# RunDbp To Do

1. Command line program to invoke a process file – DONE
2. Support for logging to database table. Columns are: – DONE
   1. processId VARCHAR
   2. taskId VARCHAR
   3. messageTime DATETIME
   4. message VARCHAR
3. FAIL command to intentionally fail and force branching. – DONE
4. ZIP command to write one or more files into a ZIP archive. See:  
   <http://www.oracle.com/technetwork/articles/java/compress-1565076.html>. – DONE
5. UNZIP command. – DONE
6. Probably a good idea to support basic file system operations, especially DELETE, RENAME, maybe COPY – DONE
7. PROCESS command to invoke another process file. Needs a separate path to tell where process files are located. - DONE
8. Implement low-level tokenizer that:
   1. doesn't need double backslash in file paths – DONE
   2. supports escaping a quote character by doubling it – DONE
   3. supports end line comments with “--“. – DONE
9. FTP command to transfer files. – DONE
10. Proper NULL handling including:
    1. assigning NULL – DONE
    2. passing NULL to PROCESS arguments – DONE
    3. expressions with NULL-valued terms – DONE
    4. “IS [NOT] NULL” testing – DONE
    5. passing NULL to “?” placeholders – DONE, NOT TESTED
11. Prevent reserved word from being used as a variable name and task name – DONE
12. Complete HEADER handling including IGNORE on OPEN and READ. – DONE
13. COLUMNS handling on READ. – DONE
14. COLUMNS handling on LOAD. – DONE, NOT TESTED
15. HOURLY, DAILY, WEEKLY, MONTHLY keywords in schedules – DONE
16. Select directory for log file in RunDbp.path.properties. (You can always set an absolute path for the file itself in RunDbp.log.properties.) – DONE
17. Control logging errors vs. programmatic messages vs. orphan warnings vs. routine start/stop messages. See SSIS options and logback. – DONE
18. Allow rolling log files with file naming convention. See logback. – DONE
19. New section CONNECTIONS to follow VARIABLES section. Declares named connections as {name} {type} where type is DATABASE, FTP, or EMAIL. Top priority is FTP. – DONE
20. Add CONNECT {connection} [TO] {parameters} with specific connection parameters for each connection type. Elaborate syntax to allow unstated parameters to be inherited from the default connection. – DONE
21. To PUT task, enhance syntax to “TO [ FTP ] {connection}” argument which uses the named FTP connection instead of the default connection. – DONE
22. To EMAIL task, add “THROUGH [ EMAIL ] {connection}” argument immediately before FROM argument which uses the named EMAIL connection instead of the default connection. – DONE
23. For all database tasks, support named connection. – DONE
24. Rename Context.nestContext() as makeChildContext(); rename Context.cloneContext() as makeNestedContext(). – DONE
25. Disconnect named connections when script ends – for root context and child contexts! – DONE
26. Reserve these keywords now: GET, FILES, PROCEDURE, EXECUTE, SYSTEM. – DONE
27. Allow omitting task name following AFTER for sequential ordering of tasks. – DONE
28. Reserve these keywords now: SEND, RECEIVE, CALL, ATTACHMENT, CLOSE. – DONE
29. Distinguish keywords from reserved words. – DONE
30. GET task to get file(s) via FTP – DONE
31. FOR FILES with wildcards, loop over files found – DONE
32. PROCEDURE data source: database-neutral stored procedure invocation with variables including OUT variables; used where SQL or STATEMENT would appear. – DONE
33. WAITFOR ASYNC: waits for all async processes spawned by any direct or indirect predecessors to complete – DONE
34. READ XLSX
35. GET and PUT tasks support for wildcards
36. Change READ INTO TABLE WITH PREFIX to run the prefix command just once instead of with every command in the batch.
37. SYSTEM command to run operating system commands. See:  
    <http://docs.oracle.com/javase/7/docs/api/java/lang/ProcessBuilder.html>
38. Configure database connection for scripting language for RUN SCRIPT. For SQL Server language, split the script into separately invoked sections separated by GO. Also respect block comments and strings split across lines.
39. Ability to invoke an arbitrary Java class that conforms to a certain interface.

# Web Service Request Task

## Requirements

Call web service for each row in a data source.

Read request parameters from the data source.

Substitute request parameters into placeholders in path, query string, or request entity.

For each request row, write a result row to a target: SQL, table, file.

