Net.Data

Administration and Programming Guide for OS/390

Version 2 Release 2, PTF UQ32535 for APAR PQ24314

Net.Data

Administration and Programming Guide for OS/390

Version 2 Release 2, PTF UQ32535 for APAR PQ24314

Note

Be sure to read the information in “Appendix D. Notices” on page 137 before using this information and the product it supports.

Oct 1999 Edition

Contents

Preface

Thank you for selecting Net.Data® Version 2.2, 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

Net.Data passes this Web page to the Web server, which in turn forwards the page over the network for display at the browser. Other members of the Net.Data family of products provide similar capabilities on machines executing the Windows NT, AIX, OS/2, AS/400, HP-UX, Sun Solaris, Linux, and Santa Cruz Operating System (SCO) operating systems.

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. In adddition, Net.Data can be executed as a servlet.

What's New?

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

What's New in Version 2?

Net.Data for OS/390 Version 2 includes performance and scalability features to meet your application's requirements, including:

- Reuse of DB2 for OS/390 connections established through the SQL language environment when using ICPAPI or GWAPI

- Integration with Work Load Manager for OS/390 in ICAPI environments
- Invocation of Net.Data without a macro (direct request)
- Minimization of extraneous white space within generated Web pages
- Ability to bypass DB2 for OS/390 message text lookups

Net.Data Version 2 also includes a number of functional enhancements:

- Language environment enhancements include the ability to execute stored procedures using the ODBC language environment.
- Macro language enhancements include:
  - Ability to place comments anywhere
  - Nested IF blocks
  - WHILE blocks
  - Ability to receive a single result set from a stored procedure
  - DBCS-enabled string and word functions
  - Support for the SQL decimal datatype in parameter lists for stored procedures
- The ability to construct a Web page by integrating database data that is encoded in one code page with data from a macro that is encoded in another code page.

What's New in Version 2.2?

Net.Data Version 2.2 provides the following enchancements:

- Performance and scalability enchancements include:
  - Ability to cache macros and include files
  - Improved processing of table variables for large result sets
  - Reuse of DB2 for OS/390 connections created by the ODBC language environment
  - Reuse of DB2 for OS/390 connections created by the SQL and ODBC language environments when using Net.Data servlets
  - Improved performance with the Net.Data built-in functions
  - Improved scalability of table variables having large result sets
- Macro Language enhancements include:
  - Ability to receive multiple result sets from stored procedures using the SQL and ODBC language environment
  - New built-in functions for table processing
  - Built-in functions that allow Net.Data applications to get and set HTTP cookies
  - Support for START_ROW_NUM, DTW_SET_TOTAL_ROWS, and TOTAL_ROWS to reduce result set sizes
  - Ability to exit the macro immediately using the DTW_EXIT built-in function
  - Ability to generate and send e-mail messages from the macro using the DTW_SENDMAIL built-in function
  - Support for INCLUDE_URL statements in WHILE blocks
  - Support for functions consisting only of Net.Data macro language statements, using the MACRO_FUNCTION language construct
  - Ability to include period (.) in the HTML section name
  - Ablility to place comments in the Net.Data initialization file
  - Changed scope of variables created within functions to local
  - Support for variable references in literal strings within function call parameter lists

  - Ability to interpret two double quotes as a single double quote within all literal strings
- Support for the Eurocurrency symbol
- Ability to invoke Net.Data using Java servlets
- Ability to disable the SHOWSQL variable with the DTW_SHOWSQL configuration variable (default is disabled)
- Ability to disable Net.Data direct request with the DTW_DIRECT_REQUEST configuration variable (default is disabled)

What's New in PTF UQ32535 for APAR PQ24314?

The PTF UQ32535 for APAR PQ24314 contains new features for Net.Data for OS/390, Version 2 Release 2. Unless otherwise specified, all features are available for Net.Data for OS/390, Version 2 Release 2 when using either DB2 for OS/390 Version 5 or DB2 UDB Server for OS/390 Version 6.

The following features are available with PTF UQ32535 for APAR PQ24314:

- Performance enhancement: dynamic Web page caching
- SQL language environment enhancement: support for DB2 large objects
- Management of cached Web pages and large objects
- Macro language enhancements:
  - Dynamic variable references
  - REPORT block in MACRO_FUNCTION
  - RETURNS clause in MACRO_FUNCTION
  - Token length restriction eliminated
  - Built-in function changes:
    - Ability to set the VALUE attribute for tags using the DTW_TB_SELECT built-in function
    - Ability to return milliseconds from the DTW_TIME built-in function
    - Support for the DTW_HEXTOCHAR built-in function
    - Support for the DTW_CHARTOHEX built-in function
    - Support for the DTW_REPLACE built-in function
- Serviceability enhancements:
  - Net.Data trace log
  - Net.Data error message log
  - Support for DTW_DEFAULT_ERROR_MESSAGE
- New configuration variables:
  - DTW_CACHE_PAGE
  - DTW_LOB_LIFETIME
  - DTW_CACHE_MANAGEMENT_INTERVAL
  - DTW_DEFAULT_ERROR_MESSAGE
  - DTW_TRACE_LOG_DIR
  - DTW_TRACE_LOG_LEVEL
  - DTW_ERROR_LOG_DIR
  - DTW_ERROR_LOG_LEVEL
- New path statement: HTML_PATH

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 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.ibm.com/software/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.

SMP/E installation information is provided in *Program Directory for Net.Data for OS/390 Version 2 Release 2*.

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 illustrates the relationship between Net.Data for OS/390, the Web server, and supported data and programming language environments.

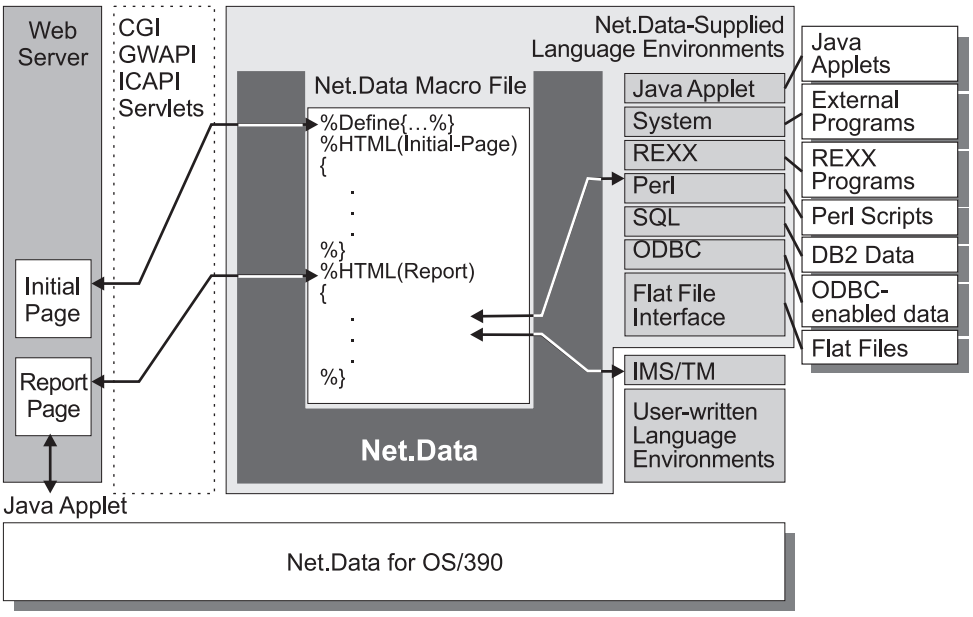


Figure 1. The Relationship between Net.Data for OS/390, the Web Server, and Supported Data and Program Sources

The Web server invokes Net.Data using CGI or a Web server application programming interface (API) 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. In addition, changes to a macro can be seen instantaneously on a browser.

Net.Data complements the extensive data management capabilities that already exist on the OS/390 operating sytem 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 and existing programming skills.
- Provides high-performance access to data that is managed by local DB2 subsystems and by remote DRDA-enabled data sources.
- 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 58.

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. Installing and Configuring Net.Data

Net.Data for OS/390 is installed using SMP/E. The *Program Directory for Net.Data for OS/390 Version 2 Release 2* describes the SMP/E installation process and accompanies the installation tape for Net.Data.

After using SMP/E to install Net.Data, you must configure Net.Data and modify your configuration for the Web server. Configuration tasks include:

- Installing and customizing the Net.Data initialization (INI) file
- Configuring Net.Data for use with CGI, ICAPI, GWAPI, or Net.Data Servlets
- Customizing the Web server configuration and environment variable files
- Setting up the Net.Data language environments
- Specifying access rights
- Enabling the message catalog

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

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 in DB2
- Whether string and word functions are DBCS enabled
- The selection of the default subsystem ID and plan name for access to DB2 and DRDA-enabled data

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 HFS files that Net.Data uses, such as macros, REXX programs, and Perl scripts.

To document the Net.Data initialiation file entries, you can use Net.Data comments. See the comment block section in the language element chapter of *Net.Data Reference*.

Installing the Net.Data Initialization File

The SMP/E install process creates the sample Net.Data initialization file named `db2www.ini` in the directory `/usr/lpp/netdata/pub`. (The SMP/E install process is described in *Program Directory for Net.Data for OS/390 Version 2 Release 2*.)

To install the Net.Data initialization file:

1. Copy the sample Net.Data initialization file to the Web server's document root directory. (The Web server's document root directory is specified in the Web server's configuration file, `/etc/httpd.conf`, by the Pass directive with request template "/*". The Web server's default document root directory is `/usr/lpp/internet/server_root/pub`, but this might have been changed when the Web server was installed. If your Web server's document root directory is different than `internet/server_root/pub`, then substitute your choice as appropriate in these instructions.)

   If you installed Net.Data in the directory `/usr/lpp/netdata`, then you can copy the initialization file by executing the following shell command under OMVS:

   ```
   cp /usr/lpp/netdata/pub/db2www.ini  /usr/lpp/internet/server_root/pub
   ```

2. Ensure that the permissions for the Net.Data initialization file are 644.

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 7
- "Path Configuration Statements" on page 13
- "Environment Configuration Statements" on page 17

The sample initialization file shown in Figure 2 contains examples of these statements.

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.

```
1   %( changes: removed RETURN_CODE parm and DTW_DEFAULT
    ENVIRONMENT statement %)
2   DB2SSID DBNC
3   DB2PLAN DTWGAV22
4   DTW_DIRECT_REQUEST NO
5   DTW_SHOWSQL NO
6   MACRO_PATH   /usr/lpp/netdata/macros;
7   EXEC_PATH    /usr/lpp/netdata/testcmd;
8   FFI_PATH     /usr/lpp/netdata/file-data;
9   ENVIRONMENT (DTW_SQL) dtwsql (IN LOCATION, DB2SSID,
    DB2PLAN, TRANSACTION_SCOPE)
10  ENVIRONMENT (DTW_ODBC) odbcdll (IN LOCATION, TRANSACTION_SCOPE)
11  ENVIRONMENT (DTW_PERL) perldll ()
12  ENVIRONMENT (DTW_REXX) rexxdll ()
13  ENVIRONMENT (DTW_FILE) filedll ()
14  ENVIRONMENT (DTW_APPLET) appldll ()
15  ENVIRONMENT (DTW_SYSTEM) sysdll ()
```

- Line 1 contains a comment
- Lines 2 - 5 define configuration variables
- Lines 6 - 8 define paths to HFS files
- Lines 9 - 15 define the environment statements that are available.

Figure 2. The Net.Data initialization file

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

Migration Note: If you are migrating from a previous version of Net.Data, check each of the configuration statement sections for a complete description of changes that you might need to make when moving to a new release of Net.Data.

- "Configuration Variable Statements"
- "Path Configuration Statements" on page 13

The following ENVIRONMENT statement changes are *required*:

- Remove the RETURN_CODE variable from the parameter list of any ENVIRONMENT statement in which it appears.
- Remove the DTW_DEFAULT ENVIRONMENT statement.

The following changes should be considered because the Net.Data initialization file has new configuration variable default values:

- If your applications require the use of the variable SHOWSQL, then change the DTW_SHOWSQL configuration variable to YES. See "DTW_SHOWSQL: Enable or Disable SHOWSQL Configuration Variable" on page 12 for syntax and examples.
- If your applications require the use of direct request invocation, then change the DTW_DIRECT_REQUEST configuration variable to YES. See "DTW_DIRECT_REQUEST: Enable Direct Request Variable" on page 11 for syntax and examples.

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 DB2 subsystems, 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:

- "DB2MSGS: DB2 Message Text Variable" on page 8
- "DB2PLAN: DB2 Plan Variable" on page 9
- "DB2SSID: DB2 Subsystem ID Variable" on page 9
- "DefaultDBCp: Default Database Code Page Variable" on page 9
- "DSNAOINI: DB2 CLI Initialization File Variable" on page 10
- "DTW_CACHE_MACRO: Caching of Macros" on page 10

Configuration variable assumptions: The sample Net.Data initialization file makes several assumptions about customizing the setting of Net.Data configuration variables. These assumptions may not be correct for your environment:

- The DB2 subsystem ID specification uses DBNC; replace this value using the DB2SSID configuration variable for your application.
- The DB2 plan specification uses DTWGAV22; replace this value using the DB2PLAN configuration variable for your application.
- If your applications require the use of the variable SHOWSQL, then change the DTW_SHOWSQL configuration variable to YES. See "DTW_SHOWSQL: Enable or Disable SHOWSQL Configuration Variable" on page 12 for syntax and examples.
- If your applications require the use of direct request invocation, then change the DTW_DIRECT_REQUEST configuration variable to YES. See "DTW_DIRECT_REQUEST: Enable Direct Request Variable" on page 11 for syntax and examples.

DB2MSGS: DB2 Message Text Variable

Specifies whether Net.Data loads DB2-provided message text for SQLCODES when using the SQL language environment to access DB2 for OS/390.

This variable does not affect MESSAGE blocks.

Syntax:

```
DB2MSGS [=] message_level
```

Where *message_level* indicates the level of DB2-provided messages that Net.Data displays. *message_level* can be set to the following values:

NONE	Specifies that Net.Data displays no message text.
ERRORONLY	Specifies that Net.Data displays message text only for negative SQLCODE values
ALL	Specifies that Net.Data displays message text for all SQLCODE values. This is the default. If a value

is provided for DB2MSGS other than one of the valid values listed above, Net.Data uses the default value of ALL.

