CHAPTER 6

Configuring Databases

Introduction

This chapter describes how to configure the Collection Manager to work with your database. The chapter also describes how to use the database infrastructure to extend the functionality of the Collection Manager.

Generating SQL Code Using the Velocity Template Language

The JDBC Adapter framework uses macros written in the Velocity Template Language (VTL) to generate all SQL code that is passed to the database server. The following sections describe the configuration file that controls the generation process.

For more information regarding VTL (which is part of the Apache Jakarta Project) go to http://jakarta.apache.org/velocity/vtl-reference-guide.html.

Table 6-1 describes VTL constructs:

Table 6-1 Summary of VTL Constructs

Directive	Syntax Example	Purpose

`#foreach`	`#foreach ($item in $collection)`
	`item is $item`
	`#end`
		Iterates over a collection, array, or map.

`#if`
`...`
`#else`
`...`
`#elseif`
	`#if ($order.total == 0)`
	`No charge`
	`#end`
		Conditional statement.

`#parse`	`#parse("header.vm")`	Loads, parses, and includes the specified template in the generated output.

`#macro`	`#macro(currency $amount)`
	`${formatter.currency($amount)}`
	`#end`
		Defines a new directive and any required parameters. The result is interpreted when used later in the template.

`#include`	`#include("disclaimer.txt")`	Includes the specified file, as is, in the generated output.
`#set`	`#set ($customer = ${order.customer})`	Assigns a value to a context object. If the context object does not exist, it is added; otherwise, it is overwritten.
`#stop`	`#if ($debug) #stop #end`	Stops template processing.

Database Configuration Files

When you initialize the Database access framework, the first file the Database access framework searches for is **nf-main.vm**, which contains definitions or pointers to all the required database SQL definitions. The location used to search for this file depends on the dbpack used in the Collection Manager. A dbpack is a collection of configuration files pertaining to a specific database installation. The adapter (according to its configuration file) selects the dbpack. The following code fragment from the **nf-jdbcadapter.conf** file configures it to work with an Oracle dbpack:

```
db_template_dir = dbpacks/oracle/9204e/
db_template_file = nf-main.vm
```

Note The directory location is interpreted relative to the main Collection Manager configuration directory (**~scmscm/cm/config**).

To make the configuration more modular, the **nf-main.vm** file generally points to other files; however this mechanism is not strictly necessary. The files can contain arbitrary definitions that can be used later, for example, in scripts. Some definitions are mandatory because the JDBC adapter uses them for its operation. These definitions are listed in Table 6-2:

Table 6-2 Mandatory VM Definitions

Object Name	Mandatory Definition
$table.sql.dropTable $table.sql.createTable $table.sql.createIndexes $table.sql.insert $table.sql.metaDataQuery	For each table, these settings control how SQL is generated for the indicated operation
$dbinfo.driverjarfile $dbinfo.driver	Location and class name for JDBC driver
$dbinfo.cmdSeparator	Pattern used to separate multiple SQL statements
$dbinfo.url $dbinfo.connOptions	URL for connecting to the database Various connection properties

Some objects representing the Collection Manager configuration in the VTL parsing context can be used in the templates. These objects are described in the following sections.

- Context Objects, page 6-4
- Application Configuration, page 6-5

Context Objects

Before any Collection Manager components (for instance, the JDBC adapter or a script), load and parse the VM templates, the Collection Manager initializes the parsing context with the following Java objects:

- The **tables** object
- The **dbinfo** object
- The **tools** object

tables Object

The **tables** object describes application-related database configuration, such as the structure of NetFlow Records that are stored in the database, the structure of the database tables and where they are stored, and the structure of any other database tables that the Collection Manager uses. The object is an array in which each row represents one of the database tables used by the Collection Manager. For each table, the row can contain the following information (not all items are relevant to all tables):

- Logical name
- Physical name
- NetFlow Record tag associated with this table
- List of fields/columns in this table, with the following attributes for each:
  - Field ID
  - Field name
  - Field native type
  - Free-form field options
- List of indexes for this table, with the following attributes for each:
  - Index name
  - Names of columns indexed
  - Free-form index options

The contents of the **tables** object can be inspected or manipulated when loading the templates. The **tables** object is initialized using the application-specific XML configuration file. See the "Application Configuration" section on page 6-5.

dbinfo Object

The **dbinfo** object describes configuration that is specific to the database, such as the parameters and the SID or schema that is used when opening a database connection. The object holds database-specific configuration options. The object contains the following information:

- The name of the JDBC class that is used as a driver for this database
- The name of the JAR file that contains the driver
- The location of the database expressed as a JDBC URL
- Free-form JDBC connection options, such as authentication data (user and password)

tools Object

The **tools** object is a container for several utility methods that you can find useful when developing templates or manipulating the context data structures.