Must be able to associate each result with its request. Therefore, allow copying one or more request parameter(s) to the result target. This means not all request parameters need to be substituted into the web service request.

## Syntax

request ::=  
REQUEST url [ HEADER header-expression ]  
{ get-request | put-request | post-request | delete-request }  
FROM data-source  
[ KEEP { request-field-reference-1 [, request-field-reference-2… ] | NONE } ]  
[ RESPONSE response-field-1 [, response-field-2…] ]  
STATUS status-name  
INTO data-target

get-request ::=  
GET [ [ WITH ] request-field-1 [, request-field-2 … ] ]

put-request ::=  
PUT [ [ WITH ] request-field-1 [, request-field-2 … ] ]  
BODY body-field-1 [, body-field-2… ]

post-request ::=  
POST [ [ WITH ] request-field-1 [, request-field-2 … ] ]  
BODY body-field-1 [, body-field-2… ]

delete-request ::=  
DELETE [ [ WITH ] request-field-1 [, request-field-2 … ] ]

url is a string expression. It is evaluated and then placeholders are found. A placeholder is a valid identifier enclosed in braces “{identifier}”.

Each request-field-n and body-field-n is a string expression. Each must evaluate to an identifier. The number of request-fields plus the number of body-fields must match the number of columns returned from data-source. At run time, the value of each column in data-source is associated to the request-field or body-field at the corresponding position. That is, the value of the first column of the data-source row is associated to the first request-field, and so on.

Each evaluated request-field may match a placeholder in the evaluated url. If so, at run time the value of the data-source column associated with the request-field is substituted for the placeholder in the url. It is not required that every request-field match a placeholder.

Each evaluated body-field is used as a field name in the web service request body. The value of the web service request field is taken from the value of the data-source column associated with the body-field.

For each row of the data-source, a web service request is made to the url with placeholder substitutions as described and with a body composed as described above. The web service response is written as a row to data-target as described below.

If the data-source is introduced by the keyword SQL, the SQL query must be terminated by the phrase END SQL, which will be followed by on the keywords KEEP, RESPONSE, or STATUS.

Each response-field-n is a string expression. Each must evaluate to an identifier. Each identifier is expected to match a field name in the response body received from the web service request.

For each web service request, a row is written to the data-target. The row is composed of the data-source column values corresponding to the request-fields followed by the web service response field values corresponding to the response-fields followed by a single integer holding the http status code.

By default, all the request-field values are retained in the target row. Optionally, a subset of the request field values can be retained and in arbitrary order by specifying the KEEP clause.

status-name is a string expression.

The column names of the target row are the evaluated request-field values followed by the evaluated response-field values followed by the evaluated status-name.

# Reserved Words

The goal is to avoid having an excessive number of reserved words while allowing good flexibility in naming including omitting keywords where possible to achieve more natural language.

The general strategy is: wherever either a keyword or a user-defined name could appear, the test for keywords takes precedence over user-defined names. The one exception is the keyword AFTER: after that keyword, first test for a user-defined name, then a keyword. To inclusively support the broadest set of user-defined names, always have an optional keyword that can introduce a user-defined name.

Full strategy as follows.

## Task Name

**TODO:**

1. Add new keyword NAME to TASK syntax as follows: TASK [[NAME] taskname] ... – DONE
2. Implement small set of reserved task names. – DONE
3. Revise the logic for recognizing anonymous tasks and naked AFTER clause. – DONE

That will allow declaring any task name that is the same as a current or future task type name.

To allow convenience of omitting the NAME keyword in the most possible scenarios, make the following promises.

* Future task type names will never include an underscore.
* Future task type names will avoid compound words. They will be of the form “Verb”. Task names are encouraged to be of the form “VerbNoun”.

With these promises, users are encouraged to select task names that either include underscores or are compound words when omitting the NAME keyword.

The above allows having only a small set of reserved words that cannot be task names:

PREVIOUS, SUCCEEDS, FAILS, COMPLETES, IF – These must not be task names to avoid syntactical ambiguity.

NAME, AFTER – These are reserved to prevent construction of valid but confusing task definitions.

END, TASK – These are always reserved in all contexts for consistency.

