DbSpec Language Reference

Based on extracts from O3.2-3 of the project "Immortal Database Access", funded by Eurostars

|  |  |  |
| --- | --- | --- |
| Note no. |  |  |
| Authors |  | Ivar Rummelhoff, Thor Kristoffersen, Bjarte M. Østvold |
| Date |  | Last edited: 2024-06-20 |

Table of Content

[1 Introduction 5](#_Toc169786972)

[1.1 Overview 5](#_Toc169786973)

[1.2 The staging database 6](#_Toc169786974)

[1.3 Database connections 6](#_Toc169786975)

[1.4 Embedded SQL 6](#_Toc169786976)

[1.5 Other embedded scripts 7](#_Toc169786977)

[1.6 Metadata 8](#_Toc169786978)

[1.7 Future use cases 8](#_Toc169786979)

[1.8 SIARD output 9](#_Toc169786980)

[1.9 Packaging 9](#_Toc169786981)

[2 How and when to write a DbSpec script 9](#_Toc169786982)

[2.1 Example work process 10](#_Toc169786983)

[2.2 Remark: Backups versus transformations 11](#_Toc169786984)

[3 Syntax and semantics 12](#_Toc169786985)

[3.1 Character encoding 12](#_Toc169786986)

[3.2 Indented blocks 12](#_Toc169786987)

[3.3 Whitespace 12](#_Toc169786988)

[3.4 Comments and “shebang” 13](#_Toc169786989)

[3.5 Parameters 14](#_Toc169786990)

[3.6 Statements 14](#_Toc169786991)

[3.7 Variables 14](#_Toc169786992)

[3.8 Types and non-Boolean expressions 15](#_Toc169786993)

[3.9 Dot operators 16](#_Toc169786994)

[3.10 String literals and string interpolation 16](#_Toc169786995)

[3.11 Interpolation in string blocks 17](#_Toc169786996)

[3.12 Set statements 18](#_Toc169786997)

[3.13 Execution statements 19](#_Toc169786998)

[3.14 For loops 19](#_Toc169786999)

[3.15 Log statements 19](#_Toc169787000)

[3.16 Boolean expressions, conditionals, and assertions 20](#_Toc169787001)

[3.17 Metadata and SIARD output 20](#_Toc169787002)

[4 Metadata 21](#_Toc169787003)

[4.1 Database level SIARD metadata 21](#_Toc169787004)

[4.2 Schema level SIARD metadata 22](#_Toc169787005)

[4.3 Type level metadata 23](#_Toc169787006)

[4.4 Table level metadata 23](#_Toc169787007)

[4.5 View level metadata 23](#_Toc169787008)

[4.6 Metadata concerning future use cases 24](#_Toc169787009)

# Introduction

DbSpec is a domain specific language (DSL) for describing the steps taken to produce data preservation artifacts, currently SIARD[[1]](#footnote-2) archives of relational databases augmented with information regarding expected future use cases. DbSpec scripts have a clearly defined syntax and semantics (chapters 3 and 4), which has been implemented in an interpreter (chapter 5).

The DbSpec language and interpreter fill a gap between the existing interactive and non-interactive tools for creating and augmenting SIARD archives in that it makes explicit and repeatable the steps for producing the archive, including integration with other tools and scripts. The steps may include packaging the resulting SIARD archive(s) and other files as an OAIS information package (SIP or AIP). Thus, DbSpec is especially useful in the context of the iDA decommissioning process. This is discussed in chapter 2.

This document describes version 1.0 of DbSpec. While we believe this is already a useful tool in many situations, it should be considered a rough prototype. Some possible improve­ments and extensions are discussed towards the end, in chapter 6 (appendix). In particular, some language features have not yet been implemented in the interpreter.

## Overview

Even though the main motivation behind DbSpec was to simplify and ensure the quality of certain steps in the iDA decommissioning process (see section 2.2), we believe it can also be useful in other contexts. That being said, the discussion below will focus on data preservation during decommissioning.

In general terms, the role of DbSpec is to integrate existing data management and transformation tools, usually in order to produce a “curated” dataset suitable for long-term storage (cf. the discussion in section 2.3). This involves:

1. *Extracting* existing data from relational databases and other sources such as flat files (or even source code).
2. *Transforming* the extracted data using merge, join and filtering operations.
3. Gathering and augmenting documentation and other *metadata*.
4. Producing *digital artifacts* suitable for archiving, in particular files in the SIARD format.
5. *Documenting* these steps for reproducibility.