You invoke the object methods by using **$tools.method(arg1, ..., argN)**, where **method** is the name of the method.

The methods are listed in Table 6-3:

Table 6-3 Methods of the tools Object

Method Name and Arguments	Function
`getTableByName (allTables, name)`	Locates the database table object whose logical name corresponds to name.
`getTableByDbTabName (allTables, name)`	Locates the database table object whose physical name corresponds to **name**.
`assignParams (sql, list_of_args)`	Replaces question mark characters in the SQL string with consecutive elements from the **list_of_args** parameter, represented as a String. This method is useful if working with templates that create SQL insert statements using the JDBC PreparedStatement string as a base.
`collapseWhitespace()`	Converts all instances of more than one consecutive white-space character to one space, and trims beginning and ending white space. This method can be useful for databases that require SQL with a minimum of newline and other white-space characters. (Sybase and Oracle do not require this conversion.)

For a sample that demonstrates how to use these tools, see the "Using the JDBC Framework in Scripts" section on page 6-10.

Application Configuration

All application-related configuration is contained in one file (**nf-dbtables.xml**), which includes the following items:

- Name and version of the application
- Name and properties of each database table, and specifically the structure of application NetFlow Records that are stored in database tables
- For each database table:
  - Names and native types of the table/NetFlow Record fields
  - Names and properties of the table indexes

This information is used primarily to populate the **tables** object in the template parsing context. See the "tables Object" section on page 6-4.

Working Sample

The **nf-main.vm** file can contain references to other VM files to support modularization (see the "Database Configuration Files" section on page 6-3). The names of these other files are arbitrary, except for the **VM_global_library.vm** file, which name is predetermined. Place macros that must be defined in this file to ensure that they are loaded at the right time. For details about this special file, see the *Velocity User Guide*.

The following sample illustrates the contents of main.vm for an Oracle setup:

```
#parse ('nf-dbinfo.vm')

#foreach ($table in $tables)
   #set ($table.sql.dropTable = "#parse ('drop_table.vm')")
   #set ($table.sql.createTable = "#parse ('create_table.vm')")
   #set ($table.sql.createIndexes = "#parse ('create_indexes.vm')")
   #set ($table.sql.insert = "#parse ('insert.vm')")
   #set ($table.sql.metaDataQuery = "#parse ('metadata.vm')")
#end
```

In this sample, the mandatory database and SQL definitions (see Table 6-2) are moved to separate files, which are loaded and parsed using the **#parse** directive.

The following sections list possible contents for the various files in the Oracle dbpack. Some of the definitions use macros that are defined in the **VM_global_library.vm** file. This file contains all macro definitions that any template uses.

Macro Definitions

The following sample illustrates definitions for the mapping between native types and SQL types, and utility macros such as the **optcomma** macro, which inserts a comma between successive elements of lists.

```
#macro (optcomma)#if ($velocityCount >1),#end#end
#macro (sqltype $field)
#set ($maxStringLen = 2000)
#if     ($field.type == "INT8") integer
#elseif ($field.type == "INT16") integer
#elseif ($field.type == "INT32") integer
#elseif ($field.type == "UINT8") integer
#elseif ($field.type == "UINT16") integer
#elseif ($field.type == "UINT32") integer
#elseif ($field.type == "REAL") real
#elseif ($field.type == "BOOLEAN") char(1)
#elseif ($field.type == "STRING") varchar2(#if($field.size <= $maxStringLen)$field.size
#else $maxStringLen #end)
#elseif ($field.type == "TEXT") long
#elseif ($field.type == "TIMESTAMP") date
#end
#end
```

dbinfo Configuration

In the following code sample, note that the only required fields are the URL and connection options (for authentication).

