SMIS TECHNICAL DOCUMENTATION REPORT

V1.0

TABLE OF CONTENTS

REVISION HISTORY

Version	Description	Date	Author
1.0	Original version of the technical documentation report.	01/09/2018	Tomasz Kurowski

ABBREVIATIONS

API	Application Programming Interface
HTML	Hypertext Markup Language
ES2016	ECMAScript 2016
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
ODM	Object Data Modelling
NoSQL	Non-SQL / Non-Relational
REST	Representational State Transfer
SCSS	Sassy Cascading Style Sheet
SMIS	Soil Management Information System
SMIS AT	SMIS Analytics Toolkit
URI	Uniform Resource Identifier
VM	Virtual Machine

1. INTRODUCTION

1.1. PURPOSE

The following document describes the functionality and architecture of the software delivered as part of the Soil Management Information System (SMIS) project. Its purpose is to provide a summary of the overall system design, its intended means of deployment, descriptions of each of the system's major components alongside their individual architectures and dependencies, and an overview of how these components interface and interact with each other to provide the SMIS functionalities. Details on the means of system administration, configuration, and deployment are also included.

The document is intended to provide an accurate overview of the software as delivered and serves as a potential introductory document for an administrator or developer seeking to modify, expand, or re-configure SMIS software either at the front-end or at the back-end. Topics dealing with configuration and deployment are covered in separate sections to provide an effective manual for administrators/developers seeking to make simple changes within the scope of already implemented options without modifying the SMIS system software, which would require a deeper understanding of the design.

1.2. SCOPE

This document covers the software design of the individual components of the SMIS system and their interactions from a technical point of view, including a discussion of the technologies used (and the resulting requirements), the code organisation and implemented interfaces.

Overviews of the modules and classes that make up the software are included, but individual functions, methods, properties or other variables are outside of the scope of this document. Those lower-level elements of the implementation, of interest primarily to developers seeking to modify or expand the software, are documented via code comments including standardised tags which allow for automated generation of interactive, up-to-date, HTML-based documentation including hyperlinks, a form of documentation more conducive for software development. Scripts used for generating and viewing this documentation are an integral part of the SMIS software system to be delivered alongside it.Topics addressed in previous documents, in particular the *Database Technical Documentation,* which covered the SMIS database design and generation process, and the *SMIS Web Interface Report,* which covered the visual side of the interface views provided by the SMIS Analytics Toolkit, are covered with the focus limited to their interactions with other components and implementation details omitted from the previous documents.

2. SYSTEM DESIGN

2.1. OVERVIEW

The SMIS software is split into two major parts, which during the course of the SMIS development project were maintained as two ostensibly independent applications with separate development cycles and version control repositories. The applications are internally referred to as SMIS AT (i.e. the SMIS Analytics Toolkit in a narrow sense) for the front-end, client-side application and visualisation of SMIS functionalities and SMIS API (i.e. the SMIS Application Programming Interface) for the back-end, server-side application which provides access to the database and analytics tools. The two applications communicate through a Representational State Transfer application programming interface (REST API) exposed by SMIS API.

This approach was chosen to facilitate a clear separation between the front-end and back-end components of SMIS with a well-defined, reusable interface between the two. During development, the SMIS AT and SMIS API applications could be deployed and tested independently, which simplified debugging and allowed for exploratory work using the SMIS database and back-end coupled with pre-existing tools (such as R machine learning and plotting libraries) before the front-end was sufficiently mature to provide the necessary visualisation of results and manipulation of system functionalities.

The back-end remaining functionally independent and available through a commonly used Representational State Transfer (REST) interface also allows for the possibility of permitting access to the SMIS database to applications, scripts, or pipelines other than the SMIS AT front-end application developed alongside it, widening the array of potential future uses of the collected data.

