CHAPTER 11

Using the Database APIs

CWCS database APIs are used primarily for installing and configuring custom databases. These APIs hide the configuration, platform details, and database management processes from applications. They allow applications to:

- Manipulate ODBC data sources
- Start and stop database processes and identify database versions
- Run SQL scripts
- Manipulate backup manifests

The following topics describe how to create a custom database and how to use the CWCS database APIs in your applications:

For more information about the CWCS database APIs, refer to:

- *CMF 1.2 Database Functional Specification*, EDCS ENG-54964
- *Database Security*, EDCS ENG-80264

For details on backing up and restoring the database, see Chapter 12, "Using Backup and Restore."

The Sybase SQL Anywhere Studio 9.0.0 Core Documentation Set is available online at http://sybooks.sybase.com/onlinebooks/group-sas/awg0900e. Recommended titles in this set include:

- *Adaptive Server Anywhere Database Administration Guide*
- *Adaptive Server Anywhere Getting Started*
- *Adaptive Server Anywhere Programming Guide*
- *Adaptive Server Anywhere SQL Reference Manual*
- *Adaptive Server Anywhere SQL User's Guide*
- *Introducing SQL Anywhere Studio*
- *MobiLink Synchronization User's Guide*
- *Ultralite Database User's Guide*

Understanding the CWCS Database

The following topics describe various aspects of the CWCS database:

What's New in This Release

The following changes have been implemented in this release:

- JConnect Upgrade: Sybase declared JConnect 4.2 End-of-Life in December 2002, and no longer supports it. Accordingly, CWCS 3.0 drops support for JConnect 4.2, and upgrades support for JConnect 5.2 to JConnect 5.5.
- Sybase Upgrade: The Sybase database engine on Solaris and Windows was upgraded to 9.0.0 + EBF. The EBF versions as of release were 9.0.0.1364 (Solaris) and 9.0.0.1366 (Windows).

Understanding the Tools

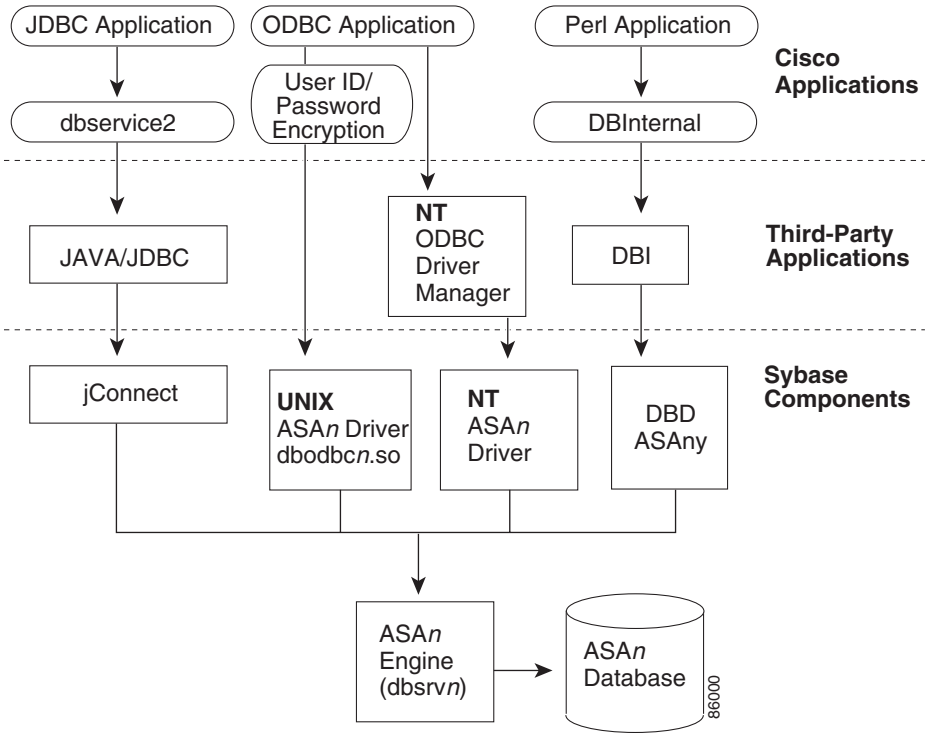
These third-party tools are required to implement the CWCS database:

Tool	Version	Description
jConnect	5.5	JDBC commands use this Sybase component to access the database engine.
JDBC	1.2	Java Database Connectivity—A set of Java APIs that provide universal data access for the Java programming language.
ODBC	2.x ODBC, 3.510 of ODBC Driver Manager	Open Database Connectivity—A standard call level interface developed by Microsoft and based on the SQL AccessGroup CLI specification. ODBC is a database-independent C language API that consists of a driver manager (supplied by the operating system) and drivers for each database vendor.
SQL Database Engine	V9.0.0 + EBF	Adaptive Server Anywhere 9.0.0 is the current version of the Sybase database present in CWCS. EBF versions as of release were 9.0.0.1364 (Solaris) and 9.0.0.1366 (Windows). This database supports the ODBC API on UNIX and Windows platforms. Includes access from Java via JDBC, C/C++ via ODBC, and Perl via DBI.
DBInternal	CWCS component	DBInternal is the link that Perl uses to access the database (on Windows platforms only).

Database Access Methods

CiscoWorks applications can communicate with the CWCS database using these methods: ODBC, JDBC, and Perl. Figure 11-1 shows how each method accesses the CWCS database.

Figure 11-1 Database Access Applications



The following topics describe these access methods:

- Types of Database Servers
- JDBC Access Methods
- ODBC Access Methods
- Perl Access Methods
- Connection Strings

Types of Database Servers

There are two types of database servers:

- dbsrv9—The network version. This allows unlimited concurrent connections, provides multi-user use, and supports client/server communication across a network.
- dbeng9—The personal version. This allows up to 10 concurrent connections. Use this simpler version when:
  - The CWCS Server is down, and therefore no other connection is on.
  - A simple task such as restore or backup needs to connect to the database to check the data.

JDBC Access Methods

JDBC (Java Database Connectivity) is a set of Java APIs that provide universal data access for the Java programming language. JDBC provides a standard interface between your application and the database server.

The JDBC commands use the Sybase component, jConnect, to access the database engine. When an application makes a database connection, the classes in dbservice2 retrieve the connection the information from the DbServer.properties file, including the database user ID and password, to construct the JDBC URL.

The dbservice2 Java classes also provide commonly-used JDBC methods. Use these Java methods, which sit on top of the JDBC APIs, *instead* of the JDBC API to shield any database-related changes from high-level applications.

Related Topics

See:

- The "About the Database Property Files and Settings" section on page 11-12.
- Sun's Java training site at the following URL:
  http://developer.java.sun.com/developer/onlineTraining/Database/JDBCShortCourse/index.html

ODBC Access Methods

ODBC (Open Database Connectivity) is a standard call level interface (CLI) developed by Microsoft and based on the SQL AccessGroup CLI specification. An industry standard, ODBC is a database-independent C language API that consists of a driver manager (supplied by the operating system) and drivers for each database vendor.

The Sybase Adaptive Server Anywhere database engine supports the ODBC API on UNIX and Windows platforms (see Figure 11-1). To access ODBC functions, the applications must be compiled and linked with the appropriate import library file. The Sybase ODBC driver makes the connection using the database user ID and password stored in the .odbc.ini file on UNIX platforms or the system registry on Windows platforms.

Related Topics

See Microsoft's online SDK site at the following URL:
http://msdn.microsoft.com/downloads/sdks/platform/database.asp

Perl Access Methods

DBI is the Perl ODBC interface. It defines a set of methods, variables, and conventions that provide a consistent database interface, independent of the actual database being used. DBI does not access any particular database; instead, it locates and loads the applicable driver modules.

The database user ID and password are stored in the .odbc.ini file on UNIX platforms or the system registry on Windows platforms.

Perl applications use the following drivers to access the database:

- On Windows platforms, Perl applications use the DBInternal driver (see Figure 11-1) to connect to the DBI module.

- On Solaris platforms, Perl applications use the DBD driver. The DBD (Database Driver) modules contain the vendor libraries and can access the actual databases; there is one DBD module for every database. For example, the driver for the Adaptive Server Anywhere database is DBD::ASAny.

Related Topics

See the Comprehensive Perl Archive Network (CPAN) web site at the following URL:
http://www.perl.com/CPAN-local/README.html

Connection Strings

Applications need to establish a connection to the database before they can interact with the database. A connection requires, at a minimum, the user ID, password, and database name. For SqlAnywhere databases and ODBC programs, this information is specified as a single parameter in the form of a connection string:

- SqlAnywhere connection strings—Used for any registered or unregistered database.

  SqlAnywhere databases use the form ENG=*xx*;CWEUID=*xx*;CWEPWD=*xx*, where:

  - ENG is the database engine name
  - CWEUID is the encrypted database user ID
  - CWEPWD is the encrypted database password

  This connection string can be made more specific by adding the DBN parameter to refer to a database attached to the database engine.

  Note If the encryption flag is ON, applications must use the encrypted user ID and password keywords, CWEUID and CWEPWD. If, however, the application still uses the plain text user ID and password (the encryption flag is OFF), the old connection string format that uses UID and PWD will still work.

- ODBC connection strings—Used only if the database is registered.

  ODBC connection strings use the form DSN=*xx*;CWEUID=*xx*;CWEPWD=*xx*, where the DSN parameter refers to a data source that contains a detailed definition of the data source. This includes the name of the database engine, engine start up parameters, the path to the database root file, and so on.

  The data source can also store the user ID and password. In this case, the connection string can be just DSN=*xx*. The data source information is kept in the registry on Windows platforms, and in the .odbc.ini file on UNIX systems.

Understanding the NMTG Database Delivery Process

Table 11-1 provides a summary of the database files that require special handling during the delivery process. *If you are not using ClearCase and the NMTG installation processes*, you will need to create a similar process for delivering these files.

Table 11-1 NMTG Database Delivery Phases and Files

Delivery Phase	Special Files
Create these files and place them in the ClearCase vob.	These files will be copied to the CD by the build process: • $*NMSROOT*/databases/orig/odbc.tmplorig: Contains the factory password for the initial database (*db_name*.dborig), the user ID and password, the engine name, and the port ID. The odbc.tmplorig file must be duplicated to the odbc.tmpl file during installation. • $*NMSROOT*/databases/orig/db.dborig: Contains the default database for your module. To create a database, use the dbinit command. Refer to later sections for details on using the dbinit command and changing the default username/password for this database.
During installation	The CWCS database is registered and System Services are enabled by default. The registration and installation process: • On Solaris only: Renames the .odbc.tmpl file to .odbc.ini and populates it with the user ID, password, and other database information for each database engine. • Renames the database template files (db.dborig) and copies them to $*NMSROOT*/databases/*dsn*.db. For example, $*NMSROOT*/databases/cmf/orig/cmf.dborig is renamed and copied to $*NMSROOT*/databases/cmf/cmf.db. • On Solaris systems: Populates the dmgtd.conf file with the database engine command for each database. It also adds the database monitor command for each database. • Updates the DBServer.properties file with the URLs for the installed suites as well as the database credential information. • On Windows only: Updates the Windows registry (if applicable) with the ODBC service information.
After installation	The following files have been modified: • Windows registry—For Windows platforms. Contains the ODBC services for the database engine. • .odbc.ini—For Solaris systems. Contains the ODBC services for the database engine. • dmgtd.conf—A Daemon Manager file that contains the database engine and database monitor commands. • DBServer.properties—Contains the JDBC URL for the installed database.

Setting Up a New Database

The following topics describe the files, settings, and processes required to create a new database:

- Creating the ODBC Database Definition File
- Creating the Backup Manifest Files
- About the Database Property Files and Settings
- Managing the Database Engine

If you want to set up a new database quickly, see the "Performing a Quick Integration" section on page 11-17.

Creating the ODBC Database Definition File

The ODBC DSN uses the database definition file, .odbc.ini, to provide database-specific information to ODBC applications. This file contains the user ID, password, and other database information.

- On Windows platforms, the ODBC definition is located under the Windows Registry. For example, the registry key for CWCS is:

  ```
  HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\cmf
  ```

  The key CWEUID contains the encrypted user ID, and the key CWEPWD contains the encrypted password.

- On Solaris platforms, the ODBC definition is located under *$NMROOT/*.odbc.ini. The environment variable ODBCINI=$NMROOT/.odbc.ini is defined by the Daemon Manager.

The following topics describe the procedures for creating a database template file:

- Creating the Database Template File
- Creating the odbc.tmplorig Template File
- Enabling Database Password Encryption

Creating the Database Template File

The registration and configuration process uses the database template file, .odbc.tmplorig, to create the to create the odbc.tmp and .odbc.ini files on Solaris, and the registry entries on Windows:

- On Solaris platforms, it renames the .odbc.tmpl file to .odbc.ini and populates it with the user ID, password, and other database information for each database engine.
- On Windows platforms, the .odbc.ini file does not exist. Instead, the odbc.tmpl file is used to populate the Windows registry with the user ID, password, and other database information for each database engine.

Note If you are a developer working inside NMTG, follow the procedure in the "Creating the odbc.tmplorig Template File" section on page 11-7.

Creating the odbc.tmplorig Template File

If you are a developer working inside NMTG, create an odbc.tmplorig database template file. The automated build processes use this file to create the corresponding .odbc.ini file on the target system. Use the following procedure to create the odbc.tmplorig file.

