639.log

Cliette file

Format:

cliett-*process_id*-*DDMMMYYYYHHMMSS*.log

Parameters:

process_id
The identifier of the cliette thread

DD
The date

MMM
The month (Jan-Dec)

YYYY
The year

HH
The hour, 24–hour clock

MM
The minutes

SS
The seconds

Example:

```
cliett-592-01Feb1999095647.log
```

Live Connection Log File Size and Rotation

The maximum size of the log file can be 1 MB. At this size, approximately 6000 log entries will fit. When the log file reaches the maximum size, the process will close the original log file, create a new log file, and continue logging to the new file.

Log files are located in the same directory as `dtwcm` and `dtwcdb2`

Live Connection Log Format

Log file entries have the following format:

```
--process_type-DD/MMM/YYYY:HH:MM:SS-PID#--
message_text
```

Parameters:

process_type
: Either `dtwcm` or `cliet`, depending upon whether the connection manager or a cliette logged the message.

DD The date

MMM The month (Jan-Dec)

YYYY The year

HH The hour (00-23)

MM The minute (00-59)

SS The number of seconds (00-59)

PID# The process ID number of the process that generated the message.

message_text
: The text of the message.

Example 1: A connection manager log entry.

```
--dtwcm-02/Mar/1999:13:43:07-330--
Creating connection manager ...successfully
Reading configuration info ...
Completing initialization ...
Initializing cm server ... successfully
Initializing NLS environment ... successfully
Detecting cliette ./dtwcdb2 for DTW_SQL:CELDIAL:  Min process(es) = 1, Priv Port = 7100.
Starting 1 cliettes for DTW_SQL:CELDIAL.
Started: ./dtwcdb2 7128 7100 7200 DTW_SQL:CELDIAL  LOG_MAX , pid: 213
1 cliettes for DTW_SQL:CELDIAL started.
...
```

Example 2: A cliette log entry.

```
--cliet-02/Mar/1999:13:43:08-335--
Cliette starting ...
Cliette: DTW_SQL:SAMPLE, database: SAMPLE, user: *USE_DEFAULT
Making a new connection to database: SAMPLE, user: *USE_DEFAULT.
Calling SQLAllocHandle for environment ...
Calling SQLAllocHandle for connection ...
Calling SQLSetConnectAttr ...
Calling SQLConnect ...
```

```
Connecting to database: SAMPLE sucessfully.
Telling CM the cliette is ready ...
Ready and waiting for command from CM ...
```

Appendix A. Bibliography

This section lists the documents referred to in this book.

- "Net.Data Technical Library"

Net.Data Technical Library

The Net.Data Technical Library is available from the Net.Data Web site at
`http://www.software.ibm.com/data/net.data/library.html`

Document	Description
• *Net.Data Administration and Programming Guide for OS/390* • *Net.Data Administration and Programming Guide for OS/2, Windows NT, and UNIX* • *Net.Data Administration and Programming Guide for OS/400*	Contains conceptual and task information about installing, configuring, and invoking Net.Data. Also describes how to write Net.Data macros, use Net.Data performance techniques, use Net.Data language environments, manage connections, and use Net.Data logging and traces for trouble shooting and performance tuning.
Net.Data Reference	Describes the Net.Data macro language, variables, and built-in functions.
Net.Data Language Environment Interface Reference	Describes the Net.Data language environment interface.
Net.Data Messages and Codes Reference	Lists Net.Data error messages and return codes.

Appendix B. Net.Data for AIX

Details for AIX are included in the README file that is shipped with Net.Data. The README file includes the following information:

- Requirements
- Installing
- Configuring
- Uninstalling

Loading Shared Libraries for Language Environments

When creating a language environment on the AIX platform, you need to load shared libraries. On AIX, the language environment is required to provide a routine that is called by Net.Data and returns the addresses of the language environment interface routines such as dtw_initialize() and dtw_execute().

Net.Data uses the dtw_fp structure to retrieve pointers to the language environment interface routines from a language environment in AIX, and has this format:

```
typedef struct dtw_fp {
    int (* dtw_initialize_fp)();   /* dtw_initialize function pointer */
    int (* dtw_execute_fp)();      /* dtw_execute function pointer    */
    int (* dtw_cleanup_fp)();      /* dtw_cleanup function pointer    */
} dtw_fp_t;
```

This structure is passed to the language environment by Net.Data as a parameter in the dtw_getFp() routine when the shared library is loaded.