On the whole the SMIS software could be said to depend on the so-called MEAN stack (the MongoDB, Express.js, Angular, and Node.js technologies working in tandem), with Angular used on the front-end (supported by other libraries as described in Section 3) and the remaining three components (MongoDB, Express.js and Node.js technologies) of the stack used at the back-end.

3. FRONT-END – SMIS AT

3.1. OVERVIEW

The SMIS AT front-end application was implemented in the TypeScript programming language following the ECMAScript 2016 (ES2016) syntax and using Google's Angular (version 6) framework, with development supported by its associated toolchain, including the Angular CLI used with scripts allowing development builds, high-performance production builds, and the automatic generation of documentation using Compodoc. The npm package manager was used to manage project dependencies.

User interface dependencies

The open source PrimeNG user interface component library was used in constructing the SMIS user interface and provided a basis for the Sassy Cascading Style Sheet (SCSS) styling used via PrimeNG themes. In particular, the high-performance data table (*TurboTable*) component was adapted for use as a central part of the more complex SMIS database browsing component (see Section 3.3), and the availability of this structure was a major reason for choosing this component library over alternatives.

Visualisation dependencies

Plotly.js, an open source graphing library built on top of the lower-level d3.js and stack.gl libraries, was used to implement custom visualisation components for several views within the application (see Section 3.3). This choice was made based on the large variety and customisability of the visualisation options available in the Plotly.js library and the relative ease of applying further modifications using plain d3.js. The existence of analogous libraries for different environments (Plotly is also available for Python, MATLAB, and R) was also considered to be an asset, as was the ease of porting visualisations between those environments which could facilitate expansion or adaptation of the SMIS platform to make use of those technologies - two of which, Python and R, are already used on the SMIS back-end.

The Cytoscape.js open source library was used for rule base graph visualisation, providing an intuitive overview of collected modelling results generated by SMIS AT based on a well-supported graphing interface.

Browser support

The Babel package was used to emulate ES2016 functionality for Web browsers which do not fully support it yet. The Web application has been tested to work on Chrome (version 58+), Firefox (50+), Safari (11+), and Edge (17+) browsers. Internet Explorer is not supported as it has been discontinued.

3.2. SERVICES

Five Angular services were implemented to provide a way for the front-end components to access external data (MongoDB database collections and configuration files) via dependency injection. These services are:

- *GrowerService* – provides methods for retrieving data stored in the *grower* collection of the SMIS database, containing data collected from growers (to be displayed in filterable tables) as well as summaries of grower data generated on-the-fly on the SMIS back-end, accessed through the SMIS API REST interface.
- *LiteratureService* – provides methods for retrieving data stored in the *literature* collection of the SMIS database, containing the curated literature data collected during the SMIS project (to be displayed in filterable tables), accessed through the SMIS API REST interface.
- *ExperimentService* – provides methods for retrieving data stored in the *experiment* collection of the SMIS database, containing the curated literature data collected in the SMIS project (to be displayed in filterable tables), accessed through the SMIS API REST interface.
- *AnalyticsService* – provides access to the analytics functionalities of the back-end through the SMIS API REST interface, either retrieving pre-existing results from the *results* collection or generating new queries on-the-fly; this process is opaque to the front-end as all generated results are saved (and indexed by the query used) and the back-end will always send a pre-existing result if available instead of re-running an analysis.
- *ConfigService* – provides access to front-end configuration options stored in JavaScript Object Notation (JSON) format files which can be modified by an administrator to change the behaviour of the user interface as described in section 6.1.

The three services corresponding to SMIS database classes were implemented as child classes of the abstract *SMISTableService* class which defines a common interface and a set of methods for interacting with the *SMISTableComponent* component used to display filterable tables of the data collections stored in the database. Additional methods for retrieving data summaries used for visualisations are implemented separately for each respective collection in their own class.