Step 1 Create the odbc.tmplorig file using these conventions:

- **File Name:** odbc.tmplorig
- **ClearCase Location:** /vob/enm_cmf/share/databases/cmf/

Step 2 Include the following lines (on Solaris platforms, expand the file names to their full path):

```
UID=cmfDBA
PWD=c2kY2k
Start=dbsrv9
DatabaseName=cmfDb
EngineName=cmfEng
```

```
CommLinks=tcpip{HOST=localhost;DOBROADCAST=NO;ServerPort=43441}
CWENCRYPTION=YES
AutoStop=yes
# note __ values are not passed through for odbc registration
# These __ values are skipped by odbcdsn.pl.
# These __ values are used for configuring db engine startup parms.
__Cache=8
__DbNTSvcLongName=CiscoWorks Cmf database engine
JdbcDriver=com.sybase.jdbc2.jdbc.SybDriver
DmPrefix=Cmf
```

The JdbcDriver line populates the JdbcDriver entry in DBServer.properties. For more information about this entry, see the "Creating the Database Template File" section on page 11-7. To register with custom switches, or specify a JDBC driver, see the "Customizing the odbc.tmpl File" section on page 11-8.

Step 3 Make these changes:

- Line 1: Replace *cmfDBA* with your user ID.
- Line 2: Replace *c2kY2k* with your password.
- Line 3: Replace *dbsrv9* with the database engine name.
- Line 4: Replace *cmfDB* with your database name. Do not include the absolute path; the full path is constructed and inserted into the .odbc.ini file. For more information about this process, see the "Understanding the NMTG Database Delivery Process" section on page 11-5).
- Line 5: Replace *cmfEng* with your engine name.
- The CommLinks line contains database engine command line parameters. Replace "43441" with the port ID for your database. To determine which port ID number to use, see the "Managing the Database Engine" section on page 11-13.
- If you do not want to enable database password encryption, remove the line, CWENCRYTPTION=YES (see the "Enabling Database Password Encryption" section on page 11-9).
- Adjust other lines as needed:
  - The Cache line is a database engine command line parameter that defines the size of the cache (default = 8 M).
  - The __DbNTSvcLongName defines the Windows Service name to be the CiscoWorks database engine.
  - The JdbcDriver line populates the JdbcDriver entry in DBServer.properties.
  - The Dmprefix is the prefix registered for this database in the CWCS Daemon Manager.

Customizing the odbc.tmpl File

If you want to register your database with custom flags, you need to include the `-Switches` entry in odbc.tmpl, with the additional switches you want specified on the same line as the property name.

If the `_Switches` property is not present, the database will be registered with the following default switches on:

- On Windows platforms: `-m -ti 0 -gm 100`
- On Solaris platforms: `-q -m -ti 0 -gm 100`

Note that if you specify custom switches and also want the default switches, you must include both on the `_Switches` line.

The following will always be present and need not be included on the `_Switches` line:

- `-c` for cache size (this is taken from the `_Cache` line) .
- `-n` for the engine name and database file name
- On Solaris only: The option `-s $ENV{PX_FACILITY}` is always added. You need not enter it with your custom switches.

For example:

```
CWEUID=r0wicBlFHWg=
CWEPWD=vJa9p8EtilQ=
Start=dbsrv9
DatabaseName=cmfDb
EngineName=cmfEng
CommLinks=tcpip{HOST=localhost;DOBROADCAST=NO;ServerPort=43441}
AutoStop=yes
CWENCRYPTION=YES
# note __ values are not passed through for odbc registration
# These __ values are skipped by odbcdsn.pl.
# These __ values are used for configuring db engine startup parms.
__Cache=8
__Switches= -u -p
__DbNTSvcLongName=CiscoWorks Cmf database engine
DmPrefix=Cmf
```

You also have the option to specify any compatible JDBC driver. To do so, add the following entries in the odbc.tmpl file:

```
"JdbcDriver=myDriver"
"DataSourceUrl=MySource"
```

When the database is registered using configureDb.pl, these entries will be added to DBServer.properties. If these entries are absent, Jconnect will be used by default.

Enabling Database Password Encryption

The encrypted password and username are stored in the .odbc.ini file on Solaris and the registry entry on Windows. It is also stored in the odbc.tmpl and DBServer.properties files.

To enable password encryption, add the `CWENCRYPTION` flag to the odbc.tmpl file. For example:

```
UID=cmfDBA
PWD=c2kY2k
Start=dbsrv9
DatabaseName=cmfDb
EngineName=cmfEng
CWENCRYPTION=YES
CommLinks=tcpip{HOST=localhost;DOBROADCAST=NO;ServerPort=43441}
AutoStop=yes
__Cache=8
__DbNTSvcLongName=CWCS Cmf database engine
```

The `CWENCRYPTION` flag:

- Indicates that an application wants to encrypt its identification information.
- Has two possible values, `YES` or `NO`. The default is `NO`.

We strongly recommend that all applications call the following command during installation:

```
$NMSROOT/objects/db/conf/ChangeDbPasswd.pl dsnname newpwd
```

Put this command in the following location:

- On Solaris platforms: postinstall
- On Windows platforms: the rul file

You can call ChangeDbPasswd.pl before or after the database is installed.

ChangeDbPasswd.pl validates passwords. Valid passwords must:

- Have a minimum of five and a maximum of 128 characters.
- Use alphanumeric characters (a-z, A-Z, 0-9) only. No special characters (e.g., #, $, %) or spaces are allowed.
- Not have a number as the first character.

If you are setting a new database password during the application install, we recommend that you validate the submitted password. If your install submits an invalid password, ChangeDbPasswd.pl will generate an appropriate information message and will *not* change the password.

Related Topics

See the:

- "Creating the Database Template File" section on page 11-7.
- "Creating the odbc.tmplorig Template File" section on page 11-7.

Creating the Backup Manifest Files

Backup manifest files are ASCII text files used by the CWCS backup and restore framework. These files contain a list of the database files or directories to be backed up.

There are two types of backup manifest files:

- The *database* backup manifest file contains a list of database file names. The backup.pl script uses this list to determine which database files to back up. The database backup manifest file is stored in the directory *$NMSROOT*/backup/manifest/*suite*/database/orig/*dsn*.txt, where:
  - *$NMSROOT* is the directory in which the product will be installed.
  - *suite* is the name of your application or suite. Often, this is the same as the *dsn*. For example: For CWCS, the *suite* and *dsn* are both "cmf", but for Campus Manager, the *suite* is "campus" and the *dsn* is "ani".
  - *dsn* is the the data source (database) name.
- The *application* backup manifest file contains a list of directories and files where application-specific data is stored. The backup.pl script uses this list to determine the application data to back up. The application backup manifest file is stored in the directory *$NMSROOT*/backup/manifest/*suite*/*app*/orig/*datafiles.txt*, where:
  - *$NMSROOT* is the directory in which the product will be installed.
  - *suite* is the name of your application or suite, as for the database backup manifest file.
  - *app* is the name of the application or module within the suite. These usually vary. For example: For Resource Manager Essentials, the *suite* is "rme", but the *app* may be "configArchive".

The following topics explain how to create both types of backup manifest files:

- Creating the Database Backup Manifest File

- Creating the Application Backup Manifest File

Note Use of the CWCS backup and restore framework requires that Campus Manager, ACLM, and DFM developers change their application backup manifest directory structure. For details, see the "CWCS Backup" section on page 12-1.

Related Topics

See the:

- "Enabling the CWCS Database Engine" section on page 11-37.
- "Using CWCS Backup" section on page 12-1.
- "Running CWCS Backups" section on page 12-3.
- "backup.pl" section on page 12-10.

Creating the Database Backup Manifest File

Use the following procedure to create the backup manifest files for your database:

Step 1 Create a *dsn*.txt file, where *dsn* is the name of your database.

Step 2 Include the following lines in *dsn*.txt, replacing all occurrences of "cmf" with your database name:

```
[cmf]
root=$ENV{NMSROOT}/databases/cmf/cmf.db
```

Step 3 Copy the *dsn*.txt file to the following directory in the runtime tree:
$NMSROOT/backup/manifest/*suite*/database/orig/*dsn*.txt

Where:

- *$NMSROOT* is the directory in which the product was installed.
- *suite* is the name of your application or suite. This is sometimes the same as the *dsn*. For example: For CWCS, the *suite* and *dsn* are both "cmf", but for Campus Manager, the *suite* is "campus" and the *dsn* is "ani". When you install CWCS, the cmf.db database is loaded and enabled by default.
- *dsn* is the name of your database.

The final runtime location of this file will be *$NMSROOT*/backup/manifest/*suite*/database/ (that is, the /database subdirectory that is the parent of /orig).

Creating the Application Backup Manifest File

Use the following procedure to create the backup manifest file for your application's files:

Step 1 Create a datafiles.txt file.

Step 2 On separate lines, list the application paths and files to be backed up. For example:

```
$ENV{NMSROOT}/lib/classpath/com/cisco/nm/cmf/servlet/cwpass

$ENV{NMSROOT}/lib/classpath/sso.properties

$ENV{NMSROOT}/lib/classpath/com/cisco/nm/dcr/dcr.ini
```

```
$ENV{NMSROOT}/lib/eds/filter/namedfilter.str
```

The `$ENV(NMSROOT)` value must precede each line (the `configureDb.pl action=install` call will replace it with the actual installation directory).

Step 3 Copy datafiles.txt to the following directory in the runtime tree:

`$NMSROOT``/backup/manifest/`*`suite`*`/`*`app`*`/orig/datafiles.txt`

Where:

- *`$NMSROOT`* is the directory in which the product was installed.
- *`suite`* is the name of the suite containing your application (often, this is the same as the *dsn*).
- *`app`* is the name of your application. Often, this is the same as *`suite`*, especially for standalone applications.

The final runtime location of this file will be *$NMSROOT*/backup/manifest/*suite*/*app*/ (that is, the /app subdirectory that is the parent of /orig).

About the Database Property Files and Settings

The following topics describe the two types of database property files:

- About the Database Server Property File
- About Private Property Files

About the Database Server Property File

DBServer.properties is the database server properties file. It contains the configuration parameters for Java Database Connectivity (JDBC) application database functions such as various debug levels, timeouts, and sleep periods, the port the database service module listens to for socket-based requests, the maximum number of database connections, and so on.

File Name	DBServer.properties
Runtime Location	*$NMSROOT*/www/classpath/com/cisco/nm/cmf/dbservice (where *$NMSROOT* is the directory in which the product was installed).
ClearCase Location	enm_cmf/share/classes/client/com/cisco/nm/cmf/dbservice/orig
Example	To see an example of the DBServer.properties file, refer to the CodeSamples directory on the CWCS SDK CD.

Typically, you should *not* change the contents of the DBServer.properties file; the information in this file is created by CMFEnable.pl. For each registered database, the CMFEnable script adds several lines to this file. For example, these lines were appended for the CMF database:

```
### dbconnection for cmf ###
DBConnection.userName.cmf=cmfDBA
DBConnection.password.cmf=c2kY2k
DBConnection.dataSourceUrl.cmf=jdbc:sybase:Tds:localhost:43441?SERVICENAME=c
mfDb
DBConnection.jdbcDriver.cmf=com.sybase.jdbc2.jdbc.SybDriver
```

You might, however, need to modify this file to change:

- The JDBC connection information
- The jConnect tuning parameters
- Any debug level, sleep, or timeout values

To make changes to the DBServer.properties file:

Step 1 Use an ASCII editor such as Notepad to update the file.

Step 2 Stop the database engine process.

Step 3 Restart the database engine process.

Related Topics

- "Enabling the CWCS Database Engine" section on page 11-37.
- "Starting a Database Engine" section on page 11-25.
- "Stopping a Database Engine" section on page 11-27.

About Private Property Files

Some applications include the database password in a private, application-specific property file. For CWCS to recognize this password, these applications must adhere to the following conventions:

- Add the location of the private property file to the odbc.tmplorig file. The key name is PropertyFile. The value must include the path of its private property file relative to *$NMSROOT*, the directory in which the product was installed. For example, the odbc.tmplorig file for ANI includes this line:

  ```
  PropertyFile=etc/cwsi/ANIServer.properties
  ```

- The key name for the database password in its property file must be "DB.password". For example, the ANIServer.properties file includes this line:

  ```
  DB.password=cwsiPWD
  ```

Related Topics

- "Creating the ODBC Database Definition File" section on page 11-7.

Managing the Database Engine

The following topics describe some important database engine management tasks:

- Understanding Port IDs
- Creating a Database Port
- Changing the Database Port
- Dynamically Allocating a Port ID

Understanding Port IDs

Because CWCS uses a database that supports cross-network operations, every database engine must have its own port ID. Every network server must define a unique port ID as a TCP/IP parameter.

At install time, the Installer framework makes a call to the CWCS service bundles enabling mechanism, CMFEnable.pl. This Perl script uses the values in the CommLinks line in the database template file, .odbc.ini, to assign each database engine its own port ID. For more information about the database template file, see the "Creating the ODBC Database Definition File" section on page 11-7.

All newly-developed databases must define a CommLinks value. The format of the CommLinks definition is:

```
CommLinks=tcpip{HOST=localhost;DBBROADCAST=NO;ServerPort=portid}
```

As part of the database registration process, CMFEnable.pl reads the .odbc.ini file and populates the port ID from the CommLinks entry to several places:

- **Database Server Command Line**

  A network server is started with a TCP/IP protocol. The command line is stored in the following locations in the runtime tree:

  - On Solaris Platforms, it is stored in:

    $NMSROOT/conf/dmgt/dmgtd.conf

  - On Windows platforms, it is stored under this Windows system entry:

    HKEY_LOCAL_MACHINE \\SYSTEM\\CurrentControlSet\\Services

- **The ODBC connection parameters**

  The ODBC driver detects a database RPC protocol and port ID from entry CommLinks. This entry is defined:

  - On Solaris Platforms, in this file:

    $NMSROOT/.odbc.ini

  - On Windows platforms, in the Windows system registry entry with this key:

    HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\dsn\CommLinks

- **The JDBC URLs**

  For both Windows and UNIX platforms, the JDBC application reads the database URL definition from this file:

  $NMSROOT/www/classpath/com/cisco/nm/cmf/dbservice/DBServer.properties

The CWCS installation code checks to see if the port ID is available, and gives a warning if it is not. This warning, however, does not stop the installation process.