Blank lines in the code separate the code into distinct fields for readability and to ease later configuration changes.

```
#set ($dbinfo.driver = "oracle.jdbc.OracleDriver")
#set ($dbinfo.driverjarfile = "ojdbc14.jar")
#set ($dbinfo.options.host = "localhost")
#set ($dbinfo.options.port = "1521")
#set ($dbinfo.options.user = "pqb_admin")
#set ($dbinfo.options.password = "pqb_admin")
#set ($dbinfo.options.sid = "avocado")
#set ($dbinfo.url =
"jdbc:oracle:thin:@$dbinfo.options.host:$dbinfo.options.port:$dbinfo.options.sid")
#set ($dbinfo.connOptions.user = $dbinfo.options.user)
#set ($dbinfo.connOptions.password = $dbinfo.options.password)
## the vendor-specific piece of SQL that will return the current
## date and time:
#set ($dbinfo.options.getdate = "sysdate")
```

SQL Definitions

The following sections provide SQL definitions:

Code for Dropping a Table

The following code sample drops a table using normal SQL syntax

```
drop table $table.dbtabname
```

Code for Creating a Table

The following code sample creates a table using normal SQL syntax. Use this definition for any customized database configuration that requires special directives to create a table. For example, you can modify the configuration to create the table in some unique tablespace or to use table partitioning.

```
create table $table.dbtabname (
#foreach ($field in $table.fields)
 #optcomma()$field.name #sqltype($field)
 #if ("$!field.options.notnull" == "true")
  not null
 #end
#end)
```

Code for Creating Indexes

The following code creates the indexes using normal SQL syntax. Use this definition to implement a customized database configuration that requires special directives to create an index. For example, you can modify the configuration to create the indexes in some unique tablespace.

```
#foreach ($index in $table.indexes)
 create index $index.name on $table.dbtabname ($index.columns)
#end
```

Code for Insert

The following code creates the JDBC PreparedStatement that corresponds to the table structure.

```
insert into ${table.dbtabname} (
#foreach ($field in $table.fields)
  #optcomma()${field.name}
#end)
values (
#foreach ($field in $table.fields)
  #optcomma()?
#end)
```

Code for Metadata Query

The following code defines a simple query that gets the table metadata (column names and types). You can use any query that returns an empty result set.

```
select * from ${table.dbtabname} where 1=0
```

Testing and Debugging

As you develop a set of templates for your database, you can see directly the results of parsing. To invoke such parsing, the JDBC adapter supports direct invocation using the Collection Manager main script **~scmscm/cm/bin/cm**.

The syntax for an invocation is:

```
~/cm/bin/cm invoke com.cisco.scmscm.netflow.adapters.jdbc.JDBCAdapter
```

Where **argument** is one of the parameters described in the following sections. You can use this mechanism whether the Collection Manager is running or not.

Additionally, you can use the query and update execution methods described in the following section to test the template results with a live database.

Parsing a String

You can parse any string as a VTL template with the complete context in place. The result of the parsing is displayed on the standard output. To parse a string, call the adapter with the -parse parameter. The following sample code illustrates a parsing mechanism (responses are shown in **bold**):

```
$ ~/cm/bin/cm invoke com.cisco.scmscm.netflow.adapters.jdbc.JDBCAdapter -parse 'xxx'
xxx
$ ~/cm/bin/cm invoke com.cisco.scmscm.netflow.adapters.jdbc.JDBCAdapter -parse
'$dbinfo.url'
jdbc:oracle:thin:@localhost:1521:avocado
$ ~/cm/bin/cm invoke com.cisco.scmscm.netflow.adapters.jdbc.JDBCAdapter -parse
'$tools.getTableByName($tables, RPT_USAGE_NF).sql.createTable'
CREATE TABLE RPT_USAGE_NF (
  TIME_STAMP      datetime,
  NF_HEAD_TIME_STAMP    integer ,
  NF_HEAD_SOURCE_ID      integer,
  NF_CLASS_ID        integer,
  NF_APPLICATION_ID     integer,
  NF_INGRESS_IF     integer,
  NF_EGRESS_IF       integer,
  NF_FLOW_DIRECTION       integer,
  NF_FLOW_START_SYSUP_TIME   integer,
  NF_FLOW_END_SYSUP_TIME    integer,
  NF_IN_PKTS      integer,
  NF_IN_BYTES     integer,
  NF_CONNECTION_COUNT_NEW     integer,
  NF_CONN_SUM_DURATION       integer
)
```

Obtaining Full Debug Information

To see a dump of all of the contents of the **tables** and **dbinfo** structures as created by the templates, use the **-debug** parameter. The **-debug** parameter displays in standard output a detailed view of all the fields, properties, and options of these structures.

