SWS: The R Application Programming Interface

Not Confidential

Doc.Ref.	:	<?>
Version	:	1.6
Date	:	11th November 2014
Authors	:	Nick Connell, Dario Fabbri, Ettore De Maio
Reviewed	:	M.Kao, R.Carleo, D.Fabbri, S.DiMichele

FAO
Viale delle Terme di Caracalla, 00153 Rome, Italy
Tel: (+39) 06 57051

Document Change Record

Issue / Revision	Date	Author	Summary of Changes
V0.1	12/09/2013	Nick Connell	Initial version
V0.2	25/09/2013	Nick Connell	Revised after extensive brainstorming of the requirements and possibilities.
V1.0	03/10/2013	Nick Connell	Revised after review.
V1.1	13/11/2013	Dario Fabbri	Some updates after the actual implementation.
V1.2	14/11/2013	Dario Fabbri	Updated the information about the variables automatically injected in the R workspace.
V1.3	08/01/2014	Dario Fabbri	Added Pivoting data structure and updated data retrieval description.
V1.4	13/03/2014	Dario Fabbri	Added block metadata APIs and description of debug infrastructure.
V1.5	14/03/2014	Nick Connell	Additional information and links for the debugging environment.
V1.6	18/11/2014	Nick Connell, Ettore De Maio	Added the GetTableData() and SetDataTable() function descriptions.

Table of Contents

1 INTRODUCTION

1.1 Background, Purpose and Scope

The Statistical Working System requirements state that R modules should be callable from within the system, with access to the database tables holding the system data.

It also states that the R should be prevented from writing directly into the core data tables, but that the R should make all changes via the main system. This allows the system to consistently maintain the change history of the data and to check user authorization and access rights.

In order to do this an API is required that encapsulates the read and write access to the database tables.

Note that the R server platform will be Linux (Redhat) rather than Microsoft Windows.

2 CONTEXT

2.1 Configuration

Each script must declare in its configuration:

- Which datasets it requires access
- It's parameters
- Whether it can be run background (on database), foreground (on a session) or both.

When a script is run it will be assigned a job ID, and will have a context information block attached to that ID. See the context block structure below.

Parameters such as the context block and any others required by the script will be passed to the script by pre-setting variable values in the R session before the script itself is run.

2.2 Run from Session

When run from a user session, the users will be presented with a pull-down list for each dataset allowing them to choose the source/target for each of those datasets:

- one of their *open* (not saved) sessions for each matching dataset
- The database

Each selected dataset source will be included in the context block accessible from the script.

When running from a session with some data supplied from the open session the script itself must be unaware as to whether the data is read/written from/to an open session or is being read/written directly from/to the database.

Note that the usual access rights will apply to data read or written from/to the database – the user's permissions will be applied. Private R modules will not be permitted to write to the database.

If an R module attempts to read data from an open session that does not contain the required data (because the session key selection is too narrow) the session should be expanded to include the required data point, but from the same timestamp period as the original session creation timestamp.

If an R module attempts to write data to an open session that does not contain the required data (because the session key selection is too narrow) thesession should be automatically expanded to include the data point.

Not confidential

3 DATA STRUCTURES

R allows the creation of custom data structures.

Examples shown below refer to Agricultural Production which has 4 axes: *area*, *item*, *element* and *year*. Other domains will have differing numbers of keys and different axis types.

3.1 The Context Block

This will be a set of R objects which contains information about the context in which the script is running. It will contain:

1. The job ID
2. The user details
3. The working directory
4. For each dataset[1] required by the script:
    a. The domain ID/Name (part of the key)
    b. The dataset ID/Name (part of the key)
    c. The key values to use
    d. The open session ID (Optional – if null the API will use the database)

The actual set of variables injected in the R workspace is the following:

Variable	Purpose
swsContext.baseRestUrl	Contains the base part of the URL that will be used internally by the API calls to perform RESTful requests directed to the main SWS application server.
swsContext.userId	The id of the user executing the R script.
swsContext.username	The system username of the user executing the R script.
swsContext.userEmail	The email address of the user executing the R script.
swsContext.token	The token (a UUID) used by the system to identify the computation execution session. Used internally by the API calls.
swsContext.executionId	The id used by the system to identify on the database the computation execution session. It is used internally by the API calls.
swsContext.computationParams	A list holding the parameters passed to the script. The list elements have a name (the parameter's name) and a value (the parameter's value). Each parameters is always passed as a string (R character data type), so in case other types are needed the R script will have to provide for explicit casting.
swsContext.datasets	A vector of objects of key class, representing the datasets on which the script has been launched, each one with the set of selected keys (if applicable). By convention, the first item of the vector is to be considered the main dataset.
swsContext.modifiedCells	This variable will only be present for interactive validation scripts. It represents the list of cells that the user has modified