The dtw_fp structure is passed as the only parameter. This structure contains a field for each supported interface, and it is the language environment's responsibility to set these fields. If the language environment is providing the specified interface, it sets the field to the function pointer of that interface. If it is not providing the specified interface, it sets the field to NULL. The dtw_getFp() routine in the program template shows a correct implementation of this routine.

In order for Net.Data to get the pointer to this routine when the shared library is loaded, the dtw_getFp routine must be the first entry point specified in the shared library's export file. A sample export file for a library called dtwsampshr.o that supports all available language environment interface routines looks like this:

```
#!dtwsampshr.o
dtw_getFp
dtw_initialize
dtw_execute
dtw_cleanup
```

Improving Performance in the REXX Environment

If you have many calls to the REXX language environment on your AIX system, consider setting the RXQUEUE_OWNER_PID environment variable to 0. Macros that make many calls to the REXX language environment can easily spawn many processes, swamping system resources.

You can set the environment variable in one of three ways:

- In the macro by using the DTW_SETENV built-in function:

  ```
  @DTW_rSETENV("RXQUEUE_OWNER_PID", "0")
  ```

- In the AIX system environment file:

  ```
  /etc/environment: RXQUEUE_OWNER_PID = 0
  ```

  This method affects the behavior of REXX for the whole machine.
- In the HTTP Web server environment file; for example, for Domino Go Webserver, insert the following statement:

  ```
  InheritEnv RXQUEUE_OWNER_PID = 0
  ```

  This method affects the behavior of REXX for the Web server.

NLS Considerations

Net.Data runs using the same codepage as the web server. For Net.Data to use the correct codepage for your Locale, the Web server must be using the correct codepage. For example, if you are using an IBM Internet Connection Server and want to use a Korean codepage, stop the server and restart it using the Korean locale:

```
stopsrc httpd
startsrc -s httpd -e "LC_ALL=ko_KR"
```

If your macro contains UTF-8 characters or you are connecting to a DB2 database containing Unicode data, set the DTW_UNICODE configuration variable in the INI file to YES. Net.Data currently supports UTF-8 characters in the macro, but not UTF-16. However, Net.Data can process database data that is encoded in either UTF-8 or UTF-16. Net.Data output is always in UTF-8.

Performance Tip: If you are running in a double-byte locale, and your string or work built-in functions always process single byte character strings, set DTW_MBMODE to NO to save unnecessary conversion.

Appendix C. Net.Data SmartGuides

The Net.Data SmartGuides are designed to provide you with a quick and easy way to create customized Net.Data applications. Simply select a SmartGuide, answer a few questions, and Net.Data creates a customized application for you.

Net.Data provides the following SmartGuides for you to use while learning how to create macros and use Net.Data features:

Drilldown
: This SmartGuide takes your existing database tables and creates a Web-enabled drilldown application that allows you to access different levels of detail of your data. Optionally, the Drilldown SmartGuide can connect to your database to collect your database information. You can customize up to five Web reports. The generated macro uses primary and foreign key information stored in your database to link your Web reports automatically.

Stored Procedure
: This SmartGuide connects to your database and retrieves a list of all stored procedures that are registered with your database. Select a stored procedure and the SmartGuide generates a Net.Data macro for you that calls your stored procedure. You can then modify the generated macro or integrate it in your existing Net.Data applications.

Contacts
: This SmartGuide generates a Web-enabled address book for storing names, addresses, phone numbers, and other important contact information. It comes with a search function that gives you quick access to your contacts. The generated contacts application can include either a brief or a detailed report. Additionally, you can include a customized report.

Check the Net.Data Web site at `http://www.software.ibm.com/data/net.data` for the latest version of the Net.Data SmartGuide package.

This appendix discusses the following topics:

Before You Begin

To run the SmartGuides and generate the Net.Data macros, you must have the following software installed:

- Java Development Kit (JDK) or Java Runtime Environment (JRE) 1.1.x
- Net.Data Version 2 or higher
- IBM Universal Database (UDB) 5.0 or higher
- REXX (required for the Drilldown SmartGuide)

Running the SmartGuides

The Net.Data SmartGuides are started from the command line and contained in the file, `NetDataSmartGuides.jar`.