Using the JDBC Framework in Scripts

You can send arbitrary SQL commands to the database for execution and view the resulting data. Viewing results helps you to perform periodic database maintenance, to monitor the contents of database tables, and to manage extra database tables.

To perform an **update** operation, call the adapter using the **-executeUpdate** parameter. To perform a query and view the results, call the adapter using the **-executeQuery** parameter.

Viewing and Setting the Time Zone Offset

The following sample of an update operation demonstrates how to change programmatically the value in the database table that contains the Cisco ASR 1000 time zone offset setting. The name of this table is usually **CONF_TZ_OFFSET_NF**; because the table can be assigned another name, it is referred to here by its logical name TZ. See the listing in the CM:NetFlow nf-dbtables.xml File, page B-2.

To avoid the need to check that the table exists first, and then update it, the table is dropped (ignoring the error status if it does not exist) and then recreated. The proper values are inserted. Since the table contains a timestamp column, you must get the current date from the database. This operation is specific to each database vendor; therefore, this example uses the preconfigured **getdate** operation that is defined in the templates.

Note the use of the tools **assignParams** and **getTableByName** to generate the SQL.

```
#! /bin/bash

this=$0
tableName=TZ

usage () {
    cat <<EOF
Usage:
   $this --status      - show currently configured TZ offset
   $this --offset=N    - set the offset to N minutes (-1440 <= N <= 1440)
   $this --help        - print this message
EOF
}

query () {
~/cm/bin/cm invoke com.cisco.scmscm.netflow.adapters.jdbc.JDBCAdapter -executeQuery "$*"
}

update () {
        ~/cm/bin/cm invoke com.cisco.scmscm.netflow.adapters.jdbc.JDBCAdapter
-executeUpdate "$*"
}

get_tz () {
    query 'select * from $tools.getTableByName($tables, "TZ").dbtabname'
}

set_tz () {
    update '$tools.getTableByName($tables, "TZ").sql.dropTable'
    update '$tools.getTableByName($tables, "TZ").sql.createTable'
    update '$tools.assignParams($tools.getTableByName($tables, "TZ").sql.insert,
[$dbinfo.options.getdate, '$1'])'
}
```

```
case $1 in
    --status)
        get_tz
        ;;
     --help)
        usage
        exit 0
        ;;
     --offset=*)
        n=$(echo $1 | egrep 'offset=[-]?[0-9]+$' | sed 's/.*=//')
        if [ "$n" ]; then
            if [ "$n" -ge -1440 -a "$n" -le 1440 ]; then
                set_tz $n &>/dev/null
                ok=1
            fi
        fi
        if [ ! "$ok" ]; then
            usage
            exit 2
        fi
        get_tz
        ;;
     *)
        usage
        exit 3
        ;;
esac
```

When an executed query returns a result set, it is displayed to standard output in tabular form with appropriate column headers.

Scalability Hints for Oracle

The following sections demonstrate ways to make database handling more scalable for the Collection Manager. These methods are specific to Oracle, and are provided to illustrate the possibilities.