[1]In this context "dataset" could also be a support data table or a mapping table

	before the validation script has been invoked. It helps the script to work as locally as possible, in order to keep the validation procedure quick and the system responsive. The variable is a data.table object with a column for each key applicable to the dataset. Each row holds the full set of coordinates of a modified cell.

3.2 Key class

A defined class object capable of contain multiple values for each key. E.g. for AP the key would have 6 elements: domain, dataset, area, item, element and year. Each of the true key elements (area, item, element, and year) will be able to contain multiple values.

In addition the class will contain a session ID which can be set to the ID of one of the user's open sessions to be used as the data source/target when accessing the particular dataset. If the session ID is null the API will use the database directly.

The session ID can be null because the user specified the database rather than an open session as the source for the dataset.

In the API it has been defined an S4 class modelled after the following signature:

```
DatasetKey <- setClass("DatasetKey",
    representation(
        domain = "character",
        dataset = "character",
        dimensions = "list",
        sessionId = "integer"))
```

Hence to access the first dataset name the following syntax can be used:

```
    primary_domain = swsContext.datasets[[1]]@domain
```

The slot named "dimensions" has to be a list of objects of class Dimension. This class has been defined as:

```
Dimension <- setClass("Dimension",
    representation(
        name = "character",
        keys = "character"))
```

The "keys" slot can contain a vector of character strings defining all the keys selected for the specified dimension. Hence the first dataset's first dimension key may be accessed (as a vector) using the syntax:

```
    Key1 = swsContext.datasets[[1]]@dimensions[[1]]
```

then the key name and codes (as a vector) vector can be accessed with:

```
    name1 = Key1@name
    codes1 = Key1@keys
```

3.3 Pivoting class

It is a class representing a single dimension used for pivoting specification. It is used in data retrieval methods to indicate the requested data pivoting. The class instances contain two values:

- *code*, that is the id of the referred dimension (i.e. geographicAreaM49)
- *ascending*, a Boolean value indicating the sort direction to be applied to the specified dimension

In order to properly specify a valid pivoting, a vector of Pivoting objects has to be passed to the data retrieval code. The vector must include all the dimensions that are defined for the target dataset. The sequence of the Pivoting objects describe the requested order of extraction of the corresponding dimensions and, in case of denormalized extractions, it puts on the columns of the returned data.table the values corresponding to the last Pivoting dimension.

The S4 definition of the class is the following:

```
Pivoting <- setClass("Pivoting",
    representation(
        code = "character",
        ascending = "logical"),
    prototype(ascending = TRUE))
```

3.4 Block metadata classes

Block metadata are represented with specialized classes. For a given set of keys, identifying a block of data in a dataset, it is possible to have more than one block metadata defined. The class representing this set of block metadata is the following:

```
BlockMetadataSet <- setClass("BlockMetadataSet",
    representation(
        keyDefinitions = "list",
        blockMetadata = "list"))
```

The “keyDefinitions” slot holds the list of dimensions that characterize the underlying dataset. Each list element has to be an instance of the following S4 class:

```
KeyDefinition <- setClass("KeyDefinition",
    representation(
        code = "character",
        description = "character",
        type = "character"))
```

The “code” slot holds the name of the dimension in the system (for instance, for the geographic dimension in the Agriculture Production dataset would assume the value of geographicAreaM49), while the “description” slot holds the extended description.

The “type” slot defines the type of the dimension. The possible values are:

- normal
- measurementUnit (for the measure dimensions)
- time (for the time dimension)

Getting back to the BlockMetadataSet class, the “blockMetadata” slot is a list of instances of the following S4 class:

```
BlockMetadata <- setClass("BlockMetadata",
    representation(
        blockId = "numeric",
        selection = "list",
        metadata = "Metadata"))
```

The class represents a single block metadata. The “blockId” slot holds the unique ID of the block metadata. The GetBlockMetadata call returns this information in order to allow update operation on block metadata, by specifying an existing ID in a SaveBlockMetadata call. Omitting this information is a SaveBlockMetadata call will create a new block metadata.

