WAD-QC 2.0

Rob van Rooij and Arnold Schilham, UMC Utrecht

20180629
v0.7
For up-to-date information, check the Wiki at
`https://bitbucket.org/rvrooij/pywad3/wiki/Home`

Abstract

WAD-QC 2.0 is an open source implementation in python of the *WAD-QC Server* `https://github.com/wadqc`: a server for automated analysis of medical images for quality control, by the *Society for Medical Physics of the Netherlands* (NVKF) `http://www.nvkf.nl`.

Contents

Chapter 1

Design

1.1 Basic requirements

Based on use-cases of WAD-QC 1.0 in several hospitals and requests for changes to the existing framework by the developers, WAD-QC 2.0 was redesigned from scratch. Many design choices for WAD-QC 2.0 turn out to be very similar to the design choices made in WAD-QC 1.0; this is partly because of bias due to experience with WAD-QC 1.0 but mostly because the choices of WAD-QC 1.0 were well founded.

The basic requirements for the design of WAD-QC 2.0 were:

(1) Use modern, up-to-date software. For ease of maintenance, use the same programming language where possible. For flexibility, use only open-source, cross-platform software.

(2) Use dedicated components: DICOM images should be stored in a PACS and access to these images should be through Query/Retrieve only. Results and parameters should be stored in a database. Introduce no hard dependencies on a specific PACS implementation or on a specific database engine.

(3) The results database should contain results and everything needed to reproduce the results (except for the images themselves): Module and parameters used and unique IDs of the images analyzed. Information not related to the results or on how to generate them should not be mixed with results data.

(4) Analysis modules should be treated as black boxes that can read DICOM images (and a configuration file with run-time parameters), and produce results in a prescribed file format.

(5) Visualization and reporting of results should be left to dedicated, flexible tool, and should not be integrated in WAD-QC 2.0.

These requirements were met as follows:

(1) WAD-QC 2.0 is completely written in python3, and is fully compatible with python2.

(2) WAD-QC 2.0 makes use of a connection layer, through which it can be coupled to multiple data sources simultaneously. Right now only usage of the DICOM server Orthanc[1] is fully implemented and tested. For interfacing with databases, WAD-QC 2.0 uses the ORM peewee [2], which out-of-the-box supports MySQL, PostgreSQL, SQLite and BerkeleyDB. As Orthanc needs SQLite or PostgreSQL as a database back-end, the only tested configurations for WAD-QC 2.0 are SQLite and PostgreSQL. The default database engine for WAD-QC 2.0 is PostgreSQL.

(3) The WAD-QC database is restructured to meet the requirements, and will be described in detail later in this document.

(4) In WAD-QC 2.0 the input and output requirements of the analysis modules have changed. The necessary changes to existing modules of WAD-QC 1.0 are only minor, for both python and non-python modules. Right now only some of the pyWAD 1.0 plugins are adjusted to become stand-alone analysis modules. Only these modules have been tested for now.

(5) A stand-alone visualization tool is included in the WAD-QC 2.0. This is not intended as a final product, but without a visualization tool WAD-QC is difficult to appreciate.

1.2 Overview of WAD-QC 2.0

Figure 1.1 shows WAD-QC 2.0 in a flow-chart.

1.2.1 Data flow

As a new dataset arrives in the PACS, the *Selector* process is triggered and the flow-chart is acted upon. The *Selector* requests the headers of the new dataset from the PACS and requests all data filters (selection rules) from the WAD-QC database. The headers are compared to the selection rules. If a match is found a new job is put in a queue for processing, detailing how the obtain the data and how to process it. The *Processor* process runs in the background, and periodically checks the queue for new jobs. The *Processor* assigns jobs to so-called *Worker* processes from a predefined pool. When a *Worker* becomes available it processes the job given: the correct dataset is requested from the PACS and the analysis is started with the provided analysis module and run-time parameters. Upon completion of the analysis, the *Worker* collects the results and stores them in the WAD-QC database.

1.2.2 Data sources

The WAD-QC framework can be connected to multiple PACS servers at the same time, but each PACS server needs to provide its own trigger mechanism for the *Selector* process. In the case of Orthanc, this is done by a lua script that invokes a *Selector* each time a new DICOM study arrives in the PACS.

1.2.3 Parallel jobs

The number of workers available to the *Processor* is set by the user, and defines the number of analysis jobs that can be run in parallel.