Performance tip: When the display of DB2 message text at the browser is not required, specifying NONE can improve performance. When the display of DB2 warning message text at the browser is not required, specifying ERRORONLY can improve performance.

DB2PLAN: DB2 Plan Variable

Specifies the default DB2 plan to be used by the SQL language environment when accessing DB2 for OS/390.

Syntax:

```
DB2PLAN [=] plan_name
```

Example: Sets the default DB2 plan name

```
DB2PLAN DTWGAV22
```

To override the initialization file setting in the macro:

1. Add the DB2PLAN variable as a parameter of the DTW_SQL ENVIRONMENT statement in initialization file as shown in the following example:

   ```
   ENVIRONMENT (DTW_SQL) dtwsql (IN DB2PLAN)
   ```

2. In the macro, set the variable DB2PLAN to the value required for the application.

DB2SSID: DB2 Subsystem ID Variable

Specifies the default DB2 subsystem ID used by the SQL language environment when accessing DB2 for OS/390.

Syntax:

```
DBS2SSID [=] subsystem_id
```

Example: Sets the default DB2 subsystem ID

```
DB2SSID DBNC
```

To override the initialization file setting in the macro:

1. Add the DB2SSID variable as a parameter of the DTW_SQL ENVIRONMENT statement in initialization file as shown in the following example:

   ```
   ENVIRONMENT (DTW_SQL) dtwsql (IN DB2SSID)
   ```

2. In the macro, set the variable DB2SSID to the value required for the application.

DefaultDBCp: Default Database Code Page Variable

Specifies the default code page that Net.Data uses when accessing database data. Net.Data uses the setting of this variable to:

- Convert SQL statement text and the values of input variables for stored procedure calls from the default file system code page to the default database code page
- Convert the values of output variables from stored procedure calls and result tables from the default database code page to the default file system code page

The Web server's configuration file (`/etc/httpd.conf`) specifies the default code page environment through DefaultFsCp and DefaultNetCp directives. The DefaultFsCp directive specifies the default file system code page on the server. This code page is the EBCDIC code page in which the Web server expects to receive

text streams from Net.Data. The DefaultNetCp directive specifies the default network code page. This code page is the ASCII code page used to encode text streams that are served by the Web server.

Performance tip: Do not configure the code page variable DefaultDBCp unless your application requires it. When you define this variable, Net.Data assumes a special conversion is necessary.

If DefaultDBCp is not specified within the initialization file, then Net.Data assumes that the code page for the data in the database is equivalent to the default file system code page and no conversions take place.

Syntax:

```
DefaultDBCp [=] code_page
```

DSNAOINI: DB2 CLI Initialization File Variable

Specifies the name of the DB2 CLI initialization file. The value of this configuration variable can either be a sequential dataset or a member of a partitioned dataset.

If you want to use the Net.Data ODBC language environment, use this variable to specify the name of your DB2 CLI initialization file. If you plan to use the ODBC language environment with ICAPI, set the MVSATTACHTYPE variable in the DB2 CLI initialization file to RRSAF. Also, set the PLANNAME variable to the same plan name as the one specified by DB2PLAN.

Syntax:

```
DSNAOINI [=] CLI_initialization_file_name
```

Example 1: A sequential dataset CLI initialization file name

```
DSNAOINI DBNC.DSNAOINI
```

Example 2:

```
DSNAOINI DBNC.CLI(DSNAOINI)
```

DTW_CACHE_MACRO: Caching of Macros

Specifies macros that are to be cached by Net.Data. This variable works with the DTW_DO_NOT_CACHE_MACRO configuration variable. See “Net.Data Caching of Macros” on page 117 for more information on using these configuration variables.

DTW_CACHE_MANAGEMENT_INTERVAL: Frequency of Web Page Caching

Sets how often automatic web page caching should occur. See “Configuring Net.Data to Automatically Manage Cached Web Pages and Large Objects” on page 29 for more information on using this configuration variable.

Syntax:

```
DTW_CACHE_MANAGEMENT_INTERVAL [=] seconds
```

DTW_CACHE_PAGE: Caching of Web Pages

Specifies pages that are to be cached by Net.Data. See “Dynamic Web Page Caching” on page 119 for more information on using this configuration variable.

Syntax:

```
DTW_CACHE_PAGE file_name_spec|path_template_spec lifetime usage_scope
```

DTW_DEFAULT_ERROR_MESSAGE: Specify Generic Error Messages

Use the DTW_DEFAULT_ERROR_MESSAGE configuration variable to specify a generic error message for applications in production. This variable provides a generic message for error conditions that are not captured in any MESSAGE block.

If you still wish to see the actual error messages generated by Net.Data, use error message logging to capture the messages. See “Net.Data Error Log” on page 126 to learn about using the error log.

If the configuration variable is not specified, Net.Data displays its own provided message for the error condition.

Syntax:

```
DTW_DEFAULT_ERROR_MESSAGE [=] "message"
```

Example: Specifies a generic message

```
DTW_DEFAULT_ERROR_MESSAGE  "This site is temporarily unavailable."
```

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 35 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_DO_NOT_CACHE_MACRO: Caching of Macros

Specifies macros that are *not* to be cached by Net.Data; all other macros are cached. This variable works with the DTW_CACHE_MACRO configuration variable. See “Net.Data Caching of Macros” on page 117 for more information on using these directives.

DTW_ERROR_LOG_DIR: Location of Error Log

Sets the directory where the error log is stored. See “Setting the Error Log File Directory” on page 126 for more information on using this configuration variable.

Syntax:

```
DTW_ERROR_LOG_DIR [=] full_directory_path
```

DTW_ERROR_LOG_LEVEL: Level of Error to Log

Sets the level of error logging. See “Setting the Level of Error Logging” on page 126 for more information on using this configuration variable.

Syntax:

```
DTW_ERROR_LOG_LEVEL [=] OFF|INFORMATION|ALL
```

DTW_LOB_LIFETIME: Length of Time LOBs Are Available

Sets how long LOB files are to be available on the filesystem. See “Configuring Net.Data to Automatically Manage Cached Web Pages and Large Objects” on page 29 for more information on using this configuration variable.

Syntax:

```
DTW_LOB_LIFETIME [=] seconds
```

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.

Syntax:

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

You can override this variable in the macro by using the DEFINE statement.

DTW_REMOVE_WS: Variable for Removing Extra White Space

Reduces the size of a dynamically generated Web page by removing superfluous white space consisting of tabulators, blanks, and new-line characters. When this variable is set to YES, Net.Data compresses a sequence of two or more white spaces to one new-line character, generating shorter HTML result pages. By compressing white space, this variable reduces the amount of data sent to the Web browser, thereby improving performance. The default is NO.

You can override this variable in the macro by using the DEFINE statement.

Tip: Defining this variable to YES affects the amount and type of white space that is displayed. Although the browser ignores extra white space in most cases, it is advised that this variable should be set to NO (or left undefined) for any macros that make use of <PRE></PRE> tags or those that set DTW_PRINT_HEADER to NO, as the resulting page might display differently.

Syntax:

```
DTW_REMOVE_WS [=] YES|NO
```

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 1 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 1. 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 be the name of the SMTP server if it is on the local system, or it can be the node and name of the SMTP server if it is on a remote system. If this variable is not set, Net.Data uses the value `SMTP` as the name of the server and assumes it is on the local system.

Syntax:

```
DTW_SMTP_SERVER server_name
```

Where `server_name` is one of the following values:

name Specifies the name of the SMTP server running on the local system. The default name is SMTP.

node.name
Specifies the node and name on which the SMTP server is running.

Example:

```
DTW_SMTP_SERVER mynode.myserver
```

DTW_TRACE_LOG_DIR: Location of Trace File

Sets the directory where the trace log is stored. See “Setting the Trace Log Directory” on page 125 for more information on using this configuration variable.

Syntax:

```
DTW_TRACE_LOG_DIR [=] full_directory_path
```

DTW_TRACE_LOG_LEVEL: Level of Trace to Log

Sets the level of trace logging. See “Setting the Level of Trace Logging” on page 125 for more information on using this configuration variable.

Syntax:

```
DTW_TRACE_LOG_LEVEL [=] OFF|APPLICATION|SERVICE
```

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 14
- “EXEC_PATH” on page 15

- “INCLUDE_PATH” on page 15
- “FFI_PATH” on page 16
- “HTML_PATH” on page 16

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

The sample Net.Data initialization file makes several assumptions about customizing the setting of Net.Data search paths. These assumptions might not be correct for your environment and require that you modify the path configuration statements:

- If your Net.Data macro directory path is different than `/usr/lpp/netdata/macros`, then replace it with your macros directory path in the MACRO_PATH statement.

  The files contained in the Net.Data `/usr/lpp/netdata/macros` directory are under SMP/E control and cannot be modified. If you modify any of these files, make the modifications to copies of the files stored in directories that you create. You must instruct Net.Data to search for these files in your private directories prior to searching the SMP/E-created directories. To do this, add your private directories in front of the SMP/E-created directories in the path statements of the `db2www.ini` file. For example, if you customize a macro that is provided during the SMP/E installation and place the macro in the directory `/u/user1/macros`, replace the default MACRO_PATH statement with:

  ```
  MACRO_PATH /u/user1/macros;/usr/lpp/netdata/macros;
  ```
- If your Net.Data external program directory path is different than `/usr/lpp/netdata/testcmd`, then replace it with your external program directory path in the EXEC_PATH statement.
- If your Net.Data flat file directory path is different from `/usr/lpp/netdata/file-data`, then replace it with your flat file directory path in the FFI_PATH statement.

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 (;).
- Each path statement can specify multiple paths. 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.

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/netdata-cgi/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 41 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/netdata-cgi/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.

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. 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.

INCLUDE_PATH

The INCLUDE_PATH configuration statement identifies one or more directories that Net.Data searches 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`.

FFI_PATH

The FFI_PATH configuration statement identifies one or more directories that Net.Data searches for an HFS 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.

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 91 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 87 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 LOCATION variable to specify the location name of the remote DBMS at which an SQL statement within a DTW_SQL function is to be executed. The value of LOCATION must be passed to the SQL language environment (DTW_SQL) so that the SQL language environment can connect to the designated remote DBMS. To pass the variable to the language environment, you must add the LOCATION variable to the parameter list of the environment statement for DTW_SQL.

There are also variables that you set as configuration variables in the initialization file, and that you can override in a macro. For example, if you want a macro to override the default settings of the DB2PLAN and DB2SSID variables when the SQL language environment is invoked, include them on the ENVIRONMENT statement for DTW_SQL.

ENVIRONMENT statement changes: If you are migrating from a previous version of Net.Data, make the following changes in the ENVIRONMENT statement section:

- Remove the RETURN_CODE variable from the parameter list of any ENVIRONMENT statement in which it appears.
- Remove the DTW_DEFAULT ENVIRONMENT statement.
- If you plan to use DB2 UDB Server for OS/390 V6 for your applications, change the name `dtwsql` to `dtwsqlv6` in 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`*`, ...)`

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 containing the language environment interfaces that Net.Data calls.

  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.

When Net.Data processes the initialization file, it does not load the language environment DLLs . Net.Data loads a language environment DLL when it first executes a function that identifies that language environment. The DLL 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 LOCATION, DB2SSID, DB2PLAN,
 TRANSACTION_SCOPE)
ENVIRONMENT (DTW_ODBC)     odbcdll   (IN LOCATION, TRANSACTION_SCOPE)
ENVIRONMENT (DTW_APPLET)   appldll   ()
ENVIRONMENT (DTW_PERL)     perldll   ()
```