The “selection” slot is a list of instances of the Dimension class, described above in the section dedicated to the key class for data and metadata calls.

The “metadata” slot is an instance of the following class:

```
Metadata <- setClass("Metadata",
    representation(
        code = "character",
        description = "character",
        language = "character",
        elements = "list"))
```

It represents the metadata associated to the block of data specified by the keys held by the “selection” slot described above. The “code” slot is the code identifying the metadata type represented (for instance ACCURACY or GENERAL), while “description” holds the description of the metadata type.

The “language” slot can be one of:

- ar - Arabic
- en – English
- es – Spanish
- fr – French
- ru - Russian
- zh – Chinese

The “elements” slot is a list of instances of the following class:

```
MetadataElement <- setClass("MetadataElement",
    representation(
        code = "character",
        description = "character",
        value = "character"))
```

In this case the “code” slot represents the metadata element type (for instance SAMPLING_ERROR or COMMENT), with the “description” holding the extended description of the metadata element.

The “value” slot holds the value associated to the metadata element.

3.5 Timeseries

This will not be a special structure but will make use of the data.table R type (a more efficient extension of DataFrame).

The content will consist of the key followed by the value(s) and the flags. This data can be requested in either normalized or denormalized form. E.g. the data.table containing normalized data would have the form:

Area	Item	Ele	Year	Value	Flag1	Flag2
250	C0111	5510	2009	12345	F	N
250	C0111	5510	2010	23451	E	
etc.						

The key columns name will be that defined by the system in the dataset configuration. In the normalized format, the key columns will always precede the observation value column. The observation value column will always have the “Value” name (first letter uppercase). The flag columns, if present, will follow the observation value and their name will be the same as defined in the dataset configuration.

For data denormalized along the time axis the form would be:

Area	Item	Ele	Value_2009	Flag1_2009	Flag2_2009	Value_2010	Flag1_2010	Etc.
250	C0111	5510	12345	F	N	23451	E	
250	C0112	5510	6789	E	N	8769		
etc.								

Which axis is denormalized would be selectable by the script writer via the API call. Also the pivoting and the sorting order of each key column can be specified through the same API call parameter.

The data retrieval API call will allow requesting metadata together with data. This is supported only in normalized format. If metadata are requested the expected data table format will be the following:

Area	Item	Ele	Year	Metadata	Metadata_Language	Metadata_Group	Metadata_Element	Metadata_Value	Value	Flag1	Flag2
250	C011 1	5510	2009	SOUR CE	en	1	DATE	2010	100	F	
250	C011 1	5510	2009	SOUR CE	en	1	COM MENT	Unkn own	200	E	
250	C011 1	5510	2009	SOUR CE	fr	2	COM MENT	Differ ent	300	F	N
250	C011 2	5510	2009	SOUR CE	zh	3	NAM E	Grain world	400		
etc.											

In this format, the key columns will precede every other column, like the above described normalized format. The metadata columns will immediately follow the key columns. The name of these columns will always be fixed:

- Metadata (the id of the metadata)
- Metadata_Language (the language in which the metadata is expressed)
- Metadata_Group (the grouping code to distinguish between different metadata with the same id put one next to the other)
- Metadata_Element (the id of the element within the metadata)
- Metadata_Value (the string representing the value of the metadata element)

Following the metadata block there are the columns for the observation value and the flags.

3.6 History

This structure will be a data.table containing the history for one or more cells.

The data will always be normalized.

The keys will be the usual for the dataset plus a version number, with the highest number being the most recent version of the cell's value. In addition to the values, the flags and the metadata the data will contain the change data (timestamp, user, etc.).

The list of returned columns will then be:

- key columns
- Version
- StartDate (the timestamp of the beginning of the validity period of the version)
- EndDate (the timestamp of the end of the validity period, NA for the current version)
- Metadata
- Metadata_Language
- Metadata_Group

- Metadata_Element
- Metadata_Value
- Value
- flag columns

3.7 Map

This data.table will contain data from a mapping table. These are tables used during the conversion of a dataset’s data from one code system to another, along a single axis.

The tables have a from/to code pair for the target axis, while the mapping can be tuned using all other axes of the dataset. Each map entry includes a split ration to handle one-to-many mappings and a factor where some scaling or unit adjustment is required. E.g.