To start a SmartGuide with the Java Development Kit (JDK):

1. Add the following line to your CLASSPATH environment variable.

   [Path]`NetDataSmartGuides.jar`

   where *[Path]* is the optional path to the `NetDataSmartGuides.jar` file.
2. If you want to use the database connect feature of the Drilldown and Stored Procedure SmartGuide, add the UDB JDBC driver to your CLASSPATH environment variable.

   For Windows NT and OS/2 operating systems, add the following line to your CLASSPATH environment vairable:

   [Path]`NetDataSmartGuides.jar;`*[UDBInstallationPath]*`\java\db2java.zip`

   For UNIX operating systems, add the following line to your CLASSPATH environment vairable:

   [Path]`NetDataSmartGuides.jar:`*[UDBInstallationPath]*`\java\db2java.zip`

   Where *[Path]* is the optional path to the NetDataSmartGuides.jar file and *[UDBInstallationPath]* is the path to your UDB installation, for example C:\SQLLIB.
3. Start the SmartGuide.
   - For UNIX operating systems, type the following command:

     `java TaskGuide LaunchPad.class`
   - For Windows NT and OS/2 operating systems, run the following file:

     `SmartGuides.cmd`

   A launchpad opens with the list of available SmartGuides as shown in Figure 26 on page 197.

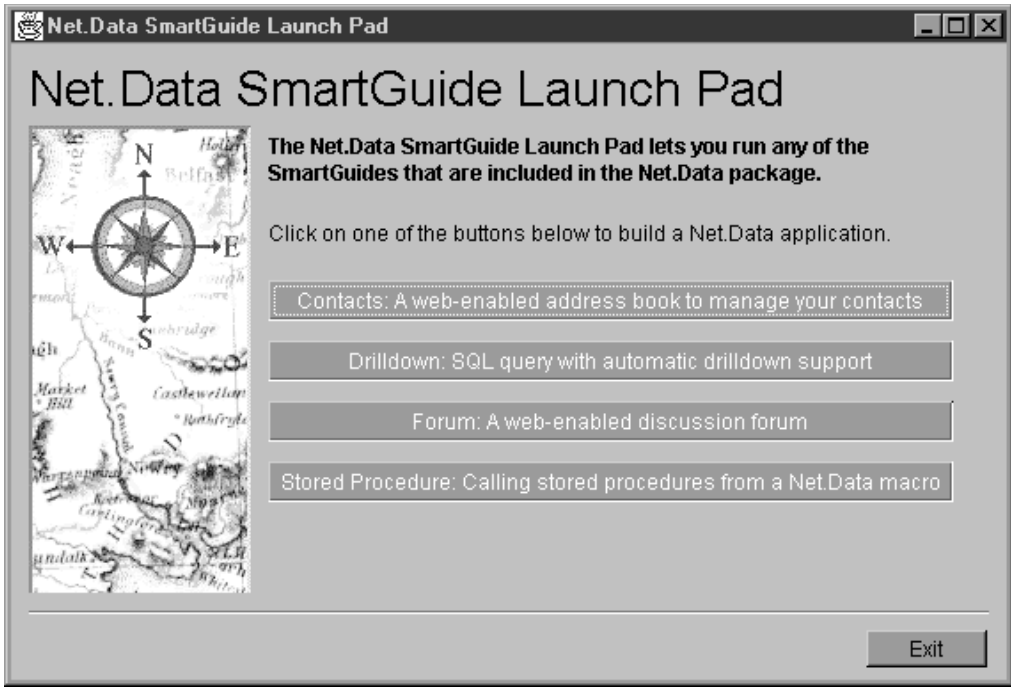


Figure 26. The SmartGuide Lauchpad

4. Click on the name of the SmartGuide that you want to run.

To start a SmartGuide with the Java Runtime Environment (JRE):

1. For the Windows NT operating system, choose **Start->Programs->Net.Data->SmartGuides**, which runs a batch file called SMARTGUIDES.BAT. For other operating systems, type the following command to run the SmartGuides:

   ```
   jre -cp [Path]NetDataSmartGuides.jar TaskGuide LaunchPad.class
   ```

   where *[Path]* is the optional path to the `NetDataSmartGuides.jar` file.