Using DbSpec, we can perform the steps (i)-(iv) in a single, executable script, thus archiving (v) at the same time.

## The staging database

SIARD is primarily a format for archiving SQL-compliant relational databases, including metadata. In many cases we do not simply want to preserve the database(s) of the decommissioned system “as is”, but a curated database – possibly based on multiple data sources and more independent of the decommissioned software – see section 2.3 for more on this. In practical terms this is achieved as follows:

1. First, we construct a “staging” database containing exactly those objects and entries we want preserved in the curated SIARD archive.
2. Next, we specify the metadata we want included in the archive that cannot be extracted from the database itself.
3. Finally, we write the SIARD archive consisting of this data and metadata to a file.

If desirable, these steps can be repeated – either in the same or in multiple DbSpec scripts – to produce multiple SIARD files with varying levels of transformation.

## Database connections

The connection to the staging database must be declared in the DbSpec script like so:

Set dbc = connection to “jdbc:sqlserver://${host}:${port}/”  
with:  
 database = “staging”  
 user = “example/${user}”  
 password = pwd

Observe that the address of the DBMS is specified using a JDBC connection string and that the properties can be specified on separate lines. The string interpolations syntax (involving ‘$’) is explained in section 3.10. One DbSpec script may use multiple database connections, e.g. in order to combine data from different data sources.

## Embedded SQL

When it comes to transforming the staging database, we rely on SQL – more precisely, the dialect used by the DBMS since we simply transmit these statements via the given database connection (after string interpolation). This is a conscious decision since features unique to the DBMS may sometimes be needed to replace non-standard database features not supported by SIARD. Example:

Execute via dbc:  
 CREATE TABLE Temp (  
 id VARCHAR(${id\_size}) NOT NULL,  
 counter INT NULL,  
 PRIMARY KEY (id));

We may also embed SQL *queries* in DbSpec, i.e. SELECT statements:

Set employee\_names = result via dbc:  
 SELECT id, name  
 FROM dbo.employees;

Here the DbSpec interpreter will bind employee\_names to a “result set”. The rows in a result set can be processed using the **For** construct:

For (id, name) in employee\_names:  
 …

## Other embedded scripts

Some data transformations require other tools than the SQL interpreter of the DBMS. For instance, the standard way of creating a backup of a PostgreSQL database involves calling the command line utility pg\_dump. In DbSpec this can be solved by calling external interpreters. Example:

Execute using “/bin/bash”:  
 pg\_dump -U “${user}” -F t customers > customers.tar

Notice that string interpolation can be used here as well, but make sure to not write passwords in scripts or pass them as command line parameters. For handling/importing data in unusual file formats, embedded Python scripts can be quite useful:

Execute using "/usr/bin/env python3":  
 import pandas as pd  
 import geopandas as gpd  
 import sqlalchemy as sa  
 gdb = 'Kommuner.gdb'  
 kommune = gpd.read\_file(gdb, layer = 'kommune')  
 navn = gpd.read\_file(gdb, layer = 'navn')  
 engine = sa.create\_engine('postgresql+psycopg2://' +  
 '${user}:${password}@localhost/staging')  
 location = pd.read\_sql(  
 sql="SELECT location\_id, latitude, longitude " +  
 "FROM location",  
 con=engine, index\_col='location\_id')  
 geo = gpd.GeoDataFrame(geometry=gpd.points\_from\_xy(  
 location.longitude, location.latitude))  
 geo = geo.set\_crs(epsg=4326)   
 geo = geo.to\_crs(kommune.crs)  
 \_, ixs = kommune.sindex.nearest(geo.geometry)  
 nk = navn[navn.sprak == 'nor'].join(  
 kommune, on='kommune\_fk', lsuffix='\_n')  
 location['name'] = nk.iloc[ixs].navn.values  
 location.name.to\_sql('location\_name', con=engine)

That being said, embedding such scripts also mean that the DbSpec script will have more system dependencies and thus be less portable. These dependencies should be clearly stated in comments at the top of the script. In complex cases, we also recommend formalizing this using a container technology.

## Metadata

The code responsible for generating the SIARD file extracts not only data from the database, but also metadata such as the database schema. However, the SIARD format also supports metadata not found in the database/DBMS itself. For this reason, we have included in DbSpec a special statement specifying metadata:

Metadata for dbc:  
 dbname = “oil2023”  
 description:  
 This database contains the preserved data of the  
 oil and gas department of Example Corporation at  
 the time of its dissolution in 2023.  
 archiver = “foobar@example.com”  
 Schema core:  
 description:  
 The schema containing the most important  
 tables in this database.  
 Table customer:  
 description = “The main customer table”  
 Column customer\_id – Internal customer id  
 …

These statements do not affect the database directly, but they have preference over the metadata extracted from the database when producing SIARD files.

