SWS: The R Application Programming Interface

Not Confidential

|  |  |  |
| --- | --- | --- |
| FAO  Viale delle Terme di Caracalla, 00153 Rome, Italy  Tel: (+39) 06 57051  Fax: (+39) 06 570 53152  http://www.fao.org |  |  |
| Doc.Ref. | : | <?> |
| Version | : | 1.7 |
| Date | : | 11th November 2014 |
| Authors | : | Nick Connell, Dario Fabbri, Ettore De Maio |
| Reviewed | : | M.Kao, R.Carleo, D.Fabbri, S.DiMichele |

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. |
| V1.7 | 23/12/2014 | Nick Connell, Ettore De Maio | Added an overview section. Extensively expanded the “getting started” information. Added the multiple R files capability. |

Table of Contents

[1 Introduction 6](#_Toc407112091)

[1.1 Background, Purpose and Scope 6](#_Toc407112092)

[1.2 Acronyms 6](#_Toc407112093)

[2 An overview of R in the SWS 7](#_Toc407112094)

[3 Context 8](#_Toc407112095)

[3.1 Configuration 8](#_Toc407112096)

[3.2 Runtime 8](#_Toc407112097)

[3.3 Run from Session 8](#_Toc407112098)

[3.4 Permissions and Access 9](#_Toc407112099)

[4 Data Structures 10](#_Toc407112100)

[4.1 The Context Block 10](#_Toc407112101)

[4.2 Key class 11](#_Toc407112102)

[4.3 Pivoting class 12](#_Toc407112103)

[4.4 Block metadata classes 12](#_Toc407112104)

[4.5 Timeseries 14](#_Toc407112105)

[4.6 History 16](#_Toc407112106)

[4.7 Map 18](#_Toc407112107)

[4.8 Code Tree 18](#_Toc407112108)

[4.9 Validation 19](#_Toc407112109)

[5 R Interface Methods 20](#_Toc407112110)

[5.1 Data Read 20](#_Toc407112111)

[5.1.1 GetData (key, flags=TRUE, normalized = TRUE, metadata = FALSE, pivoting) 20](#_Toc407112112)

[5.1.2 GetData (tag, flags=”<Y/N>”, denormalized=”<axis-name>”) 20](#_Toc407112113)

[5.1.3 GetHistory (key, pivoting) 21](#_Toc407112114)

[5.1.4 GetMapping (key) 21](#_Toc407112115)

[5.1.5 GetSupportData (key) 22](#_Toc407112116)

[5.1.6 GetCodeList (domain, dataset, keyname, IDs) 22](#_Toc407112117)

[5.1.7 GetCodeTree (domain, dataset, keyname, roots) 22](#_Toc407112118)

[5.1.8 GetBlockMetadata(key) 23](#_Toc407112119)

[5.2 Data Write 24](#_Toc407112120)

[5.2.1 SaveData (domain, dataset, data, metadata) 24](#_Toc407112121)

[5.2.2 SaveValidation (domain, dataset, validation) 24](#_Toc407112122)

[5.2.3 DeleteValues (domain, dataset, key, comment) 24](#_Toc407112123)

[5.2.4 ClearMetadata(domain, dataset, key) 24](#_Toc407112124)

[5.2.5 SaveBlockMetadata(domain, dataset, blockMetadata) 25](#_Toc407112125)

[5.3 Database Access to Private Schema 26](#_Toc407112126)

[5.3.1 GetTableData 26](#_Toc407112127)

[5.3.2 SetTableData 27](#_Toc407112128)

[6 Writing and debugging R scripts 29](#_Toc407112129)

[6.1 Introduction to Development 29](#_Toc407112130)

[6.1.1 Steps 29](#_Toc407112131)

[6.2 Setting up the client environment 30](#_Toc407112132)

[6.2.1 Installation of R 30](#_Toc407112133)

[6.2.2 Installation of the client certificate 30](#_Toc407112134)

[6.2.3 Installation of prerequisite packages 31](#_Toc407112135)

[6.2.4 Installation of the faosws package 31](#_Toc407112136)

[6.3 Starting development of a new modules 32](#_Toc407112137)

[6.3.1 Create and upload the XML configuration file 32](#_Toc407112138)

[6.3.2 Launching a test session 32](#_Toc407112139)

[6.3.3 Authoring the script 35](#_Toc407112140)

[6.3.4 Deploying a module 36](#_Toc407112141)

[6.3.5 Advanced configuration 37](#_Toc407112142)

# Introduction

## Background, Purpose and Scope

The Statistical Working System (SWS) 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.