   A launchpad opens with the list of available SmartGuides as shown in Figure 26.
2. If you want to use the database connect feature of the Drilldown and Stored Procedure SmartGuide, type the following command:

   For Windows NT and OS/2 operating systems:

   ```
   jre -cp [Path]NetDataSmartGuides.jar;[UDBInstallationPath]
     \java\db2java.zip TaskGuide LaunchPad.class
   ```

   For UNIX operating systems:

   ```
   jre -cp [Path]NetDataSmartGuides.jar:[UDBInstallationPath]
     \java\db2java.zip TaskGuide LaunchPad.class
   ```

   Where *[Path]* is the optional path to the NetDataSmartGuides.jar file and *[UDBInstallationPath]* is the path to your UDB installation, for example C:\SQLLIB.
3. Click on the name of the SmartGuide that you want to run.

Appendix D. Building SQL Statements with Net.Data SQL Assist

The Net.Data SQL Assist is a Java-based SQL statement builder that provides an easy-to-use GUI to guide you through the process of building SQL statements. With Net.Data SQL Assist you can:

- Build SELECT, INSERT, UPDATE, and DELETE statements (including SELECT DISTINCT)
- Build multiple conditions using value lookup, AND or OR, and type sensitive entry fields
- Define table JOINS (inner, right outer, and left outer)
- Select columns to view
- Select the sort order
- Enter user-defined variables to be used in conditions, values, and sorting

After building the SQL statement, you can:

- Save the SQL statement as a file
- Generate and save a macro containing the SQL statement
- Copy the SQL statement or macro to the clipboard

This appendix discusses the following topics:

- "Before You Begin"
- "Running Net.Data SQL Assist"

Before You Begin

To run the Net.Data SQL Assist, you must have the following software installed:

- Java Development Kit (JDK) or Java Runtime Environment (JRE) 1.1.x
- A JDBC enabled database

Refer to your database documentation for further details on accessing the data source through JDBC and any other server start-up that might be required. For example, when accessing a DB2 UDB v5.0 datasource remotely, the database server must have the JDBC server (db2jstrt) running.

Running Net.Data SQL Assist

Net.Data SQL Assist is started from the command line and contained in the file, {*inst_dir*}/assist/NetDataAssist.jar.

To start Net.Data Assist with the Java Development Kit (JDK):

Enter the following command to start Net.Data Assist:

```
java -classpath %CLASSPATH%;{inst_dir}/assist/NetDataAssist.jar NetDataAssist
```

To start Net.Data Assist with the Java Runtime Environment (JRE):

Enter the command to start Net.Data Assist:

```
jre -cp %CLASSPATH%;{inst_dir}/assist/NetDataAssist.jar NetDataAssist
```

Click on the **Next** button to navigate through the windows used to logon, construct an SQL statement, and generate a Net.Data macro.

Appendix E. Using NetObjects Fusion NOF Plug-ins with Net.Data Servlets

Net.Data provides a NetObjects Fusion plug-in for the Net.Data servlets. Net.Data servlets are described in “Net.Data Servlets” on page 72.

You can use NetObjects Fusion (NOF) to integrate your existing Net.Data macros, which will provide better integration with your Web site management and an easy-to-use graphical user interface.

This appendix discusses the following topics:

About the NetObjects Fusion Plug-in

The NetDataServlet.NFX plug-in works with the Net.Data servlets. This NOF plug-in supports the invocation of an existing Net.Data macro or of a single Net.Data function. The plug-in generates the HTML to invoke Net.Data as a servlet or a server-side-include (SSI). When the Web server invokes Net.Data, the Net.Data macro or function is run. Use the Net.Data servlet plug-in to embed either of the macro and function servlets into an NOF-managed Web site, which Figure 27 on page 202 describes. The plug-in provides many of the basic macro or function parameters, with defaults selected to automate building a macro. To use the plug-in, you must install and configure NOF and the plug-in files.