## Future use cases

In addition to the usual SQL metadata, DbSpec allows the user to bundle with the SIARD file details on how to implement a certain form of expected future use cases: database lookups based on user provided parameters. These are also grouped within Metadata statements:

Metadata for dbc:  
 …  
 Command:  
 title = "Get list of transactions for a customer"  
 Parameters:  
 customer\_id – The system ID of the customer  
 Body:  
 SELECT t.transaction\_id, t.title  
 FROM core.transactions t  
 WHERE t.customer\_id = $${customer\_id};  
 …

This also shows an alternative interpolation syntax, which is safer if customer\_id may contain double quotes or other special characters (see section 3.11).

## SIARD output

Finally, DbSpec scripts will typically specify that one or more SIARD files are to be produced:

Output dbc to "oil2023.siard"

Otherwise, the metadata statements will have no effect. Since the DbSpec script is processed sequentially, there is also no point in placing metadata or transformation statements after the output statement(s).

Since Command declarations are not part of the SIARD standard, the Output statement will gather those in a separate file, see section 3.17.

## Packaging

SIARD files will often be packaged together with other files – typically an OAIS[[2]](#footnote-3) information package (SIP or AIP). If so, we recommend including the DbSpec script itself in this package, as well as the file(s) containing Command declarations. DbSpec does not (yet) have a separate statement for producing such packages, but if the relevant package can be produced using a script, we recommend including this as an external script in the DbSpec script. Thus, all the steps needed to produce the package will be documented in one place and can easily be repeated if necessary.

# How and when to write a DbSpec script

As of this writing, there is no support for writing DbSpec scripts in existing text editors or integrated development environments (IDEs). Whereas it would be helpful and should be straightforward to add syntax highlight­ing for DbSpec to some editors[[3]](#footnote-4), the key to writing efficient, correct, and readable DbSpec scripts is to develop the embedded scripts using specialized development environments. For complex SQL scripts or queries in particular, we recommend first using an interactive environment such as SQL Server Management Studio[[4]](#footnote-5), Oracle SQL Developer[[5]](#footnote-6) or pgAdmin[[6]](#footnote-7).

## Example work process

The actual steps involving DbSpec in a decommissioning process may look something like this:

1. Obtain a backup of the main relational database that should be preserved.  
   If the system is still in production, this can be a preliminary backup.
2. Obtain access to a DBMS instance – preferably the same version as the backup.
3. Create an empty DbSpec script.
4. Write and run a script or shell command for restoring the backup to a staging database in this instance.
5. Copy the script into the DbSpec script as an external command, using string interpolation for values that should not be fixed such as usernames and passwords.
6. Write and run scripts for importing other relevant data into the staging database – e.g. descriptions for codes used in the database tables – and copy these scripts into the DbSpec script, also with interpolation where appropriate.
7. Write and run scripts for transforming the staging database so that it contains the data and database objects we want in the SIARD archive and nothing more. Copy these scripts into the DbSpec script as well.
8. Include Assert statements in the DbSpec script as “sanity checks”, e.g. ensuring that no table is empty.
9. Add one or more Metadata statements augmenting the automatically extracted metadata for the database (see chapter 4). In particular, one should usually write descriptions for every table and column.
10. Add one or more Metadata statements with expected future use cases, i.e. Command declarations.
11. Add an Output statement for producing the SIARD archive and companion Command file.
12. Add one or more external scripts for packaging these files together with the initial backup file, the DbSpec script itself and other relevant files into an information package, e.g. according to the CITS SIARD specification[[7]](#footnote-8).
13. Make sure the DbSpec script runs correctly from start to finish.
14. Obtain the final versions of the backup files and other relevant data, make sure the DbSpec script uses these files, and run the DbSpec interpreter one last time in order to produce the final information package.

In practice, this is usually more of an iterative process. For instance, (13) will often reveal issues so that one has to go back to a previous step.

## Remark: Backups versus transformations

Even though a DbSpec script can transform the original database beyond recognition, there may be reasons for keeping the changes minimal. If possible, we suggest preserving all of the following:

1. a complete backup of the system being decommissioned, including databases, binaries, and other files,
2. the source code of the system,
3. a curated SIARD archive of the data and metadata of the system, possibly transformed from multiple data sources in order to simplify future use.