## Acronyms

SWS: Statistical Working System

API: Application Programming Interface (something that allows two bits of software to communicate)

R-API: The SWS R API (A set of function calls)

RHEL: Red Hat Enterprise Linux (A server grade linux operating system)

VM: Virtual Machine

XML: eXtensible Markup Language (described as self-describing data)

XSD: XML Schema Definition (some rules for what XML can contain)

# An overview of R in the SWS

The R code will use various functions supplied by the R-API packages to access the data in the SWS database. The functions are documented here.

When an R module is run within the SWS (or in debug mode), the system supplies information in the R environment about parameter values and key selections made by the user. The variables used to contain this information are also described here.

R modules running within the system are executed in an [Rserve](http://rforge.net/Rserve/) container running on a dedicated virtual machine which forms part of the SWS platform. The VM is running RHEL 6.5.

Since all data is accessed through the API function calls, the SWS automatically maintains metadata for changed cells, such as date, time, user, etc. It also enforces permissions on the data as applied to the user running the module.

Ad-hoc tables can be created in the database by the administrators and accessed via the R, however mostly this is discouraged since such tables are not visible via the SWS user interface, only via the R modules. R modules are expected to access SWS datasets and metadata (which can all be seen via the user interface).

An R module ready to be included in the SWS consists of a zip file containing:

1. An XML configuration file which describes the module to the SWS and the end users.
2. The R main script.
3. A directory containing additional R functions called by the main script.

The dataset configuration information is required to access the data in the R modules, this is defined by the SWS administrators, and will be posted on the [Userspace](http://workspace.fao.org/tc/sws/userspace/SitePages/Home.aspx) collaboration web-site. This information includes:

* The domain name
* The dataset name
* The dimension names
* The observation status flag names

A development/debugging mode is available, where the R can be run in a developer’s personal development environment (i.e. on their local computer) yet still use the API, and have the appropriate environment set up (as if the module was actually running within the SWS). This facility requires access to the SWS main server; such access from outside the FAO firewall requires additional permissions (which can be requested).

R modules uploaded by an ordinary user (i.e. not an SWS administrator) can only change values in a user session, not in the main database. These are referred to as ***private*** modules, though this is a slight misnomer, since the uploader can grant permissions to any user to run such modules. ***Private*** module names appear in the list prefixed with the uploader’s login name.

Administrators can upload modules and mark them as ***core***. This status allows them to make changes to the database directly. These are expected to be modules developed by statisticians, to perform core operational computations (e.g. the SUA/FBS computation) but to be run by clerks as part of their day-to-day work.

# Context

## Configuration

Each script must declare in its configuration in an XML file:

* Its name and description (this is what the users will see in the user interface)
* Which datasets it requires access to
* Its parameters
* Whether it can be run background (on database), foreground (on a session) or both.

An annotated template XML configuration file and the schema XSD can be found [here](http://workspace.fao.org/tc/sws/userspace/Shared%20Documents/R%20Development/RConfigurationTemplate.xml) and [here](http://workspace.fao.org/tc/sws/userspace/Shared%20Documents/R%20Development/rScript.xsd), respectively. The schema file will only be useful if you have access to a validating XML editor/checker. Note that the template XML file is deliberately badly formed to force R developers to fill in fields as required.

## Runtime

When a script is run it will be assigned a job ID, and will have a context information block attached to that ID. The context block contains information such as the user name, and email, what parameter values were set, and which dimension keys were selected by the user; its structure is described 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.