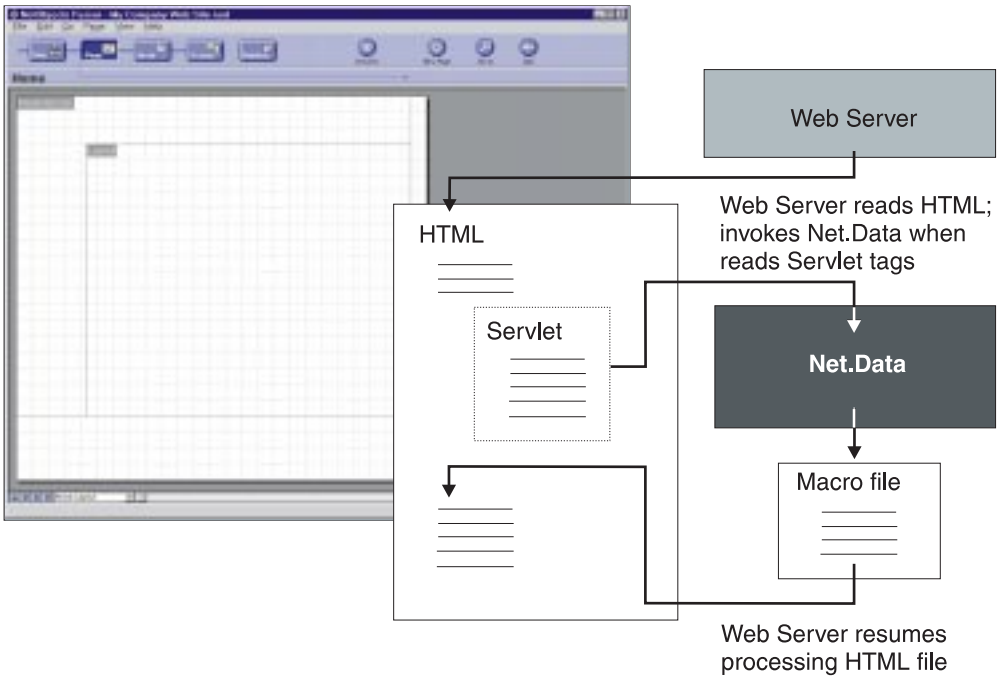


Figure 27. Net.Data Servlets Plug-ins

Installing the NetObjects Fusion Plug-in

Software and Hardware Requirements:

The NOF plug-ins require NetObjects Fusion Version 2.0 or later.

To Install: Copy the <inst_dir>\fx\NetDataServlet.nfx and <inst_dir>\fx\NetDataServlet.gif to your <NetObjects Fusion>\components\ directory.

Setting Up the Net.Data Plug-in for NetObjects Fusion

You can change the properties of the servlet with which you are working using NOF.

1. Open NetObjects Fusion.
2. From the NetObjects Fusion's (NOF) **Tools** palette, select the **NetObjects Components** button:  The plug-in buttons are displayed on the bottom of the **Tools** palette.
3. From these six **Tools** palette buttons, select the **NetObjects Components** button: 
4. On the NOF canvas, mark the area to specify where you want to place the selected plug-in. This is where the results of the servlet will be displayed. The Installed Components window opens, displaying a list of plug-ins from which to choose. If the servlet plug-in is not in the list, use the path and file name fields to specify the plug-in file name: `NetDataServlet.NFX` to use with the macro or function servlet
5. Select the servlet plug-in from the list and click on the **OK** push button.
   The plug-in becomes an object on the NOF canvas.

Modifying the Plug-in Properties

You can modify the macro and function servlets using the Net.Data servlet plug-in.

To modify the Net.Data servlet with NOF:

1. On the NOF canvas, mark the area to specify where you want to place the selected plug-in. This is where the results of the servlet will be displayed. The Installed Components window opens, displaying a list of plug-ins from which to choose. If the servlet plug-in is not in the list, use the path and file name fields to specify the plug-in file name, `NetDataServlet.NFX` to use with the macro or function servlet.
2. Select the Net.Data servlet plug-in from the list and click on the **OK** push button.

   The plug-in becomes an object on the NOF canvas.
3. Select and customize the properties of the Net.Data servlet plug-in:
   a. Select the Net.Data servlet plug-in on the NOF canvas. The NOF **Properties** palette opens, displaying the plug-in properties as shown in Figure 28.