Preserving A means that it is possible – in theory at least – to resurrect the whole system. It also means that potential errors introduced in C can be remedied. To some extent, this can also be achieved by a SIARD archive of the relational database “as is” (C’). However, there is also potential data loss when producing such archives. In particular, such archives will not contain data stored outside the database or hardcoded in the source code. The software available to restore such archives also have limited functionality. In particular, the documentation of the SiardToDb tool of the SIARD Suite[[8]](#footnote-9) states:

*Uploading only creates tables and types and attempts to enable unique and foreign key constraints. No other database objects are created.*

Thus, if there is a choice between preserving a complete, native database backup (A) or a raw SIARD archive (C’), the former may be a better choice even though the risk of format obsolescence is presumably higher, especially for RDBMSs that are not widely used. That being said, unless the storage requirements are an issue, one can keep both. DbSpec is arguably geared towards producing curated SIARD archives (C), but it can also produce archives that are closer to the original (C’). In fact, a single DbSpec script can produce multiple archive files at varying levels of transformation if desirable.

The reason why we suggest produce a curated archive (C’) now, is to make it easier to use correctly and thus more useful. At the time of the decommissioning, one is more likely to find someone that understands the system than in the future. As most systems accumulate technical debt over time, writing database queries that are guaranteed to give the correct result can become nearly impossible for someone not familiar with the quirks of the system.

# Syntax and semantics

In this chapter we shall explain the concepts and components of the DbSpec language in more detail. Since blocks are delimited using indentation, it is not – strictly speaking – a context-free language. However, this can be solved using a lexical analyzer (lexer) that converts changes in indentation to indent and “dedent” tokens. This approach is taken in the current DbSpec parser, see section 5.1.

## Character encoding

DbSpec scripts must use the UTF-8 character encoding, but they *should* *not* start with the two bytes known as the “byte order mark” (BOM). As of this writing, either line feed (LF, \n) or line feed + carriage return (CRLF, \r\n) can be used as newline.

## Indented blocks

A characteristic of DbSpec – seen in the examples in chapter 1 – is that blocks are delimited using increased indentation. This is similar to programming languages such as Python and Haskell, but DbSpec is more strict with regards to whitespace. This is explained in section 3.3.

Some blocks contain statements, for instance the body of a For loop. We also use indented blocks for (potentially) multi-line strings and embedded source code in other languages. We shall refer to these two kinds as *string blocks*. Finally, we use special blocks for expressing metadata, see chapter 4.

The DbSpec parser does not accept completely empty blocks. Unlike in Python a properly indented comment is sufficient, but this trick does not work in string blocks.

## Whitespace

In DbSpec horizontal tabulation (HT, \t) is the only whitespace character allowed at the beginning of lines. There is one exception to this rule, however: String blocks can use any whitespace character for indentation as long as they occur after the HT characters defining the block. Example:

Execute viadbc:  
 HT DELETE FROM Temp  
 HT WHERE counter >= $${min}  
 HT AND counter <= $${max};

This example also illustrates that it is easier to write correct DbSpec scripts using editors that can highlight the HT characters.

DbSpec scripts may contain empty lines, usually to make the script more readable. An empty line is not considered to end the current block. Example:

Letiso\_time = result using "/usr/bin/env python3":  
 HT from datetime import datetime  
  
 HT print(datetime.now().isoformat(), end='')

Otherwise, deleting “trailing whitespace” might break a script.

To avoid ambiguity, consecutive keywords and identifiers must be separated by at least one space character. Except in fixed constructs, we may use HT characters instead. Thus, in the previous example, we could have written “**Let** HT iso\_time”, but not “**result** HT **using**”.

In general, DbSpec does not allow newlines within statements except when it contains a block (which is always indicated using a colon). Very long lines can usually be avoided using variables with short names, but long lines are not considered problematic in DbSpec. In particular, it is sometimes appropriate to use long lines in string blocks to avoid newlines in the resulting string or script. Still, these strings and scripts will always contain at least one final newline character as well as empty lines corresponding to the empty lines before the next statement or comment.

## Comments and “shebang”

A DbSpec script may contain lines starting with the character ‘#’, possibly preceded with HT characters in order to line up with the current block structure. These lines are code comments, intended for human readers, ignored by the DbSpec interpreter. However, on some platforms one can make the script executable by setting the file execute bit and starting the script with the line:

#!/usr/bin/env dbspec

Do not indent this line. If used, ‘#!’ – known as “shebang” – must be the first two bytes of the file. Thus, there should also not be a byte order mark (BOM) in the file.