In addition the Rserve has a number of pre-set variables, some of which are described below. One of these gives the local server path to a shared drive on the server:

R\_SWS\_SHARE\_PATH

## Run from Session

When run from a user session, the users will be presented with a pull-down list for each dataset described in the module XML configuration allowing them to choose the source/target for each of those datasets. This can be either:

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

Each of the user’s selected dataset sources will be included in the context block accessible from the script via the variables described below.

When running from a session with some data supplied from the open session the script itself does not need to aware as to whether the data is read/written from/to an open session or is being read/written directly from/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 will be automatically expanded by the SWS 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) the session will be automatically expanded to include the data point.

## Permissions and Access

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.

# 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.

## 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]](#footnote-1) required by the script:
   1. The domain ID/Name (part of the key)
   2. The dataset ID/Name (part of the key)
   3. The key values to use
   4. 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 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. |

## 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

## 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))

## 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.

## 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 | C0111 | 5510 | 2009 | SOURCE | en | 1 | DATE | 2010 | 100 | F |  |
| 250 | C0111 | 5510 | 2009 | SOURCE | en | 1 | COMMENT | Unknown | 200 | E |  |
| 250 | C0111 | 5510 | 2009 | SOURCE | fr | 2 | COMMENT | Different | 300 | F | N |
| 250 | C0112 | 5510 | 2009 | SOURCE | zh | 3 | NAME | 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.

## 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

## 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.

## 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. |  |

## 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. |  |  |  |  |  |

# R Interface Methods

## Data Read

### 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.

**Note:** The *key* parameter is expected to be one of the dataset definitions passed via the *swsContext.datasets* context variable. If this is the case then the data passed back may be either from the database, or from an open user session (according to the user’s choice when running the module). However is it is required to force a read from the database, or if a dataset not listed in the module configuration is to be accessed a key may be constructed “manually” by the R code or the R may simply set the *sessionID* variable to NULL.

### 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.

### 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.

### 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.

### 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.

### 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”))

### 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]](#footnote-2)

### 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.

## 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.

### 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.

### 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.

### 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

### 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.

### 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.

## 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

### 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) |

### 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).

# Writing and debugging R scripts

## Introduction to Development

The process of writing an R script to be deployed in the Statistical Working System using the R plugin system appears complex when first attempted, however after the initial setup of a module in development mode, the SWS development environment is relatively unobtrusive.

The general idea is to have the script developer performing the initial development and debug on their own machine, using their preferred R development environment; once the script is ready (or good enough) to be deployed it can be uploaded into the SWS and tested in situ.

To achieve this the SWS provides two things:

1. An environment setup facility which emulates the module being embedded in the SWS system.
2. A remote access facility which allows the developer to use the R-API calls directly from their own development machine/environment.

### Steps

1. Install the HTTPS certificates (one time only).
2. Install the FAO SWS packages and dependencies (one time only).
3. Create and upload the XML configuration file.
4. Generate debug token.
5. Create the R module with the SWS development environment setup function call.
6. Start developing.

An R module ready for upload to the SWS consists of a zip file containing:

1. An XML file which describes the module to the system, and the end users.
2. The R main module (the one that the SWS will execute).
3. A directory containing additional R function called by the main module.

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](http://workspace.fao.org/cio/projects/esstssws/Shared%20Documents/RDevelopment).

## Setting up the client environment

### 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.

### Installation of the client certificate

The R API communicates with the SWS application server through 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 Userspace web-site in a zipped directory called *Certificate.R\_Directory* here:

<http://workspace.fao.org/tc/sws/userspace/Shared%20Documents/R%20Development>

### 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.

### Installation of the faosws package

The R API package provides a set up functions allowing the R modules to access the Statistical Working System and its database. The actual API package and a number of custom support packages must be installed on the developer client to allow development and debugging of R modules. All packages can be found on the Userspace portal here:

<http://workspace.fao.org/tc/sws/userspace/Shared%20Documents/R%20Development>

The order of installation is:

1. faoswsUtil
2. faoswsExtra
3. faoswsFlag
4. faosws

## Starting development of a new modules

### Create and upload the XML configuration file

The configuration file defines the environment which will be set up when the module runs inside the SWS, hence in order to simulate that environment the configuration file is the first thing to create.