Figure 28. The Net.Data Servlet Properties Palette

Properties for the Net.Data servlets:

You can customize the following properties:

Servlet name

Select the name of the servlet you want to invoke: Function or Macro. Depending on which servlet name you select, different properties are displayed.

Servlet type
Select the type of servlet you want: SSI, HREF, or FORM Submit Button. Depending on which type of servlet you want, different properties are displayed.

Submit label
If you select a FORM Submit Button type, specify the text for the submit label. Otherwise, this property is not displayed.

Server URL
If you select an HREF servlet type, specify the server URL to the servlet-enabled Web server. If you select SSI, this property is not displayed.

Macro name
If you select the macro servlet name, specify the name of a existing Net.Data macro to execute. Otherwise, this property is not displayed.

HTML block name
If you select the macro servlet name, specify the name of HTML block in the Net.Data macro to run. Otherwise, this property is not displayed.

Function type
If you select the function servlet name, select the type of function to execute: Function or SQL. Otherwise, this property is not displayed.

Language env
If you select the function servlet name, specify the Net.Data language environment to use. Otherwise, this property is not displayed.

Statement
If you select the function servlet name, specify the statement to execute. Otherwise, this property is not displayed.

Database
If you select the function servlet name, specify the name of the database to use. Otherwise, this property is not displayed.

# of other parameters
Specify the number of other parameters to pass to Net.Data (maximum: 25). For each parameter, enter the name of the parameter and its value (optional).

Before HREF text
If you select an HREF servlet type, specify the text to appear before the text to appear before the <A HREF> HTML tag. Otherwise, this property is not displayed (optional).

Inside HREF text
If you select an HREF servlet type, specify the text to appear inside of the <A HREF> HTML tag. Otherwise, this property is not displayed (optional).

After HREF text
If you select an HREF servlet type, specify the text to appear after the text to appear after the <A HREF> HTML tag. Otherwise, this property is not displayed (optional).

SQL Reminder!

If you select an HREF servlet type and specify an SQL function type, a message displays with a reminder that the HREF SQL statement should use a plus (+) character for any space ( ) characters. This text cannot be changed, nor is it displayed after the page is published. Otherwise, this property is not displayed.

b. After you have defined the properties for your page, click on the **Publish** push button to build and publish the Web pages with the Net.Data servlet NOF plug-in.

Note: If you select a SSI servlet type, your Web page file extension should be `.shtml`. You can set this as your page default from the NOF **Properties** palette, selecting the **Page** notebook tab, clicking on the **Custom names** push button, and entering `.shtml` in the **Extension Type** field.

Publishing Servlets with the NOF Plug-in

After you have set the properties for your page, click on the **Publish** push button to build and publish the Web pages with the plug-in.

Appendix F. Net.Data Sample Macro

This sample macro application displays a list of employees names from which the application user can obtain additional information about an individual employee by selecting the employee's name from the list. The macro uses the SQL language environment to query the EMPLOYEE table for both the employee names and the information about a specific employee.

The macro file uses an include file that contains the DEFINE block for the macro.

Figure 29 on page 208 shows the sample macro. Figure 30 on page 210 shows the include file.

```
%{************************ Sample Macro ****************************
*   FileName = sqlsamp1.d2w                                          *
*   Description:                                                     *
*     This Net.Data macro queries...                          *
*         - The EMPLOYEE table to create a selection list of         *
*          employees for display at a browser                        *
*         - The EMPLOYEE table to obtain additional information      *
*          about an individual employee                              *
*                                                                    *
**********************************************************************%}
%{*****************************************************************************
*   Include for global DEFINEs -                                                *
*****************************************************************************%}
%INCLUDE "sqlsamp1.hti"
%{*****************************************************************************
*  Function:  queryDB              Language Environment: SQL                  *
*  Description: Queries the table designated by the variable myTable and      *
*  creates a selection list from the result.  The value of the variable       *
*  myTable is specified in the include file sqlsamp1.hti.                     *
*****************************************************************************%}
%FUNCTION(DTW_SQL) queryDB() {
 SELECT FIRSTNME FROM $(myTable)
%MESSAGE {
    -204:  {<p><b>ERROR -204: Table $(myTable) not found. </b>
           <p>Be sure the correct include file is being used.</b>
           %} : exit
    +default: "WARNING $(RETURN_CODE)" : continue
    -default: "Unexpected ERROR $(RETURN_CODE)" : exit
%}

%REPORT {
<select name=emp_name>
%ROW{
<option>$(V1)
%}
</select>
%}
%}

%{*****************************************************************************
*  Function: fname         Language Environment: SQL                          *
*  Description: Queries the table designated by the variable myTable for      *
*               additional information about the employee identified by the   *
*               variable emp_name.                                            *
*****************************************************************************%}
%FUNCTION(DTW_SQL) fname(){
SELECT FIRSTNME, PHONENO, JOB FROM $(myTable) WHERE FIRSTNME='$(emp_name)'
%MESSAGE {
  -204: "Error -204: Table not found "
  -104: "Error -104: Syntax error"
   100: "Warning 100: No records" : continue
  +default: "Warning $(RETURN_CODE)" : continue
  -default: "Unexpected SQL error" : exit
%}
%}
```