From	To	Area	Ele	Year	Split	Factor	Comment
H1001	C0111	*	*	*	1.00	1.00	Simple 1-to-1 map
H1005	C0112	*	*	*	1.00	1.00	
H100610	C0113	*	*	1990	1.00	1.00	Year specific
H100620	C0113	*	*	1990	1.00	0.80	n-to-1 with scaling factor

Each mapping table will be specific to a single mapping operation hence the from/to keys will be from known axes (the example above is commodities, HS to CPC); the script writer is expected to be familiar with the table and axes involved in a given mapping.

3.8 Code Tree

This data.table will contain data describing the tree structure of an axis. Each row will contain a parent code and a list of children. E.g.

Parent	Children (List)
C01	C011, C012,C013
C011	C0112,C0113
C012	C0121,C0122,C0123,C0124
Etc.	

3.9 Validation

This will also make use of the R data.table.

Scripts must use this structure to report validation failures back to the system for logging and display.

This data must always be normalized.

The content will consist of the key followed by the validation failure severity and a descriptive comment (created by the R script).

Area	Item	Ele	Year	Severity	Description
250	C0111	5510	2009	1	The value deviates by >10% but <20% from the previous year value.
250	C0111	5510	2010	3	The value indicates an excessive yield given the area planted: <some value> hectares
etc.					

4 R INTERFACE METHODS

4.1 Data Read

4.1.1 GetData (key, flags=TRUE, normalized = TRUE, metadata = FALSE, pivoting)

Parameters:

key: A key_class object

flags: [optional, default=TRUE] If true then flags are returned otherwise not

normalized: [optional, default=TRUE] If true then data are returned in normalized format, otherwise the format is denormalized

metadata: [optional, default=FALSE] If true then metadata are returned otherwise not

pivoting: [optional, if omitted the datatset default is assumed] A pivoting class object vector

Return:

A data table containing the data matching the key (may be empty).

If the denormalized value is not set then the data is normalized; if set the data is denormalized along the axis specified.

If the pivoting vector is present, the dimensions are extracted in the specified order and applying the requested sort direction. This affects both normalized and denormalized extractions. For normalized extractions only the order of the dimensions is influenced, while for denormalized extraction the effect is more evident since the last dimension specified in the pivoting vector gets its values developed along the column of the generated data.table result object.

4.1.2 GetData (tag, flags=”<Y/N>”, denormalized=”<axis-name>”)

Parameters:

tag: A data tag name

flags: [optional, default=”Y”] If true then flags are returned otherwise not

denormalized: [optional] If set it must be the name of an axis along which to denormalize the data.

Return:

A data table containing the data matching the tag (may be empty).

The tag defines a snapshot of data including the keys and timestamp.

If the denormalized value is not set then the data is normalized; if set the data is denormalized along the axis specified.

4.1.3 GetHistory (key, pivoting)

Parameters:

key: A key_class object

pivoting: [optional, if omitted the datatset default is assumed] A pivoting class object

Return: A data table of observation objects containing the values and flags through history with the associated history metadata.

4.1.4 GetMapping (key)

Parameters:

key: A key object defining the required entries.

Return:

A data table containing the map entries matching the key (may be empty).

This is expected to be used only by advanced script writers. The key will be a custom key layout matching the axes of the mapping table.

The key may make use of wildcards to widen the number of results matched.

Map tables are always read-only.

4.1.5 GetSupportData (key)

Parameters:

key: A key object defining the required entries.

Return:

A data table containing the support data entries matching the key (may be empty).

This is expected to be used only by advanced script writers. The key will be a custom key layout matching the axes of the support table; the script writer is expected to be familiar with the layout and content of the table.

The key may make use of wildcards to widen the number of results matched.

Support tables are always read-only.

4.1.6 GetCodeList (domain, dataset, keyname, IDs)

Parameters:

domain: The domain for which the code list is required

dataset: The dataset for which the code list is required

keyname: The name of the key for which the code list is required

IDs: [optional] A list of codes for which the key data is required

Return:

A data table containing the key codes and matching labels.

Example:

```
ds=swsContext.datasets[[1]]
key1=ds@dimensions[[1]]
tree1=GetCodeTree(ds@domain,ds@dataset,key1@name,c("4", "6"))
```

4.1.7 GetCodeTree (domain, dataset, keyname, roots)

Parameters:

domain: The domain for which the code tree is required

dataset: The dataset for which the code tree is required

keyname: The name of the key for which the code tree is required

roots: [optional] A list of root codes for the tree. Default is all those with no parent node.