The following topics describe how CWCS allocates port IDs:

- Creating a Database Port
- Changing the Database Port
- Dynamically Allocating a Port ID

Related Topics

See the "Enabling the CWCS Database Engine" section on page 11-37.

Creating a Database Port

CWCS reserves ports 43441 through 43549 for database servers. Table 11-2 shows the ports permanently allocated to CWCS and existing applications. To permanently allocate a database port for your application, contact the CWCS database team via the support alias, embu-db-interest@cisco.com.

If you are using Solaris platforms and CWCS-supplied scripts like configureDB.pl, the scripts will check for and select a free port automatically, as follows:

1. If you specified a port in odbc.tmpl, the scripts will check /etc/services to see if this port is already in use. Then:
   a. If there is no entry for the port in /etc/services, the scripts will assume the port is free and will select it.
   b. If there is an entry for the port, the script will:
   - Assume that another application is using it.
   - Pick a port from the dynamic range 43461-43480.
   - Check to see if that port is free. It will select the first port in that range for which there is no entry in /etc/services.
   - If none of the ports in the range 43461-43480 are free, the script returns an error.
2. If there is no port specified in odbc.tmpl, a free port from the range 43461-43480 is picked up and selected.
3. The scripts enter the selected port in /etc/services. For example:

```
cscocmfdb        43441/tcp       # CSCO NM cmf database
```

Note that port checking and /etc/services updates are *not* available if you are using Windows platforms or non-CWCS configuration scripts. For details, consult DDTS defects CSCsa09950 and CSCsa11233.

Table 11-2 Permanently Allocated Database Server Ports[1]

Port	Application	Contact
10033[2]	IDS MC	Patti Abkowitz (abkowitz)
10033[2]	Sec Mon	Patti Abkowitz (abkowitz)
10033[2]	PIX MC	Joel Klein (jfklein)
10033[2]	AUS	Jared Smith (jarsmith)
10033[2]	QPM	Oren Fridler (ofridler)
10033[2]	Router MC	Yardena Meymann (ymeymann)
43441	CWCS	Vikram Rao (vikram)
43442	RME	Vivi Zhang (vzhang)
43443	Campus (ANI)	Suresh Pathamadai (pssuresh)
43444	Service Level Manager	Sajan Mathew (samathew)
43445	Kilner (FH)	Pavan Kumar Mirla (pavankm)
43446	Kilner (Inventory)	Pavan Kumar Mirla (pavankm)
43447	Kilner (EPM)	Pavan Kumar Mirla (pavankm)
43448	Kilner (AMA)	Shiva Shankar (shaj)

Table 11-2 Permanently Allocated Database Server Ports[1] (continued)

Port	Application	Contact
43449	PIF	Shiva Shankar (shaj)
43450[3]	PIF HTTP Server	Shiva Shankar (shaj)
43451	WAN Performance Utility	Mathangi Kuppusamy (mathangi)
43453	Performance Monitor	Wei W. Wang (weiwa)
43454	Security Auditor	Mark Lu (malu)
43455	RME NG	Venunadh Veerala (vveerala)
43458	CVM	Auro Dharmapuram (auro)
43459	Pairpoint	Subbu Chandrasekaran (csubrama)
44341	IPM	Pavan Kumar Mirla (pavankm)

1. For the most current list, see http://wwwin-embu.cisco.com/embueng/database_ports.htm.
2. These databases share the same database server (SqlCoreDBServer).
3. Not a database, but this port is permanently allocated to the application specified.

Changing the Database Port

To use another port number for your database engine, you can use the configureDb utility from the command line to assign a new port ID. Use this utility only when:

- The Daemon Manager is down.
- The database is registered and enabled.

Caution Use this utility with caution. It is not intended to provide another interface for database registering, and should be called only after the subsystem is registered. For more information about the configureDb utility, see the "configureDb.pl" section on page 11-53.

To change the port number:

Step 1 At the command line, enter (all on one line):

/bin/perl $*NMSROOT*/**objects/db/conf/configureDb.pl action=upgrade** *dsn*=**`$dsn`** *portid*=**`$portid`**

Where:

- *$NMSROOT* is the directory in which the product was installed.
- *$dsn* is your database name.
- *$portid* is the new port ID you want to assign.

The utility updates all required files and Windows system registry entries.

Dynamically Allocating a Port ID

Database port IDs can be assigned dynamically during installation. Dynamic port ID assignments, however, can increase development and troubleshooting times dramatically.

For example, when you are working in a network environment, one machine may already have three installed databases and another only has two databases. When you add another database without specifying a port ID, the machines each assign the next available port ID number. This means your new database now has two different port ID numbers. If your network has more than just two machines, the problem is exponentially more difficult.

If the database does not come up on one machine, you cannot merely look at another machine and compare settings. This is because it is possible that one file has an old port ID or the port ID is missing. This happens most often during the initial development phase when most settings are manually entered.

The preferred method for allocating a port ID is to coordinate your port ID assignments with the CWCS database group by sending a request to the embu-db-interest alias.

Performing a Quick Integration

The CWCS database property and other files permit flexible configuration for a wide range of application data-storage needs. However, if you plan to follow the basic CWCS database configuration (whether or not you plan to include backup and restore), it is a relatively simple task to create a new database for your application and integrate it with CWCS.

The following procedure describes the minimum set of steps you must perform to get a new database set up and integrated. It summarizes all of the tasks described in detail in the topics under the "Setting Up a New Database" section on page 11-6.

Step 1 Create the following files (per the guidelines given in the "Creating the ODBC Database Definition File" section on page 11-7 and the "Creating the Backup Manifest Files" section on page 11-10):

$*NMSROOT*/databases/*dsn*/orig/odbc.tmplorig

$*NMSROOT*/databases/*dsn*/orig/odbc.tmpl

$*NMSROOT*/databases/*dsn*/orig/*dsn*.dborig

$*NMSROOT*/backup/manifest/*suite*/database/orig/*dsn*.txt

Where:

- *dsn* is the data source (database) name.
- *suite* is the name of your application or suite. Often, this is the same as the *dsn*. For example: For CWCS, the *suite* and *dsn* are both "cmf", but for Campus Manager, the *suite* is "campus" and the *dsn* is "ani".

For example, for a VMS database with "vms" as the *suite* and *dsn*, you would create the following files:

$*NMSROOT*`/databases/vms/orig/odbc.tmplorig`
$*NMSROOT*`/databases/vms/orig/odbc.tmpl`
$*NMSROOT*`/databases/vms/orig/vms.dborig`
$*NMSROOT*`/backup/manifest/vms/database/orig/vms.txt`

Step 2 Edit the database backup manifest file (*dsn*.txt) so that it points to your database path, as follows:

```
[dsn]
root=$ENV{NMSROOT}/databases/dsn/dsn.db
```

For example, for the VMS database, the database backup manifest contents would look like this:

```
[vms]
root=$ENV{NMSROOT}/databases/vms/vms.db
```

Step 3 Add any other databases to the database backup manifest file. For example, RME's rmeng.txt file looks like this:

```
[rme]
root=$ENV{NMSROOT}/databases/rmeng/rmeng.db
SyslogFirst=$ENV{NMSROOT}/databases/rmeng/SyslogFirst.db
SyslogSecond=$ENV{NMSROOT}/databases/rmeng/SyslogSecond.db
SyslogThird=$ENV{NMSROOT}/databases/rmeng/SyslogThird.db
```

Step 4 Run the following commands to install, register, and create a DbMonitor process for your database:

```
$NMSROOT/objects/db/conf/configureDb.pl action=install dsn=dsn
$NMSROOT/objects/db/conf/configureDb.pl action=reg dsn=dsn dmprefix=<ur_dmprefix>
```

For example, to install and register the VMS database, you would run these commands:

```
$NMSROOT/objects/db/conf/configureDb.pl action=install dsn=vms
$NMSROOT/objects/db/conf/configureDb.pl action=reg dsn=vms dmprefix=<ur_dmprefix>
```

Note If you do not want include your database in the CWCS backup and restore processes, you can simply delete the backup manifest file $*NMSROOT*/backup/manifest/*suite*/database/*dsn*.txt. This file is created from $*NMSROOT*/backup/manifest/*suite*/database/orig/*dsn*.txt when you run the configureDb.pl action=install/action=reg commands.

Using the Sybase Database

The following topics describe some of the typical database tasks you might perform:

For more information, refer to:

- The wrapper classes in dbservice2. DBClient.ExecuteUpdate and DBClient.ExecuteSelect are two database manipulation classes for the update, delete, query and create functions.
- *Sybase Adaptive Server Anywhere Reference Manual,* Chapter 4, "Database Administration Utilities."
- Sun's JDBC manual.

Before You Begin

When you create a Perl application that interfaces with the database APIs, follow these guidelines:

- Use "my" variables whenever possible—These variables have a true local scope; a local variable in Perl is not truly local as in the C language.
- Always specify "use strict"—This generates errors at compile time for any variables not properly defined in scope and any typos in variables masquerading as null values.
- Always run Perl using `perl -w {your script}`—This generates warnings at runtime for variables that have not been initialized prior to being used.
- Always check for errors from every DBI call.
- Do not confuse DBI with dbi.pl—dbi.pl contains code specific to the Resource Manager Essentials (RME) database only, and its routines are primarily used in reports. DBI is a general-purpose public domain API contained in DBI.pm.

For guidelines when creating JDBC or ODBC applications, refer to the third-party manuals for those interfaces.

Setting Up Your Environment

Before you can initialize your database or run any Sybase utilities, you must set up your environment:

- On Windows platforms, if you have installed CMF 1.2 or higher, your environment settings have already been created. The only settings you may need to update will be those that apply to any custom databases.
- On Solaris platforms, set the following environment variables:

```
setenv SATMP /tmp/.SQLAnywhere
setenv ASANY $NMSROOT/objects/db
setenv LD_LIBRARY_PATH $NMSROOT/objects/db/lib:$NMSROOT/lib
setenv PATH ${PATH}:$NMSROOT/bin:$NMSROOT/objects/db/bin
```

  where *$NMSROOT* is the directory in which the product was installed.

Initializing a New Database

Follow the procedure appropriate for your database platform.

On Windows Platforms

To initialize a database on Windows:

Step 1 Log in as a local administrator and open a DOS window.

Step 2 Enter:

```
cd $NMSROOT/objects/db/win32
```

where *$NMSROOT* is the directory in which the product was installed.

Step 3 Initialize the database:

```
dbinit -j [-b] [-c] [-p page-size] dbName.db
```

For example, the CWCS database is initialized using this command:

```
dbinit -j -b -c -p 1024 cmf.db
```

On Solaris Platforms

To initialize a database on Solaris:

Step 1 Log in as root.

Step 2 From the command line, set the environment variables (see the"Setting Up Your Environment" section on page 11-19).

Step 3 Enter:

```
cd $NMSROOT/objects/db/bin
```

where *$NMSROOT* is the directory in which the product was installed.

Step 4 Initialize the database:

```
dbinit -j [-b] [-c] [-p page-size] dbName.db
```

For example, the CWCS database is initialized using this command:

```
dbinit -j -c -b -p 1024 cmf.db
```

Related Topics

- For more information about the dbinit utility, see the "dbinit" section on page 11-54.
- See the *Sybase Adaptive Server Anywhere Reference Manual,* Chapter 4, "Database Administration Utilities," section "The Initialization Utility."

Creating a New Database

After you have initialized the database, make the following changes to this file:

Step 1 Change the user ID and password. The default database user ID is DBA and default password is SQL.

Warning **If you do not change these values, you will create a security hole.**

This procedure is described in the "Step 1: Change the User ID and Password" section on page 11-21.

Step 2 Create and populate DbVersion and DbVersionHistory.

This procedure is described in the "Step 2: Create and Populate DbVersion and DbVersionHistory" section on page 11-21.

Step 3 Copy the database file to the required directory locations.

This procedure is described in the "Step 3: Install the Database Files" section on page 11-23.

Related Topics

See the "Initializing a New Database" section on page 11-19.

Step 1: Change the User ID and Password

Use the changepwd.sql script to change the user ID and password. An SQL script that must run within the Sybase dbisqlc utility, changepwd.sql, changes only the password in the database, *not* the passwords in related configuration files such as obdc.ini and odbc.tmplorig. A copy of this script is included in the CodeSamples directory on the SDK CD.

Follow the procedure appropriate for your database platform.

On Windows Platforms

To change the user ID and password on Windows:

Step 1 Log in as a local administrator and open a DOS window.

Step 2 Enter (on one line):

```
dbisqlc -q -c "uid=DBA;pwd=SQL;dbf=newdb.db" read changepwd.sql newuid newpwd
```

On Solaris Platforms

To change the user ID and password on Solaris:

Step 1 Log in as root.

Step 2 Set the environment variables (see the "Setting Up Your Environment" section on page 11-19).

Step 3 Enter (on one line):

```
dbisqlc -q -c "uid=DBA;pwd=SQL;dbf=newdb.db" read changepwd.sql newuid newpwd
```

Step 2: Create and Populate DbVersion and DbVersionHistory

Database restore and upgrade utilities require some means of identifying installed device families and the current database version. Two tables track the schema version, dropins and incremental device support:

- DbVersion—This table contains the latest or most recent entries. The schema for this table is:

```
create table DbVersion (
    Componentvarchar(30) not null,
    subComponentvarchar(30) not null
    VersionStringvarchar(20)
    Descriptionvarchar(50)
    InstallDatetimestamp not null default current timestamp,
    Primary key (component, subComponent)
);
```

The information in DbVersion table is updated when IDs are added to the system.