For all four services (*GrowerService, LiteratureService, ExperimentService and AnalyticsService*) which interact with the back-end via the REST interface, the implementations are extremely simple and limited to processing the input (stringification and URI encoding of JSON objects), formatting and making a GET request via HTTP, followed by returning an *Observable* (dependent on the RxJS library) which allows asynchronous access to the request results. Data processing and most checks for input correctness are carried out on the back-end.

3.3. USER INTERFACE MODULES AND COMPONENTS

The front-end implementation has been split into the following modules:

- AppModule – top-level (root) module of the application, imports all other modules and contains the *MainMenuComponent* which implements the sidebar used for site navigation and the *NotificatorComponent* used for displaying notifications on the main page. Uses the *ConfigService* to retrieve configuration settings used by other modules.
- AppRoutingModule – router module which defines the rules for site navigation.
- SMISTableModule – contains the *SMISTableComponent*, which extends the PrimeNG *TurboTable* to allow for advanced filtering and support server-side pagination.
- QueryConstructorModule – contains the *QueryConstructorComponent* which is a complex form allowing for the selection of dependent/independent variables to be used in machine learning, as well as the addition of filters used to limit analytics to a subset of the database. Individual parts of the *QueryConstructorComponent* interface can be hidden and have their values hard-coded (preselected variables, mandatory filter fields, preselected method).
- FactorModule – contains the *FactorBarsComponent* which uses the Plotly.js graphing library to display bar plots based on the output of an *AnalyticsService* query. The bar plots display details of the variable distribution and show a breakdown of categorical variable coefficients on clicking a column corresponding to such a variable. The *FactorBarsComponent* is used for visualising the results of established queries.
- GrowerModule – contains the *GrowerComponent,* which shows a filterable data table (*SMISTableComponent*) of grower data, the *HectarageComponent* which visualises the hectarages for different crops/varieties present in the database using a Plotly.js line plot, and the *YieldOverviewComponent*, which visualises a breakdown of normalised yield by variety using a Plotly.js bar plot. Uses the *GrowerService*.

- LiteratureModule – contains the *LiteratureComponent,* which shows a filterable data table (*SMISTableComponent*) of literature data, including hyperlinks to papers. Uses the *LiteratureService*.
- ExperimentModule – contains the *ExperimentComponent*, which shows a filterable data table (*SMISTableComponent*) of experimental data. Uses the *ExperimentService*.
- QueryModule – contains the *QueryEditorComponent*, which uses a fully customisable (i.e. no hidden or hard-coded options) *QueryConstructorComponent* component to construct queries and execute queries, and the *ResultTableComponent*, which displays regression results in a simple paginated table. Uses the *GrowerService*, *LiteratureService*, *ExperimentService*, and *AnalyticsService* and imports the *QueryConstructorModule*.
- GraphModule – contains the *RuleBaseComponent* which displays a Cytoscape.js graph of Established Query results with a network of identified connections between variables. A limited *QueryConstructorComponent* can be used to filter graph contents. Uses the *AnalyticsService* and imports the *QueryConstructorModule*.
- EstablishedQueryModule – Contains one component for each of the implemented Established Queries. Each of those components contains a limited *QueryEditorComponent* set up to use dependent/independent variables specific to its Established Query and require particular filters during modelling, with the results displayed by a *FactorBarsComponent*. Uses the *GrowerService*, *LiteratureService*, *ExperimentService*, and *AnalyticsService* and imports the *QueryConstructorModule* and *FactorModule*.

4. BACK-END – SMIS API

4.1. OVERVIEW

The SMIS API back-end application was implemented in the TypeScript programming language following the ES2016 syntax, with documentation automatically generated using TypeDoc and packages managed using npm. The application runs in an Node.js environment (via ts-node / pm2), with the Express.js framework used to implement routing for a REST API which is deployed through an HTTP server. The SMIS MongoDB (version 3.4+) database is accessed by the application through the Mongoose Object Data Modelling (ODM) library, which enforces a set of well-defined schemas on the NoSQL data in order to effectively use them to conduct analyses and serve REST requests.