An annotated template can be found [here](http://workspace.fao.org/tc/sws/userspace/Shared%20Documents/R%20Development/RConfigurationTemplate.xml), and an XML schema XSD can be found [here](http://workspace.fao.org/tc/sws/userspace/Shared%20Documents/R%20Development/rScript.xsd). The schema will only be useful if you use a validating XML editor or syntax checker.

The configuration you create at this point is not necessarily the final version, each time you upload a version of the module/XML the SWS will overwrite the previous version; however the name defined in the XML is the key, used to identify the module, so that needs to remain constant to avoid leaving multiple copies under different names loaded in the system.

As an initial version you should at least:

* Name the module
* Set up initial parameters
* Define the main dataset

Once you have a valid XML configuration file, add this to a zip file (on its own).

Go to the user interface of the SWS [here](http://hqlqasws1.hq.un.fao.org:8080/sws/).

Click on the  button, then select the *Plugins management* tab.

Click the *New* button and select your zip file containing the XML, then select *Save*.

Once saved, select your script from the list at the top of the panel; click the  button and start typing your user name – select you name and select the *Save permissions* button.

### Launching a test session

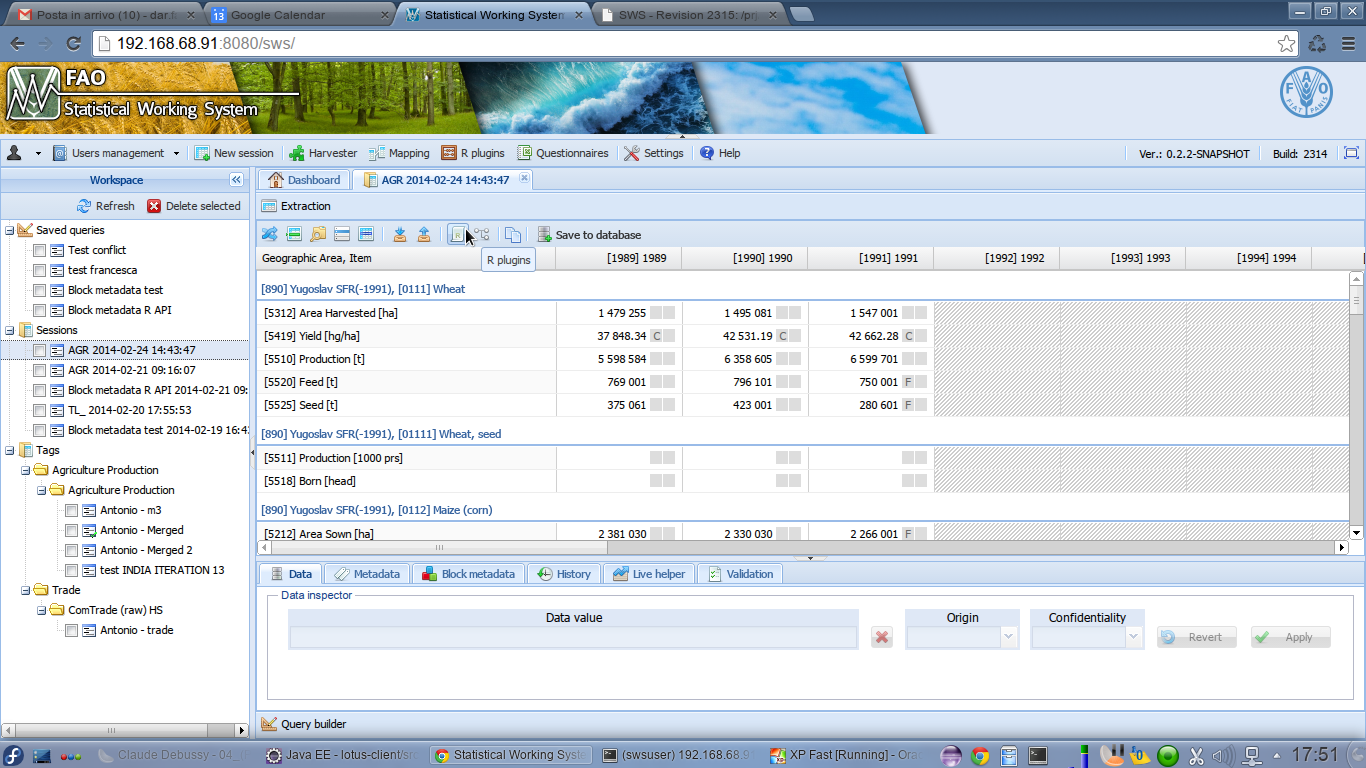
Using the configuration file in combination with the current state of user sessions, the SWS passes various pieces of information to the R modules via the *swsContext* variables set in the R environment before module execution starts.