- DbVersionHistory—This table contains a history of the changes to the database. The schema for this table is:

```
create table DbVersionHistory (
    Componentvarchar(40) not null,
    subComponentvarchar(30) not null
    VersionStringvarchar(20)
    Descriptionvarchar(50)
    InstallDatetimestamp,
    Primary key (component, subComponent, InstallDate)
);
```

The DbVersionHistory table is updated with an update/delete trigger on the DbVersion table without requiring applications to insert rows into the DbVersionHistory table.

The content of the DbVersion and DbVersionHistory tables is controlled by individual applications. Although it might not seem very useful when a new application is developed, you can use the Component, subComponent, and VersionString fields to distinguish databases from several application versions.

Follow the procedure appropriate for your platform.

On Windows Platforms

To create and populate the DbVersion and DbVersionHistory tables on Windows:

Step 1 Log in as a local administrator and open a DOS window.

Step 2 Enter:

```
dbisqlc -q -c "uid=newuid;pwd=newpwd;dbf=newdb.db" read dbversion.sql
```

On Solaris Platforms

To create and populate the DbVersion and DbVersionHistory tables on Solaris:

Step 1 Log in as root.

Step 2 Set the environment variables (see the "Setting Up Your Environment" section on page 11-19).

Step 3 Enter:

```
dbisqlc -q -c "uid=newuid;pwd=newpwd;dbf=newdb.db" read dbversion.sql
```

Step 3: Install the Database Files

After you have initialized the database, changed the user ID and password, populated the DbVersion and DbVersionHistory tables, you are ready to install the database.

To install the database files:

Step 1 Copy the database file to the database directory. This directory contains the working database.

- Database file name: *dsn*.db
- Location: *$NMSROOT*/databases/*suite*

  where *$NMSROOT* is the directory in which the product was installed.
- Example: *$NMSROOT*/databases/cmf/cmf.db

Step 2 Copy the database file to the /orig database directory. This directory contains a copy of the original database, which can be used to reinitialize a corrupted database.

- Database file name: *dsn*.dborig
- Location: *$NMSROOT*/databases/*suite*/orig
- Example: *$NMSROOT*/databases/cmf/orig/cmf.dborig

Note If you are an NMTG developer, just copy the *dsn*.dborig file to the backup (/orig) directory. The automated build processes use this file to create the corresponding .db file on the target system.

Also be sure you have set up the backup manifest files, as explained in "Creating the Backup Manifest Files" section on page 11-10.

Step 3 Once you have set up the database files, you can use configureDb.pl to installand register database:

- To install the database, run the following command

  ```
  perl configureDb.pl action=install <dsn=database>
  ```

  This command copies the database file from the /orig directory to the runtime directory.
- To register the database, run the following command:

  ```
  perl configureDb.pl action=reg <dsn=database> <dmprefix=prefix>
  ```

  This command registers the database with the Daemon Manager, including populating the .odbc.ini or Windows odbc registry, updating dmgtd.conf or the Windows services registry, and updating the DBServer.properties file.

Step 4 If needed, you can use the same script to re-install, uninstall, register or unregister the database:

- To reinstall the database, run the following command

  ```
  perl configureDb.pl action=install <dsn=database>
  ```
- To uninstall the database, run the following command

  ```
  perl configureDb.pl action=uninstall <dsn=database>
  ```

  This command removes the database file from the runtime directory.
- To re-register the database, run the following command:

  ```
  perl configureDb.pl action=reg <dsn=database> <dmprefix=prefix>
  ```
- To unregister the database. run the following command:

```
perl configureDb.pl action=unreg <dsn=database> <dmprefix=prefix>
```

This command unregisters the database with Daemon Manager.

Related Topics

See the:

- "Restoring a Corrupt Database" section on page 12-18
- "Reinitializing a Database" section on page 11-35.

Updating the Database Password

Use the dbpasswd.pl utility at runtime to change the database password. This utility will replace the old password in the odbc.tmpl file and populate the new password to:

- The .odbc.ini file
- The Windows registry
- The DbServer.properties file
- If the entry already exists, any customer-specified Java property file

Note that dbpasswd.pl validates submitted passwords. Valid passwords must:

- Have a minimum of five and a maximum of 128 characters.
- Use alphanumeric characters (a-z, A-Z, 0-9) only. No special characters (e.g., #, $, %) or spaces are allowed.
- Not have a number as the first character.

Follow the procedure appropriate for your platform.

On Windows Platforms

To change the password on Windows:

Step 1 Log in as a local administrator and open a DOS window.

Step 2 Stop the Daemon Manager by entering:

```
net stop crmdmgtd
```

Step 3 Run the dbpasswd.pl utility (see the "dbpasswd.pl" section on page 11-57).

Step 4 Enter the new password.

Step 5 Verify the new password.

Step 6 Start the Daemon Manager by entering:

```
net start crmdmgtd
```

On Solaris Platforms

To change the password on Solaris:

Step 1 Log in as root.

Step 2 Set the environment variables (see the "Setting Up Your Environment" section on page 11-19).

Step 3 Stop the Daemon Manager by entering:

```
/etc/init.d/dmgtd stop
```

Step 4 Run the dbpasswd.pl utility (see the "dbpasswd.pl" section on page 11-57).

Step 5 Enter the new password.

Step 6 Verify the new password.

Step 7 Start the Daemon Manager by entering:

```
/etc/init.d/dmgtd start
```

Starting and Stopping Database Engines

The SqlAnywhere embedded API does not provide database auto-start and auto-stop capabilities. Therefore, when Perl DBI applications are run on Solaris platforms, the database must be started explicitly.

The following topics explain how to start and stop a database engine from a Perl application:

- Starting a Database Engine
- Stopping a Database Engine

Starting a Database Engine

You can start the database engine from the CWCS Desktop, on Windows or on Solaris. Follow the procedure appropriate for your platform, below.

From the CWCS Desktop

Typically, you would start the database engine (or any other process) from the CWCS desktop:

Step 1 Select **Server Configuration > Administration > Process Management > Start Process**.

The Start Process dialog appears.

Step 2 Select the name of the database engine from the list box (for example, CmfDbEngine).

Step 3 Click **Finish**.

On Windows Platforms

When creating a new database using a Windows machine, you might have to change the registry entries before starting the database engine. To change the registry entries and start the database engine:

Step 1 Log in as local administrator.

Step 2 You can use these dialogs in the Control Panel window to add or modify registry entries:

- To start, stop, and configure services (the database engine is registered as a service), select **Start > Settings > Control Panel > Services**.
- To maintain the ODBC data sources and drivers, select **Start > Settings > Control Panel > ODBC Data Sources > System DSN**.

Step 3 Or, you can set the Windows registry values directly. The registry entries are located in two places:

- The registry entry used by Windows systems is at:

  `My Computer/HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/ Services/CmfDbEngine`
- The registry entries used by the Daemon Manager are located at:

  `My Computer/HKEY_LOCAL_MACHINE/SOFTWARE/Cisco/ResourceManager/ CurrentVersion/Daemons/*`

  There is one registry entry for each application that uses the Daemon Manager.

Step 4 To start the database engine, open a DOS window and enter (all on one line):

```
-x tcpop{HOST=localhost;DOBROADCAST=NO;ServerPort=portID} -m -ti 0 -gm 100 -c 8M -n
yourdbEng $NMSROOT\databases\yourdb\yourdb.db -n yourdbDb
```

where:

- *$NMSROOT* is the directory in which the product was installed.
- *portID* is the port number assigned to your database.

For example, to start the CWCS database engine, enter:

```
-x tcpop{HOST=localhost;DOBROADCAST=NO;ServerPort=43441} -m -ti 0 -gm 100 -c 8M -n cmfEng
$NMSROOT\databases\cmf\cmf.db -n cmfDb
```

On Solaris Platforms

When creating a new database on a Solaris machine, you can use this procedure during the prototyping phase to start the database engine:

Step 1 Log in as root.

Step 2 Set the SATMP, SQLANY, and LD_LIBRARY_PATH environment variables (see the "Setting Up Your Environment" section on page 11-19).

Step 3 To start the database engine process, enter (all on one line):

```
$NMSROOT/objects/db/bin/dvsrv9 -x tcpip{HOST=localhost;DOBROADCAST=NO;ServerPort=portID} -q -s local0 -m -ti
0 -gm 100 -gc 5 -c 8M -ht -gss 9900 -n yourdbEng $NMSROOT/databases/yourdb/yourdb.db -n yourdbDb
```

where:

- *$NMSROOT* is the directory in which the product was installed
- *yourdb* is the name of your database engine.
- *portID* is the port number assigned to your database.

For example, to start the CWCS database, enter (all on one line):

```
$NMSROOT/objects/db/bin/dvsrv9 -x tcpip{HOST=localhost;DOBROADCAST=NO;ServerPort=43441} -q -s local0 -m -ti 0
-gm 100 -gc 5 -c 8M -ht -gss 9900 -n cmfEng $NMSROOT/databases/cmf/cmf.db -n cmfDb
```

Note The engine name and database name must be used in the connection string to connect to the specific database.

The first transaction against the database creates the transaction log.

Related Topics

See the "Connection Strings" section on page 11-5.

Stopping a Database Engine

You can stop a database engine from the CWCS Desktop, on Windows or on Solaris. Follow the procedure appropriate for your platform, below.

From the CWCS Desktop

To stop the database engine from the CWCS desktop:

Step 1 Select **Server Configuration > Administration > Process Management > Stop Process**.

The Stop Process dialog appears.

Step 2 Select the name of the database engine from the list box (for example, CmfDbEngine).

Step 3 Click **Finish**.

On Windows Platforms

To stop the database engine on Windows:

Step 1 Log in as a local administrator and open a DOS window.

Step 2 To stop the database engine process, use one of these options:

- If the SqlAnywhere Console is present, click **Shutdown**.
- If the database is running as a service, use the Windows service control manager. To access this dialog, select **Start > Control Panel > Services**.

Caution DO NOT use the Windows Task Manager unless the process is hanging.

On Solaris Platforms

To stop the database engine on Solaris:

Step 1 Log in as root.

Step 2 Set the SATMP, SQLANY, and LD_LIBRARY_PATH environment variables (see the "Setting Up Your Environment" section on page 11-19).

Step 3 Stop the database engine process:

```
$SQLANY/bin/dbstop -s uid=username;pwd=password;eng=dbEngineName;dbn=dbName
```

Caution Do *not* use the kill command unless the process is hanging.

Creating and Closing Database Connections

Once the database engine is started, there are separate procedures for connecting to and disconnecting from the database. Connection parameters are specified using connection strings. For more information about connection strings, see the "Connection Strings" section on page 11-5.

The following topics describe how to connect to a database:

- Connecting to a Database
- Closing a Database Connection

Connecting to a Database

You can connect to a database from a Java application, C or C++ application, or Perl script. Follow the procedure appropriate for your application, below.

In a Java Application

- Use this call to load the JDBC 5.5 driver, com.sybase.jdbc2.jdbc.SybDriver:

  ```
  Class.forName(com.sybase.jbdc2.jdbc.SybDriver)
  ```

- The JDBC Sybase component jConnect uses a URL-style syntax:

  ```
  jdbc:sybase:Tds:localhost:42341?SERVICENAME=dbName
  ```
  where *dbName* is the name of the database.

In a C or C++ Application

SQLConnect is the simplest ODBC connection function. It accepts the following connection strings format:

```
"uid=xxx;pwd=yyy;dsn=ddd"
```

SQLDriverConnect can be used to replace SQLConnect. It supports data sources that require more connection information than the arguments in SQLConnect, and data sources that are not defined in the system information. One advantage of using SQLDriverConnect is that we can provide "con=myConnectionName" as part of the connection string to identify the connection. This is useful for debugging and performance analysis.

Instead of using either of these functions, however, your applications can use the SQLSecureConnect connection API. SQLSecureConnect acts as a wrapper around SQLDriverConnect to accept and process connection strings that have an encrypted username and password (the custom tokens CWEUID, CWEPWD). It reads the .odbc.ini or registry entry to get either the encrypted or plain text user information. It decodes this information and passes the user ID and password to SQLDriverConnect.

Note If encryption is disabled (ENCRYPTION=NO), you can continue to use SQLDriverConnect. SQLSecureConnect, however, supports both encrypted and plain text user IDs and passwords.

If the connection string contains the user ID and password, SQLSecureConnect passes the connection string directly to SQLDriverConnect. If the user ID and password are missing, DSN must be present in the connection string. Other parameters in the connection string are carried over.

SQLSecureConnect uses the following syntax:

```
#include "dbencrypt.h"
SQLRETURN SQLSecureConnect(
    SQLHDBC            hdbc,
    SQLHWND            hwnd,
    SQLCHAR ODBCFAR * szConnStrIn,
    SQLSMALLINT        cbConnStrIn,
    SQLCHAR ODBCFAR * szConnStrOut,
    SQLSMALLINT        cbConnStrOutMax,
    SQLSMALLINT ODBCFAR * pcbConnStrOut,
    SQLUSMALLINT       fDriverCompletion)
```

(The syntax and semantics of this API are identical to SQLDriverConnect. See the SQLDriverConnect documentation for details.)