A set of pipelines implemented in the R programming language is used to provide machine learning functionality to SMIS API. The *r-script* npm module is used to transfer data (in the form of JSON objects) between the R scripts and the TypeScript application. For the sake of performance, only a query (rather than data) is passed as the input, and the R scripts fetch data directly from the database using the *mongolite* R package instead of relying on the back-end's Mongoose schemas.

A semi-independent part of the back-end is the Model Building Pipeline implemented in Python which generates a derivative *model* collection based on the SMIS stored data. This collection is used as an intermediary between the raw SMIS data and analytics pipelines. The raw data is interpreted and transformed into groups of derived variables (e.g. the counts of pesticide applications per year on a specific field or the rotational context of a field) which are directly usable in the machine learning regression pipeline. As the Model Building Pipeline needs to be executed only once per database update, it is not strictly integrated with the rest of the back-end and requires manual execution of a Python script (this is already completed in the delivered version of the SMIS database).

4.2. REST API AND ROUTING

The SMIS API application starts an Express.js HTTP server which listens for requests to be handled by one of four available Express routers:

- GrowerRouter
- LiteratureRouter
- ExperimentRouter
- AnalyticsRouter

Each of the routers contains middleware which process the input parameters, parsing them from Uniform Resource Identifier (URI) strings to JSON objects and validating their values (this includes verifying the correctness of pagination requests for the three routers serving requests from data table components). After input validation the router queries the MongoDB database through the corresponding Mongoose schema object for queries which do not require additional processing (note that this can also be the case for analytic queries – see Section 4.4) or passes the input to an R pipeline through the *r-script* interface.

It should be noted that while from a routing (or data flow) perspective it would be straightforward to retrieve data during the routing (taking advantage of Mongoose and structured schema classes) and pass them to the R pipeline for processing, this approach was found to make the *r-script* package

potentially unstable when dealing with large volumes of data. Therefore, validated queries are instead passed to the R pipeline which retrieves the needed data from the SMIS database on its own.

It is possible that future updates to *r-script* might resolve the instability issue, in which case it might be better to use the more straightforward implementation instead (which is still present in the code base but disabled by a comment block).

Once a result of either a database query or an R script execution is received, it is sent as a JSON response through the HTTP interface. For novel machine learning results which gave a valid output the result is also persisted in the MongoDB database, identified by the contents of the query which created them.

4.3. MONGODB DATABASE

The SMIS MongoDB (version 3.4+) database as used by the SMIS API contains five collections (the full database also includes dictionary collections used for parsing; this is described in the *Database Technical Documentation*) listed below:

- *grower* collection – grower data rows.
- *literature* collection – curated literature rows.
- *experiment* collection – experimental data rows.
- *model* collection – collections of variables derived from grower data by the Model Building Pipeline (see Section 4.5), used for machine learning.
- *rule* collection – results of machine learning.

The SMIS API treats the database as essentially static except for the *rule* collection which saves each generated result of a valid analytics query. The *model* collection can also be re-created by re-running the Model Building Pipeline (see section 4.5), but for an unchanged database the contents of the collection will be unchanged as well.

4.4. MEMOISATION / CACHING

The SMIS API is designed not to repeat analytics which it has already executed for the same database and query. Generated results are stored in the *rule* collection and the router verifies whether a pre-existing result is available for a particular query (and retrieves it if it already exists). A new analysis is started through *r-script* only if no pre-existing results are found. This has the dual use of speeding up the operation of the application and collecting all the generated results which can then be displayed together as a graph which visualises all the connections identified among variables of interest.

4.5. ANALYTICS

Model Building Pipeline

The Model Building Pipeline is a Python script which converts the raw data stored in the *grower* collection (which represents individual farm field operations as recorded in Gatekeeper grower datasets) into groups of variables used in machine learning. These are either straightforward summaries (e.g. counting the number of fungicide applications in a year on a particular farm field), more complex variables such as the *n*-th previous crop (which requires the algorithm to identify the full rotation used), or even inferred variables not directly present in grower data (e.g. possible incidence of compaction problems inferred from timing and frequency of subsoiling operations). Prior knowledge and expert opinion was used in formulating possible inference strategies for variables not directly recorded in the source data.