After the shebang line (if present) one would typically include a few comment lines concerning copyright and the intended use of the file. Example:

# Copyright © 2023 Example software – All Rights Reserved  
#  
# This file describes the data extraction and transformation  
# performed as part of the decommissioning of system Foo in   
# June 2023, resulting in SIARD archive file Foo\_2023.siard.  
#  
# For information about the DbSpec language used in this file  
# see https://www.dbspec.org.

DbSpec does not allow comments on the same line as other code or within string blocks. Comments within an embedded script must follow the conventions of the script language. Example:

Set customer\_ids = result via dbc:  
 -- This is an SQL comment  
 SELECT id  
 FROM customer;

## Parameters

A DbSpec script may declare a list of script parameters, in which case this must be the first statement in the file. Example:

Parameters:  
 port - The port of the DBMS  
 # Undocumented parameters:  
 user  
 password

For each parameter one may include an official documentation string after a hyphen (-). These strings are not allowed to span multiple lines, but the lines may be as long as you want. Parameters are variables with string values, see section 3.7. Since there is no way to specify default parameter values, one must provide a complete list of arguments in order to run the script. How this is done is discussed in chapter 5.

## Statements

Except for comments and the optional parameter list, a DbSpec script consists of a list of *statements*, which the interpreter will attempt to execute sequentially. If there is an error or failed assertion, the interpreter will terminate the execution immediately. Thus, it sometimes makes sense to include code in the script that cleans up potential remains of a previous, unsuccessful execution first.

DbSpec does not allow more than one statement on each line; and there is no symbol for terminating or separating statements (such as “;” in the languages descending from ALGOL).

## Variables

The way variables are handled in DbSpec is similar to a single function scope in the programming language Python (before let). More precisely:

* Variables do not have to be declared before assignment.
* Variables do not have a fixed type.
* There is a single, global variable scope.

Any valid Python identifier[[9]](#footnote-10) can be used as a DbSpec variable name. Thus, a variable name cannot start with a digit (0-9), but otherwise it can be any combination of underscore (\_), digits, and letters in many alphabets. DbSpec does not have any reserved words that cannot be used as variable names, but it can be confusing with variable names like “Let” and “For”. Variable names in DbSpec are case sensitive.

A variable is *bound* if it is either a parameter or has been set either with an explicit Set statement or as one of the iteration variables in a For loop. After a For loop, the iteration variables will retain the values of the last iteration – unless the result set was empty, in which case the variables will retain the values they had before the loop, if any.

Any reference to an unbound variable is an error and terminates the execution.

## Types and non-Boolean expressions

In the current version of DbSpec (v1.0) a non-Boolean expression is either:

* a variable instance (i.e. a variable name),
* an integer constant (matching /-?[0-9]+/),
* a string literal – possibly with string interpolation (see section 3.10),
* a “dotted expression” of the form <non-Boolean expression>.<dot operator>  
  (see section 3.9).

This might be extended in the future. During the execution, the value of each non-Boolean expression will have one the following types:

1. Unicode string
2. Integer (arbitrarily large or small)
3. Connection: an open JDBC database connection, see section 3.12.
4. Result set: the result of an SQL queries executed via a database connection.

Thus, the value of each bound variable must also have one of these types. The initial value of a parameter is always a string. In some situations, it makes sense to convert it to an integer. Example:

**Set** port = port.stripped.as\_integer

## Dot operators

The DbSpec “dot operators” are stripped, as\_integer and size:

* If the value of exp is a string, then the value of exp.stripped is the string with whitespace characters removed from both ends.
* If the value of exp is a string matching the regular expression /-?[0-9]+/, then the value of exp.as\_integer is the corresponding integer.
* If the value of exp is a result set, then exp.size is its number of rows.

If one of these operators is applied to a value with the wrong type, the execution terminates with an error.

## String literals and string interpolation

String literals in DbSpec are delimited using double quotes ("). They are not allowed to span multiple lines, but newline characters may be expressed using escape sequences. The allowed escape sequences are as follows:

* \uXXXX, where each X is a hexadecimal digit (/[0-9a-fA-F]/) of a Unicode character. As in Java, “representing supplementary characters in the range U+010000 to U+10FFFF requires two consecutive Unicode escapes” [[10]](#footnote-11).
* \' (single quote), \" (double quote), \\ (backslash), and \$ (dollar sign).
* \b (backspace, BS), \f (form feed, FF), \r (carriage return, CR),  
  \n (linefeed, LF), and \t (horizontal tab, HT).