For example, the following SQL statements allocate memory for an environment handle, initialize the ODBC call level interface, allocate memory for a connection handle, and use SQLSecureConnect to connect to the database:

```
SQLHENV     henv;
SQLHDBC     hdbc;
SQLHSTMT    hstmt;
SQLRETURN   retcode;
char *      newconn = (char *)malloc(MAX_DSN_LEN);

/*Allocate environment handle */
retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);

if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
   /* Set the ODBC version environment attribute */
   retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION,
             (void*)SQL_OV_ODBC3, 0);

      if (retcode == SQL_SUCCESS ||
       retcode == SQL_SUCCESS_WITH_INFO) {
      /* Allocate connection handle */
      retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);

      if (retcode == SQL_SUCCESS ||
          retcode == SQL_SUCCESS_WITH_INFO) {
         /* Set login timeout to 5 seconds. */
         SQLSetConnectAttr(hdbc, (void*)SQL_LOGIN_TIMEOUT, 5, 0);

         /* Connect to data source */
         retcode = SQLSecureConnect(
           hdbc,
           0,
```

```
                (SQLCHAR*)"dsn=cmf",
                SQL_NTS,
                (SQLCHAR*)newconn,
                MAX_DSN_LEN,
                &len,
                SQL_DRIVER_NOPROMPT);

            if (retcode == SQL_SUCCESS
                || retcode == SQL_SUCCESS_WITH_INFO){
                /* Allocate statement handle */
                retcode = SQLAllocHandle(SQL_HANDLE_STMT,
                          hdbc, &hstmt);

                if (retcode == SQL_SUCCESS
                    || retcode == SQL_SUCCESS_WITH_INFO) {
                /* Process data */
                    ;
                    ;
                    ;

                   SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
              }
                SQLDisconnect(hdbc);
        }
          SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
    }
 }
                SQLFreeHandle(SQL_HANDLE_ENV, henv);
…
```

In a Perl Application

Use SqlAnywhere strings (such as Isql, dbstop, and dbvalid):

```
"uid=xxx;pwd=yyy;eng=engName;dbn=dbname;"
```

For example, the following Perl code fragment connects to the CWCS database:

```
use DBI;
my $dbh = DBI->connect ("','uid=DBA;pwd=SQL;dsn=cmf','','Sqlany');
```

In this example, DBI will resolve the user ID and password in both encryption modes:

```
use DBI;
my $dbh = DBI->connect ("','dsn=cmf','','Sqlany');
```

The last argument is the DBD driver to be loaded. This parameter is ignored on Windows.

Closing a Database Connection

You can close a database connection from a Java application, C or C++ application, or Perl script. Follow the procedure appropriate for your application.

In a Java Application

Use `dbc.close`.

In a C or C++ Application

The following SQL statements close the database connection, release the connection handle, free all memory allocated for the handle, close the ODBC driver, and release all memory associated with the driver:

```
SQLDisconnect (dbc);
SQLFreeConnect (dbc);
SQLFreeEnv (env);
```

In a Perl Application

Use `$dbh->disconnect();`

Examining the Contents of a Database

You can access your data using the Sybase utility, dbisqlc, or the dbreader utility:

- The Sybase utility, dbisqlc, is more difficult to use but does not require installing CWCS. For information about using dbisqlc, refer to the Sybase documentation.
- The dbreader utility requires that you first install CWCS.

The following topics describe how to use the dbreader utility to access your database:

- Creating a DSN
- Accessing Your Data

Creating a DSN

The dbreader utility uses ODBC to access your database. Before you can use dbreader to access your data, you must create a data source name (DSN) from the ODBC Manager using the steps below.

Note If you have CWCS installed on your desktop, the DSN was created at install time; you can skip this procedure.

Step 1 Launch the Windows Control Panel.

Step 2 Click **ODBC Data Source**.

Step 3 Click **tab-System DSN**.

The name of your database engine should not appear.

Step 4 Click **Add**.

Step 5 Click **CiscoWorks Embedded Database**.

Step 6 Click **Finish**.

The ODBC Configuration for Adaptive Server Anywhere dialog box appears.

Step 7 Enter the name of your database engine in the **Data Source Name** field.

Step 8 Click **Login**.

Step 9 Enter your user ID and password.

Step 10 Click the **Database** tab.

Step 11 Complete the following fields:

- Server name: *enginename*Eng
- Database name: *enginename*Db
- Database file: *$NMSROOT*/databases/*enginename*/*enginename*.db

Step 12 Click **OK**.

Accessing Your Data

After you have created a data source name, you can access the contents of your database:

Step 1 Start CiscoWorks and log in with the appropriate user ID and password.

Step 2 In your browser's address or location bar, enter the following URL:

`http://hostname:portid/dbreader/dbreader.html`

The Ad-hoc Retrieval of Database dialog box appears.

Step 3 Enter the database user ID and password. (Do not confuse them with the CWCS admin user ID and password.)

Step 4 Enter the database name (for example, cmf, rme, or ani).

Step 5 To read the database, perform one of the following options:

- Leave **SQL statement to execute** blank, then select **Get Database Table**s. This option retrieves all the Cisco tables in that database. Click on the table name to drill down to the table data.
- Enter SQL statements in **SQL statement to execute**, then select **Execute SQL Statemen**t.

Backing Up Your Database

CWCS provides two different backup options:

- Back up now

  To run a backup immediately, select **Server Configuration > Admin > Backup > Back Up Data Now**.

- Scheduled backup

  To schedule a backup, select **Server Configuration > Admin > Backup > Schedule Backup**.

When you use either option, all installed application groups are backed up; you are not allowed to select specific application groups.

For more information about using these dialogs, click **Help** at the bottom of the dialog box.

You can also use the backup.pl script to back up the database and all installed application groups. The backup.pl script also uses the backup manifest files to determine which files and directories to back up. For more information on this script, see the "backup.pl" section on page 12-10.

Related Topics

See the:

- "Using the Sybase Database" section on page 11-18.
- "Creating the Backup Manifest Files" section on page 11-10.

Debugging and Troubleshooting the Database

The following topics describe how to use the database utilities to troubleshoot database problems:

- Managing Database Log Files
- Ensuring Sufficient Temporary Space
- Optimizing Query Processing
- Verifying a Database
- Reinitializing a Database
- Cleaning Up Other Application Files

For information about backing up and restoring a database, see Chapter 12, "Using Backup and Restore."

Managing Database Log Files

There are two database files that, while transparent to the user, may require attention from the engineer during database development:

- Transaction log—Contains a record of the database transactions for this session.

  This file is erased whenever the database engine shutdown process is successful and the database server is running with the -m option. When an exception occurs, however, this file remains. You can use the contents of this file to help troubleshoot database problems.

- Temporary files—Used for intermediate result sets, and stored in the *$SATMP* or *%TMPDIR%* directory.

  These files are erased during typical database operations. They do require a certain amount of swap space, however, which may be overrun during debugging and testing cycles. When exceptions occur during database development and testing, consider cleaning these directories *only* if you are certain that CWCS is the only application using them.

Ensuring Sufficient Temporary Space

The Sybase version supplied with this release of CWCS introduced the `temp_space_limit_check` option. When `temp_space_limit_check=on` and there is insufficient temp space for a database connection, the connection will fail. If this option is set `off` and there is insufficient temp space for a connection, the database server will crash.

This option is set `on` for the CWCS database by default. To set this option on for your database, run the SQL statement `set option public.temp_space_limit_check='on'`. To check the option setting, run `select connection_property('temp_space_limit_check')`.

Optimizing Query Processing

The Sybase option `OPTIMIZATION_GOAL` controls how query processing is optimized. It has two allowed values:

- `first-row`: Returns the first row as quickly as possible.
- `all-rows`: Minimizes the cost of returning the complete result set

In previous versions of CWCS, the default setting was `first-row`. In this release, the default setting is `all-rows`.

If you create your CWCS database using older Sybase binaries and rebuilding them to the Sybase 9.x format, you will retain `first-row` as the default OPTIMIZATION_GOAL setting. If you create your database file using Sybase 9.x binaries, the default setting will be `all-rows`.

We recommend that you:

1. Determine which OPTIMIZATION_GOAL setting your application is using. To do so, run the SQL statement `select connection_property('OPTIMIZATION_GOAL')`.
2. Evaluate the preformance of the modules in your application under that setting.
3. Change to the opposite setting and evaluate its performance. To change the setting, run `set option public.OPTIMIZATION_GOAL='all-rows'` or `set option public.OPTIMIZATION_GOAL='first-row'`.
4. Based on your test results, choose whether to change this option setting for your database, as needed.

You can find additional information in the following Sybase documentation:

- *Sybase Adaptive Server Anywhere Database Administration Guide*: See the section "Optimization_Goal option" on page 613.
- *Sybase Adaptive Server Anywhere SQL Reference*: See the "FROM clause" section on page 445. This section discusses the FASTFIRSTROW table hint.

Verifying a Database

Use the dbvalid script to validate the integrity of a database. To run dbvalid, enter:

```
cd $NMSROOT/databases/dbfile
dbvalid -c uid=$uid;pwd=$pwd;dbf=dbfile
```

where:

- *$NMSROOT* is the directory in which the product was installed.
- *$uid* and *$pwd* are the user ID and password for your database.
- *dbfile* is the database engine name.

The dbvalid utility returns one of the following responses:

- No problem.
- One or more tables have been corrupted.

  See the "Restoring a Corrupt Database" section on page 12-18.
- Cannot bring up the database engine

  If you have a log file, try this:

  ```
  rm -f rme.log
  dbeng9 -f rme.db
  ```

  This forces the RME database to start up without a transaction log file. Then call dbvalid to revalidate the database.

  You can also try running the configureDb/dbvalid command without passing the entire connection string as an argument.

Related Topics

See the:

- "Types of Database Servers" section on page 11-3.
- "dbvalid" section on page 11-60.
- "configureDb.pl" section on page 11-53

Reinitializing a Database

If the data in a database is totally corrupted or not important, you can copy a clean database from the orig directoryover the existing database. You will lose the application data. However, this can be useful if your application database is the only problem the application is having, since you will not need to re-install the application.

When run, dbRestoreOrig.pl prompts for user confirmation, warning that all data will be lost. If your application is using this script internally and you do not want this prompt to appear, add the `opt=y` argument to the dbRestoreOrig.pl call. For example:

```
$NMSROOT/bin/dbRestoreOrig.pl dsn=cmf dmprefix=Cmf opt=y
```

You can also use the call to dbRestoreOrig.pl to clean up application configuration and other data stored in the file system. For details, see the "Cleaning Up Other Application Files" section on page 11-36.

If the data in the database is important, and you have been using a backup framework for regular maintenance, you can use a the corresponding restore framework to recover it. For information about the restoring a database from a regularly made backup, see the "Using CWCS Restore" section on page 12-4.

On Windows Platforms

To reinitialize the database on Windows:

Step 1 Stop the Daemon Manager by entering:

```
net stop crmdmgtd
```

Step 2 Enter on one line:

```
NMSROOT/bin/dbRestoreOrig.pl dsn=dsn dmprefix=dmprefix
```

where:

- *NMSROOT* is the directory in which the product was installed.
- *dbn* is your database name.
- *dmprefix* is the prefix registered for this database in the CWCS Daemon Manager.

On Solaris Platforms

To reinitialize the database on Solaris:

Step 1 Stop the Daemon Manager by entering:

```
/etc/init.d/dmgtd stop
```

Step 2 Enter:

```
NMSROOT/bin/dbRestoreOrig.pl dsn=$dbn dmprefix=$dmprefix
```

where:

- *NMSROOT* is the directory in which the product was installed.
- *dsn* is your database name.
- *dmprefix* is the prefix registered for this database in the CWCS Daemon Manager.

Cleaning Up Other Application Files

The script dbRestoreOrig.pl (see the "dbRestoreOrig" section on page 11-59) lets you reinitialize a corrupt database without touching the application's file system. For example, if MyApp has configuration files, logs, archives, or images stored outside of the database, all of these files will still be in the same locations after you reinitialize MyApp's database using dbRestoreOrig.pl.

If you want to eliminate these file system leftovers at the same time you reinitialize the database, you can create a script to do so and get dbRestoreOrig.pl to execute it, as follows:

1. Define a script to be executed after the database is reinitialized.

   The script must implement a Cleanup::doCleanup() function that meets your requirements, and should return zero for success and a non-zero value for failure (see Example 11-1).

2. Store the script as Cleanup.pm in*$NMSROOT*/databases/*dsn*/scripts/(where *dsn* is your application's data source name).
3. Run dbRestoreOrig.pl with the appropriate *dsn*. The utility will restore the orig database.
4. Before exiting, dbRestoreOrig.pl will look for the Cleanup.pm file. If the utility finds this file, it will execute the Cleanup.pm file's Cleanup::doCleanup() as its last act.

Example 11-1 Cleanup.pm Script

```
sub doCleanup{
LogError( "Inside CMF Cleanup\n") ;
# add your code here
# return 0 for success and non-zero for failure
return 0 ;
}
1
```

Database API Command Reference

The following topics describe how to integrate your database into CWCS:

Enabling the CWCS Database Engine

Your custom database runs independently from the CWCS database engine. When using CWCS, however, you will also be using the CWCS database engine.

The database engine is part of the Common Services, but it is not enabled by default. You must enable it before you can use it. If your application requires services from the CWCS database engine, remember to register for this service at installation.

For instructions, refer to the "Registering for CWCS Services" section on page 5-4. If you prefer to request services after installation, refer to the "Enabling New Service Bundles from the Command Line" section on page 5-5.

Note If you have *installed* the CWCS database but have not *enabled* it, you will *not* have access to any ODBC or JDBC commands. Perl also uses ODBC commands, so it will not work either.