A developer can simulate that environment by mimicking what the user is expected to enter, then using a developer specific feature of the system.

Once you have uploaded the XML configuration file and set permissions, your R module will appear in the available modules for the main dataset (as defined in the configuration file).

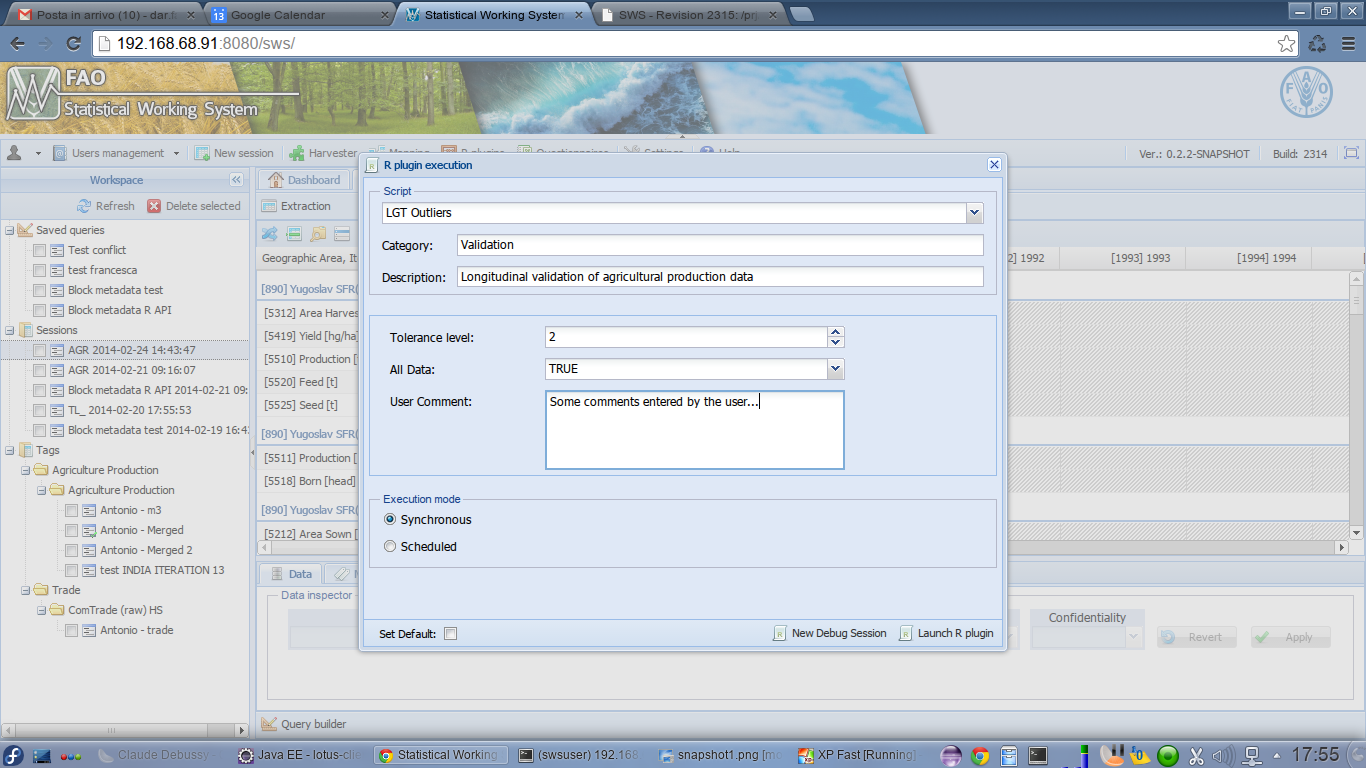
Using the SWS system user interface, open a session on that main dataset, select a small subset of suitable dimension key values and run the query. This will create a session on that main dataset.