Return:

A data table containing the parent codes and lists of child codes in the tree (see the Code Tree structure above).

Example:

```
ds=swsContext.datasets[[1]]
```

```
key1=ds@dimensions[[1]]
tree1=GetCodeTree(ds@domain,ds@dataset,key1@name,c("953"))
```
[2]

4.1.8 GetBlockMetadata(key)

Parameters:

key: A key_class object

Return:

A BlockMetadataSet object holding all the block metadata found in the specified portion of data. All the block metadata having a keys definition intersecting the keys passed as an argument to this call are returned.

[2] Code 953 is the world from the Agricultural Production domain area code list, flat structure.

4.2 Data Write

Note: The column headings in data structures submitted to the API to be written back to the database or session must be the same as those obtained when the data is read. The script writer is expected to know what these column headings are for the output dataset concerned.

4.2.1 SaveData (domain, dataset, data, metadata)

Parameters:

domain: The target domain identifier

dataset: The target dataset identifier

data: [optional] A data table object containing keys, data and flags.

metadata: [optional] A data table object containing metadata

The data may be normalized or denormalized. It must conform to the dataset data table definition described above with the addition of an extra column containing comments to attach to the data values.

The metadata is always normalized.

4.2.2 SaveValidation (domain, dataset, validation)

Parameters:

domain: The target domain identifier

dataset: The target dataset identifier

validation: A data table object containing keys and validation data.

The data table must conform to the validation data table definition described above.

4.2.3 DeleteValues (domain, dataset, key, comment)

Parameters:

domain: The target domain identifier

dataset: The target dataset identifier

key: A key_class object defining observations to delete.

comment: A mandatory comment on the delete operation

4.2.4 ClearMetadata(domain, dataset, key)

Remove all metadata from a set of values.

Parameters:

domain: The target domain

dataset: The target dataset

key: The key sets of the values from which to clear metadata.

4.2.5 SaveBlockMetadata(domain, dataset, blockMetadata)

Parameters:

domain: The target domain identifier

dataset: The target dataset identifier

blockMetadata: Can be specified using an instance of the BlockMetadataSet class, or through a list of BlockMetadata objects.

4.3 Database Access to Private Schema

Since it is impossible to foresee and handle every possible database table layout and content, the faosws package includes functions to allow access to ad-hoc tables in the PostgreSQL database underlying the system. These functions handle all the connection and formatting tasks, leaving the users' code uncluttered. Tables being accessed must have been set up and have appropriate access permissions set – this is a manual, system administrator task performed directly on the database.

Access is achieved by using the following two functions:

- GetTableData
- SetTableData

4.3.1 GetTableData

This function allows the retrieval of data from a database table in a specific schema. The user is expected to know the location and layout of the table.

Usage:

```
Data <- GetTableData (schemaName, tableName, whereClause = NULL,
selectColumns = NULL)
```

This queries the specified table and returns a data.table object containing the specified data, or NULL if no matching data was found. The default of both *whereClause* and *selectColumns* are omitted is to return the entire content of the table.

If the SQL request and resulting query is incorrect, SQL error details are pushed into the error message list (using stop()) and can be retrieved using geterrmessage();execution is aborted.

Arguments:

- **schemaName**, MANDATORY: a string containing the name of the schema to be accessed for the query;
- **tableName**, MANDATORY: a string containing the name of the table to be read by the query;
- **whereClause**, OPTIONAL: a string containing whichever SQL clauses to take place after the "FROM" section (usually WHERE conditions). Example values:
  - "WHERE (id IN (1,2,3,4) AND descr = 'A') OR (id = 5 AND status = 0) "
  - "ORDER BY descr DESC"
  - "WHERE descr = 'aaa' ORDER BY id LIMIT 5"

  if NULL, nothing is appended in the SQL query composition after the FROM section, in which case all rows will be returned from the table;
- **selectColumns**, OPTIONAL: the list of columns to be returned by the query. it can be:
  - NULL (equivalent to "SELECT *", returning all columns in their native order)
  - a list of strings, such as: "list('id', 'descr')", which should match column names of the table being queried

The data.table column names are set to the names of the database table columns returned.