In DbSpec we also allow string interpolation in every string literal, using the notation

${<non-Boolean expression>}

where the inner expression must evaluate to a string or an integer. Unlike most programming languages, DbSpec also allows string literals in such expressions, and so on. Example:

"a${"b ${23} b"}a" has the same value as "ab 23 ba".

The interpolation symbol $ can be changed using the following declaration:

Interpolation symbol = '<single unicode character>'

The interpolation symbol cannot be a whitespace character or a double quote ("). Changing the interpolation symbol affects the source code after the declaration, regardless of scope or evaluation order. Example:

For (x, y) in result\_set:  
 Set z = "(${x}, ${y})"  
 Interpolation symbol = '€'  
 Log "Pair: €{z}"

## Interpolation in string blocks

We also allow string interpolation in string blocks, both those denoting strings and embedded scripts. Example:

Execute using "/bin/bash":  
 pg\_dump -U "${user}" -F t customers > customers.tar

The fact that the default interpolation symbol ($) has a special meaning in many languages is the main reason for changing this symbol (as explained in the previous section). Even more so since string blocks do not have escape sequences; ‘\’ is interpreted as is.

However, changing the interpolation symbol does not solve the main issue with string interpolation in scripts, namely that the string contents may interfere with the syntax of the embedded script. For instance, the example above will not behave as expected if (the value of) user is a string containing a double quote ("). As a partial solution to this problem, string blocks may also contain a second form of interpolation:

$${<non-Boolean expression>}

where we use two interpolation symbols in a row. Currently, this is only supported for embedded SQL scripts. Using this notation in other string blocks is undefined.

If an SQL script contains the second form of interpolation ($$), the interpreter will create a prepared statement[[11]](#footnote-12) with question marks (?) at each location and make a list of the expression values. Both are then sent to the DMBS via the given database connection. This way, one does not have to worry about quotation marks or strings interfering with the SQL syntax. Example:

Execute viadbc:  
 DELETE FROM Temp  
 WHERE id <> $${user}  
 AND counter > $${max};

This form of interpolation can only be used where the SQL interpreter would accept a string or numeric literal, so writing "$${user}" (including the quotes) would be an error. As with the simple interpolation ($), the value of the expression must be a string or an integer. Both forms of interpolations can be used in the same SQL script if necessary.

## Set statements

The values of variables can be assigned – i.e. set or changed – using **Set** statements. They come in five variants. The most direct assignment is:

Set <variable> = <non-Boolean expression>

which evaluates the expression and makes the result the current value of the variable. Multiline strings can be assigned using a (possibly multi-line) string block:

Set <variable>:  
 <string block>

Next, we use separate forms of the Set statement to capture result sets from SQL queries and output when executing other external scripts:

Set <variable> = result via <connection>:  
 <string block with embedded SQL>

Set <variable> = result using <interpreter>:  
 <string block with embedded script>

Here <connection> and <interpreter> are non-Boolean expressions that must evaluate to a connection or string, respectively.

Finally, we use a form of the Set statement for creating and assigning (JDBC) database connections. Observe that the list of key-value pairs is optional:

Set <variable> = connection to <connection string>

Set <variable> = connection to <connection string> with:  
 <key-value pairs>

The key-value pairs are separated using newlines and each have one of two forms:

<identifier> = <non-Boolean expression>

<identifier>:  
 <string block>

The valid identifier names are the same as valid variable names, see section 3.7. The interpreter URL-encodes the key-value pairs before attaching them to the connection string as query strings. If this does not work as expected, try writing the connection string explicitly instead, i.e. dropping the list of key-value pairs. The exact format of the connection string depends on the DBMS. For instance, the database name is usually a part of the path, but not for Microsoft SQL Server.

For readability, the DbSpec grammar allows a newline before with. In general, we only allow spaces and tabs within statements, except after colons (where newline is mandatory).

## Execution statements

Execution statements are similar to the statements for capturing the results of executing SQL or other external scripts (see section 3.12), except that we are not interested in the output:

Execute via <connection>:  
 <string block with embedded SQL>

Execute using <interpreter>:  
 <string block with embedded script>

The scripts may still fail, though, in which case the interpreter will terminate with an error message.