```
ENVIRONMENT (DTW_FILE)     filedll   ()
ENVIRONMENT (DTW_REXX)     rexxdll   ()
ENVIRONMENT (DTW_SYSTEM)   sysdll    ()
```

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 SQL and ODBC Language Environments" on page 20

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.ibm.com/software/data/ims/about/imsweb/document/`
2. Install IMS TCP/IP OTMA Connection (IMS TOC) on your host system. For information about installing and using IMS TCP/IP OTMA Connection, see:

   `http://www.ibm.com/software/data/ims/about/imstoc/document/index.html`
3. Create the transaction DLL.
   a. Generate the C++ code, makefile (DTW*proj*.mak), and Net.Data macros (DTW*proj*.d2w) files from the HFS source for your transaction with the IMS Web Studio tool.
   b. Build the executable form of the transaction DLL using the generated make file.
4. Copy the transaction DLL file (DTW*proj*.dll) and the Net.Data macro (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 14 for more information.)
   b. Place the transaction DLL or shared library in a directory from which the Web server retrieves DLLs.
5. 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 SQL and ODBC Language Environments

The SQL language environment (DTW_SQL) and the ODBC language environment (DTW_ODBC) use the DB2 load module library SDSNLOAD. The Net.Data SQL and ODBC language environments require that this library reside in LINKLIST or that it be specified in the STEPLIB DD statement of the Web server start-up procedure. The name and location of the Web server start-up procedure depends on your system configuration.

Required:

- Create a plan for Net.Data before using the Net.Data SQL and ODBC language environments to call stored procedures or to execute other types of SQL statements. The binds required for creating this plan depend on the language environments that you plan to use and the version of DB2 you are using.
- The SQL and ODBC Language Environments require RRS Attach Facility when using Net.Data with ICAPI, GWAPI, and Net.Data Servlets. Make sure the RRS Attach Facility is installed for DB2 and OS/390 RRS is installed and configured properly.

Use one of the following approaches to bind the Net.Data DBRM into a package.

- Use the sample JCL for binding the Net.Data. The samples bind DBRM into a package, create a Net.Data plan that supports the use of the SQL language environment, and grant EXECUTE authority on the plan to PUBLIC. The sample JCL can be found in one of the following jobs:

  DTWBIND For the use of the SQL language environment with DB2 for OS/390 V5

  DTWBIND6 For the use of the SQL language environment with DB2 UDB Server for OS/390 V6

  DTWOBIND For the use of the ODBC language environment, or both the ODBC language environment and the SQL language environment, with DB2 for OS/390 V5

  DTWOBND6 For the use of the ODBC language environment, or both the ODBC language environment and the SQL language environment, with DB2 UDB Server for OS/390 V6
- If you plan to use both the SQL and ODBC language environments, bind the DBRMs for DB2 CLI into the same plan as the Net.Data DBRM. Sample JCL for binding the Net.Data DBRM and the DB2 CLI DBRMs into a package, for creating a Net.Data plan that supports the use of the SQL and ODBC language environments, and for granting EXECUTE authority on the plan to PUBLIC can be found in DTW220.SDTWBASE(DTWOBIND).

You might need to make some minor changes to the sample JCL in order to successfully execute the JCL within your environment:

- The prefix of the SDSNEXIT and SDSNLOAD dataset names specified in the STEPLIB DD statement depend on the version of DB2 that you are using and might be incorrect for your installation.
- The values specified by the SYSTEM option of the DSN command and the PLAN option of the RUN command might also be incorrect for your installation. The SYSTEM option of the DSN command specifies the name of the DB2 subsystem and should be identical to the subsystem ID used for your applications.
- The PLAN option of the RUN command specifies the name of the application plan for the DSNTIAD program. If you plan to use stored procedures, you might also need to bind the packages for the stored procedures into the Net.Data plan.

Make whatever modifications are appropriate and submit the JCL.

Managing Connections to DB2

Application programs like Net.Data must connect to DB2 for OS/390 to access DB2-managed data or to execute DB2 stored procedures. When using ICAPI, GWAPI or Net.Data Servlets, Net.Data accomplishes this objective by using the Resource Recovery Services Attachment Facility (RRSAF), which is provided as part of the DB2 product. Because establishing a connection to a DB2 subsystem involves significant overhead, the reuse of existing connections is an attractive alternative to recreating a new connection for each user request.

Net.Data supports the reuse of connections that are used by the SQL and ODBC language environments when Net.Data is configured for use with ICAPI, GWAPI, or Net.Data servlets. When a Web server thread processes a Net.Data user request that requires access to DB2, Net.Data connects to DB2 and creates a DB2 thread. The DB2 thread remains as long as the Web server is running. When the Web server assigns subsequent requests to this Web server thread, and access to DB2 is needed, Net.Data reuses the existing DB2 thread. Net.Data modifies the DB2 plan name and user ID, and switches to a new subsystem ID as needed to match the requirements of the request. The number of DB2 threads created increases until the number of DB2 threads matches the number of Web server threads. At this point, the steady state operation of the server is reached. Net.Data reuses the existing DB2 threads, and no new DB2 threads are created.

No configuration of Net.Data is required for the use of connection management facilities. However, if you want to use Work Load Manager (WLM) to manage the Web server address spaces that process Net.Data requests, some additional WLM configuration is needed.

Workload Management Considerations

Work Load Management (WLM) is a component of the OS/390 operating system that provides facilities to define, implement, and monitor system performance against business goals. WLM allocates resources for processing work by using policies that you define, in order to better ensure that the performance and scalability of your applications meets your requirements.

When you configure Net.Data for use with ICAPI or GWAPI, either IBM Internet Connection Server or Lotus Domino Go Webserver let you use WLM to establish policies to manage your Net.Data workload. You can establish these policies by specifying application environments and WLM transaction classes for processing URL requests that match a given template.

An added advantage of using WLM is that you can make changes to the Net.Data initialization file (`db2www.ini`) and have them take effect without having to stop and restart the Web server, by using the WLM REFRESH or WLM QUIESCE commands.

For more information about WLM, refer to *OS/390 MVS Planning: Workload Management*, GC28-1761.

For further information about configuring IBM Internet Connection Server and Lotus Domino Go Webserver for use with WLM, refer to:

- *IBM Internet Connection Secure Server Webmaster's Guide Version 2 Release 2 for OS/390*, GC31-8490

- *Lotus Domino Go Webserver Webmaster's Guide Version 4.6.1 for OS/390*, SC31-8643

Configuring Net.Data 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.

Unless you modified the directory structure or name when you created the hierarchical file system (HFS) directory for Net.Data, the SMP/E install process installed the Net.Data executable files and DLLs in the directory `/usr/lpp/netdata/cgi-bin`. Because `/usr/lpp/netdata` is not your Web server's root directory, the Web server cannot process client requests for Net.Data unless you make some additional modifications to the Web server's configuration.

To modify the Web server:

1. Stop the Web server.
2. Use either of the following approaches to complete the installation of the executable files and DLLs.
   - **Using Net.Data Directories**
     a. Add an Exec directive to the Web server's configuration file, `/etc/httpd.conf`, that redirects Net.Data requests to the `/usr/lpp/netdata/cgi-bin` directory. For example:

        ```
        Exec   /netdata-cgi/*       /usr/lpp/netdata/cgi-bin/*
        ```

     b. Add your Net.Data `cgi-bin` directory to the LIBPATH statement of the Web server's environment variables file, `/etc/httpd.envvars`. If your Net.Data `cgi-bin` directory is `/usr/lpp/netdata/cgi-bin`, then your LIBPATH statement should be similar to the following statement:

        ```
        LIBPATH=/usr/lpp/internet/bin:/usr/lpp/netdata/cgi-bin
        ```

   - **Using Web Server Directories**
     a. Move the executable files and DLLs (`appldll`, `db2www`, `dtwle`, `dtwlei`, `dtwsql`, `dtwsqlv6`, `filedll`, `odbcdll`, `perldll`, `rexxdll`, `sysdll`) to the Web server's `cgi-bin` directory. The Web server default `cgi-bin` directory is `/usr/lpp/internet/server_root/cgi-bin`.

        The Web server's default root directory is specified by the ServerRoot directive in the Web server's configuration file, `/etc/httpd.conf`, and might have been changed when the Web server was installed. The Web server's default `cgi-bin` directory is specified by an Exec directive in the Web server's configuration file and might also have been changed when the Web server was installed. If your Web server's root directory is different than `/usr/lpp/internet/server_root`, or if your Web server's `cgi-bin` directory is different than `/usr/lpp/internet/server_root/cgi-bin`, substitute your choices as appropriate in these instructions.

     b. Add the Web server's `cgi-bin` directory to the LIBPATH statement of the Web server's environment variables file, `/etc/httpd.envvars`. If your Web server `cgi-bin` directory is `/usr/lpp/internet/server_root/cgi-bin`, your LIBPATH statement should be similar to the following statement:

        ```
        LIBPATH=/usr/lpp/internet/bin:/usr/lpp/internet/server_root/cgi-bin
        ```

3. Ensure that the permissions for the Net.Data executable files and DLLs and for each directory in the path to the executable files and DLLs are 755.
4. Restart the Web server

Restriction: Do not specify more than one of the following directories in the LIBPATH statement of the Web server's environment variables file.

- cgi-bin
- icapi-lib
- servlet-lib

For more detail on installing the Web server and on Web server configuration file directives, refer to the following publications:

- *IBM Internet Connection Secure Server Planning for Installation Version 2 Release 2 for OS/390*, GC31-8489-00
- *IBM Internet Connection Secure Server Webmaster's Guide Version 2 Release 2 for OS/390*, GC31-8490-00
- *Lotus Domino Go Webserver Planning for Installation Version 4.6.1 for OS/390*, SC31-8642
- *Lotus Domino Go Webserver Webmaster's Guide Version 4.6.1 for OS/390*, SC31-8643

Configuring Net.Data for Use with ICAPI or GWAPI

Using a Web server application programming interface (API) rather than CGI can improve the performance of Net.Data considerably. With these APIs, Net.Data reuses connections to DB2. Net.Data creates DB2 threads and keeps them active for the life of the process.

Any macro that executes successfully using CGI will execute successfully using ICAPI or GWAPI. No modifications need to be made to these macros.

Unless you modified the directory structure or name when you created the HFS directory for Net.Data, the SMP/E install process installed the Net.Data executable files and DLLs in the directory `/usr/lpp/netdata/icapi-lib`. Because `/usr/lpp/netdata` is not your Web server's root directory, the Web server cannot handle client requests for Net.Data unless you make some additional modifications to the Web server's configuration.

To modify the Web server:

1. Stop the Web server.
2. Use either of the following approaches to complete the installation of the executable files and DLLs.
   - **Using Net.Data Directories**
     a. Add a ServerInit directive to the Web server's configuration file, `/etc/httpd.conf`, that instructs the Web server to perform Net.Data-specific initialization when the Web server executes its initialization routines. One possible ServerInit directive is:

        ```
        ServerInit   /usr/lpp/netdata/icapi-lib/db2www:dtw_init
        ```

     b. Add a Service directive to the Web server's configuration file, `/etc/httpd.conf`, that redirects Net.Data requests to the `/usr/lpp/netdata/icapi-lib` directory. One possible Service directive is:

        ```
        Service  /netdata-cgi/db2www*  /usr/lpp/netdata/icapi-lib/db2www:dtw_icapi*
        ```

     c. Add your Net.Data `icapi-lib` directory to the LIBPATH statement of the Web server's environment variables file, `/etc/httpd.envvars`. If your

Net.Data `icapi-lib` directory is `/usr/lpp/netdata/icapi-lib`, then your LIBPATH statement should be similar to:

```
LIBPATH=/usr/lpp/internet/bin:/usr/lpp/netdata/icapi-lib
```

- **Using Web Server Directories**
  a. Move the executable files and DLLs (`appldll`, `db2www`, `dtwle`, `dtwlei`, `dtwsql`, `dtwsqlv6`, `filedll`, `odbcdll`, `perldll`, `rexxdll`, `sysdll`) to the Web server's `cgi-bin` directory. The Web server default `cgi-bin` directory is `/usr/lpp/internet/server_root/cgi-bin`.
  b. Add a ServerInit directive to the Web server's configuration file, `/etc/httpd.conf`, that instructs the Web server to perform Net.Data-specific initialization when the Web server executes its initialization routines. One possible ServerInit directive is:

```
ServerInit /usr/lpp/internet/server_root/cgi-bin/db2www:dtw_init
```

  c. Add a Service directive to the Web server's configuration file, `/etc/httpd.conf`, that redirects Net.Data requests to the `/usr/lpp/internet/server_root/cgi-bin` directory. One possible Service directive is:

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

  d. Add the Web server's `cgi-bin` directory to the LIBPATH statement of the Web server's environment variables file, `/etc/httpd.envvars`. If your Web server `cgi-bin` directory is `/usr/lpp/internet/server_root/cgi-bin`, your LIBPATH statement should be similar to:

```
LIBPATH=/usr/lpp/internet/bin:/usr/lpp/internet/server_root/cgi-bin
```

  Restriction: Do not specify more than one of the following directories in the LIBPATH statement of the Web server's environment variables file:

  - `cgi-bin`
  - `icapi-lib`
  - `servlet-lib`

3. Ensure that the permissions are 755 for the Net.Data executable files and DLLs and for each directory in the path to the executable files and DLLs.
4. Restart the Web server.

For more detail on installing the Web server and on Web server configuration file directives, refer to the following publications:

- *IBM Internet Connection Secure Server Planning for Installation Version 2 Release 2 for OS/390*, GC31-8489
- *IBM Internet Connection Secure Server Webmaster's Guide Version 2 Release 2 for OS/390*, GC31-8490
- *Lotus Domino Go Webserver Planning for Installation Version 4.6.1 for OS/390*, SC31-8642
- *Lotus Domino Go Webserver Webmaster's Guide Version 4.6.1 for OS/390*, SC31-8643

Configuring Net.Data for Use with Java Servlets

Servlets are Java classes that perform a role similar to that of CGI programs or Web server API plug-ins. Servlets run on a Java servlet-enabled Web server and extend the server's capabilities, much like the way Java applets run on a browser and extend the browser's capabilities. Use the following steps to configure your environment to invoke Net.Data through this Java servlet interface.

Unless you modified the directory structure or name when you created the HFS directory for Net.Data, the SMP/E install process installed the Net.Data DLLs and NetDataServlets.jar file in the directory `/usr/lpp/netdata/servlet-lib`. Because `/usr/lpp/netdata` is not your Web server's root directory, the Web server cannot handle client requests for Net.Data unless you make some additional modifications to the Web server's configuration.

To modify the Web server:

1. Enable the Web server to run servlets. (See your Web server documentation for instructions on registering and using servlets.)
2. Use either of the following approaches to complete the installation of the executable files and DLLs.
   - **Using Net.Data Directories**
     a. Add your Net.Data `servlet-lib` directory to the LIBPATH statement of the Web server's environment variables file, `/etc/httpd.envvars`. If your Net.Data `servlet-lib` directory is `/usr/lpp/netdata/servlet-lib`, then your LIBPATH statement should be similar to:

        ```
        LIBPATH=/usr/lpp/internet/bin:/usr/lpp/netdata/servlet-lib
        ```

     b. Add the `NetDataServlets.jar` file to the CLASSPATH statement of the Web server's environment variables file, `/etc/httpd.envvars`. If the Net.Data `servlet-lib` directory is `/usr/lpp/netdata/servlet-lib`, then your CLASSPATH statement should be similar to the following statement:

        ```
        CLASSPATH=/usr/lpp/JDK1.1/lib/classes.zip:/usr/lpp/netdata/servlet-lib/
           NetDataServlets.jar
        ```

   - **Using Web Server Directories**
     a. Move the Net.Data DLLs (`libdtwndapi.so`, `appldl`, `dtwle`, `dtwlei`, `dtwsql`, `dtwsqlv6`, `filedll`, `odbcdll`, `perldll`, `rexxdll`, `sysdll`) and the `NetDataServlets.jar` file to the Web server's `cgi-bin` directory. The Web server default `cgi-bin` directory is `/usr/lpp/internet/server_root/cgi-bin`.

     b. Add the `NetDataServlets.jar` file to the CLASSPATH statement of the Web server's environment variables file, `/etc/httpd.envvars`. Your CLASSPATH statement should be similar to the following statement:

        ```
        CLASSPATH=/usr/lpp/JDK1.1/lib/classes.zip:/usr/lpp/internet/server_root\
        /cgi-bin/NetDataServlets.jar
        ```

     c. Add the Web server's `cgi-bin` directory to the LIBPATH statement of the Web server's environment variables file, `/etc/httpd.envvars`. If your Web server `cgi-bin` directory is `/usr/lpp/internet/server_root/cgi-bin`, your LIBPATH statement should be similar to:

        ```
        LIBPATH=/usr/lpp/internet/bin:/usr/lpp/internet/server_root/cgi-bin
        ```

3. Ensure that the permissions are 755 for the Net.Data executable files and DLLs and for each directory in the path to the executable files and DLLs.
4. After configuring Net.Data, configure the IBM WebSphere Application Server to work properly with Net.Data servlets. Use one of the following methods:
   - Update the `jvm.properties` file by adding the full pathname of the file `NetDataServlets.jar` to the property named `ncf.jvm.classpath`.
   - You can use the IBM WebSphere Application Server Manager:
     a. From your web browser, launch the IBM WebSphere Application Server Manager login screen.
     b. Login to the manager and click **Manage**.
     c. From the **Setup** menu, select **Basic**.

d. Add the full pathname of the file `NetDataServlet.jar` to the Java classpath shown on the page.
e. Select **No** to the question, ″Use system classpath?″

5. Restart the Web server.

Restriction: Do not specify more than one of the following directories in the LIBPATH statement of the Web server's environment variables file:

- `cgi-bin`
- `icapi-lib`
- `servlet-lib`

For more detail on installing the Web server and on Web server configuration file directives, refer to the following publications:

- *Lotus Domino Go Webserver Planning for Installation Version 5.0 for OS/390*, SC31-8690-00
- *Lotus Domino Go Webserver Webmaster's Guide Version 5.0 for OS/390*, SC31-8691-00

Enabling the Message Catalog

Net.Data for OS/390 provides English, Japanese, and Korean message catalogs. You enable and specify these message catalogs in the Web server's environment variables file.

Unless you modified the directory structure or name when you created the hierarchical file system (HFS) directory for Net.Data, you have already installed the Net.Data English, Japanese, and Korean message catalogs in the files `/usr/lpp/netdata/C/d2w.cat`, `/usr/lpp/netdata/Ja_JP/d2w.cat`, and `/usr/lpp/netdata/Ko_KR/d2w.cat`, respectively.

If you did modify the directory structure or name, substitute your choice for `/usr/lpp/netdata` in the following steps:

1. To enable the use of Net.Data message catalogs, add `/usr/lpp/netdata/%L/%N` to the NLSPATH statement in the Web server's environment variables file. Your NLSPATH statement should be similar to:

   ```
   NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lpp/internet/%L/%N:/usr/lpp/netdata/%L/%N
   ```

2. To select the specific catalog that Net.Data uses, specify the value of the LANG statement in the Web server's environment variables file, `/etc/httpd.envvars`. The syntax of the statement is

   ```
   LANG = locale
   ```

   Use Table 2 to specify the correct value for *locale*.

Table 2. LANG statement values

	English	Japanese	Korean
LANG =	C	Ja_JP	Ko_KR

Granting Access Rights to Files and Data Sets 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 and datasets that are referenced in a Net.Data macro and to the macro that a URL references. This means that these files must be in MVS datasets or HFS files and directories 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 DB2 CLI initialization file specified by the DSNAOINI configuration variable
- 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 macros 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, write, and execute files in the `/tmp` HFS directory

Managing Cached Web Pages and Large Objects

If you set up Net.Data to cache Web pages (see “Dynamic Web Page Caching” on page 119) or if Net.Data accesses many large objects (LOBs) (see “Using Large Objects” on page 91), some management of the temporary files is necessary. Net.Data provides features for automatically managing cached Web pages and LOBs according to the settings of configuration variables. Or, you can use a Net.Data-provided macro to manage your Web pages and LOBs in more sophisticated ways based on criteria such as macro names, HTML block names, and creation times. Management of cached Web pages and large objects is available when using ICAPI, GWAPI, or Net.Data Servlets.

- “Setting-up DB2: Creating the Web Page Dependency Table”
- “Configuring Net.Data to Automatically Manage Cached Web Pages and Large Objects” on page 29
- “Using a Net.Data-provided Macro for More Advanced Management” on page 29

Setting-up DB2: Creating the Web Page Dependency Table

You need to create the dynamic Web page dependency table and install several stored procedures before you can begin to use Net.Data to manage your cached Web pages and large objects.

1. The dynamic Web page dependency table contains information about the LOB files stored in HFS and about the relationship that these files may have, if any, to Web pages stored in the dynamic Web page cache.
   a. Create the LOB dependency table, SYSIBM.DTWCACHEDEPS, using the SQL found in DTW220.SDTWSPUF(DTWCRCCH). This file also includes SQL statements to create a database, called DTWCACHE, and a tablespace, called DTWTBSP1, for the Web dependency table.

The database and tablespace have the same names as those created when you enable Net.Data for Web page caching (see “Step 2: Set Up the Web Page Cache” on page 121).

b. Define the stored procedure used to insert the dependency information into SYSIBM.DTWCACHEDEPS. The stored procedure is found in DTW220.SDTWLOAD(DTWDEPIN).

1) Copy the stored procedure into your stored procedure library.

2) Define the stored procedure into DB2 UDB Server for OS/390 V6 using the SQL found in DTW220.SDTWSPUF(DTWCCHV6).

c. Bind the stored procedure’s DBRM DTW220.SDTWDBRM (DTWV22DP) into the package DTWCACHEPKG using a user ID with INSERT, SELECT, and DELETE privileges on SYSIBM.DTWCACHEDEPS. The user IDs associated with the requests that retrieve LOBs must have EXECUTE privilege on the package DTWCACHEPKG.

The JCL for this step is located in DTW220.SDTWBASE(DTWDEPBD).

2. Create the stored procedure that performs automatic management of the Web page cache and LOBs.

a. Define the stored procedure. The stored procedure can be found in DTW220.SDTWLOAD(DTWCLEAN).

1) Copy the stored procedure into your stored procedure library.

2) Define the stored procedure.

- The SQL to define the stored procedure into DB2 for OS/390 V5 is found in DTW220.SDTWSPUF(DTWCLNV5).
- The SQL to define the stored procedure into DB2 UDB Server for OS/390 V6 is found in DTW220.SDTWSPUF(DTWCLNV6).

b. Bind the stored procedure’s DBRM DTW220.SDTWDBRM (DTWV22CL) into the package DTWCACHEPKG using a user ID with DELETE privilege on SYSIBM.DTWCACHEDPAGES and SYSIBM.DTWCACHEDEPS. The user IDs that execute the macro must have EXECUTE privilege on the package DTWCACHEPKG.

The JCL for this step is located in DTW220.SDTWBASE(DTWCLNBD).

3. Create the stored procedure that is used for more advanced management of the Web page cache and LOBs.

a. Define the stored procedure. The stored procedure is found in DTW220.SDTWLOAD(DTWMANCL).

1) Copy the stored procedure into your stored procedure library.

2) Define the stored procedure.

- The SQL to define the stored procedure into DB2 for OS/390 V5 is found in DTW220.SDTWSPUF(DTWCLNV5).
- The SQL to define the stored procedure into DB2 UDB Server for OS/390 V6 is found in DTW220.SDTWSPUF(DTWCLNV6).

b. Bind the stored procedure’s DBRM DTW220.SDTWDBRM (DTWV22MN) into the package DTWCACHEPKG using a user ID with DELETE privilege on SYSIBM.DTWCACHEDPAGES and on SYSIBM.DTWCACHEDEPS. The user IDs that execute the cleanup_cache.dtw must have EXECUTE privilege on the package DTWCACHEPKG.

The JCL for this step is located in DTW220.SDTWBASE(DTWMANBD).

Configuring Net.Data to Automatically Manage Cached Web Pages and Large Objects

You can configure Net.Data to automatically manage cached Web pages and LOBs based on expiration time values. Net.Data uses the lifetime values for dynamic Web pages specified on the DTW_CACHE_PAGE directives and the setting of the DTW_LOB_LIFETIME configuration variable to determine the expiration time values. The setting of DTW_CACHE_MANAGEMENT_INTERVAL specifies the frequency with which automatic cache management occurs. When Net.Data performs automatic cache management, it removes all objects from the cache having expiration times older than the current time.

To specify the caching interval:

Use the DTW_CACHE_MANAGEMENT_INTERVAL configuration variable to specify the minimum number of seconds between successive invocations of the Net.Data stored procedure that performs automatic cache management.

Syntax:

```
DTW_CACHE_MANAGEMENT_INTERVAL [=] seconds
```

To specify the LOB lifetime:

Use the DTW_LOB_LIFETIME configuration variable to specify the minimum number of seconds that LOB files are available in HFS. For more information on DB2 LOBs, see “Using Large Objects” on page 91.

Syntax:

```
DTW_LOB_LIFETIME [=] seconds
```

Where *seconds* is the minimum number of seconds that LOB files are available. The effective lifetime of a LOB file is the larger of the DTW_LOB_LIFETIME value and the lifetime of any Web page that references it.

Using a Net.Data-provided Macro for More Advanced Management

Net.Data automatically manages cached Web pages and the LOBs they reference using expiration times. Net.Data also offers more advanced styles of management through a Net.Data-provided macro, manage_cache.dtw. With this macro, you can:

- Delete expired dynamic Web pages and LOBs that have been cached.
- Delete cached Web pages and related LOB files based on HTML block and creation time criteria, as well as delete all expired objects that have been cached.
- Delete LOB files from HFS and related Web pages based on creation time criteria, as well as delete all expired objects that have been cached.

To begin the macro, invoke the BEGIN HTML block of the manage_cache.dtw located in the directory `/usr/lpp/netdata/macros`. To learn more about how to invoke a macro, see “Chapter 4. Invoking Net.Data” on page 41.

To delete expired dynamic Web pages and large objects that have been cached:

1. Invoke the BEGIN HTML block of the manage_cache.dtw macro.
2. Click the **Delete dynamic Web pages and large object files that have expired** choice.

3. Click the **EXECUTE** push button to proceed, or select **Back to the beginning** to return to the main page and cancel your request.

To delete cached Web pages and related LOB files:

1. Invoke the BEGIN HTML block of the manage_cache.dtw macro.
2. Click on the **Delete selected dynamic Web pages and related large object files** choice.

   This choice lets you specify a filter and timestamp values for the cached Web pages you want to delete. All expired cached Web pages and LOBs as well as all LOBs referenced by the Web pages are deleted.
3. Optionally type a string in the **Enter the ACTUAL_KEY filter** field that matches any part of the *ACTUAL_KEY* for the Web pages to be deleted. This string acts as filter for selecting the cached Web pages Net.Data deletes. The string can contain up to 250 characters.

   For example, when the following string is entered:

   ```
   /netdata/macros/my_macro.d2w/report
   ```

   Net.Data deletes all cached Web pages that have an *ACTUAL_KEY* value containing this string. For a detailed description of *ACTUAL_KEY*, refer to Table 3 on page 32.
4. Optionally click on the **Starting CREATION_TIME** check box and enter a timestamp value. Net.Data deletes all cached Web pages that have creation times greater than or equal to this timestamp value, and that have creation times less than or equal to the **Ending CREATION_TIME**, if specified. If no **Ending CREATION_TIME** value is specified, than Net.Data deletes all cached Web pages that have creation times greater than or equal to the **Starting CREATION_TIME** value.

   For example, when the following timestamp is entered:

   ```
   Starting CREATION_TIME:

     Year 1999  Month 03  Day 23 Hour 14  Minute 00  Second 00
   ```

   Net.Data deletes all cached Web pages that were created on March 23, 1999 at 2:00 PM or later, up to and including the value of **Ending CREATION_TIME**, if specified.
5. Optionally click on the **Ending CREATION_TIME** check box and enter a timestamp value. Net.Data deletes all cached Web pages that have creation times less than or equal to this timestamp value, and that have creation times greater than or equal to the **Starting CREATION_TIME**, if specified. If no **Starting CREATION_TIME** value is specified, than Net.Data deletes all cached Web pages that have creation times less than or equal to the **Ending CREATION_TIME** value.

   For example, when the following timestamp is entered:

   ```
   Ending CREATION_TIME:

     Year 1999  Month 03  Day 23 Hour 23  Minute 59  Second 59
   ```

   Net.Data deletes all cached Web pages that were created on March 23, 1999 at 11:59:59 PM or earlier, starting with the value of **Starting CREATION_TIME**, if specified.

If the **Enter the ACTUAL_KEY filter** field is empty and neither of the check boxes are checked, Net.Data deletes only expired cached Web pages and LOBs.

6. Click the **EXECUTE** push button to proceed, or select **Back to the beginning** to return to the main page and cancel your request.

To delete large objects in HFS and related Web pages

1. Invoke the BEGIN HTML block of the manage_cache.dtw macro.
2. Click on the **Delete selected large object files and related dynamic Web pages** choice.

   This choice lets you specify timestamp values for the LOBs you want to delete. All expired Web pages and LOBs as well as all Web pages referenced by LOBs are deleted.
3. Optionally click on the **Starting CREATION_TIME** check box and enter a timestamp value. Net.Data deletes all LOB files that have creation times greater than or equal to this timestamp value, and that have creation times less than or equal to the **Ending CREATION_TIME**, if specified. If no **Ending CREATION_TIME** value is specified, than Net.Data deletes all LOB files that have creation times greater than or equal to the **Starting CREATION_TIME** value.

   For example, when the following timestamp is entered:

   ```
   Starting CREATION_TIME:

     Year 1999  Month 03  Day 23 Hour 14  Minute 00  Second 00
   ```

   Net.Data deletes all LOBs that were created on March 23, 1999 at 2:00 PM or later, up to and including the value of **Ending CREATION_TIME**, if specified.
4. Optionally click on the **Ending CREATION_TIME** check box and enter a timestamp value. Net.Data deletes all LOB files that have creation times less than or equal to this timestamp value, and that have creation times greater than or equal to the **Starting CREATION_TIME**, if specified. If no **Starting CREATION_TIME** value is specified, then Net.Data deletes all LOB files that have creation times less than or equal to the **Ending CREATION_TIME** value.

   For example, when the following timestamp is entered:

   ```
   Ending CREATION_TIME:

     Year 1999  Month 03  Day 23 Hour 23  Minute 59  Second 59
   ```

   Net.Data deletes all LOB files that were created on March 23, 1999 at 11:59:59 PM or earlier, starting with the value of **Starting CREATION_TIME**, if specified.

If the neither of the check boxes are checked, Net.Data deletes only expired LOB files and cached Web pages.

5. Click the **EXECUTE** push button to proceed, or select **Back to the beginning** to return to the main page and cancel your request.

Web page cache table and Web page dependency table descriptions

If you want to manage the contents of the dynamic Web page cache and LOB files using techniques different from those provided by Net.Data, you can access the SYSIBM.DTWCACHEDPAGES and SYSIBM.DTWCACHEDEPS DB2 tables. The columns for these two tables are described below along with their data types and descriptions.

Table 3. SYSIBM.DTWCACHEDPAGES Table. Contains the Web pages that are cached and information about them.

INDEXED_KEY: CHAR(250)	Indexed key for cached page: the first 250 characters of the actual key
ID: INTEGER	Identifier: an identifier derived from the key
ACTUAL_KEY: VARCHAR(4000)	The actual key of the cached page: the path information, macro, HTML block name, query string, and form data of the request that generated the page
CREATOR: CHAR(8)	User ID of creator: user ID associated with the request that created the cached page
CREATION_TIME: TIMESTAMP	Creation timestamp: date and time of creation of cached page. It is the same as the CREATION_TIME for any LOBs that the Web page references
EXPIRATION_TIME: TIMESTAMP	Expiration timestamp: date and time of the expiration of the cached page (value of CREATION_TIME + lifetime value from the DTW_CACHE_PAGE directive)
SIZE: INTEGER	Size: size of cached page in bytes
USAGE_SCOPE: SMALLINT	Usage scope: a value of 1 means that the page has a PUBLIC usage scope and a value of 2 means that the page has a PRIVATE usage scope
ORDINAL_POSITION: INTEGER	Ordinal position of segment: the ordinal position of the Web page segment within the complete cached page
PAGE_SEGMENT VARCHAR(28100) FOR BIT DATA	Dynamic Web page segment: the ASCII encoded Web page segment

Table 4. SYSIBM.DTWCACHEDEPS Table. Contains information about the LOBs that are referenced by Web pages.

INDEXED_KEY: CHAR(250)	Indexed key for Web page: first 250 characters of the actual key
ID: INTEGER	Identifier: an identifier derived from the actual key
ACTUAL_KEY: VARCHAR(4000)	The actual key of the cached page: the path information, macro, HTML block name, query string, and form data of the request that generated the page
FILENAME: VARCHAR(1024)	Fully qualified HFS filename for the LOB
CREATION_TIME: TIMESTAMP	Creation timestamp: date and time of the creation of the LOB. Is the same as the CREATION_TIME for the Web page that references this LOB
EXPIRATION_TIME: TIMESTAMP	Expiration timestamp: date and time for expiration of the LOB (value of CREATION_TIME + DTW_LOB_LIFETIME configuration value when dynamic web page caching not in use; value of CREATION_TIME + max(DTW_LOB_LIFETIME configuration value, lifetime value from DTW_CACHE_PAGE directive) when dynamic web page caching in use).
SIZE: INTEGER	Size: size of the LOB in bytes

Chapter 3. Keeping Your Assets Secure

Internet security in an OS/390 environment 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

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 OS/390 Firewall Technologies or equivalent firewall products that execute in the OS/390 environment.

OS/390 Firewall Technologies is a tool kit that you can use to implement various security architectures and strategies. It includes the following tools:

- IP filters
- Proxy servers
- Socks servers
- Domain name service (DNS)
- Virtual private networks

For more detail on how to install and configure your firewall in a secure manner, refer to *IBM Firewall Toolkit for OS/390 Guide and Reference*, SC24-5835.

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.

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.

In an OS/390 environment, a user ID can become associated with the thread or process that is handling a Net.Data request in one of three ways:

Client-based authentication
: The user is prompted for a local OS/390 user ID and password at the client. The Web server then invokes the local security subsystem (such as RACF) to authenticate the user. If successfully authenticated, the supplied user ID is associated with the request. Use of the special Web server %%CLIENT%% access control user ID enables this type of authentication.

Server-based authentication
: The user ID of the Web server is associated with each request and the user is not prompted for a user ID or password. This choice is not recommended because of the level of authority usually associated with the Web server's user ID. Use of the special Web server %%SERVER%% access control user ID enables this type of authentication.

Surrogate authentication
: A surrogate user ID that has the authority to access some predefined collection of resources is associated with the client request. This type of authentication requires the creation of surrogate user IDs with access authority that is appropriate for a group of users or class of requests.

The approach that the Web server uses for associating a user ID with a client request is specified when the Web server is configured. For additional detail on access control user IDs, on installing the Web server, and on using the Protect, Protection, DefProt, and UserId directives to configure the Web server, refer to:

- *IBM Internet Connection Secure Server Planning for Installation Version 2 Release 2 for OS/390*, GC31-8489
- *IBM Internet Connection Secure Server Webmaster's Guide Version 2 Release 2 for OS/390*, GC31-8490
- *Lotus Domino Go Webserver Planning for Installation Version 4.6.1 for OS/390*, SC31-8642
- *Lotus Domino Go Webserver Webmaster's Guide Version 4.6.1 for OS/390*, SC31-8643

Using Authorization

Authorization provides a user with complete or restricted access to an object, resource, or function. Data sources such as DB2 and HFS 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 34. 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 macros, executable files, include files, or other HFS 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. Installing and 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 36.

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 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 12 for syntax and additional information.
- Use the DTW_ASSIGN() function as described in “Macro Development Techniques” on page 36.

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 11 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 se

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 12 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 64 for more information about hidden variables.

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)
- Lotus Domino Go Web server (GWAPI)
- Internet Connection Server (ICAPI)
- Java Servlets

See “Chapter 2. Installing and Configuring Net.Data” on page 5 to learn more about configuring Net.Data for these interfaces. 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.

- “Invoking Net.Data using CGI, ICAPI, or GWAPI”
- “Invoking Net.Data with Java Servlets” on page 50

Invoking Net.Data using CGI, ICAPI, or GWAPI

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 3 on page 42 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.

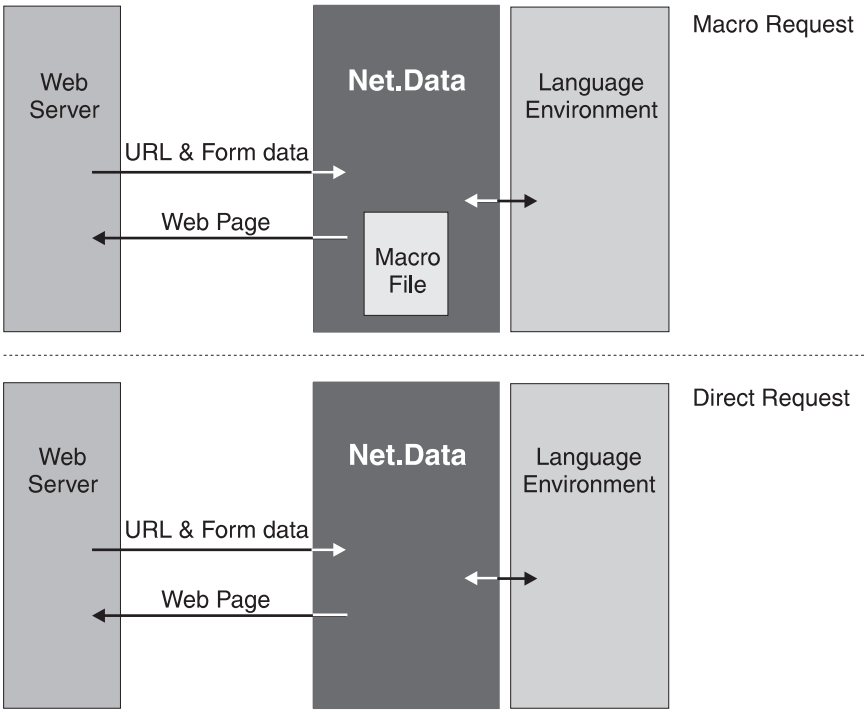


Figure 3. Macro Request Versus Direct Request

The syntax for invoking Net.Data when it is configured for use with ICAPI or GWAPI is the same as the syntax for invoking Net.Data when it is configured for use with CGI. 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, ICAPI, or GWAPI.

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)”
- “Invoking Net.Data without a Macro (Direct Request)” on page 45

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. The examples assume that Net.Data was configured using Net.Data directories, as previously described in “Configuring Net.Data for Use with CGI” on page 22 and “Configuring Net.Data for Use with ICAPI or GWAPI” on page 23.

- 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 load modules. For example, `/netdata-cgi/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 14 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/netdata-cgi/db2www/myMacro.d2w/report">
.
.
.
</A>
```

Example 2: Invoking Net.Data using a form

```
<FORM METHOD=POST
 ACTION="http://server/netdata-cgi/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" on page 44
- "HTML Forms" on page 44

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/netdata-cgi/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:

```
%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 78 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” 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="/netdata-cgi/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:

```
%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 11 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, 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. It assumes that Net.Data was configured using Net.Data directories, as previously described in “Configuring Net.Data for Use with CGI” on page 22 and “Configuring Net.Data for Use with ICAPI or GWAPI” on page 23, and illustrates the context in which you specify a direct request for the Perl language environment.

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

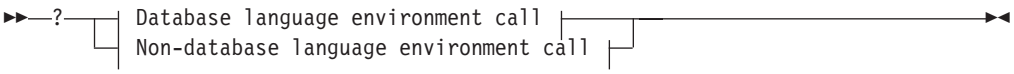
Where *direct_request* represents the direct request syntax. For example, the following HTML link contains the direct request:

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

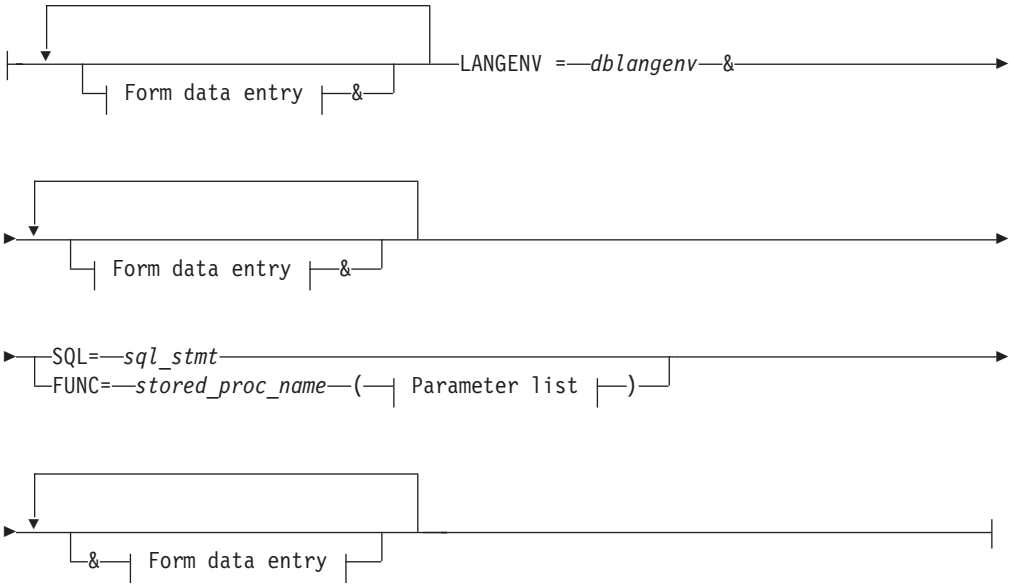
Direct Request Syntax

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

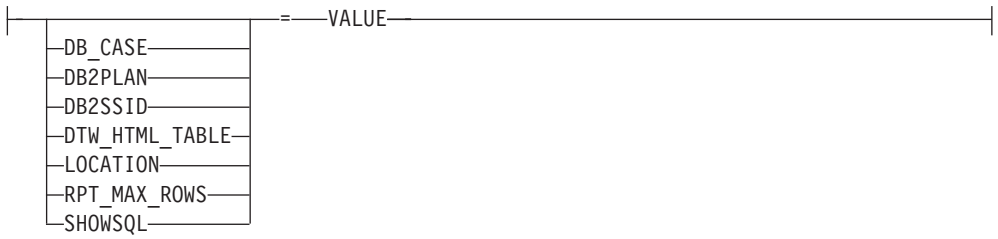
Syntax



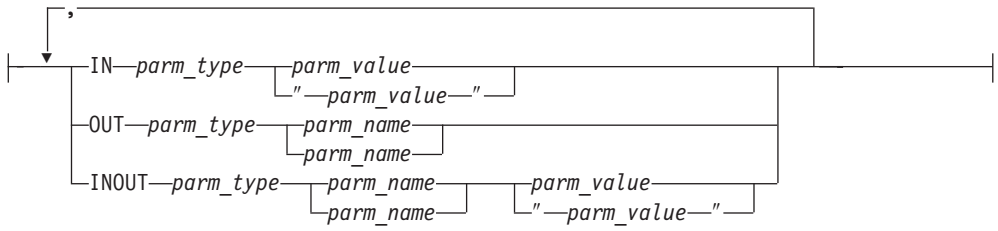
Database language environment call:



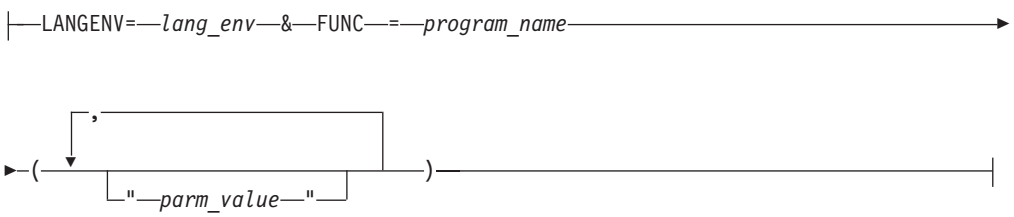
Form data entry:



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.

DB_CASE

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

DB2PLAN

Specifies the DB2 plan to be used when accessing the local DB2 subsystem.

DB2SSID

Specifies the DB2 subsystem ID to be used when accessing the local DB2 subsystem.

DTW_HTML_TABLE

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

LOCATION

Specifies the name of the remote server to which the local DB2 subsystem should pass the SQL request.

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.

VALUE
Specifies the value of the Net.Data variable.

LANGENV
Specifies the target language environment for the SQL statement or stored procedure call.

dblangenv
The name of the database language environment:

- DTW_SQL
- DTW_ODBC

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. The examples assume that Net.Data was configured using Net.Data directories as previously described in “Configuring Net.Data for Use with CGI” on page 22 and “Configuring Net.Data for Use with ICAPI or GWAPI” on page 23

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/netdata-cgi/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/netdata-cgi/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