Figure 29. Sample macro (Part 1 of 3)

```
%{************************************************************************
*  HTML block: INPUT            Title: Dynamic Query Selection        *
*                                                                      *
*  Description: Queries the EMPLOYEE table to create a selection list *
*               of the employees for display at the browser           *
************************************************************************%}
%HTML(INPUT) {
<html>
<head>
<title>Generate Employee Selection List</title>
</head>
<body>
<h3>$(exampleTitle)</h3>
<p>This example queries a table and uses the result to create
a selection list using a <em>%REPORT</em> block.
<hr>
<form method="post" action="report">
@queryDB()<input type="submit" value="Select Employee">
</form>
<hr>
</body>
</html>
%}
```

Figure 29. Sample macro (Part 2 of 3)

```
%{************************************************************************
*  HTML block:    REPORT                                                *
*  Description: Queries the EMPLOYEE table to obtain additional information *
*               about an individual employee                           *
************************************************************************%}
%HTML(REPORT) {
<html>
<head>
<title>Obtain Employee Information</title>
</head>
<body>
<h3>You selected employee name = $(emp_name)</h3>
<p>Here is the information for that employee:
<PRE>
@fname()
</PRE>
<hr><a href="input">Return to previous page</a>
</body>
</html>
%}

%{     End of Net.Data macro 1 %}
```

Figure 29. Sample macro (Part 3 of 3)

```
================================================================================
%{*************************** Include File ********************************
*   FileName = sqlsamp1.hti                                                    *
*   Description:                                                               *
*     This include file provides global DEFINEs for the sqlsamp1.d2w           *
*     Net.Data macro.                                                          *
*******************************************************************************%}
%define {
    emp_name   =""
    reposition = sign
    exampleTitle = "Sample Macro"
    myTable = "EMPLOYEE"
    DATABASE = "sample"
%}

%{     End of include file  %}
```

Figure 30. Include file

Appendix G. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this publication to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is as your own risk.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

_W92/H3_
555 Bailey Avenue
P.O. Box 49023
San Jose, CA 95161-9023
_U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX
AS/400
DB2
DB2 Universal Database
DRDA
DataJoiner
IBM
IMS
Language Environment
MVS/ESA
Net.Data
OS/2
OS/390
OS/400
OpenEdition

The following terms are trademarks of other companies as follows:

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

UNIX is a registered trademark in the United States and/or other countries licensed exclusively through X/Open Company Limited.

Lotus and Domino Go Webserver are trademarks of Lotus Development Corporation in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Glossary

absolute path. The full path name of an object. Absolute path names begins at the highest level, or ″root″ directory (which is identified by the forward slash (/) or back slash (\) character).

API. Application programming interface. Net.Data supports three Web server APIs for improved performance over CGI processes.

applet. A Java program included in an HTML page. Applets work with Java-enabled browsers, such as Netscape Navigator, and are loaded when the HTML page is processed.

application programming interface (API). A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or licensed program. Net.Data supports the following proprietary Web server APIs for improved performance over CGI processes: ICAPI, GWAPI, ISAPI, and NSAPI.

BLOB. Binary large object.

cache. A part of memory or disk space that contains recently accessed data, designed to speed up subsequent access to the same data. The cache is often used to hold a local copy of frequently used data that is accessible over a network.

caching. The processes of storing frequently-used results from a request to the Web server locally for quick retrieval, until it is time to refresh the information.