Compiling and Running a Database

When you are developing new applications, remember that:

- Client and client-server Java classes are stored in *$NMSROOT*/www/classpath
- Server-only Java classes are stored in *$NMSROOT*/lib/classpath

Use the procedure appropriate for your platform.

On Windows Platforms

To compile an application that accesses a database on Windows, enter:

```
$dev:\enm_jdk\jdk1.2.1\NT\bin\javac -classpath
$NMSROOT\lib\classpath;$NMSROOT\www\classpath appname.java
```

where:

- *$dev* is the location of the JDK.
- *$NMSROOT* is the directory in which the product is installed.
- *appname* is the name of your application

To run an application that accesses a database on Windows, enter:

```
$NMSROOT\lib\jre2\bin\java -classpath
.;$NMSROOT\www\classpath;$NMSROOT\lib\classpath test
```

On Solaris Platforms

To compile an application that accesses a database on Solaris, enter (all on one line):

```
java -classpath .:$NMSROOT/www/classpath:$NMSROOT/lib/classpath testappContent-Type:
text/plain; charset="us-ascii"
Content-Disposition: attachment; filename="compile.sh"
```

where *$NMSROOT* is the directory in which the product was installed.

To run an application that accesses a database on Solaris, enter (all on one line):

```
javac -classpath
.:$NMSROOT/lib/classpath:$NMSROOT/www/classpath:$NMSROOT/lib/jre/lib/rt.jar testapp.java
```

Related Topics

See:

- Chapter 3, "Understanding the CWCS Directory Structure."
- The "Understanding the Java Application Launch Process" section on page 4-1.

Code Samples

The following topics contain assorted code samples that illustrate various database tasks:

- Using Java to Read a Database
- Using ODBC to Access a Table
- Using Perl to Access a Database

Using Java to Read a Database

The following example shows how to use these JDBC API wrappers from the dbservice2 package:

- DBClient—Connect a database engine and perform database-level and SQL statement-level operations.
- DBResult—Examine the query results.
- DBException—Handle exception cases.

For more information about these wrappers, see the "Using JDBC API Wrappers" section on page 11-41.

Example 11-2 Using Java to Read a Database

```
import java.io.*;
import java.util.*;
import com.cisco.nm.cmf.dbservice2.*;
import java.sql.*;


public class testapp {
   static DBClient dbc = null;
   static DBResult dbr1 = null;    // data base results from select and update operations
   static Vector row;

   public static void main(String args[]) {
       System.out.println("test application for DB");
```

```
    try {
        dbc = new DBClient("testapp", "rme", 1);
        //appName, debugLevel (1 means turn on debug messages)
        // gets the URL from the property file and connects to the database.
        // Also creates an SQL statement handle.
    } catch (DBException ex) {
        System.out.println("Error Message: " + ex.getMessage());
        if (ex.isSqlError()) {
            System.out.println("SQL ERROR CODE: " + ex.getSqlErrorCode());
            System.out.println("SQL STATE: " + ex.getSqlErrorCode());
        }
        System.out.println("test application failed at: ");
        ex.printStackTrace();
        return;
   } catch (ClassNotFoundException ex) {
        System.out.println("Error Message: " + ex.getMessage());
        System.out.println("test application failed at: ");
        ex.printStackTrace();
        return;
   }
try {
        dbr1 = dbc.executeSelect("select * from dbversion");
        if (dbr1 == null) {
            System.out.println("no data");
        } else {
            dbr1.toFirst();

            int count = 0;
            String Component;
            String SubComponent;
            String VersionString;
            String Description;
            String InstallDate;
            while ((row = dbr1.getRow() ) != null) {  //iterate through all the test results
                Component = row.elementAt(0).toString();
                SubComponent = row.elementAt(1).toString();
                VersionString = row.elementAt(2).toString();
                Description = row.elementAt(3).toString();
                InstallDate = row.elementAt(4).toString();
                System.out.println(Component + " " + SubComponent + " "
                                + VersionString + " " + Description + " "
                                + InstallDate + "\n\n");
                count++;
            }
        }
    } catch (DBException ex) {
        System.out.println("Error Message: " + ex.getMessage());
        if (ex.isSqlError()) {
            System.out.println("SQL ERROR CODE: " + ex.getSqlErrorCode());
            System.out.println("SQL STATE: " + ex.getSqlErrorCode());
        }
        System.out.println("test application failed at: ");
        ex.printStackTrace();
    }
    try {
        dbc.close();
    } catch (DBException ex) {
        System.out.println("Error Message: " + ex.getMessage());
        if (ex.isSqlError()) {
            System.out.println("SQL ERROR CODE: " + ex.getSqlErrorCode());
            System.out.println("SQL STATE: " + ex.getSqlErrorCode());
        }
        System.out.println("test application failed at: ");
```

```
            ex.printStackTrace();
        }
        System.out.println("test application finished. ");
    }
}
```

Using ODBC to Access a Table

The following code fragment shows how to use ODBC calls to access a device group table.

Example 11-3 Using ODBC Calls to Access a Device Group Table

```
strcpy((char *)connStr, "uid=DBA;pwd=SQL;dsn=rme";
if (SQLAllocEnv( &henv ) != SQL_SUCCESS ) { /* handle error */
if (SQLAllocConnect( henv, &hdbc ) != SQL_SUCCESS) {
    print_error(); exit(1); }
rc = SQLSecureConnect( hdbc, 0, connStr, SQL_NTS,
        outBug, 256, &outBufLen, SQL_DRIVER_NOPROMPT);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) { ... }

/* read and print from dev_group table */
rc = SQLAllocStmt(hdbc, &hstmt);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) { ... }
sqlStr = (UCHAR *) "select * from dev_group";
if (SQLExecDirect (hstmt, sqlStr, SQL_NTS) != SQL_SUCCESS) { ... }
SQLNumResultCols(hstmt, &nresultcols);

for (i = 0; i < nresultcols; i++) {
    SQLDescribeCol(hstmt, i + 1, colname, (SWORD) sizeof(colname),
...
}
```

Using Perl to Access a Database

The following code fragment shows how to use Perl to fetch data from the database.

Example 11-4 Using Perl to Access a Database

```
use CRM;
use lib "$ENV{NMSROOT}/lib/perl/install";
use InstallUtility;
use lib "$ENV{NMSROOT}/lib/perl/db";
use Cisco::DbUtils;
use dbinternal;

    my $dbh;
    my $cur;
    my $dsn = "cmf";
    my @data;
    my $Component, $SubComponent, $VersionString, $Description, $InstallDate;
    $dbh = &dbinternal::connect("dsn=$dsn");
    if ( !defined($dbh) ) {
```

```
        print "\n\n ERROR testdb.pl:Couldn't connect to the database $dsn\n";
        return 0;
    }

    $cur = $dbh->prepare("SELECT * from DbVersion");
    if($cur) {
        $cur->execute();
        while (@data = $cur->fetchrow) {
            ($Component,$SubComponent,$VersionString,$Description,$InstallDate) = @data;
            print "$Component,$SubComponent,$VersionString,$Description,$InstallDate \n";
            $cnt++;
        }
        $cur->finish;
    } else {
        print STDERR "Unable to select items from DB :$DBI::errstr";
    }

    $dbh->disconnect;
1;
```

Using JDBC API Wrappers

The dbservice2 package contains six classes. You can write a JDBC application by referencing three of them:

- DBClient allows you to connect to a database engine and perform database-level and SQL statement-level operations.
- DBResult lets you examine the query results.
- DBException handles exception cases.

We strongly recommend that you use the DBUtil.getDBConnection(java.lang.String dbName) API to get a java.sql.Connection object. While it is still supported, the existing DBConnection class (see the "DBConnection" section on page 11-44) is being deprecated.

DBClient

This topic describes the DBClient constructor and its public methods.

DBClient Constructor Summary

```
public DBClient (String dbName) throws ClassNotFoundException,
DBException.
```

Same as ("noAppName", dbName, DBUtil.getDBServiceProperties(), 0);

This class is a wrapper for the java.sql.Connection and java.sql.Statement classes. It extracts JDBC connection information from the DBServer.properties file and creates a connection. It also creates java.sql.Statement objects so the application can perform the java.sql.Statement operation.

Table 11-3 DBClient Method Summary

Returns	Syntax and Description
`synchronized void`	`close () throws DBException` A wrapper for Connection.close. It also closes the statement it owns.
`synchronized void`	`commit () throws DBException` A wrapper for Connection.commit.
`void`	`createPreparedStatement (String sql) throws DBException` Creates a PrepareStatement object.
`void`	`disableRetryOnLockedRow ()` Disallows retry on locked row.
`void`	`enableRetryOnLockedRow (Integer maxTryCount, Long tryInterval)` Allows retry on locked row. If RetryOnLockedRow is true, an SQL statement will try up to maxTryCount or until the statement is executed successfully.
`synchronized DBResult`	`executeSelect (String sql) throws DBException` A wrapper for Statement.executeQuery. Tries to rebuild a connection if the old connection is dropped. If it fails, it retries up to MaxTryCount if RetryOnLockedRow is true.
`synchronized DBResult`	`executeUpdate (String sql) throws DBException` A wrapper for Statement.executeUpdate. Tries to rebuild a connection if the old connection is dropped. If it fails, it retries up to MaxTryCount if RetryOnLockedRow is true.
`synchronized int`	`getDebugLevel () throws DBException` Gets the debug level.
`void`	`reOpenStaleConnection (Boolean reOpenStaleConnection) throws DBException` Reconnects to a database if the connection drops for any reason by setting the reOpenStaleConnection flag to true.
`synchronized void`	`rollback () throws DBException` A wrapper for Connection.rollback.
`synchronized void`	`setAutoCommit (boolean autoCommit) throws DBException` A wrapper for Connection.setAutoCommit.
`synchronized void`	`setDebugLevel (int debugLevel) throws DBException` Sets the debug level.
`synchronized void`	`setTransactionIsolation (int ti) throws DBException` A wrapper for Connection.setTransactionIsolation.
`synchronized void`	`setTransactionIsolation (Integer ti) throws DBException` A wrapper for Connection.setTransactionIsolation.

DBResult

The DBResult class is a wrapper for ResultSet. This topic provides a summary of its constructors and public methods.

DBResult Constructor Summaries

`public DBResult (int rowsAffected)` A wrapper for ResultSet.

`public DBResult (ResultSet resultSet) throws SQLException, DBException` Copies and stores data in private storage.
`public DBResult (ResultSet resultSet, int storageType) throws SQLException, DBException` Copies and stores data in private storage.

Table 11-4 DBResult Method Summary

Returns	Syntax and Description
`int`	`getAffectedRows() throws DBException` Returns the number of updated records after an update operation.
`int`	`getColCount() throws DBException` Returns column count.
`int`	`getColDisplaySize (int col) throws DBException` Given a column position, returns a column display size by fetching column metadata stored in this object.
`int`	`getColType (int col) throws DBException` Given a column position, returns a column type by fetching column metadata stored in this object.
`Vector`	`getResultVector() throws DBException` Returns all records.
`synchronized Vector`	`getRow()` Moves the cursor down and fetches the next record.
`int`	`getRowCount() throws DBException` Returns the number of records after a fetch operation.
`void`	`toFirst()` Moves the cursor to the first row in result set.
`void`	`toPrintWriter (PrintWriter pw) throws IOException` Sends formatted result set to an output stream.
`String`	`toString ()` Converts the object to string with “\t” as element delimiter and “\r\n” as line delimiter.
`String`	`toString (String elementDelimiter, String lineDelimiter, String beginDelimiter, String endDelimiter)` Converts the object to string-supplied delimiters.

Class DBUtil

The DBUtil class loads the JDBC property file. Table 11-5 contains a summary of its public methods.

Table 11-5 DBUtil Method Summary

Returns	Syntax and Description
`static void`	`debugPrint (String app, String catg, String message)` Prints formatted message with arguments.
`static String`	`extractStackTrace (Exception ex)` Prints stack.
`static Properties`	`getDBServiceProperties() throws DBException` Returns an object of class Properties with a pair of (name, value) as (_DBS__PROPS, “com/cisco/nm/cmf/dbservice/DBServer.properties”).

Table 11-5 DBUtil Method Summary

Returns	Syntax and Description
`static String`	`getPropertiesFileName()` Returns the value of environment variable BG_DBPARAMS if it is defined. Else it returns `bgdbparam.ini`.
`static Properties`	`loadPropertiesFromFile (String propertiesFile) throws IOException, FileNotFoundException` Loads the class Properties from DBConst._DBS_PROPS.
`static void`	`printExceptionDetails (Exception e)` Prints contents of the exception object.
`static void`	`printExecuteDebugStmt (long startTime, long stopTime, String sql)` Prints formatted message with arguments.

In CS3.0 SP2, two flavors of the new API, executeSqlStmt() have been created.

One takes default delimiter (;), and the other takes the customized delimiter.

Flavor A

The executeSqlStmt() API is used to execute a bunch of SQL statements septarated by the customized delimiter.

@param dsn—The DSN for the database

@param schemaFileName—The file containing the schema to create the tables. It uses getResourceAsStream to locate the file. Hence this file must be present in the classpath.

@param component—The name of the component/product eg. Kilner/Campus

@param subComponent—The name of the subcomponent eg. OGS/DCR

@param versionString— The version number of the release eg: 1.0/1.1

@param description—A brief description of the module/schema

@param delimiter—Delimiter for the SQL Statements.

@return—Throws an exception if the operation fails for any reason

Flavor B:

The executeSqlStmt() API is used to execute a bunch of SQL statements separated by the default (;) delimiter. This API takes all the parameters spcified in the Flavor A, except the delimiter. Here, the default delimiter is semicolon (;). This API does not support to create procedures

DBConnection

One instance is constructed in the DBClient construct method. An application developer can choose to skip this section.