```
<FORM METHOD="POST"
 ACTION="http://server/netdata-cgi/db2www/">
<INPUT TYPE=hidden NAME="LANGENV" VALUE="DTW_SQL">
<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/netdata-cgi/db2www/?LANGENV=DTW_SQL&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/netdata-cgi/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/netdata-cgi/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/netdata-cgi/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/netdata-cgi/db2www/?LANGENV=DTW_SQL&FUNC=MY_STORED_PROC
   (IN+CHAR(30)+Salaries)&DTW_HTML_TABLE=YES
```

Invoking Net.Data with Java Servlets

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 perform CGI-like functions. 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).

Net.Data provides servlets that you can use to invoke Net.Data macros, single SQL statements, stored procedures, and functions on OS/390. The servlets can be executed from both a URL and as a Server-Side-Include (SSI). Net.Data provides two servlets:

MacroServlet (com.ibm.netdata.servlets.MacroServlet)
Executes a Net.Data macro.

You can run macros through Server-Side-Includes (SSI) to embed multiple macros in your HTML file.

Function Servlet (com.ibm.netdata.servlets.FunctionServlet)
Invokes Net.Data without a macro by specifying:

- 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.

The function servlet provides direct request capability, but using a Java interface. See “Invoking Net.Data without a Macro (Direct Request)” on page 45 for more information.

Invoking Net.Data using MacroServlet

You can call this servlet from either a URL or an SSI in an HTML file.

Syntax and Examples

- URL:

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

  For example:

  ```
  http://server/servlet/com.ibm.netdata.servlets.MacroServlet?MACRO=companies.d2w&
  BLOCK=gatherinfo
  ```

- SSI:

  ```
  <servlet code="com.ibm.netdata.servlets.MacroServlet">
    <param name="MACRO" value="my_macro">
    <param name="BLOCK" value="my_block">
    <param name="parmn" value="valuen">
  </servlet>
  ```

  For example:

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

Parameters

MACRO
Required. Specifies the path to an existing Net.Data macro.

BLOCK
Specifies the name of the HTML block in the specified Net.Data macro to execute. The default block is `report`.

parmn
Specifies any additional parameters that your macro requires.

Invoking Net.Data using FunctionServlet

You can call this servlet from either a URL or an SSI in an HTML file, and with it you can invoke either a function, SQL statement, or stored procedure.

Syntax and Examples

- URL:
  - Invoking a function:

    ```
    http://server/servlet/com.ibm.netdata.servlets.FunctionServlet?LANGENV=language&
    FUNC=function_name&parmn=valuenn
    ```

    For example:

    ```
    http://server/servlet/com.ibm.netdata.servlets.FunctionServlet?LANGENV=DTW_REXX&FUNC=custinp
    ```

  - Invoking an SQL statement:

    ```
    http://server/servlet/com.ibm.netdata.servlets.FunctionServlet?LANGENV=database_lang&
    SQL=SQL_statement&parmn=valuenn
    ```

    For example:

    ```
    http://server/servlet/com.ibm.netdata.servlets.FunctionServlet?LANGENV=DTW_SQL&SQL=select+la
    ```

  - Invoking a stored procedure:

    ```
    http://server/servlet/com.ibm.netdata.servlets.FunctionServlet
    ?LANGENV=DTW_SQL&FUNC=stored_procedure_name(parameter_list)
    ```

    For example:

```
http://server/servlet/com.ibm.netdata.servlets.FunctionServlet
?LANGENV=DTW_SQL&FUNC=myStoredProc(IN+CHAR(20)+"inval")
```

- SSI:
  - Invoking a function:

```
<servlet code="com.ibm.netdata.servlets.MacroServlet">
  <param name="LANGENV" value="language">
  <param name="FUNC" value="function_name">
  <param name="parmn" value="valuen">
</servlet>
```

  - Invoking an SQL statement:

```
<servlet code="com.ibm.netdata.servlets.MacroServlet">
  <param name="LANGENV" value="language">
  <param name="SQL" value="SQL_statement">
  <param name="parmn" value="valuen">
</servlet>
```

  - Invoking a stored procedure:

```
<servlet code="com.ibm.netdata.servlets.MacroServlet">
  <param name="LANGENV" value="language">
  <param name="FUNC" value="stored_procedure">
  <param name="parmn" value="valuen">
</servlet>
```

For example:

```
<servlet code="com.ibm.netdata.servlets.FunctionServlet">
  <param name="LANGENV" value="DTW_SQL">
  <param name="FUNC"    value="myStoredProc(IN CHAR(20) invalue)">
</servlet>
```

Parameters

LANGENV
Specifies the Net.Data language environment that is called to process the function (for example, DTW_SQL or DTW_REXX).

FUNC
Specifies the name of the program that contains the function to be executed, or in the case of a stored procedure, the stored procedure name and parameter. For example, my_rexx, where my_rexx is the name of an executable REXX file. Use the **parmn** keyword to specify input parameters to the function.

SQL
Specifies an SQL statement or stored procedure name that accesses a database, for example, `"select * from employee"`.

parmn
Specifies any additional parameters that the function requires.

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 4.

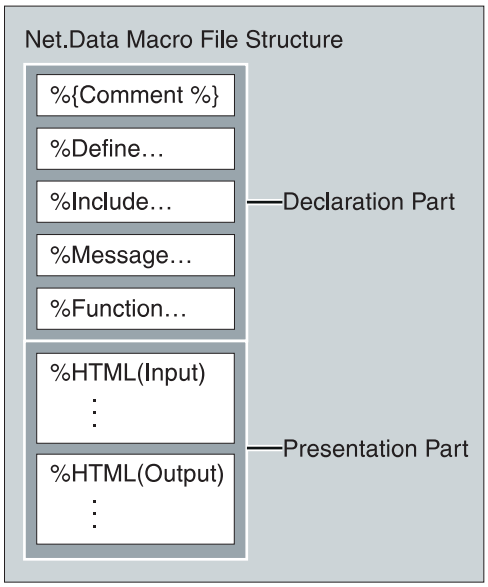


Figure 4. Macro Structure

- 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.

This chapter examines the different blocks that make up a Net.Data macro and methods you can use for writing the macro.

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*.
- 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 56.

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 c

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="/netdata-cgi/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 the language environments and back. The variable names, values, and literal strings that are passed are called tokens. Net.Data puts no limit on the size of the tokens and will pass any token that the memory of your system can handle. Individual language environments, however, might provide restrictions on the token size.

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”
- “Defining Variables” on page 59
- “Referencing Variables” on page 61
- “Variable Types” on page 61

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 44 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/netdata-cgi/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.

To use variables as part of your text presentation statements, reference them in the HTML blocks of your macro.

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 dynamically generate a variable name by including variable references, strings, and function calls within a variable reference. If you reference a dynamically-generated variable that does not follow the variable name rules, Net.Data resolves the reference to an empty string. See *Net.Data Reference* for more information on variable references.

Example: Dynamically generates a variable reference for a field value of a row

```
%WHILE (INDEX < NUM_COLS) {
 $(V$(INDEX))
 @DTW_ADD(INDEX, "1", INDEX)
 %}
```

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.

Variable Types

You can use the following types of variable in your macros.

- “Conditional Variables” on page 62

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 65 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.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 15 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 tstamp=%EXEC "date"

%FUNCTION(DTW_SQL) myQuery() {
SELECT CUSTNO, CUSTNAME from dist1.customer
%REPORT{
%ROW{
<A HREF="/netdata-cgi/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="/netdata-cgi/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{
%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 87 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” for accessing individual elements in a table within a REPORT block, and see “Table Functions” on page 76 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.

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 DB2 subsystems,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)
%}
```

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”
- “Calling Functions” on page 73
- “Calling Net.Data Built-in Functions” on page 73

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]
  ...

  [message-block]
%}
```

MACRO_ FUNCTION block:

```
%MACRO_FUNCTION function-name([usage] parameter, ...) [RETURNS(return-var)] {
  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 73 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 87 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

- 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.

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 63.

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 78.(In the FUNCTION block, multiple report blocks can only be used in the SQL and ODBC language environments).

message-block

Defines the MESSAGE block, which handles any messages returned by the FUNCTION block. See “Message Blocks” on page 71.

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.

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 5 on page 74 shows how the Net.Data built-in functions and the macro interact.

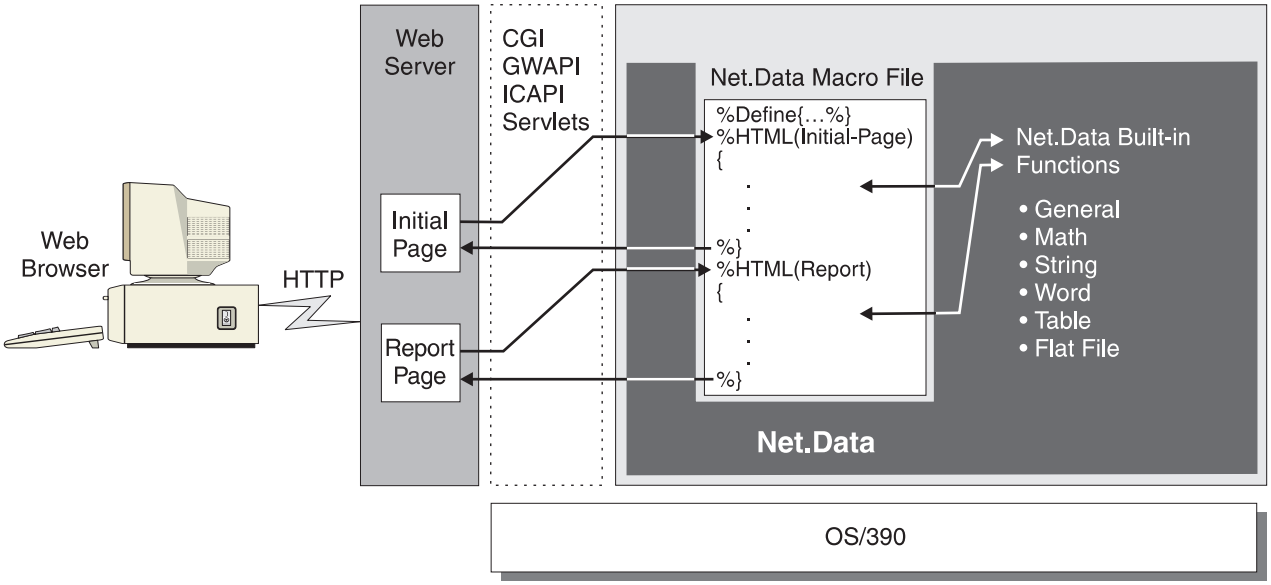


Figure 5. Net.Data Built-in Functions

Built-in functions can return their results in three ways, depending on its prefix:

- **DTW_ and DTWF_:** The results of the call are returned in an output parameter or no result is returned. (**DTWF_** is the prefix for flat file functions.)
- **DTW_r and DTWF_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 require 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.

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="/netdata-cgi/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/netdata-cgi/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 66. 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 use multiple REPORT blocks with the DTW_SQL language environment or the DTW_ODBC language environment when a function calls a stored procedure that returns multiple result sets. See “Stored Procedures” on page 93.

To use multiple REPORT blocks, place a result set name on the stored procedure CALL statement for each result set. If more result sets are returned from the stored procedure than the number of REPORT blocks you have specified, then default reports are generated for each result set that is not associated with a REPORT block. This assumes that you have not disabled default report processing by specifying the Net.Data built-in function, DTW_DEFAULT_REPORT = ″NO″.

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

```
%FUNCTION (dtw_sql) myStoredProc () {
     CALL myproc (table1, table2)  %}
```

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 = "YES"
%FUNCTION(dtw_sql) myStoredProc (OUT table1) {
   CALL myproc (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.

Guidelines and Restrictions for Multiple REPORT Blocks: Use the following guidelines and restrictions when specifying multiple REPORT blocks in a 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 = ″YES″. 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.

Restrictions:

- Multiple REPORT blocks can only be used in functions using the DTW_SQL or DTW_ODBC language environments when the function calls a stored procedure that returns multiple result sets.
- 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.
- Multiple REPORT blocks cannot be specified for the same table.

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”
- “Looping Constructs: WHILE Blocks” on page 84

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 == "OS390")
  %IF (version == "1.3")
     %INCLUDE "os390v1r3_def.hti"
  %ELIF (version == "2.0")
     %INCLUDE "os390v2r1_def.hti"
  %ELIF (version == "2.2")
     %INCLUDE "os390v2r2_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, 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 subsystem, and the REXX language environment lets you invoke REXX programs. You can also use the SYSTEM language environment to execute an external program that, for example, uses the External CICS Interface (EXCI) interface to execute a CICS program.

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). 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 6 shows the relationship between the Web server, Net.Data, and the Net.Data language environments.

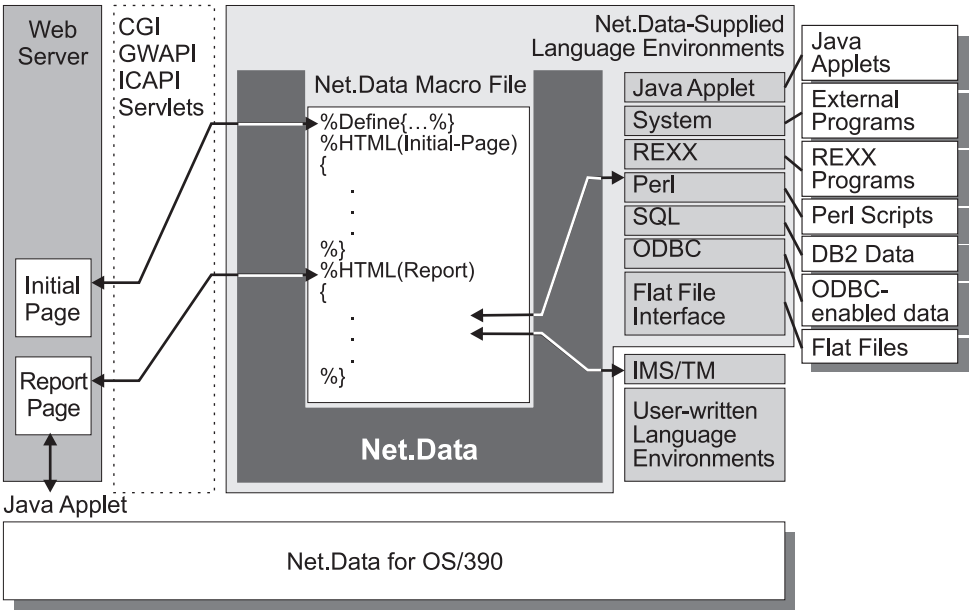


Figure 6. The Net.Data Language Environments

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 88
- "Calling a Language Environment" on page 88
- "Data Language Environments" on page 89
- "Programming Language Environments" on page 102

For configuration information about the Net.Data-provided language environments, see "Setting Up the Language Environments" on page 19.