The R script execution panel is opened by clicking on the button depicted in the following screenshot:

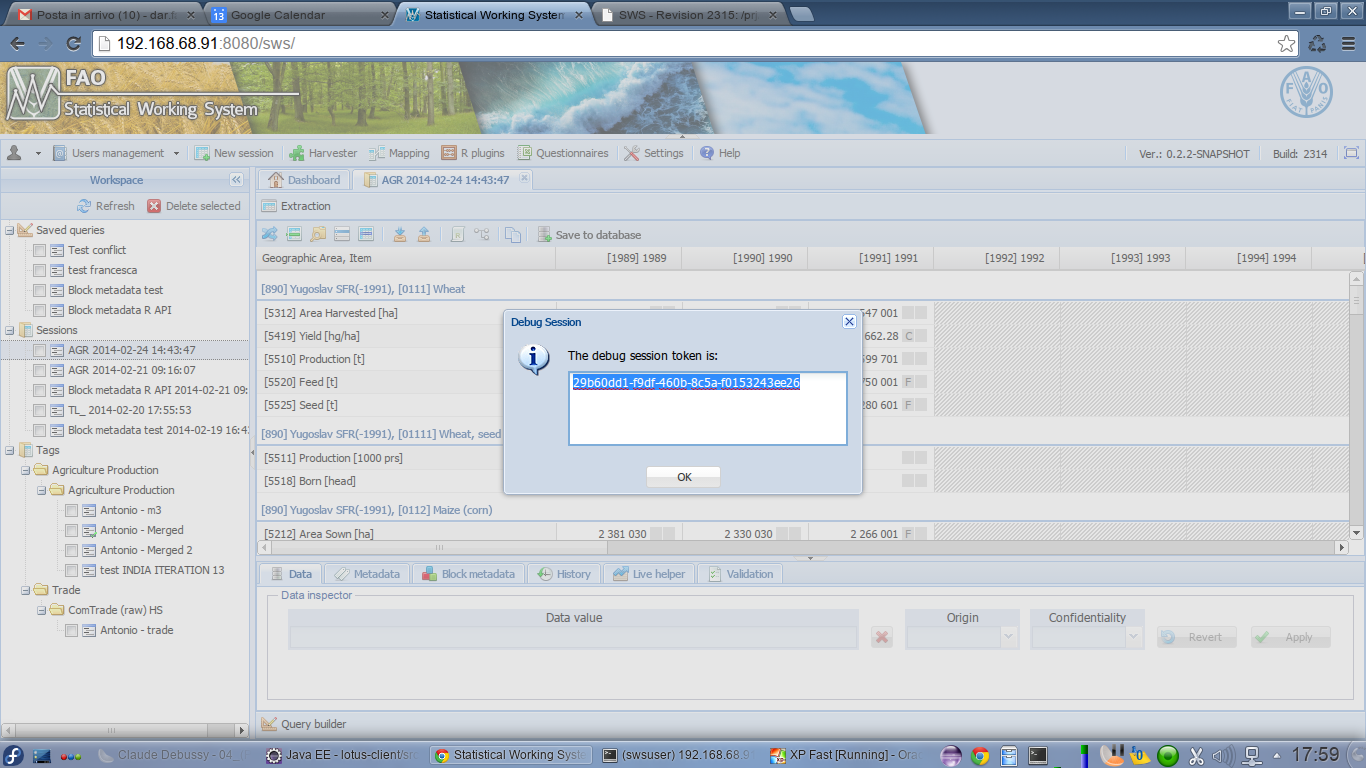


Select your new script from the pull-down list and the panel will re-configure to show your script options, such as parameters and secondary datasets.