## For loops

DbSpec scripts may contain For loops for iterating through result sets:

For (<comma-separated variable list>) in <result set>:  
 <body: list of statements>

Here <result set> is a non-Boolean expression which must evaluate to a result set. Thus, at least for now, it will be a variable. The comma-separated list of variables should contain distinct variable names for each column in the result set. For each row the “body” statements will be executed after assigning the variables to the string representations of the corresponding cell values. In other words, one has to use .as\_integer to get the corresponding integer value also when a column has an integral SQL type.

Beware that if any of the cell values is NULL, the corresponding variable becomes unbound. This is currently the only way a bound variable can become unbound in DbSpec. Also observe that since there is a single, global variable scope, the variable bindings after a loop will reflect the last row – except if the result set is empty, in which case the variable bindings will be unaffected.

Iterating through result sets is often useful, but the user is warned against doing this with large result sets since high performance has so far not been a priority for the DbSpec interpreter.

## Log statements

When encountering a **Log** statement, the interpreter will evaluate the expression and print the result to the standard output, provided the expression was of type string or integer. Trying to print other types is an error. Syntax:

Log <non-Boolean expression>

It is recommended to include some Log statements – especially when the DbSpec script has many or long-running steps. It can also be a good idea to log the initial parameter values and other information useful for debugging.

## Boolean expressions, conditionals, and assertions

Even though DbSpec does not have a type of Boolean values (true or false), it does have a rudimentary kind of Boolean expressions, namely:

<non-Boolean expression> <comparison> <non-Boolean expression>

where the value of both expressions must be integers and <comparison> is either

==, !=, <, >, <=, or >=.

Here ‘!=’ means “is not equal to”. In other words, there is no support (yet) for combining Boolean expressions using conjunction, disjunction, or negation. Boolean expressions are used in assertions and conditionals:

Assert <Boolean expression>

If <Boolean expression>:  
 <then-block: non-empty list of statements>  
Else:  
 <else-block: non-empty list of statements>

In both cases the execution terminates with an error if either non-Boolean expression is not an integer. This also happens with the Assert statement if the two integers do not have the stated relation. Similarly, the conditional statement executes either the then-block or the else-block depending on the comparison. The Else branch is optional.

Users should include assert statements is to ensure that the DbSpec script works as expected. Example:

# Ensure that every customer is represented in temp  
Let missing\_customers = result via dbc:  
 SELECT t.id  
 FROM temp t  
 WHERE t.id NOT IN (SELECT id from Customer);  
Assert missing\_customers.size == 0

## Metadata and SIARD output

The Output statement produces a SIARD archive for a database via a database connection:

Output <connection> to <file>

Here <connection> must be a variable bound to a connection, <database> must be the name of a database which is reachable through this connection, and <file> must be a string-valued expression. Example:

Output dbc to "siard\_output/${dbname}.siard"

The SIARD metadata can be augmented or overridden using Metadata statements, as long as they occur before the Output statement. This is discussed in the next chapter.

# Metadata

As mentioned above, Metadata statements augment or override the metadata written to the SIARD archives by subsequent Output statements.

Metadata for <connection>.<database>:  
 <database metadata block>

If multiple Metadata statements refer to the same database connection and database, they are combined, with the last statement having precedence in case of conflict. The database metadata block is a list of declarations, where each declaration is either:

* a declaration of a single piece of metadata for the database as a whole or the archival process, see section 4.1,
* metadata for one of the schemas in the database, see section 4.2,
* or metadata describing expected future use cases, see section 4.4.

In this chapter we go through these one by one, but in the database metadata block the three types of declarations can be mixed freely.

## Database level SIARD metadata

Of the SIARD metadata fields concerning the database as a whole, the following can be specified/overridden in the database metadata block:

archiver, archiverContact, dataOwner, dataOriginTimespan, dbname, description, lobFolder

Those that are underlined are mandatory in the SIARD archive, but that does not necessarily mean that they must be specified in the DbSpec script. For instance, dbname will default to database name (as written on the first line of the Metadata statement). For detailed information about each field, the user should consult the SIARD documentation.

Throughout the Metadata statement fields can be specified in one of two ways:

<field> = <non-Boolean expression>

<field>:  
 <string block>

These are the same alternatives as for the key-value pairs when creating a connection, see section 3.12. In the first case, the expression must evaluate to a string or an integer. String interpolation can be used in both cases. Example:

Metadata for dbc:  
 dbname = "prod2023"  
 description:  
 Praesent et posuere ex. ${user} congue mauris eget   
 sapien dignissim, non vehicula ipsum finibus.   
 Vestibulum iaculis quam eget urna tempor pharetra.   
 Proin et congue orci. Nulla laoreet, nibh ut   
 condimentum mattis, dui mauris maximus nisi,   
 vestibulum ornare est purus sed nibh.   
 archiver = "${user}@example.com"

Beware that in this example, the SIARD description will include 6 newline characters. As discussed in section 3.3, it might be a better idea to write the whole description on one long line.