For information about improving performance when using the language environments, see "Chapter 7. Improving Performance" on page 117.

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 89
- "Programming Language Environments" on page 102

Table 5 provides a brief description of each language environment.

Table 5. 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.
ODBC	DTW_ODBC	The ODBC language environment executes SQL statements through an ODBC interface for access to multiple database management systems.
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.
System	DTW_SYSTEM	The System language environment supports executing commands and calling external programs.

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"
- "Flat File Interface Language Environment" on page 100
- "IMS Web Language Environment" on page 101

Relational Database Language Environments

Net.Data provides relational database language environments to help you access your relational data sources. The SQL statements you provide to access the relational data are executed as dynamic SQL. For more information on dynamic SQL, see your DB2 documentation.

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:

Verify that the location of your CLI initialization file is specified in the configuration variable DSNAOINI. To learn how to set the DSNAOINI configuration variable, see "DSNAOINI: DB2 CLI Initialization File Variable" on page 10.

Verify that the following configuration statement is in the initialization file, on one line.

```
ENVIRONMENT (DTW_ODBC) odbcdll ()
```

Allowed variables on the ENVIRONMENT statement:
TRANSACTION_SCOPE, LOCATION

Restrictions:

- SQL statements in the inline statement block can be 32 KB.

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 ()
```

Allowed variables on the ENVIRONMENT statement:
TRANSACTION_SCOPE, LOCATION, DB2SSID, DB2PLAN

Important: See "Setting up the SQL and ODBC Language Environments" on page 20 to learn how to set up the SQL language environment.

Restriction: SQL statements in the inline statement block can be up to 32 KB.

The following sections describe how to use these language environments:

- "Managing Transactions in a Net.Data Application"
- "Using Large Objects" on page 91
- "Stored Procedures" on page 93
- "Relational Database Language Environment Example" on page 99

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.

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")
```

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, because issuing a commit after each transaction affects the performance of your database application and should only be done when necessary.

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.

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 UDB Server for OS/390 Version 6 tables and incorporate them into your dynamic Web pages by using the Net.Data SQL language environment.

When the language environment executes an SQL SELECT statement or a stored procedure that returns a LOB, it does not assign the object to a V(*n*) table processing variable or a Net.Data table field. Instead, it stores the LOB in an HFS file that Net.Data creates and returns only the name of the file in the V(*n*) table processing variable or a Net.Data table field. In your Net.Data macro you can use the name to reference the LOB file; for example, you can create an HTML anchor element with a hypertext reference or an image element containing a URL for the file. Net.Data places the file containing the LOB in the directory specified by the HTML_PATH path statement, located in the Net.Data initialization file (db2www.ini). Write access to the LOB file is limited to the user ID associated with the Net.Data request that retrieved the LOB.

The file name for the LOB is dynamically constructed, and has the following form:

name[.*extension*]

Where:

name Is a dynamically generated 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 determines the extension by looking for a signature in the first few bytes of the LOB file. Table 6 on page 92 shows the LOB extensions used by the SQL language environment:

Table 6. LOB extensions used in the SQL language environment

Extension	Object Type
.bmp	bitmap image
.gif	graphical image format
.jpg	joint photographic experts group (JPEG) image
.tif	tagged image file format
.ps	postscript
.mid	musical instruments digital interface (midi) audio
.aif	AIFF audio
.avi	audio visual interleave audio
.au	basic audio
.ra	real audio
.wav	windows audio visual
.pdf	portable document format
.rmi	midi sequence

If the object type for the BLOB is not recognized, no extension is added to the file name.

When Net.Data returns the name of the file containing a LOB, it prefixes the file name with the string `/tmplobs/` using the following syntax:

`/tmplobs/`*`name`*`.[`*`extension`*`]`

This prefix permits you to locate your LOB directory in a directory other than the Web server's document root directory.

To ensure that references to LOB files are correctly resolved, add the following Pass directive to your Web server's configuration file:

```
Pass    /tmplobs/*    <HTML_PATH>
```

<HTML_PATH> is the value specified for the HTML_PATH path statement in the Net.Data initialization file.

Planning tips:

- Consider using the facilities provided by Net.Data to manage LOBs in HFS.

  Net.Data stores each LOB that it receives from DB2 in an HFS file in the directory specified by the HTML_PATH path statement. Because a LOB can be up to 2 gigabytes in size, these files can quickly consume a considerable amount of disk storage. Net.Data provides automatic management of LOBs based on expiration time, and a macro that allows a system administrator to manage LOBs in a more sophisticated fashion using the creation time of the LOBs. See "Managing Cached Web Pages and Large Objects" on page 27 for additional information.
- Consider using the HFS file system structure to more effectively manage your LOB files.
  - Put the directory specified by the HTML_PATH path statement in its own HFS data set and manage the size of the data set using MVS allocation size and extents.
  - Place your HFS data set on a device that has capacity and performance characteristics that match the needs of your applications.

Restriction: Net.Data does not support UPDATE and INSERT SQL statements for large objects.

Example: The following application uses an MPEG audio (.mpa) file. Because the SQL language environment does not recognize this file type, an EXEC variable is used to append the .mpa extension to the file name. A user of this application must click on the file name to invoke the MPEG audio file viewer.

```
%DEFINE{
docroot="/usr/lpp/internet/server_root/html"
myFile=%EXEC "mv $(docroot)$(filename) $(docroot)$(filename).mpa"
%}

%FUNCTION(DTW_SQL) queryData() {
  SELECT Name, IDPhoto, Voice FROM RepProfile
  %REPORT{
    <P>Here is the information you selected:</P>
    %ROW{
      @DTW_ASSIGN(filename, @DTW_rSUBSTR(V3, @DTW_rLASTPOS("/", V3)))
      $(myFile)
      $(V1) <IMG SRC="$(V2)">
            <A HREF="$(V3).mpa">Voice sample</A><P>
    %}
  %}
%}

%HTML(REPORT){
@queryData()
%}
```

If the RepProfile table contains information about Kinson Yamamoto and Merilee Lau, then the execution of the REPORT block will add the following HTML to the Web page being generated:

```
<P>Here is the information 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.

- The value of V1 is a person's name, which is character data.
- The value of V2 is the name of a GIF file containing the photo of the person. The image is displayed inline within the generated Web page.
- The value of V3 is the name of an MPA file containing a sample of the person's voice. Because Net.Data does not recognize the MPA file format, it does not add an extension to the file name when it creates the file for the LOB in the directory specified by HTML_PATH. This example illustrates the use of an EXEC variable to add the .mpa extension to the file name. The voice sample is played when the user clicks on text ″Voice sample″, which is a hyperlink text.

Access rights for LOBs:

Ensure that the user ID or user IDs under which Net.Data executes have write access to the directory specified by HTML_PATH.

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"
- "Calling a Stored Procedure"
- "Passing Parameters" on page 95
- "Processing Result Sets" on page 96

Stored Procedure Syntax: The syntax of the stored procedure uses the FUNCTION statement, the CALL statement, and optionally a REPORT block.

```
%FUNCTION (DTW_lang_env) function_name ([IN datatype arg1, INOUT datatype arg2,
    OUT tablename, ...]) {
  CALL stored_procedure [(resultsetname, ...)]
[%REPORT [(resultsetname)] { %}]
...
[%REPORT [(resultsetname)] { %}]
[%MESSAGE %}]

%}
```

Where:

lang_env
: Is the name of the language environment being invoked. It can be DTW_SQL or DTW_ODBC.

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 7. 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 7. Stored Procedures Data Types

CHAR	FLOAT	SMALLINT
DECIMAL	INTEGER	VARCHAR
DOUBLE	GRAPHIC	VARGRAPHIC
DOUBLEPRECISION		

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).

   ```
   %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 {  ...   %}
           ...
      %}
      %REPORT (table1) {
           ...
           %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 81 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 REPORT block. 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.

Relational Database Language Environment Example

The following example shows how you can call the relational database language environments from your macros:

SQL and ODBC

The following example shows a macro with a DTW_SQL function definition that calls an SQL stored procedure. For the ODBC language environment, substitute `DTW_ODBC` for `DTW_SQL` where it appears. 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>
%}
```

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)   filedll    ()
```

See "Environment Configuration Statements" on page 17 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)
%}
```

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 or shared library 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, 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.ibm.com/software/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)      hwsdll    ()
```

See “Environment Configuration Statements” on page 17 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 19 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”
- “Perl Language Environment” on page 108
- “REXX Language Environment” on page 111
- “System Language Environment” on page 113

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 and Data Sets Accessed by Net.Data” on page 27 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)   appldll    ()
```

See “Environment Configuration Statements” on page 17 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");
            }
        }
    }
```

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)     perldll   ()
```

See “Environment Configuration Statements” on page 17 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)     rexxdll   ()
```

See “Environment Configuration Statements” on page 17 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 15 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*.

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)   sysdll    ()
```

See “Environment Configuration Statements” on page 17 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 15 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"
- "Net.Data Caching of Macros"
- "Suppressing DB2 for OS/390 Messages" on page 122
- "Optimizing the Language Environments" on page 122

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.

Net.Data Caching of Macros

Use macro caching to improve throughput and reduce CPU utilization. When macro caching is enabled, preprocessed macros are cached in memory when the macros are first invoked. These preprocessed versions are then available for reuse, thereby eliminating the costs associated with reading and the macros from HFS and processing them each time they are requested.

Caching Considerations

Please note the following items regarding caching of macros:

- Caching is available when using ICAPI, GWAPI, or Net.Data Servlets.
- The cached version of a macro is available to a requestor that has read permission for the file containing the macro.
- The amount of memory that the preprocessed version of the macro uses is approximately twice the size of the macro itself.
- You can control the amount of memory that will be used for the caching of macros by using the caching configuration variables.

Enabling Macro Caching

You enable macro caching by adding caching configuration variables to the Net.Data initialization file (db2www.ini). If you add the DTW_CACHE_MACRO variable, the DTW_DO_NOT_CACHE_MACRO variable, or both variables to the Net.Data initialization file, then caching is enabled. If you do not add either variable, then no macros will be cached.

If the DTW_CACHE_MACRO and DTW_DO_NOT_CACHE configuration variables both specify the same macro, then the macro is not cached by Net.Data.

Defining Which Macros to Cache

The DTW_MACRO_CACHE configuration variable specifies macros that are to be cached.

Set this configuration variable in the Net.Data initialization file.

Syntax:

```
DTW_CACHE_MACRO [=] filename_or_pathtemplate;...
```

Where *filename_or_pathtemplate* is either:

- A fully qualified macro name.
- A path template, which is a directory path followed by /*. If a path template is used, all macros in the directory and its subdirectories will be cached.

Example 1: If you want all of the macros in `/u/user1/macros` and its subdirectories to be cached, set the configuration variable as follows:

```
DTW_CACHE_MACRO /u/user1/macros/*
```

Example 2: If you want to cache all macros in the DIR1 and DIR2 directories and the individual macro `sql.dtw`, the DTW_CACHE_MACRO path might look like this:

```
DTW_CACHE_MACRO /u/user1/macros/DIR1/*;/u/user2/macros/sql.dtw;/u/user2/macros/DIR2/*
```

Defining Which Macros to Not Cache

The DTW_DO_NOT_CACHE_MACRO configuration variable specifies which macros are not to be cached.

Set this configuration variable in the Net.Data initialization file. If the Net.Data initialization file contains this variable, and does not contain the DTW_CACHE_MACRO variable, then all macros will be cached except for those listed in the DTW_DO_NOT_CACHE_MACRO variable.

Syntax:

```
DTW_DO_NOT_CACHE_MACRO [=] filename_or_pathtemplate;...
```

Where *filename_or_pathtemplate* is either:

- A fully qualified macro name.
- A path template, which is a directory path followed by /*. If a path template is used, then none of the macros in the directory or its subdirectories will be cached.

Example 1: If you want all of your macros to be cached except the `adminset.d2w` macro, you would set the configuration variable as follows:

```
DTW_DO_NOT_CACHE_MACRO  /u/user1/macros/adminset.d2w
```

Example 2: If both caching configuration variables are set in the initialization file, the DTW_DO_NOT_CACHE_MACRO takes precedence. For example, suppose the variable settings appear as follows:

```
DTW_CACHE_MACRO        /u/user1/user_macros/*;/u/user1/admin_macros/*
DTW_DO_NOT_CACHE_MACRO /u/user1/admin_macros/adminset.d2w
```

The macros in the directories user_macros and admin_macros will be cached except for the macro adminset.d2w. Even though this macro is in the admin_macros directory, it will not be cached because the setting for DTW_DO_NOT_CACHE_MACRO overrides the setting for DTW_CACHE_MACRO.

Dynamic Web Page Caching

Net.Data can cache dynamic Web pages, thereby improving performance by eliminating the cost of reconstructing repeatedly requested Web pages. Web page caching is available when using ICAPI, GWAPI, or Net.Data Servlets.

Net.Data caches ACSII encoded Web pages in DB2. Using caching directives, you can specify which Web pages are cached, how long they remain valid in the cache, and the degree to which the reuse of the cached pages is restricted. Net.Data provides automatic management of the cache based on expiration time, and a macro that allows a system administrator to manage the cache based on macro and HTML block names and on the creation time for the cached page.

A cached page is static. Its content depends on the state of the data stores and business logic at the time the Web page was created. Subsequent changes to the data stores and business logic do not affect the content of the cached page.

Caching Considerations

A number of factors determine which Web pages should be cached. The decision on whether to cache a Web page varies from application to application.

General recommendations:

- Cache pages that are requested repeatedly.
- Cache pages that do not change frequently.
- Do not cache pages for macros that make changes to data sources. If a cached Web page is used to respond to a Net.Data request, the macro is not run and no changes are made to the data sources.

Enabling Dynamic Web page Caching

Use the following steps to configure Net.Data for caching:

Step 1: Specify the Web Pages to be Cached: DTW_CACHE_PAGE

Use one or more DTW_CACHE_PAGE directives to identify the Web pages that are to be cached. Specify the directives in the Net.Data configuration file.

Syntax:

```
DTW_CACHE_PAGE file_name_spec|path_template_spec lifetime usage_scope
```

Where:

file_name_spec
: Is the specification of one or all of the HTML blocks within a macro. To specify one HTML block, use the fully qualified name of the macro and the

block name. To specify all HTML blocks, use the fully qualified macro name and the suffix /*. Net.Data caches all Web pages that it creates by executing this HTML block. Specify this value or *path_template_spec*, but not both.

path_template_spec
Is the specification of HTML blocks within macros, using a *path template* for one or more directories containing macros. A path template must contain the suffix /*. Net.Data caches all Web pages that it creates by executing an HTML block of any macro contained within any of these directories. Specify this value or *file_name_spec*, but not both.

lifetime
Specifies the number of seconds that a cached Web page is valid

usage_scope
Specifies the degree to which the reuse of the Web page is restricted. Reuse is granted or denied based on the user ID associated with the Net.Data request. Usage_scope can have one of the following values:

PUBLIC
The cached page can be reused for any request associated with a user ID that is authorized to execute the macro.

PRIVATE
The cached page can be reused for any request associated with the user ID that was associated with the request that originally cached the Web page.

You can specify this directive multiple times. Specify one DTW_CACHE_PAGE directive for each *file_name_spec* or *path_template_spec* value. If DTW_CACHE_PAGE directives conflict with each other, the first directive specified takes precedence.

A cached page is reused for a request if the URL, the form data, and the query string of the request match the URL, form data, and query string of the request that originally cached the page.

Examples:

Example 1: Specifies the caching of any Web pages generated when Net.Data executes the specified HTML block

```
DTW_CACHE_PAGE /u/USER1/macros/main.d2w/output  3600 PUBLIC
```

In this example, Net.Data caches the Web pages generated when it executes the `output` HTML block in the macro `main.d2w`, located in the `/u/USER1/macros` directory. The Web pages have PUBLIC scope, and remain valid for 1 hour.

Example 2: Specifies the caching of any Web pages generated when Net.Data executes any HTML block in the specified macro

```
DTW_CACHE_PAGE  /u/USER1/macros/main.d2w/*  3600 PUBLIC
```

In this example, Net.Data caches any Web pages Net.Data generates when it executes any HTML block in the macro `main.d2w`, located in the `/u/USER1/macros` directory. The Web pages have PUBLIC scope, and remain valid for 1 hour.

Example 3: Specifies the caching of any Web pages generated when Net.Data executes HTML blocks in macros located in one or more directories

```
DTW_CACHE_PAGE  /u/USER1/macros/*  3600 PUBLIC
```

In this example, Net.Data caches any Web pages Net.Data generates when it executes any HTML block in any macro located in the /u/USER1/macros directory or its subdirectories. The Web pages have PUBLIC scope, and remain valid for 1 hour.

Example 4: Specifies the caching of any Web page generated by all Net.Data macros

```
DTW_CACHE_PAGE  /*  3600  PUBLIC
```

In this example, Net.Data caches all Web pages Net.Data generates. The Web pages have PUBLIC scope, and remain valid for 1 hour.

Example 5: Specifies multiple Web page caching directives

```
DTW_CACHE_PAGE  /u/USER1/macros/main/*  1800  PUBLIC
DTW_CACHE_PAGE  /u/USER1/macros/special/daily_news.d2w/*  43200  PUBLIC
DTW_CACHE_PAGE  /u/USER1/macros/special/employee_stats.d2w/*  3600  PRIVATE
```

In this example, Net.Data caches all Web pages generated from any HTML block in any macros located in the /u/USER1/macros/main/ directory. The Web pages have PUBLIC scope and remain valid for 30 minutes. All Web pages generated by the daily_news.d2w macro in the directory /u/USER1/macros/special/ have PUBLIC scope and remain valid for 12 hours. All Web pages generated by the employee_stats.d2w macro in the directory /u/USER1/macros/special/ have PRIVATE scope and remain valid for 1 hour.

Step 2: Set Up the Web Page Cache

Set up the table used to cache Web pages.

1. Create the Web page cache table, SYSIBM.DTWCACHEDPAGES, using the SQL found in DTW220.SDTWSPUF(DTWCRCCH). This file also includes SQL statements to create a database, called DTWCACHE, and a tablespace, called DTWTBSP1, for the Web page cache.

   In a data sharing environment, the CREATE TABLESPACE statement that creates the tablespace for SYSIBM.DTWCACHEDPAGES and SYSIBM.DTWCACHEDEPS should specify GBPCACHE CHANGED.
2. Define the stored procedure used to insert the cached pages into SYSIBM.DTWCACHEDPAGES. The stored procedure is found in DTW220.SDTWLOAD(DTWCCHIN).
   a. Copy the stored procedure into your stored procedure library.
   b. Define the stored procedure.
      - The SQL to define the stored procedure when using DB2 for OS/390 V5 is found in DTW220.SDTWSPUF(DTWCCHV5).
      - The SQL to define the stored procedure when using DB2 UDB Server for OS/390 V6 is found in DTW220.SDTWSPUF(DTWCCHV6).
3. Bind the stored procedure's DBRM DTW220.SDTWDBRM (DTWV22IN) into the package DTWCACHEPKG using a user ID with INSERT, SELECT, and DELETE privileges on SYSIBM.DTWCACHEDPAGES. The user IDs associated with the requests that cache pages must have EXECUTE privilege on the package DTWCACHEPKG.

   The JCL for this step is located in DTW220.SDTWBASE(DTWCCHBD).

After you have set up this table and stored procedure, you can begin caching Web pages.

Suppressing DB2 for OS/390 Messages

You can improve the performance of Net.Data for OS/390 when using the SQL language environment by suppressing DB2 messages from non-zero SQLCODEs. Use the DB2MSGS configuration variable to indicate the level of messages that is necessary for your application. Within production environments, you can bypass DB2 message lookups by setting DB2MSGS to NONE. When DB2MSGS is set to NONE or ERRORONLY, you can still catch non-zero SQLCODEs with MESSAGE blocks within your macro. See *Net.Data Reference* to learn how to use the MESSAGE block in your macro.

To specify the messaging level, use the DB2MSGS configuration variable in the Net.Data initialization file.

Possible values:

`DB2MSGS [=] message_level`

Where *message_level* indicates the level of DB2 messages provided by Net.Data and can be specified as follows:

NONE Specifies that Net.Data provides no messages.

ERRORONLY Specifies that Net.Data provides messages only for negative SQLCODE values.

ALL Specifies that Net.Data provides messages for all SQLCODE values. This is the default. If a value is provided for DB2MSGS other than one of the valid values listed above, Net.Data uses the default value of ALL.

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 123

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.
- 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 SQL language environment are described. To learn about DB2 performance considerations, visit the web at: *http://review.ibm.com/software/data/db2/performance*

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 is 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. Serviceability Features

The following sections describe new tracing and error reporting features for Net.Data.

Net.Data Trace Log

Net.Data provides trace data about the execution of your macro that is recorded in the trace log. You can specify where the trace log is stored and what level of tracing is recorded. Use trace information for debugging macros and providing information when working with your IBM service representative. See *Net.Data Messages and Codes Reference* for a list of Net.Data trace messages.

Configuring Net.Data for Tracing

To configure Net.Data for tracing, you need to set configuration variables to specify where the trace log is stored and what level of trace data Net.Data needs to capture.

Setting the Trace Log Directory

The name of the trace log is `netdata.trace`. Use the DTW_TRACE_LOG_DIR configuration variable to specify the directory in which the trace file is stored.

Syntax:

```
DTW_TRACE_LOG_DIR [=] full directory
path
```

Example:

```
DTW_TRACE_LOG_DIR /usr/lpp/internet/server_root/logs
```

Setting the Level of Trace Logging

Determine the level of tracing that Net.Data logs by setting the value of the configuration variable, DTW_TRACE_LOG_LEVEL.

Syntax:

```
DTW_TRACE_LOG_LEVEL [=] OFF|APPLICATION|SERVICE
```

Where:

OFF Specifies that no trace data is captured in the trace log. This is the default value.

APPLICATION
Net.Data writes application-level trace messages to the trace log.

SERVICE
Net.Data writes all trace messages to the trace log.

Example:

```
DTW_TRACE_LOG_LEVEL SERVICE
```

Trace Log Format

The format of a trace log entry is:

```
[DD/MMM/YYYY:HH:MM:SS] [macro][PID#][TID#][UID] trace_message
```

Where:

DD/MMM/YYYY:HH:MM:SS
: Is a timestamp indicating when the trace entry was created.

macro
: Is the name of the macro that generated the trace message.

PID#
: Is the process ID of the request that generated the trace message.

TID#
: Is the thread ID number of the request that generated the trace message.

UID
: Is the user ID associated with the request that generated the trace message.

trace_message
: Is the text of the trace message.

Access Rights

To successfully write trace messages to the trace log file, the user IDs under which Net.Data executes must have:

- Write authority on the log directory specified in the DTW_TRACE_LOG_DIR configuration variable.
- Execute authority on all directories in the path, including the log directory.

Net.Data Error Log

Net.Data provides the ability to capture Net.Data and DB2 error messages in a log file. You can specify where the error log file is stored and what type of error messages are logged.

Configuring Net.Data for Error Message Logging

To configure Net.Data for error message logging, you need to set configuration variables to specify where the log file is stored and what level of error messages Net.Data needs to capture.

- “Setting the Error Log File Directory”
- “Setting the Level of Error Logging”

Setting the Error Log File Directory

The name of the error log file is `netdata.error.log`. Use the DTW_ERROR_LOG_DIR configuration variable to specify the directory in which the trace file is stored.

Syntax:

```
DTW_ERROR_LOG_DIR [=] full directory
path
```

Example:

```
DTW_ERROR_LOG_DIR /usr/lpp/internet/server_root/logs
```

Setting the Level of Error Logging

Determine the type of messages that Net.Data logs by setting the value of the configuration variable, DTW_ERROR_LOG_LEVEL.

Syntax:

```
DTW_ERROR_LOG_LEVEL [=] OFF|INFORMATION|ALL
```

Where:

OFF Specifies that no error messages are captured in the error message log. This is the default value.

INFORMATION
Net.Data logs messages marked ″Information″ (I).

ALL All messages are logged.

Example:

```
DTW_ERROR_LOG_LEVEL ALL
```

Error Log File Format

The format of a log file entry is:

```
[DD/MMM/YYYY:HH:MM:SS] [macro][PID#][TID#][UID] error_message
```

Where:

DD/MMM/YYYY:HH:MM:SS
Is a timestamp indicating when the log entry was created.

macro Is the name of the macro that generated the error message.

PID# Is the process ID of the request that generated the error message.

TID# Is the thread ID of the request that generated the error message.

UID Is the user ID associated with the request that generated the error message.

error_message
Is the text of the error message.

Access Rights

To successfully write error messages to the error log, the user IDs under which Net.Data executes must have:

- Write authority on the log directory specified in the DTW_ERROR_LOG_DIR configuration variable.
- Execute authority on all directories in the path, including the error log directory.

Appendix A. Bibliography

Net.Data Technical Library

The Net.Data Technical Library is available from the Net.Data Web site at
`http://www.ibm.com/software/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.
Program Directory for Net.Data for OS/390 Version 2 Release 2	Describes SMP/E installation and configuration of Net.Data for OS/390

Related Documentation

The following documents might be useful when using Net.Data and related products:

- *Accessing DB2 for OS/390 Data from the World Wide Web*, Maria Sueli Almeida, Charles E. Lewis, Uwe Sager, Pilar Sandoval
- *IBM Internet Connection Secure Server Planning for Installation Version 2 Release 2 for OS/390*, GC31-8489
- *IBM Internet Connection Secure Server Webmaster's Guide Version 2 Release 2 for OS/390*, GC31-8490
- *Lotus Domino Go Webserver Planning for Installation Version 4.6.1 for OS/390*, SC31-8642
- *Lotus Domino Go Webserver Webmaster's Guide Version 4.6.1 for OS/390*, SC31-8643
- *OS/390 MVS Planning: Workload Management*, GC28-1761

Appendix B. Configuring Net.Data for OS/390 to Access DataJoiner

You can use Net.Data for OS/390 with DataJoiner to access remote databases such as DB2/6000, Oracle, and Sybase. This section describes how to configure your system for use with DataJoiner for AIX Version 1.2 with PTF U447593 or DataJoiner for HP-UX Version 1.1.

Configuration steps:

1. Enter the information needed in the communications database (CDB) for remote communication to DataJoiner. Information on the CDB is in *DB2 Installation Guide*.
2. Bind the Net.Data DBRM to the remote location where DataJoiner is installed using the BIND PACKAGE command.
3. Bind the Net.Data DBRM to DB2 using the BIND PLAN command. Use the PKLIST option to include the package created at the remote location.
4. Modify the Net.Data initialization file, which is in the Web server's document root directory, to specify the LOCATION variable as an input variable to SQL functions. The new DTW_SQL environment statement looks like this:

   ```
   ENVIRONMENT (DTW_SQL) dtwsql (IN LOCATION)
   ```

Net.Data macros that access remote data using DataJoiner must specify a value for LOCATION. This example Net.Data macro queries a remote database through DataJoiner:

```
%{ ****************** Define Block ******************************** %}
%DEFINE {
  DB2SSID="NDA1"
  LOCATION="QMFDJ00"
  DTW_DEFAULT_REPORT="YES"
%}

%{ ****************** Function Definition Block ******************* %}
%FUNCTION(DTW_SQL) selectall() {
  SELECT * FROM $(tabnam)
%}

%{ ****************** HTML Block: Table_Input ********************** %}
%HTML(Table_Input) {
<Title>DJ Test #1</Title>
<Body>
<h1 align=center>Table Selection</h1>
<br>
<form method="post" action="Column_Output">
<p>Enter Table Name: <input type="text" name="tabnam"></p>
<p><input type="submit"></p>
</form>
</Body>
%}

%{ ****************** HTML Block: Column_Output ******************* %}
%HTML(Column_Output) {
<Title>DJ Test #1</Title>
<Body>
@selectall()
</Body>
%}
```

Appendix C. 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 uses an include file, which contains the DEFINE block for the macro.

Figure 7 on page 134 shows the sample macro. Figure 8 on page 137 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 7. 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 7. 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 7. 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   =""

    exampleTitle = "Sample Macro"
    myTable = "EMPLOYEE"
    DATABASE = "sample"
%}

%{     End of include file  %}
```

Figure 8. Include file

Appendix D. 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 and GWAPI.

CGI. Common Gateway Interface.

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.

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.

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.

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.

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.ibm.com/software/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

E

F

G

H

I

J

L

M

T

U

V

W

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.