The DBConnection construct gets the URL from the JDBC property file and establishes a connection to a database engine (default = rme). This topic summarizes the constructor and its public methods.

DBConnection Constructor Summary

`public` **`DBConnection`** `(String dbName) throws SQLException, DBException, FileNotFoundException, IOException, ClassNotFoundException` Builds a connection with dbName.

Table 11-6 DBConnection Method Summary

Returns	Syntax and Description
`void`	`activate (String client)` Activates a client by adding the client name to the internal hash table.
`void`	`clearWarnings() throws SQLException` A wrapper for Connection.clearWarnings
`void`	`close() throws SQLException` A wrapper for Connection::close();
`void`	`commit() throws SQLException` A wrapper for Connection::commit();
`Statement`	`createStatement() throws SQLException` A wrapper for connection::createStatement();
`boolean`	`getAutoCommit() throws SQLException` A wrapper for Connection::getAutoCommit();
`String`	`getCatalog() throws SQLException` A wrapper for Connection.getCatalog
`String`	`getClientName()` Returns a client name.
`int`	`getDebugLevel() {` Gets the Cisco debug level.
`DatabaseMetaData`	`getMetaData() throws SQLException` A wrapper for Connection::getMetaData();
`int`	`getTransactionIsolation() throws SQLException` A wrapper for Connection.getTransactionIsolation
`SQLWarning`	`getWarnings() throws SQLException` A wrapper for Connection.getWarnings
`void`	`inactivate()` Deactivates a client by removing the client name from the internal hash table.
`boolean`	`isClosed() throws SQLException` A wrapper for Connection:: isClosed();
`boolean`	`isReadOnly() throws SQLException` A wrapper for Connection.isReadOnly
`String`	`nativeSQL (String sql) throws SQLException` A wrapper for Connection:nativeSQL(sql);
`CallableStatement`	`prepareCall (String sql) throws SQLException` A wrapper for Connection.prepareCall(sql);
`PreparedStatement`	`prepareStatement (String sql) throws SQLException` A wrapper for connection::prepareStatement(sql);
`void`	`rollback() throws SQLException` A wrapper for Connection::rollback();
`void`	`setAutoCommit (boolean autoCommit) throws SQLException` A wrapper for Connection::setAutoCommit(autoCommit);
`void`	`setCatalog (String catalog) throws SQLException` A wrapper for Connection.setCatalog
`void`	`setClientName (String client)` Sets a client name.

Table 11-6 DBConnection Method Summary (continued)

Returns	Syntax and Description
void	`setDebugLevel (int debugLevel)` Sets the Cisco debug level.
void	`setReadOnly (boolean readOnly) throws SQLException` A wrapper for Connection.setReadOnly.
void	`setTransactionIsolation (int ti) throws SQLException` A wrapper for Connection.setTransactionIsolation

Using CWCS Perl APIs

The database APIs are implemented in Perl for portability between platforms. Because these APIs are primarily used by install and database administration scripts such as backup and restore, they will not be implemented in other languages.

Programming Tips for Perl APIs

- All applications that use these APIs must include this line at the beginning of their file:

  ```
  use Cisco::DbUtils;
  ```
- These APIs return 0 for success and 1 for error. The variable *Cisco::Dbutil::errstr* is null if the return value is 0 and contains the error string if the return value is 1.
- For more information about connection strings, see the "Connection Strings" section on page 11-5.

Perl API Summaries

The following tables summarize the Perl APIs.

Table 11-7 Database Process and File Management Perl APIs

Returns	Syntax and Description
int	`CheckDb ($connstr);` Checks if the specified database accepts connections. This is done by opening a connection to the database.
int	`StartDb ($connstr, $cacheSize, $timeout, $numRetries)` Starts the database engine as a process on the specified database.
int	`StopDb ($connstr, $timeout, $numRetries);` Stops the specified database engine process. Programs must be sure there are no active connections to the database.

Table 11-8 Miscellaneous Perl APIs

Returns	Syntax and Description
int	`addManifestFiles ("SUITE=xxx;SWITCH", \@parm2, \$manifestPath);` Creates the manifest files used by the backup, restore, and move database utilities. This routine works in one of two ways, depending on the SWITCH field of the first parameter.
int	`check_create ($Dir)` Verifies the existence of the specified directory and creates it if it does not exist.

Table 11-8 Miscellaneous Perl APIs (continued)

Returns	Syntax and Description
int	deleteDbVersionData (*$dbh, $component, $subComponent*); Using the database handle and the primary keys, this routine deletes a row from the DbVersion table.
int	deleteManifestFiles ("SUITE=*xxx*;SWITCH", *$manifestPath*); Deletes the datafiles.txt or the appropriate dbfiles.txt file.
int	getDbVersionData (*$dbh, $comp, $subcomp, $rhDdata*); Using the database handle and the primary keys, this routine retrieves a row from the DbVersion table. The complete row entry is returned in $rhData.
int	getManifestFiles ("SUITE=*xxx*;SWITCH", \@parm2, *$manifestPath*); Retrieves the contents of the datafiles.txt or dbfiles.txt file.
int	setDbVersionData (*$dbh, $rhData*); Writes a row to the DbVersion table.
int	unloadDbVersionData (*$dbh, $file*); Unloads the contents of the DbVersion table into the specified file.

addManifestFiles

```
$ret = addManifestFiles ("SUITE=xxx;SWITCH", \@parm2, \$manifestPath);
```

Creates the manifest files used by the backup, restore, and move database utilities. This routine works in one of two ways, depending on the SWITCH field of the first parameter.

Input Arguments

`SUITE` Name for a group of applications. Used for backup and restore purposes.

`SWITCH`
- APP—Application directory name. Indicates the location of the datafiles.txt file.
- DSN—The ODBC data source name of the database. Indicates the location of the dbfiles.txt file.

`@parm2`
- If SWITCH = "APP=*xxx*" then @parm2 is a pointer to a list of paths required by the application for backup.

  In this case, the routine creates a file called datafiles.txt in the directory *$NMSROOT*/backup/manifest/*suite*/*app*, where *$NMSROOT* is the directory in which the product was installed.

  The contents of the file will be the contents of @parm2 (the list of paths containing files to be backed up).
- If SWITCH = "DSN=*xxx*" then @parm2 is a pointer to an array of database file paths. In this case, the routine creates a file called dbfiles.txt in *$NMSROOT*/backup/manifest/*suite*/database. The contents of the file will be the DSN value followed by the contents of @parm2 (the list of database files).

`$manifestPath` The complete path of the manifest file. This path information is used to register the file with the UNIX packaging mechanisms.

check_create

```
$ret = check_create ($Dir);
```

Verifies the existence of the specified directory and creates it if it does not exist.

Input Arguments

`$Dir`	The name of the directory.

Return Values

`0`	Directory exists or was created successfully.
`1`	Directory cannot be accessed or failed to create directory.

checkDb

```
$ret = CheckDb ($connString);
```

Checks to see if the specified database accepts connections. This is done by opening a connection to the database. If the connection is opened (the routine is successful), the connection is immediately closed.

Input Arguments

`$connString`	**SQL style:** ENG=*xx*;DBN=*xx*;UID=*xx*;PWD=*xx* where ENG, DBN, UID, PWD are the database engine name, database name, database user ID and password. **ODBC style:** DSN=*xx* where the engine corresponding to the ODBC data source *xx* is checked.

Return Values

`0`	Connection accepted.
`1`	Connection refused.

deleteDbVersionData

```
$ret = deleteDbVersionData ($dbh, $component, $subComponent);
```

Deletes a row, defined by the database handle and primary keys, from the database.

Input Arguments

`$dbh`	Database handle.
`$component`	A primary key into the Schema Version tracking table in the database.
`$subcomponent`	A primary key into the Schema Version tracking table in the database.

deleteManifestFiles

```
$ret = deleteManifestFiles("SUITE=xxx;SWITCH", \$manifestPath);
```

Deletes the manifest files used by the backup, restore and move database utilities. This routine works in one of two ways, depending on the SWITCH field of the first parameter.

Input Arguments

`SUITE`	Name for a group of applications. Used for backup and restore purposes.
`SWITCH`	• If SWITCH = "DSN=*yyy*" then the dbfiles.txt file in the *$NMSROOT*/backup/manifest/*suite*/database is deleted. *$NMSROOT* is the directory in which the product was installed. • If SWITCH = "APP=yyy" then the datafiles.txtfile in the directory *$NMSROOT*/backup/manifest/*suite*/*app* is deleted.
`$manifestPath`	The complete path of the manifest file. This path information is used to register the file with the UNIX packaging mechanisms.

getDbVersionData

```
$ret = getDbVersionData ($dbh, $comp, $subcomp, $rhDdata);
```

Returns the complete row entry in the Schema Version tracking table, identified by the database handle and primary keys, in a hash table.

Input Arguments

`$dbh`	Database handle.

`$comp`	Component—a primary key into the Schema Version tracking table in the database.
`$subcomp`	Subcomponent— a primary key into the Schema Version tracking table in the database.

Output Arguments

`$rhDdata`	Pointer to a hash table containing column name and value pairs.

Example

```
$dbh = DBI->connect('DSN=xx', undef, undef, 'Sqlany');
$ret = getDbVersionData($dbh, 'Main', 'Baseline', $rhData);
```

where $rhData is a pointer to a hash whose definition looks like this:

```
$rhData = {'Component' => 'Main', 'SubComponent' => 'Baseline',
           'VersionString' => '208',
           'Description' => 'Database Schema Model Version',
           'InstallDate' => '08/21/1998'};
```

getManifestFiles

```
$ret = getManifestFiles ("SUITE=xxx;SWITCH", \@parm2, \$manifestPath);
```

Places the contents of either the datafiles.txt or the dbfiles.txt file into an array. This routine works in one of two ways, depending on the SWITCH field of the first parameter.

Input Arguments

`SUITE`	Name for a group of applications. Used for backup and restore purposes.
`SWITCH`	• APP—Application directory name. Indicates the location of the datafiles.txt file. • DSN—The ODBC data source name of the database. Indicates the location of the dbfiles.txt file.
`@parm2`	• If SWITCH = "APP=*xxx*" then @parm2 contains the contents of the datafiles.txt file, which is the list of file paths containing the files to be backed up. • If SWITCH = "DSN=*yyy*" then @parm2 contains the contents of the dbfiles.txt file.
`$manifestPath`	The complete path of the manifest file. This path information is used to register the file with the UNIX packaging mechanisms.

setDbVersionData

```
$ret = setDbVersionData ($dbh, $rhData);
```

Adds or modifies a row in the DbVersion table.

Input Arguments

`$dbh`	Database handle.
`$rhData`	Contains the row to be inserted or updated.

StartDb

```
$ret = StartDb ($connString, $cacheSize, $timeout, $numRetries);
```

Starts the database specified in the connection string. The connString argument must have dsn=xxx component.

Input Arguments

`$connString`	**SQL style:** "ENG=xx;DBN=xx,UID=xx;PWD=xx" where ENG, DBN, UID, PWD are the database engine name, database name, database user ID, and password. **ODBC style:** "DSN=xx". where the engine corresponding to the ODBC data source *xx* is stopped.
`$cacheSize`	Size (in megabytes) of the database engine. Default = 16M.
`$timeout`	Number of seconds the routine waits for the database to start on each try. Default = 5 seconds.
`$numRetries`	Number of times the routine attempts to start the database before returning an error. Default = 10 retries.

Example

This example starts the database with a cache size of 16M.

```
$ret = StartDb("eng=upgrade; dbn=Essentials; dbf=/opt/CSCOpx/objects/db/px.db; uid=sa;
pwd=c2Ky2k", "16");
```

StopDb

```
$ret = StopDb ($connString, $timeout, $numRetries);
```

Stops the specified database engine process. Programs must be sure there are no active connections to the database.

Input Arguments

`$connString`	**SQL style:** `ENG=xx;UID=xx;PWD=xx` where ENG, UID, PWD are the database engine name, database user ID, and password. **ODBC style:** `DSN=xx` where the engine corresponding to the ODBC data source *xx* is stopped.
`$timeout`	Number of seconds the routine waits for the database to stop on each try. Default = 5 seconds.
`$numRetries`	Number of times the routine attempts to stop the database before returning an error. Default = 10 retries.

unloadDbVersionData

```
$ret = unloadDbVersionData ($dbh, $file);
```

Writes the contents of the DbVersion table into the specified file. Use this routine for backing up data and restoring utilities to match database versions.

Input Arguments

`$dbh`	Database handle.

Output Arguments

`$file`	Complete path to the file.

Using the Database Utilities

The following topics describe the available database utilities:

Utility Name	Description
backup.pl	Backs up the database file to a specified directory. It also backs up the files that are listed in manifest files into specific directory locations. For details, see backup.pl, page 12-10.
configureDb.pl	Performs several functions, including installing and uninstalling the database.
dbinit	Creates an empty database, assigns the default user ID and password, and specifies various options.
dbMonitor	Ensures that a database can accept connections.
dbpasswd.pl	Changes the password field in the database configuration files.
DBPing	Ensures that multiple database engines can accept connections at startup.

Utility Name	Description
dbreader.pl	A web-based Perl database query and manipulation utility intended to supplement the Sybase dbisqlc.
dbRestoreOrig	The dbRestoreOrig utility restores a canned database from the orig directory.
dbvalid	A Sybase utility that determines if the database is valid.
restorebackup.pl	Restores a previous backup. For details, see restorebackup.pl, page 12-11.
runIsql	Runs the SqlAnywhere ISQL utility on the database identified by the connection string. The SQL commands are read from the script file.

configureDb.pl

This utility has several functions:

`perl configureDb.pl action={install|uninstall} <dsn=`*`database`*`>`
If `action=install`, copies the database file from the /orig to the runtime directory. If `action=uninstall`, removes the database file.