The software converts between database column types and R data types as follows (java /PostgreSQL DB /R):

PostgrSQL	R
int8	numeric
int4	integer
int2	integer
numeric	numeric
float4	numeric
float8	numeric
varchar	character
bpchar	character
text	character
bool	logic
date	Date
timestamp	POSIXct (no timezone)

4.3.2 SetTableData

The function allows altering the content of a table for a specified schema

Usage:

```
SetTableData (schemaName, tableName, data, replace = FALSE, purge = FALSE,
purgeFilter = NULL)
```

This sends rows contained in a "data.table" to the system to be checked and applied to the specified table. The action taken when supplied keys match existing records is controlled by the *replace* parameter. It is also possible to delete all/some records prior to saving the new data using the *purge* and *purgeFilter* parameters.

- **schemaName**, MANDATORY: a string containing the name of the schema to be accessed for the query;
- **tableName**, MANDATORY: a string containing the name of the table to be read by the query;
- **data**, MANDATORY: a non empty data.table containing records with values to be appended/replaced on the table. It is not necessary supply all the columns that are in the database table, BUT at all columns that compose the primary key (if any) must be supplied. If no primary key is set for the table, all the records will be appended as they are. Column names in the data.table must match the column names in the database table.
- **replace**, DEFAULT=FALSE: if TRUE, when a row in the data.table matches (on primary key) a corresponding record in the database table, the database record is UPDATED from the data.table entry; if FALSE such matching records will be skipped (i.e. not updated in the database table).
- **purge**, DEFAULT=FALSE: if TRUE, before any inserts/updates, rows are deleted from the database table; which rows depends on the *purgeFilter* parameter (see below).
- **purgeFilter**, DEFAULT=NULL: can be specified only for purge=TRUE. This allows the user to specify a filter for the deletion in the form of an SQL WHERE clause.

Specifying *purge=TRUE* and *purgeFilter=NULL* will remove all rows from the table before the update.

The system checks if the data passed as data.table are either conforming to the expected column type to which any cell refers, or can be parsed (e.g. a string containing "1" to the number 1, or a string containing "01-01-2001" to the corresponding date by assuming the format "dd-MM-yyyy"). If the conversion fails, or any other SQL error is arisen by the underlying DB (e.g. a column constraint for NOT NULL), the function aborts (using stop()) with the error message being placed on the R error message queue.

If otherwise the function succeed, statistics on the updates are returned, indicating how many records have been:

- deleted (if purge was TRUE)
- updated (if replace was TRUE and rec found by key)
- inserted (if rec not found by key)
- skipped (if replace was FALSE and rec found by key)

SetDataTable uses the same data types mapping as GetTableData (see above).

4.3.3 RPostgreSQL direct access

- Direct access to the RPostgreSQL database is deprecated by the introduction of the GetTableData and SetTableData, which allow a more secure and streamlined way to access ad-hoc data. However the setup will be maintain for some time for backwards compatibility. If used, the package usage in the content of the SWS system has to conform to the following directives: Installing the RPostgreSQL package on the R server.
- Setting up R variables in the R environment for the database location and connection information.
- Providing a private schemas as required (initially one named `ess`).
- Providing a read-only user (named `ess_user`)
- Providing R module developers with read/write users for the schema (same as their network login names, minus the domain).

Example connection and query code:

```
Con <- dbConnect(PostgreSQL(), user=R_SWS_DATABASE_USER,
     password=R_SWS_DATABASE_USER_PASSWD, dbname=R_SWS_DATABASE_NAME,
     host=R_SWS_DATABASE_HOST)

out <-dbGetQuery(Con, "select count(*) from ess.item_agg_grp")
country <-dbGetQuery(Con, "select count(*) from ess.area_yr_agg_grp")
```

Warning: **DO NOT** use hard coded connection or user names in the R modules, only use these variables. Connection details will differ between QA and production environments, while hard coded user names and passwords compromise security.

When developing an R module using this facility you should add the following lines to your local $HOME/.Rprofile:

```
R_SWS_DATABASE_NAME="sws_test_upload"
R_SWS_DATABASE_HOST="hqlqasws2.hq.un.fao.org"
R_SWS_DATABASE_USER="ess_user"
R_SWS_DATABASE_USER_PASSWD="ess_user"
```

The `ess_user` has read-only access to the database tables in the `ess` schema. For development and to load tables you may substitute your own user name and password in your `.Rprofile` file which will have read-write access. Your user name will be the same as your standard system login (without the domain name) however your password will be specifically created on request to the SWS administrators and sent to you.

Note this facility will only function in a development environment when inside the FAO firewall.