Set the parameters as expected by the users, but instead of running the script, click on the *New debug session* button:



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 then displayed:



This token will be used by the script in developer/debug mode to create the simulated R module environment, as shown in the next section. Copy and paste the token into a text file.

You can dismiss the various SWS windows at this point, close the session and log out of the SWS user interface.

**Note:** Do not delete the session from your environment (is should still be present in you *Workspace* area in the user interface).

### 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 their local 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.

An example R module can be found here: [HelloWorld.R](http://workspace.fao.org/tc/sws/userspace/Shared%20Documents/Examples/R/HelloWorld.R).

Following is a snippet of R script using this special call, this assumes you saved the token in a file called *token.txt* in the same directory as the R:

library(data.table)

library(faosws)

## Set up for the test environment

tokenFile = "token.txt"

TEST\_MODE=FALSE

if (file.exists(tokenFile))

{

token = readChar(tokenFile,file.info(tokenFile)$size)

token = sub('[\n\r]','',token) # account for MS CRLF

GetTestEnvironment("https://hqlqasws1.hq.un.fao.org:8181/sws",

token)

TEST\_MODE=TRUE

options(error=utils::recover)

}

str(swsContext.datasets[[1]])

Once this environment setup function has been called, all other R-API functions can be called exactly as they will be once the module is uploaded to the SWS. For example to get the data selected by the user from the main dataset:

# 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.

### Deploying a module

Once you have the module working locally you can deploy it to the SWS as a private module, only visible to you and the administrators. You can also grant permissions to other users via the R module management panel in the SWS user interface.

Place both the XML configuration file and the R module into a zip file. This can then be uploaded as described above via the R module management facilities of the SWS user interface.

To execute the module follow the steps listed above, except instead of using the *New debug session* button, actually run the module on the session.

Note: The final return/output from the module (if non-NULL) will be displayed on the screen in a popup when the module finishes. E.g. of the last line of your module is “Hello World!”, that text will appear on the screen when the module runs.

Before packaging the script for deployment on the SWS, it is preferred to remove or comment out the *GetTestEnvironment* call, however the snippet shown above makes this unnecessary since no *token.txt* file is present on the server.

Unless you have SWS administrator status, when you deploy your R module it will not be able to write data to the main database, only to a user session. Once you are happy that you module is working correctly (assuming it is to be a core module which changes the database data) you should supply the R code and XML file to an SWS administrator and request that is uploaded as a core module.

### Advanced configuration

By examining the template XML configuration file it can be seen that various options are available.

**Parameter types**

Three types of parameter can be set:

* A simple string.
* A number, which can be limited by min and max values.
* A list, in which a list of options is included in the configuration for users to select from.

**Secondary datasets**

A main dataset is mandatory – this where the module is called from – however any number of secondary datasets can also be included, allowing the users to choose between the database and open sessions as the data source/target.

**Multiple R script files**

The uploaded zip file may contain a sub-directory containing additional R scripts. These are pre-loaded into the R environment before the R from the root of the zip is executed (more than one R script in the root of the zip is invalid). The order in which the secondary files are loaded is not fixed, so they should either contain *variables* or *function declarations* used by the main script, not unwrapped R code.

All files in the sub-directory must be R, and further nesting of directories below the first level is not allowed.

1. In this context “dataset” could also be a support data table or a mapping table [↑](#footnote-ref-1)
2. Code 953 is the world from the Agricultural Production domain area code list, flat structure. [↑](#footnote-ref-2)