`perl configureDb.pl action=rebuild dsn=`*`database`*
Rebuilds database files to 9.0.0 format. Can be called during upgrade from older versions.

`perl configureDb.pl action={reg|unreg} dsn=`*`database`*` dmprefix=`*`prefix`*
If `action=reg`, registers the database, including populating the .odbc.ini or Windows odbc registry, updating dmgtd.conf or the Windows services registry, and updating the DBServer.properties file. If `action=unreg`, unregisters the database.

`perl configureDb.pl action=upgrade dsn=`*`database`*` portid=`*`number`*
Checks the database version and upgrades to 9.0.0 format.

`perl configureDb.pl action=upgradeall`
Upgrades every database to 9.0.0 format.

`perl configureDb.pl action=validate dsn=`*`database`*
Use the dbvalid utility to validate the specified database.

`perl configureDb.pl action=reg dsn=`*`database`*` dmprefix=`*`prefix`*` dbmonitor=no`
Do not register dbMonitor as the default database monitor. This option is intended for use only when you want to substitute your own database monitor, such as DBPing.

Input Arguments

`dsn` The data source name (for example, cmf or rme)

`dmprefix` The system name (for example, CWCS or Essentials). For example, to register the Essentials database, enter:

```
perl configureDb.pl action=reg dsn=rme dmprefix=Essentials
```

The configureDb.pl script always validates dmprefix against the dmprefix value in odbc.tmpl. If the two do not match, the script will throw a warning and use the odbc.tmpl value . If there is no dmprefix entry in odbc.tmpl, the given entry is used and added to odbc.tmpl.

To find the value of dmprefix:

- On Solaris platforms, go to *$NMSROOT*/objects/dmgt/dmgtd.conf and look for {$dmprefix}DbEngine.
- On Windows platforms, use regedit to access HKEY_LOCAL_MACHINE > SYSTEM > Current ControlSet > Services > {$dmprefix}DbEngine.

`portid` The port ID number. This must be a 16-bit integer smaller than 65535.

dbinit

```
dbinit -j [-b] [-c] [-p page-size] dbName.db
```

Note This procedure should be performed by developers only.

The dbinit utility is a Sybase utility that:

- Creates an empty database with a system catalog and system stored procedures.
- Assigns the following default user ID and password: DBA, SQL.
- Specifies the page size, transaction log file, case sensitivity and blank padding options.

Prerequisites

- CMF 1.2 or higher must be installed.

Input Arguments

`dbName` The name of the new database. The .db extension is required.

Switches

Use `dbinit -help` for additional help with switches.

`-j` Do not install runtime Java classes.

`-b` Pads blanks in strings for comparisons.

`-c` Enforces case sensitivity for all string comparisons.

Note Case sensitivity cannot be changed later.

`-p page-size` Sets the page size.

Note Page size cannot be changed later.

dbMonitor

```
dbMonitor dsn -app daemon_registered_name -dbserver database_registered_server_name
[-sterror "start_error_interval"] [-stretry "start_retry_number"]
[-sleep "sleep_interval"] [-error " error_sleep_interval"]
[-retry "error_retry_number"][-debug] [-nodisplay]
```

DbMonitor is the default database-engine process monitor. It ensures that a database can accept connections and avoid race conditions in database connections during engine startup, and periodically monitors the database to ensure that the connection is still available. Each database requires a separate DbMonitor process to monitor it. For example, all CWCS applications depend on the DbMonitor for the CWCS database; an application with its own database will require an additional DbMonitor instance for that database, and only that application will be dependent on that instance. Only the ODBC version of DbMonitor is used. See the "DBPing" section on page 11-56 for an alternative database monitor.

DbMonitor performs these functions:

- On startup, DbMonitor attempts to connect to the database and periodically selects from a common database table. If successful, it notifies the Daemon Manager to start the dependent applications.
- After the initial startup, DbMonitor periodically monitors the database by attempting selects from the common table. If it is unsuccessful, it sends a message to the front end and notifies the Daemon Manager to terminate the dependent applications.
- Important: All daemons dependent on the database should place a dependency on the corresponding DbMonitor and not on the database. This is because Daemon Manager does not accurately reflect the state of the database, but DbMonitor does.
- A DbMonitor entry is automatically created for each database that is registered using the configureDb.pl utility.

Input Arguments

`dsn` Data source name (for example, cmf).

Switches

`-app` Registered process name (for example, CmfDbMonitor).

`-dbserver` Registered database server name (for example, CmfDbEngine).

`-sterror` Number of seconds to sleep before next connection try. Optional.

`-stretry` Number of times to try to make a connection. Optional.

`-sleep` Number of seconds to sleep before checking the database engine. Optional.

`-error`	Number of seconds to sleep before next fetch for a valid connection. Optional.
`-retry`	Number of times to try a fetch before declaring the connection is down. Optional.
`-debug`	Flag to send more debug information to log file. Optional.
`-nodisplay`	Flag to disable all log information. Optional.

DBPing

```
$NMSROOT/bin/cwjava com.cisco.nm.cmf.dbservice2.DBPing -name daemon_registered_name -dsn
database_server_name_list [-maxtry tries] [-timeout time] [-debug]
```

DBPing is an alternative to dbMonitor. It ensures that all specified database engines are successfully initialized at startup. It *does not* poll the databases for connectivity thereafter. If all database engines are up and responding, the DBPing process notifies Daemon Manager that DBPing itself is up and running; otherwise, it will notify Daemon Manager that DBPing is down. All other daemons requiring database operations should register themselves as dependent on DBPing.

DBPing performs these functions:

- On startup, DBPing attempts to connect to each database in the -dsn list. If all of these attempts are successful, it notifies the Daemon Manager to start any dependent applications. If any of them are down, DBPing will generate an error
- Important: All daemons dependent on the databases in the -dsn list should place a dependency on DBPing.
- You can use DBPing only if you use the the configureDb.pl utility's `dbmonitor=no` option.

Input Arguments

`-name`	Registered DBPing process name (for example, DFMDbMonitor).
`-dsn`	A comma-delimited list of database server names (for example: `DFMDbEngine, CmfDbEngine`).

Switches

`-maxtry`	Number of times toattempt to make a connection with each of the database servers named in the -dsn list. Optional.
`-timeout`	Amount of time (in milliseconds) between attempts to make a connection with each of the database servers named in the -dsn list. Optional.
`-debug`	Flag to send more debug information to log file. Optional.

Remarks

DBPing provides a monitor that can verify the status of multiple database engines at startup. It is intended for situations where your application has multiple databases, and you want to ensure that all database engines are fully initialized before Daemon Manager starts up any processes that depend on them. To use DBPing for this purpose:

1. Register each of your application databases using the `dbmonitor=no` option of configureDb.pl. This will prevent automatic creation of a DBMonitor instance for each of these databases. For example:
   `perl configureDb.pl action=reg dsn=MyAppDB1 dmprefix=prefix dbmonitor=no`
   `perl configureDb.pl action=reg dsn=MyAppDB2 dmprefix=prefix dbmonitor=no`
2. During your application installation, install and register with Daemon Manager a process that makes use of DBPing. Make sure you:
   a. Assign your DBPing process a unique `-name` value. For example: `$NMSROOT/bin/cwjava com.cisco.nm.cmf.dbservice2.DBPing -name MyAppDBMon -dsn MyAppDB1, MyAppDB2 -maxtry 10 -timeout 5000 -debug`. You must use this same name (for example, `MyAppDBMon`) when registering the DBPing process with Daemon Manager.
   b. Use a DmgrRegisterWR (for C and C++) or DmgrRegisterJavaWR (for Java) call to register the DBPing process with Daemon Manager. Both registration calls allow you to specify a `-timeout` value for the DBPing process. For more information, see the "DmgrRegister" section on page 21-51.
   c. Pass with your registration call a `-timeout` value that is less than the combined value of your DBPing process `-maxtry` and `-timeout` values. For example, if your DBPing process call specifies `-maxtry 10` and `-timeout 5000`, your DmgrRegisterWR or DmgrRegisterJavaWR -timeout value should be less than 50000.
3. Set all other processes that depend on the databases in the DBPing `-dsn` list with a dependency on your DBPing process.
4. Unregister the DBPing process if the application is uninstalled.

dbpasswd.pl

This utility changes the password field in the following database configuration files:

- The odbc.tmpl file.
- The ODBC registry (.odbc.ini for Solaris, or the registry for Windows).
- The database service property file (DBServer.properties) and, if specified, the private property file.

You can use this utility in a variety of ways, as shown below.

Table 11-9 dbpasswd.pl Usage Summary

Format	Action
`dbpasswd.pl all`	Changes all database passwords.
`dbpasswd cfile=configurefile`	Overrides database settings based on the configuration file specified.
`dbpasswd.pl listdsn`	Lists all available data sources in the product.
`dbpasswd.pl dsn=odbc_datasource`	Changes the database password.
`dbpasswd.pl dsn=odbc_datasource npwd=new_password`	Changes the database password to `new_password`.

Table 11-9 dbpasswd.pl Usage Summary

Format	Action
`dbpasswd.pl dsn=odbc_datasource encryption=yes`	Encrypts the database password.
`dbpasswd.pl dsn=odbc_datasource encryption=yes npwd=new_password`	Changes the database password to `new_password` and encrypts the database password.

Prerequisites

- Daemon Manager must be shut down.
- On UNIX platforms, you must be logged in as root.
- On Windows platforms, you must be logged in as part of the local administrator group.
- This utility assumes that the databases and data sources are properly configured.

Input Arguments

`all` Validates all registered databases and changes the password for each database.

`cfile` Overrides the database settings as specified in the configuration file mentioned

`dsn` Contains the ODBC data source name (DSN) of the database whose password will be changed. If the *dsn* argument is present, change the password for the specified database.

`encryption` Encrypt the user name and password for the specified database. Possible values: YES, NO (default). The values are *not* case sensitive.

`npwd` Optional. If present, replace the password in the registry entry, odbc.tmpl, and the property file with *`new_password`*.

`listdsn` If present, this must be the only argument. Lists all data source names in the Solaris .odbc.ini or Windows registry and quits. To be included in this list, a database must be enabled.

Remarks

Since dbpasswd.pl also tries the password in the odbc.tmplorig file if the value in the ODBC registry fails, it can be run as the last manual step in the existing database repair schemes that require copying the factory database from the *orig* directory.

The dbpasswd.pl utility validates passwords. Valid passwords must:

- Have a minimum of five and a maximum of 128 characters.
- Use alphanumeric characters (a-z, A-Z, 0-9) only. No special characters (e.g., #, $, %) or spaces are allowed.
- Not have a number as the first character.

Related Topics

See the "Updating the Database Password" section on page 11-24.

dbreader.pl

This is a web-based Perl database query and manipulation utility intended to supplement the Sybase dbisqlc.

Note Dbreader cannot be invoked from the command line; it can only be run from a browser.

For information about using dbreader, see the "Examining the Contents of a Database" section on page 11-31.

Runtime Location

$NMSROOT/htdocs

dbRestoreOrig

```
dbRestoreOrig.pl dsn=dsnname dmprefix=dmprefixname [npwd=newpassword]
```

The dbRestoreOrig utility:

- Restores the pre-canned database from the orig directory.
- Changes the file permissions.
- Populates the encrypted or plain-text user ID and password to .odbc.ini or the registry entry.
- Populates the user ID and password to the property file for JDBC access (CWCS 2.2 and later).

Note This utility was first introduced in CWCS 2.2. The JDBC property files were included in previous CWCS releases.

Input Arguments

`dsnname` Data source name (for example, cmf).

`dmprefixname` Used to construct the database engine process name. For example:

- For CWCS: dmprefix is Cmf and the CWCS database engine name is CmfDbEngine.
- For RME: dmprefix is Essentials, and the RME database engine name is EssentialsDbEngine.

The dmprefix argument is initially configured using configureDb.pl. To determine the value of dmprefixname, see the "configureDb.pl" section on page 11-53.

Caution There is a check to validate this argument against the dmPrefix property in odbc.tmpl. If this property is absent, the user is prompted for action.

`newpassword` Optional. If present, replace the contents of the current password with *newpassword.*

dbvalid

```
dbvalid -c "uid=$uid; pwd=$pwd; dbf=$dbf"
```

If a process cannot bring up the database engine and settings are okay, run this Sybase utility to determine if the database is valid. If the database is not totally corrupted, it might be possible to recover some of the data.

This script is located in *$NMSROOT*/objects/db/bin, where *$NMSROOT* is the directory in which the product is installed.

Note This is a Sybase utility. configureDb.pl provides a wrapper to use this utility by providing only the dsn name as an argument. configureDb.pl will invoke dbvalid with the correct connection string.

Prerequisites

On Solaris platforms, set the environment variables first (see the "Setting Up Your Environment" section on page 11-19).

Switches

`-c`	Connection string.

Input Arguments

`uid`	User ID
`pwd`	User password
`dbf`	Database file

runIsql

```
runIsql ($scriptFile, $connString);
```

Runs the SqlAnywhere ISQL utility on the database identified by the connection string. The SQL commands are read from the script file.

Input Arguments

`$scriptFile` Complete path name to the script file that contains the SQL commands.

`$connString` **SQL style:** ENG=*xx*;DBN=*xx*;UID=*xx*;PWD=*xx*

where ENG, DBN, UID, PWD are the database engine name, database name, database user ID and password. Connection string must be complete or this routine will fail.

ODBC style: DSN=*xx*

Contains either the DSN or the DSN plus the user ID and password. In either case, it is converted to SQL-style before the ISQL call.