5 WRITING AND DEBUGGING R SCRIPTS

The process of writing an R script to be deployed in the Statistical Working System using the R plugin system poses some challenges. The general idea is to have the script author working on her machine, using a standalone R installation; once the script is ready (or good enough) to be deployed it can be uploaded into the SWS and tested using the available tools.

The R developer must be able to test the above described API calls locally before deploying the script in the SWS, otherwise the write/test cycle might easily become very difficult, especially since when the script executes in the R server within the system the developer has no access to the intermediate states or the output log and cannot see the results of the data call.

It is therefore necessary to have a facility that allows a direct and powerful interaction with the SWS system in the code writing and debugging phase of the development.

This chapter describes this facility.

The various files mentioned here and required for setting up your R environment can be found on the project Workspace here.

5.1 Setting up the client environment

5.1.1 Installation of R

It is necessary to have R installed on the client machine; the version must to be 3.0.0 or higher.

It is also possible to use RStudio, the setup process should be similar to the one described here for the plain R installation, but a full guide on this is outside the scope of the present document.

5.1.2 Installation of prerequisite packages

The R/SWS interface is implemented in an R package called "faosws". This package depends on a number of other packages available in the CRAN which must be pre-installed in your R instance.

RCurl

It is the package responsible for the REST call performed by the R API to the application server of the Statistical Working System. It wraps an immensely famous library called libcurl, that needs to be available on the system before installing the RCurl package. Since its installation depends on the platform, it is necessary to refer to the RCurl page in the CRAN for the installation details.

http://cran.mirror.garr.it/mirrors/CRAN/web/packages/RCurl

RJSONIO

Since the data exchanged between the R API and the SWS application server is in JSON format, the RJSONIO package has been used to convert JSON data to and from native R data structures.

http://cran.mirror.garr.it/mirrors/CRAN/web/packages/RJSONIO

data.table

It is an extension of the standard data.frame, over which offers better performances in terms of data manipulation and sorting.

http://cran.mirror.garr.it/mirrors/CRAN/web/packages/data.table

The data.table package itself is dependent on the packages Rcpp, plyr and reshape2; these also require installation in your R environment.

5.1.3 Installation of the faosws package

The R API package especially created for the Statistical Working System needs to be installed on the developer client. It can be obtained as a package through the CIO or in source code on the FAO Subversion server (having the right privileges) at the following URL:

https://svn.fao.org/projects/SWS/prj/R/trunk/faosws/

5.1.4 *Installation of the client certificate*

The R API communicates with the SWS application server through REST services protected using HTTPS and X.509 certificates. In order to be able to perform service calls, the API must have access to a recognized client certificate locally installed.

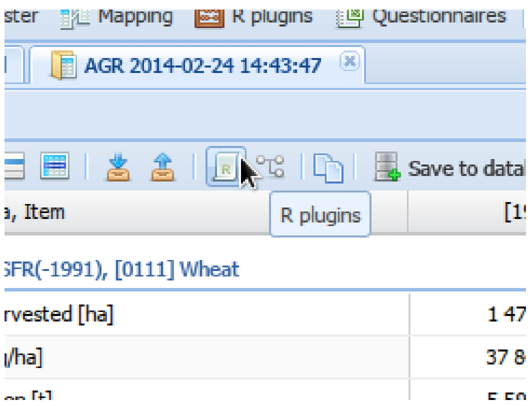
Special certificates for R script testing will be issued by the CIO. Once obtained the certificate and its key, they have to be saved in the user's local home folder under an .R subfolder (on a Unix machine this ~/.R, on a Windows XP/7/8 computer this is C:\Users\<username>\My Documents\.R). The two files are named:

- client.crt
- client.key

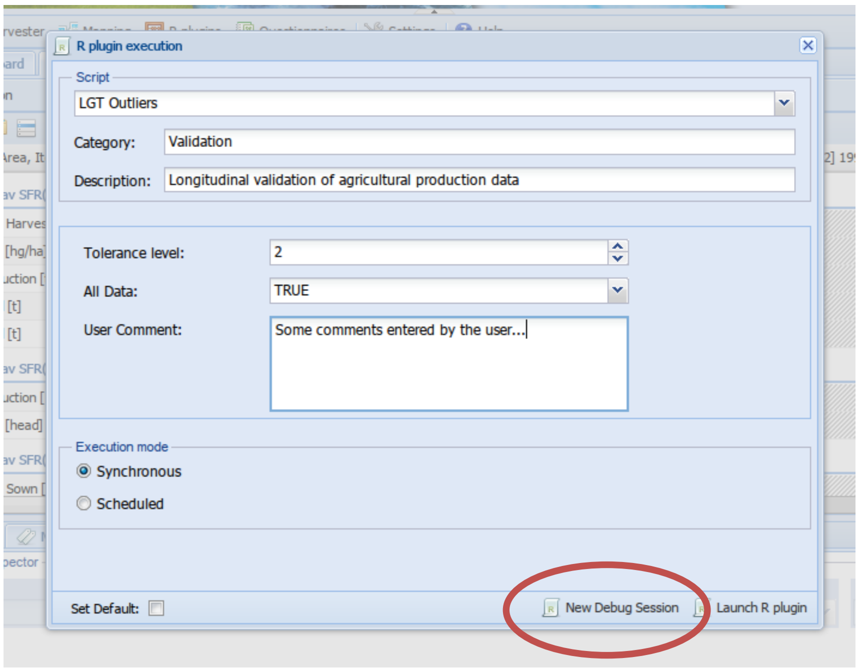
and can be found in the workspace directory mentioned above.

5.2 Launching a test session

In order to fully simulate the environment that the script will encounter during its normal execution, it is necessary that the user performs a number of choices on the system even if she wants to start a test session. The R script execution panel is opened by clicking on the button depicted in the following screenshot:



The user sets the parameters just like she would in a normal script execution:


R plugin execution

Script: LGT Outliers

Category: Validation

Description: Longitudinal validation of agricultural production data

Tolerance level: 2

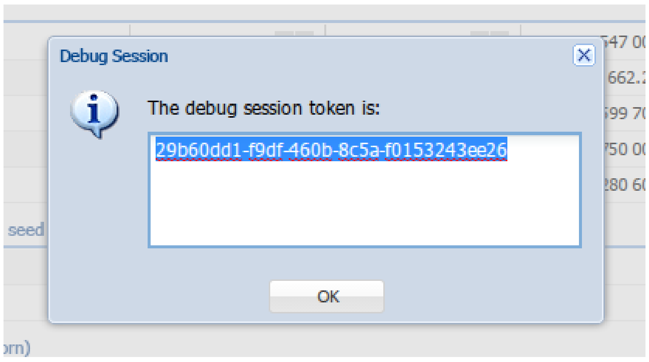
All Data: TRUE

User Comment: Some comments entered by the user...

Execution mode: (● Synchronous) ○ Scheduled

Set Default: ☐

Once the script is chosen and the parameters fully set, the user selects the “New debug session” option highlighted in the picture. The system creates all the data structures supporting the script execution and marks them for the test/debug session. A token generated for the created session is reported to the user:


Debug Session

The debug session token is:

29b60dd1-f9df-460b-8c5a-f0153243ee26

This token will be used by the script author for the R environment setup, as shown in the following chapter.

In the described situation, the R script was already fully deployed and configured in the system. When the author wants to start the development of a new script, this has to be configured in the system using the normal deploy mechanism. The only difference is that the deployed ZIP file, instead of being composed of an XML descriptor and the R code, only contains the XML descriptor.

5.3 Authoring the script

With the environment properly set up and a test session configured on the Statistical Working System, the author can start the development of the script on her machine. The only thing to do is include a special API method call at the beginning of the script, in order to load from the server all the environment information that will enable the correct execution of the subsequent API calls.

Following is a snippet of R script using this special call:

```
require("faosws")

# Setup test environment
#
GetTestEnvironment(
    "https://hqlqasws1:8181/sws",
    " 29b60dd1-f9df-460b-8c5a-f0153243ee26")

# Execute the data call.
#
data <- GetData(
    key = swsContext.datasets[[1]],
    flags = TRUE, normalized = TRUE)
print(data)
```

The code shows the usage of the *GetTestEnvironment* method that performs the test environment initialization. It can be noted that the base part of the server URL needs to be passed as the first parameter of the method; in the example the QA server is referred but for the real value to be used it is necessary to consult with CIO. The second argument passed is the test session token obtained from the SWS user interface as described above.

It is also interesting to observe that the *GetData* call uses the *swsContext.datasets* global variable to query the SWS using exactly the keys selected by the user in it R execution session creation; this is a normal pattern for a real R script to be deployed on the SWS. The global variable, along with all the others described in the first section of the present document, is set by the *GetTestEnvironment* call.

Before packaging the script for deployment on the SWS, it is necessary to remove or comment the *GetTestEnvironment* call.