## Variable Name

**TODO:**

1. Enhance TaskSetParser to recognize only a name token immediately followed by open parenthesis with no intervening space as a function call. Only in that case check for reserved function names. – DONE
2. Implement small set of reserved variable names. – DONE

The above will allow declaring any variable name without fear of name conflict with current or future function names. Then only the following are reserved words that cannot be variable names:

NULL, CASE – These must not be variable names to avoid syntactical ambiguity.

END, TASK – These are always reserved in all contexts for consistency.

Special considerations:

WHEN – This will be used in the future in the CASE construct. If a variable is named “when”, there can be ambiguity in “CASE [expression] WHEN…”. To avoid the ambiguity, the rule is if WHEN appears where the keyword is allowed, it will be interpreted as the keyword. To use variable “when” as the expression, enclose it in parentheses as “CASE (when) WHEN…”.

YEAR, MONTH, DAY, WEEKDAY, HOUR, MINUTE, SECOND – These are keywords when they appear as the first argument of the DATEPART function. Otherwise they can be used as variable names. They also may be future function names.

## Connection Name

**TODO:**

1. Modify all task syntax so that wherever a connection name is allowed, it can optionally be preceded by the keyword “CONNECTION”. Also, allow the phrase “DEFAULT CONNECTION” in those contexts. – DONE
2. Remove any other syntactical constructs around connection names in task definitions. – DONE
3. Implement small set of reserved connection names. – DONE

I don’t believe there strictly need to be any reserved connection names, but reserve the following to prevent construction of valid but confusing task definitions:

CONNECTION, DEFAULT, STATEMENT, SQL, TABLE, SCRIPT, VALUES, PROCEDURE, FILE, FILES, VARIABLE, END, TASK

# File Refactoring

Objectives:

1. Simplify the code, unwinding the callbacks - DONE
2. Separate concerns: no expression evaluation at the read / write level - DONE
3. Make it easy to support additional file types, including the following. - DONE
4. Support writing and reading single column text files with no character translation - DONE
5. Support writing Excel friendly CSV (“CSVX”) in addition to the current pseudo-typed CSV
6. Modify TSV to replace tab, carriage return, and line feed with space on output
7. Support reading TSV

## Don’t Do

1. CREATE LOG command to create database log? DON’T DO IT – CREATE TABLE can be database vendor specific. Just run a normal RUN task with a CREATE TABLE statement to create the log table with the standard column names. Also, user can decide how wide they want to make each column.

# CheckDbp

Takes a RunDbp argument string and validates it:

1. Confirm valid syntax of the script
2. Confirm supplied arguments are correct type for the parameters
3. Check whether any argument(s) is/are a valid schedule string.

# SrvWrap

Wraps any Windows console executable as service. – DONE

# ServeDbp

Wraps a DBPA script invocation as a Windows service. This is just a batch file that uses SrvWrap.exe with a Java command line to invoke RunDbp with the script and its arguments. Presumably it is a continuously running script, e.g., with a schedule task. Starting the service starts the script, stopping the service sends an interrupt to the script.

# ManageDbp

DBPA script manager exposing a web service interface wrapped as a Windows service. Duplicates SQL Server Agent functionality, including creating schedules that invoke scripts on a scheduled basis. This is a better solution than requiring every script to implement its own scheduling.

This was formerly ControlDbp, which now has a new function.

## Capabilities

1. Upload script (web service only) – DONE
2. Validate a script – DONE
3. Download script (web service only) – DONE
4. List available scripts – DONE
5. CRUD for script launch configurations – DONE
6. Start a script from launch configuration – DONE
7. List running scripts – DONE
8. Stop a running script – DONE
9. BONUS: Generate a standard report script

TODO:

1. Rename Controller to Manager – DONE
2. APIs for managing schedules. – DONE
3. Associate each configuration with zero or more schedules. A schedules can be shared among multiple configurations. – DONE
4. Expose all public methods in RESTful API. – DONE
5. File upload and download for scripts and properties. – DONE
6. Implement scheduled runs of scripts. – DONE
7. Set up default context properties distinct from properties used by the manager. In fact, the manager only uses a database connection from a context, so it really doesn’t need a full context. It does need protection from database timeouts and stale connections, but these don’t need to be provided by the Context object. It can use the DatabaseConnection object. – DONE
8. The manager should use classic Java logging (e.g.,logback) for itself, but not for the scripts it runs.

# ControlDbp To Do

A command line program that talks via web service to ManageDbp. – DONE

## Commands

See “DBPA manager web service URIs.xlsx”. All commands listed there are implemented.