## Schema level SIARD metadata

Within the database metadata block the metadata concerning a database schema can be specified in one of three ways:

Schema <schema name>

Schema <schema name> - <short description>

Schema <schema name>:  
 <schema metadata block>

Of these, the first does not actually do anything; and the second is equivalent to

Schema <schema name>:  
 description = "<short description>"

except that in the simple variant (with -) there is no string interpolation. This is similar to the parameter descriptions discussed in section 3.5.

In the third variant, the schema metadata block is a set of declarations, much like the database metadata block, but at this level description is the only field that can be specified directly in DbSpec; and if present, this declaration must come first. In addition, the schema metadata block may contain metadata declarations for types, tables, and views within the schema. This is discussed in the next three sections.

## Type level metadata

The SIARD type level metadata is intended for the named advanced or structured types in the schema. Again, description is the only field one can specify directly. Thus, each type declaration has one of four forms, where the first does not do anything and the others are more or less equivalent:

Type <type name>

Type <type name> - <short description>

Type <type name>:  
 description = <non-Boolean expression>

Type <type name>:  
 description:  
 <string block>

## Table level metadata

Again, there are simple declarations that do nothing or only specify the description. Moreover, description is the only supported field; and if specified, that must happen first.

Table <table name>:  
 <optional description=… or description:…>  
 <table metadata block>

The table metadata block consists of three sorts of metadata declarations for columns, keys, and check constraints, respectively. For keys and checks DbSpec only allows specifying the description. Thus, we have similar declarations as for Type. Unless there is need for string interpolation, it makes most sense to use these forms:

Key <key name> - <short description>

Check <check name> - <short description>

Column declarations also has only one field, but in some cases the column may consist of multiple fields, which can have their own descriptions. Example:

Column issue\_details:  
 description = "Multifield column containing issue details"  
 Field severity – Minor, major or critical  
 Field attachments - Blob containing attachments, if any

## View level metadata

The view declarations in a schema metadata block are similar to the table declarations, except that they never mention keys or checks. In other words, they have one optional field, description, followed by column declarations. Example:

View current\_delivery\_issues:  
 description = "An aggregate view of current issues"  
 Column customer\_name - The full name of the customer  
 Column order\_date  
 Column issue\_details:  
 description = "Issue details"  
 Field severity – Minor, major or critical  
 Field attachments - Blob containing attachments

## Metadata concerning future use cases

Finally, the database metadata block may contain one or more “command” declarations, each describing a future use case in terms of a parameterized SQL query:

Command:  
 title = <non-Boolean expression>  
 Parameters:  
 <parameter list>  
 Body:  
 <SQL query (string block)>

Here the parameter list is optional and has the same format as the parameter list for the whole script, see section 3.5. Furthermore, the query is similar to the other SQL scripts in a DbSpec script, but the semantics is different: Since the arguments corresponding to the command parameters will only be available in the future, interpolations will only be partially resolved by the interpreter; and instead of executing a query now, it is saved in a companion file to the SIARD file (with the suffix .roae) together with the title and parameter list.

1. See <https://dilcis.eu/content-types/siard>. [↑](#footnote-ref-2)
2. See <https://www.iso.org/standard/57284.html>. [↑](#footnote-ref-3)
3. This is discussed in section 6.3. [↑](#footnote-ref-4)
4. See <https://learn.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms>. [↑](#footnote-ref-5)
5. See <https://www.oracle.com/database/sqldeveloper/>. [↑](#footnote-ref-6)
6. See <https://www.pgadmin.org/>. [↑](#footnote-ref-7)
7. See <https://dilcis.eu/content-types/cs-siard>. [↑](#footnote-ref-8)
8. See <https://www.bar.admin.ch/bar/en/home/archiving/tools/siard-suite.html>. [↑](#footnote-ref-9)
9. See e.g. <https://peps.python.org/pep-3131>. [↑](#footnote-ref-10)
10. See <https://docs.oracle.com/javase/specs/jls/se16/html/jls-3.html#jls-3.3>. [↑](#footnote-ref-11)
11. See e.g. <https://en.wikipedia.org/wiki/Prepared_statement>. [↑](#footnote-ref-12)