Cache Manager. The program that manages a cache for one machine. It can manage multiple caches.

CGI. Common Gateway Interface.

cliette. A long-running process in Net.Data Live Connection that serves requests from the Web server. The Connection Manager schedules cliette processes to serve these requests.

CLOB. Character large object.

commitment control. The establishment of a boundary within the process that Net.Data is running under where operations on resources are part of a unit of work.

Common Gateway Interface (CGI). A standardized way for a Web server to pass control to an application program and receive data back.

Connection Manager. An executable file, `dtwcm`, in Net.Data that is needed to support Live Connection.

cookie. A packet of information sent by an HTTP server to a Web browser and then sent back by the browser each time it accesses that server. Cookies can contain any arbitrary information the server chooses and are used to maintain state between otherwise stateless HTTP transactions. *Free Online Dictionary of Computing*

current working directory. The default directory of a process from which all relative path names are resolved.

database. A collection of tables, or a collection of table spaces and index spaces.

database management system (DBMS). A software system that controls the creation, organization, and modification of a database and access to the data stored within it.

DATALINK. A DB2 data type that enables logical references from the database to a file stored outside the database.

data type. An attribute of columns and literals.

DBCLOB. Double-byte character large object.

DBMS. Database management system.

Domino *Go* **Web server.** The Web server offered by Lotus Corp. and IBM, that offers both regular and secure connections. ICAPI and GWAPI are the interfaces provided with this server.

firewall. A computer with software that guards an internal network from unauthorized external access.

flat file interface. A set of Net.Data built-in functions that let you read and write data from plain-text files.

GWAPI. *Go* Web server API.

HTML. Hypertext markup language.

HTTP. Hypertext transfer protocol.

hypertext markup language. A tag language used to write Web documents.

hypertext transfer protocol. The communication protocol used between a Web server and browser.

ICAPI. Internet Connection API. *See* .

Internet. An international public TCP/IP computer network.

Intranet. A TCP/IP network inside a company firewall.

ISAPI. Microsoft's Internet Server API.

Java. An operating system-independent object-oriented programming language especially useful for Internet applications.

language environment. A module that provides access from a Net.Data macro to an external data source such as DB2 or a programming language such as Perl.

Live Connection. A Net.Data component that consists of a Connection Manager and multiple cliettes. Live Connection manages the reuse of database and Java virtual machine connections.

LOB. Large object.

middleware. Software that mediates between an application program and a network. It manages the interaction between a client application program and a server through the network.

NSAPI. Netscape API.

null. A special value that indicates the absence of information.

path. A search route used to locate files.

path name. Tells the system how to locate an object. The path name is expressed as a sequence of directory names followed by the name of the object. Individual directories and the object name are separated by a forward slash (/) or back slash (\) character.

Perl. An interpreted programming language.

persistence. The state of keeping an assigned value for an entire transaction, where a transaction spans multiple Net.Data invocations. Only variables can be persistent. In addition, operations on resources affected by commitment control are kept active until an explicit commit or rollback is done, or when the transaction completes.

port. A 16-bit number used to communicate between TCP/IP and a higher level protocol or application.

registry. A repository where strings can be stored and retrieved.

relative path name. A path name that does not begin at the highest level, or ″root″ directory. The system assumes that the path name begins at the process's current working directory.

TCP/IP. Transmission Control Protocol / Internet Protocol.

transaction. One Net.Data invocation. If persistent Net.Data is used, then a transaction can span multiple Net.Data invocations.

Transmission Control Protocol / Internet Protocol. A set of communication protocols that support peer-to-peer connectivity functions for both local and wide-area networks.

URL. Uniform resource locator.

uniform resource locator. An address that names a HTTP server and optionally a directory and file name, for example: `http://www.software.ibm.com/data/net.data/index.html`.

unit of work. A recoverable sequence of operations that are treated as one atomic operation. All operations within the unit of work can be completed (commited) or undone (rolled back) as if the operations are a single operation. Only operations on resources that are affected by commitment control can be committed or rolled back.

Web server. A computer running HTTP server software, such as Internet Connection.

Index

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.