A variable generation function was implemented for each of the variables. The sequence of operations for each individual farm field is retrieved and sorted by date, then processed by each of the variable generating functions. Variables collected for each field are then persisted to the *model* collection in the SMIS database, from which they can be retrieved for use in machine learning.

The pipeline interacts with the SMIS database via the Python PyMongo library rather than the SMIS REST API for performance reasons. In the case of deploying SMIS using a Virtual Machine (VM) or a Docker image, the Model Building Pipeline is available in the home directory and can be executed manually (although for the default SMIS deployment this is not necessary, as the *models* collection already exists).

Machine Learning Pipeline

Two machine learning pipelines have been developed in R, one using multiple linear regression, and another using the random forest algorithm. Only the former is used by default, as it has been more thoroughly tested by the SMIS development team.

The pipelines receive a query object (a JSON object converted into an R object using the *jsonlite* package) which contains a *dependent_variable* data field, a list of *independent_variables* (predictors) to be used, and a *filter* object containing MongoDB search terms used to select a subset of data to be used in the analysis. The *mongolite* package is used to retrieve filtered data from the *model* collection

in the SMIS database, and the variables defined by the query object are extracted to be used in regression.

Predictors undergo feature selection using a stepwise algorithm for multiple linear regression or the Boruta algorithm (*Boruta* R package) for random forest. Variable importance is then investigated using the *caret* package. The pipelines return a list of variables (predictors) sorted by their variable importance and accompanied (for multiple linear regression) by coefficients for each variable (or each value of a categorical variable) which are used to generate visualisations in the front-end.

5. DEPLOYMENT

5.1. OVERVIEW

Two primary options for deploying the SMIS applications have been developed: a Virtual Machine (VM) and a Docker image, both using Ubuntu Server with all dependencies installed and both SMIS applications deployed using the *pm2* process manager for load balancing. While internally the SMIS AT and SMIS API applications use different ports, the containers use a reverse proxy setup (HAProxy) to expose both applications on port 80 (by default SMIS AT is at http://localhost:80, and SMIS API is at http://localhost/api:80). The port can be forwarded on the host machine so that the SMIS Web application can be made available over a network.

Both the Docker image and the VM contain a "*configure_smis.sh*" shell script in their home directories. The script allows an administrator to change the host URL and ports to be used through a series of prompts.

6. MANUAL

6.1. CONFIGURATION FILES

The look and behaviour of certain user interface elements within the SMIS AT Web application can be changed by modifying configuration files which are stored in the *assets/config* directory of the front-end application, also accessible (as the *config* directory link) through the home directory of the Docker image used in deployment.

The configuration files and their associated options are described below.

Data table column configuration

Three configuration files (*grower.col.cfg.json, experiment.col.cfg.json, literature.col.cfg.json*) control the columns and their associated filter options displayed by each of the three data tables used for data browsing (see Section 3.3). An example of a (simplified) grower column configuration file is shown below:

```
{
    "Date": { "hidden": false, "position": 1, "filter_type": "date" },
    "Crop": { "hidden": false, "filter_type": "multi",
              "filter_label": "All Crops", "position": 3 },
    "Variety": { "select": true, "filter_type": "multi",
                 "filter_label": "All Varieties", "position": 4 },
    "Heading": { "header": "Field Operations", "select": true,
                 "filter_type": "multi",
                   "filter_label": "All Field Operations",
                 "position": 2 },
    "Yield": { "select": false },
    "Product Name": { "select": false },
    "Texture": { "ignore": true }
}
```

Key – value pairs match a database column (data field) with a list of options describing how (and if) the column should be represented in a data table (e.g. `"Texture": { "ignore": true }` are the options for the Texture column). Each of the options has a list of valid values it can take. The full list of options is shown in Table 1.

