CHAPTER 6

Configuring Databases

This module describes how to configure the Cisco Service Control Management Suite (SCMS) Collection Manager (CM) to work with your database, and how to use the database infrastructure of the CM to extend its functionality.

Quick Start Guide

To use an external database with the CM, it is necessary to change basic connection parameters such as the IP address and port on which the database is deployed. To configure these parameters, use the dbconf.sh script. See Configuring Databases, page 4-4.

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 used to control 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 **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 CM. A dbpack is a collection of configuration files pertaining to a specific database installation. The adapter (in accordance with its configuration file) selects the dbpack. The following code fragment from the **jdbcadapter.conf** file configures it to work with an Oracle dbpack:

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

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

To make the configuration more modular, the **main.vm** file generally points to other files; however this is not strictly necessary. The files can contain arbitrary definitions that can later be used, 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, and various connection properties

Some objects representing the CM configuration in the VTL parsing context are available to be used in the templates. These objects are described in the following sections.

- Context Objects, page 6-3
- Application Configuration, page 6-5

Context Objects

Before the VM templates are loaded and parsed by any CM components (for instance, a TA or JDBC adapter, or a script), the parsing context is initialized 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 RDRs that should be 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 CM might use. The object is an array in which each row represents one of the database tables used by the CM. For each table, the row may contain the following information (not all items are relevant to all tables):

- Logical name
- Physical name
- RDR 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 Application Configuration, 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 to be used when opening a database connection. The object holds database-specific configuration options. It contains the following information:

- The JDBC class name to be used as a driver for this database
- The name of the JAR file containing 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 might find useful when developing templates or manipulating the context data structures.

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

The included 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 characters to one space, and trims beginning and ending white space. This method may be useful for databases that require SQL with a minimum of newline and other white-space characters. (Sybase and Oracle do not require this.)

For a sample that demonstrates how to use these tools, see Using the JDBC Framework in Scripts, page 6-8.

Application Configuration

All application-related configuration is done in one file (**tables.xml**) that includes the following items:

- Name and version of the application
- Name and properties of each database table, and specifically the structure of application RDRs that are to be stored in database tables
- For each database table:
  - Names and native types of the table/RDR fields
  - Names and properties of the table indexes

This information is used primarily to populate the **tables** object in the template parsing context. See tables Object, page 6-3.

Working Sample

The **main.vm** file can contain references to other VM files to support modularization (see Database Configuration Files, page 6-2). The names of these other files are arbitrary, except for the **VM_global_library.vm** file whose name is predetermined. Place macros that need to 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 ('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, to be 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 should contain all macro definitions used by any template.

- Macro Definitions, page 6-6
- dbinfo Configuration, page 6-6
- SQL Definitions, page 6-6

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 ($dbinfo.options.sid = "apricot")
#set ($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

- Code for drop table, page 6-7
- Code for create table, page 6-7
- Code for create indexes, page 6-7
- Code for insert, page 6-7
- Code for metadata query, page 6-7

Code for drop table

The following code sample drops a table using normal SQL syntax

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

Code for create table

The following code sample creates a table using normal SQL syntax. Any customized database configuration that requires special directives for table creation can be implemented using this definition. For example, you can modify it 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 create indexes

The following code creates the indexes using normal SQL syntax. A customized database configuration requiring special directives for index creation can be implemented using this definition. For example, you can modify it 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 corresponding 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 is used to get the table metadata (column names and types). Any query that returns an empty result set can be used.

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

Testing and Debugging

While you develop a set of templates for your database, it is useful to be able to see the results of parsing directly. To enable, this, the JDBC adapter supports direct invocation using the CM main script **~scmscm/cm/bin/cm**.

The general syntax for an invocation is:

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

where **argument** is one of the flags described in the following sections. You can use this mechanism whether or not the CM is running.

Additionally, the query and update execution methods described in the following section can be used to test the template results against a live database.

- Parsing a String, page 6-8
- Obtaining Full Debug Information, page 6-8

Parsing a String

Any string can be parsed 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 flag. Examples appear below (responses are shown in **bold**):

```
$ ~/cm/bin/cm invoke com.cisco.scmscm.adapters.jdbc.JDBCAdapter -parse 'xxx'
xxx
$ ~/cm/bin/cm invoke com.cisco.scmscm.adapters.jdbc.JDBCAdapter -parse '$dbinfo.url'
jdbc:oracle:thin:@localhost:1521:apricot
$ ~/cm/bin/cm invoke com.cisco.scmscm.adapters.jdbc.JDBCAdapter -parse
'$tools.getTableByName($tables, "LUR").sql.createTable'
create table RPT_LUR (
 TIME_STAMP   date
   ,RECORD_SOURCE   integer
   ,LINK_ID   integer
   ,GENERATOR_ID   integer
   ,SERVICE_ID   integer
   ,CONFIGURED_DURATION   integer
   ,DURATION   integer
   ,END_TIME   integer
   ,UPSTREAM_VOLUME   integer
   ,DOWNSTREAM_VOLUME   integer
   ,SESSIONS   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** flag. When this flag is used, a very detailed view of all the fields, properties, and options of these structures is printed to standard output.

Using the JDBC Framework in Scripts

You can send arbitrary SQL commands to the database for execution and view the resulting data. This may be useful for periodic database maintenance, monitoring the contents of database tables, managing extra database tables, or any other purpose.

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

Viewing and Setting the SCE Time Zone Offset

The following sample of an update operation demonstrates how to programmatically change the value in the database table holding the Service Control Engine (SCE) time zone offset setting. The name of this table is usually **JCONF_SE_TZ_OFFSET**; because the table may be assigned another name, it is referred to here by its logical name TZ. See the listing in tables.xml File, page A-1.

To avoid the need to first check that the table exists and then update it, the table is dropped (ignoring the error status if it does not exist) and then recreated, and 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.adapters.jdbc.JDBCAdapter -executeQuery "$*"
}

update () {
        ~/cm/bin/cm invoke com.cisco.scmscm.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 a result set is returned by an executed query, 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 CM. These are specific to Oracle, and are provided as hints to illustrate the possibilities.

- Using Custom tablespaces, page 6-10
- Using Table Partitioning, page 6-11

Using Custom tablespaces

Suppose you create several tablespaces and wish to distribute the CM tables among them. Specify the tablespace to be used for each table in the file **tables.xml**. For one table, the definition looks like this (note the code in **bold**):

```
<rdr name="LUR" dbtabname="RPT_LUR" tag="4042321925" createtable="true">
   <options>
      <option property="tablespace" value="tspace1"/>
   </options>
   <fields>
      <field id="1" name="TIME_STAMP" type="TIMESTAMP">
      <!-- (other field declarations) -->
      <field id="10" name="DOWNSTREAM_VOLUME" type="UINT32"/>
      <field id="11" name="SESSIONS" type="UINT32"/>
   </fields>
   <indexes>
      <index name="RPT_LUR_I1" columns="END_TIME">
         <options>
            <option property="clustered" value="true"/>
            <option property="allowduprow" value="true"/>
            <option property="tablespace" value="tspace2"/>
         </options>
      </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 CM; 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, you can create a partitioned option for the table in the **tables.xml** file in a similar manner to the example in the previous section (Using Custom tablespaces, page 6-10). 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 Using the JDBC Framework in Scripts, page 6-8) to issue the appropriate **alter table drop partition** and **alter table add partition** SQL commands.