Using Custom Tablespaces

If you create several tablespaces and wish to distribute the Collection Manager tables among them, specify the tablespace to use for each table in the file **nf-dbtables.xml**. The following sample code represents the definition for one table (note the code in **bold**):

```
<rdr name="USAGE_NF" dbtabname="RPT_USAGE_NF" tag="Usage"
                createtable="true">
                <fields>
                    <field id="1" name="TIME_STAMP" type="INT32">
                    <options>
                       <option property="source" value="timestamp" />
                    </options>
                </field>
                <field id="2" name="NF_HEAD_TIME_STAMP" type="INT32">
                    <options>
                         <option property="source" value="nf_timestamp" />
                    </options>
                </field>
                <field id="3" name="NF_HEAD_SOURCE_ID" type="INT32">
                    <options>
                              <option property="source" value="recordsource" />
                    </options>
                </field>
                <field id="4" name="NF_CLASS_ID" type="UINT32" /> <!-- For Future Releases-->
                <field id="5" name="NF_APPLICATION_ID" type="INT32DIG10" />
                <field id="6" name="NF_INGRESS_IF" type="UINT32" />
                <field id="7" name="NF_EGRESS_IF" type="UINT32" />
                <field id="8" name="NF_FLOW_DIRECTION" type="UINT8" />
                <field id="9" name="NF_FLOW_START_SYSUP_TIME" type="UINT32" />
                <field id="10" name="NF_FLOW_END_SYSUP_TIME" type="UINT32" />
                <field id="11" name="NF_IN_PKTS" type="UINT64" >
                    <options>
                         <option property="source" value="unsigned64" />
                    </options>
                </field>
```

```
            <field id="12" name="NF_IN_BYTES" type="UINT64" >
              <options>
                  <option property="source" value="unsigned64" />
              </options>
            </field>
            <field id="13" name="NF_CONNECTION_COUNT_NEW" type="UINT32" />
            <field id="14" name="NF_CONN_SUM_DURATION" type="UINT64" >
              <options>
                  <option property="source" value="unsigned64" />
              </options>
            </field>
        </fields>
        <indexes>
            <index name="RPT_USAGE_NF_I1" columns="NF_HEAD_TIME_STAMP">
                <options>
                    <option property="clustered" value="true" />
                    <option property="allowduprow" value="true" />
                </options>
            </index>
            <index name="RPT_USAGE_NF_I2" columns="NF_APPLICATION_ID">
            </index>
        </indexes>
    </rdr>
```

This sample adds the required tablespaces (**tspace1** and **tspace2**) for the index and for the table. There is no preconfigured meaning to the option **tablespace** in the Collection Manager. Any new option name could have been used. Its meaning is derived from its subsequent use in the templates.

To create the table in the correct tablespace, modify **create_table.vm** as follows:

```
create table $table.dbtabname (
#foreach ($field in $table.fields)
 #optcomma()$field.name #sqltype($field)
 #if ("$!field.options.notnull" == "true")
  not null
 #end
#end)
#if ("$!table.options.tablespace" != "")
 TABLESPACE $table.options.tablespace
#end
```

To create the index in its own tablespace, modify **create_indexes.vm** as follows:

```
#foreach ($index in $table.indexes)
 create index $index.name on $table.dbtabname ($index.columns)
#if ("$!index .options.tablespace" != "")
  TABLESPACE $index.options.tablespace
 #end
#end
```

Using Table Partitioning

To implement rolling partitioning for a particular table on a weekly basis, create a partitioned option for the table in the **nf-dbtables.xml** file in a manner similar to the example in the previous section (see the "Using Custom Tablespaces" section on page 6-12). Then augment the **create_table.vm** code as follows (note the code in **bold**):

```
create table $table.dbtabname (
#foreach ($field in $table.fields)
#optcomma()$field.name #sqltype($field)
#if ("$!field.options.notnull" == "true")
not null
#end
#end)
#if ("$!table.options.partitioned" != "")
partition by range (timestamp)
(partition week_1 values less than (to_date ('01-JAN-2005 00:00:00','DD-MON-YYYY
HH24:MI:SS')),
partition week_2 values less than (to_date ('08-JAN-2005 00:00:00','DD-MON-YYYY
HH24:MI:SS'))
partition week_3 values less than (to_date ('15-JAN-2005 00:00:00','DD-MON-YYYY
HH24:MI:SS'))
partition week_4 values less than (to_date ('22-JAN-2005 00:00:00','DD-MON-YYYY
HH24:MI:SS')) );
#end
```

Because Oracle does not accept nonconstant expression for the time boundaries, the values must be hardwired for the time the tables are created.

Create a cron job to roll the partitions (delete an old partition and create a new one) on a weekly basis. This cron job runs a script that calls the command-line interface of the JDBC Adapter (as explained in the "Using the JDBC Framework in Scripts" section on page 6-10) to issue the appropriate **alter table drop partition** and **alter table add partition** SQL commands.