TABLE 1: DATA TABLE COLUMN CONFIGURATION OPTIONS

Option	Valid values	Description
header	Any character string.	Controls the column header used in the data table. By default SMIS simply uses the same name the column has in the database.
hidden	*true*, *false*	Controls whether the column is shown or hidden when a user first views the table. When *hidden* is *true*, the column is not displayed in the table but can be selected (provided *ignore* is *false*) from the table. The default is *true*.
position	Any positive integer.	Controls the starting position of the column in the data table. Columns are ordered (left to right) from the lowest position to the highest, although a user can freely rearrange them. The default is 0.
filter_type	"*text*", "*date*", "*multi*", "*numerical*"	Controls the type of filtering available for this column. The default setting is "*text*". "*text*": the user can enter (case-insensitive) filter text into a text box.

		"*date*": the user can select a range of dates from a calendar. "multi": the user can check any number of elements in a searchable list. "numerical": the user can select a number range (from lowest to highest).
filter_label	Any character string.	Used only when filter_type is "*multi*". Controls the label next to the checkbox which allows the user to (de)select all values from the multiple selection box. By default, the label is simply "All".
ignore	*true*, *false*	Controls whether the column is included in the Web interface. If *ignore* is *true*, the column will be shown neither in the table nor in the column selector. This option is *false* by default, but note that columns not included in the configuration file will be ignored as if the *ignore* option was set to true.

It should be noted that while columns present in the database collection are not displayed by default and therefore using the "*ignore*: *true*" option may seem redundant, it does in fact serve an important function. As the JSON format does not support comments (which could be used to hide undesired columns), every column present in the database is listed in the configuration files, with all but the ones the administrators want to show to the users being marked with *ignore: false*, making it easy to add/remove any individual column from the view.

An empty option list (e.g. `"Field Name": {}`) is essentially equivalent to:

```
"Field Name": {
      "header": "Field Name", "hidden": true,
      "position": 0, "filter_type": "text",
      "ignore": false
}
```

Note that the configuration files are part of the front-end application executed on the client side, so a user could potentially modify their own copy to gain access to columns which the administrators have opted to ignore in the interface. This means ignoring columns to hide them from users should ***not*** be considered a means of securely ensuring data remains unavailable to users, but only a means of hiding unnecessary detail in the interface. The contents of the database were sanitised and anonymised during the database construction and do not contain sensitive data by design.

User interface configuration

Several interface user interface configuration options stored as key-value pairs can be changed by an administrator by editing the *ui.cfg.json* file. The options are described in Table 2.

TABLE 2: USER INTERFACE CONFIGURATION OPTIONS

Option	Valid values	Description
store_session_layout	*true, false*	Controls whether layout changes made by a user (e.g. reordering or hiding data table columns) should be kept for that user for the duration of the session. If *false*, the layout is restored to its default state when a view is refreshed. The default is *true*.
notification_area	*"top", "local"*	Controls the location of notifications shown to the user (e.g. ones instructing them on how to zoom out after a user zoomed in or informing the user that an analysis failed). If set to "top", the notifications are all displayed in a common area on the top of the page. If set to "local", individual components display their own notifications. The default is "*local"*.
grower_data_summary	*"rows", "hectares"*	Controls whether the amount of data available for a given filter setting in the grower data table (displayed at the bottom) should be expressed in terms of data rows or hectares. The default is "*rows*".
smis_api_url	Any valid HTTP URL.	The URL and port corresponding to the SMIS API. This is set automatically when running the configuration script prior to deploying the application but can also be changed manually.
machine_learning_pipeline	"*mlr*", "*rf*"	Controls the machine learning pipeline requested by the front-end. "*mlr*": Multiple Linear Regression pipeline. "*rf*": Random Forest pipeline. The default is "*mlr*".