1.2.4 IO layers

As can be seen in Figure 1.1, there are dedicated *IO layers* to access the PACS and the WAD-QC database (green boxes). That way the WAD-QC framework is made independent from a specific type of PACS or database engine, and by forcing all access to go through those layers, the integrity of the PACS and the database is easier to maintain

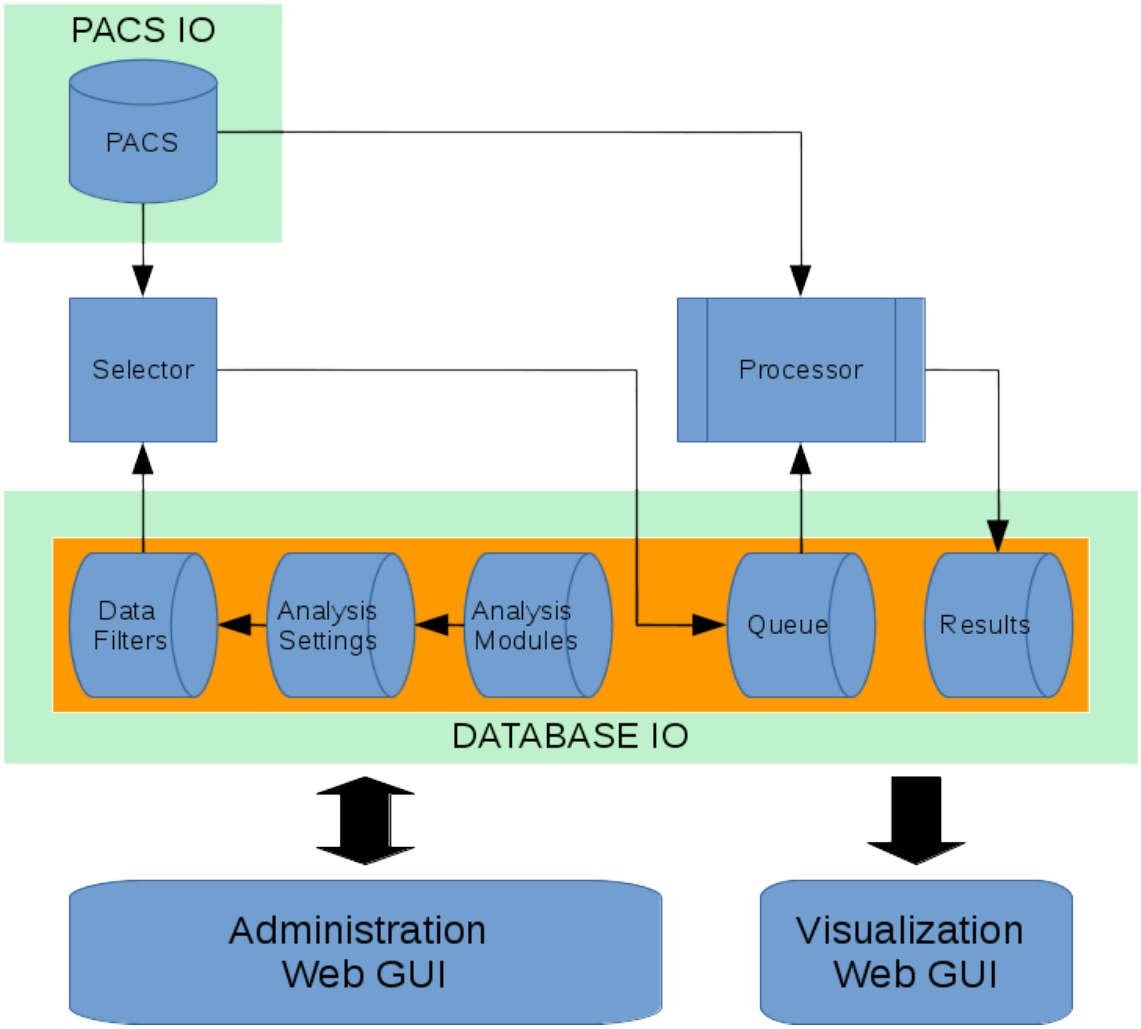

Figure 1.1: Flowchart of WAD-QC 2.0. The green boxes indicate specialized IO layers, and the orange box indicates the WAD-QC database.

1.2.5 WAD-QC database

The orange box in Figure 1 shows the actual WAD-QC database. It consists of three main parts:

(1) Data Filters and Analysis Settings: the selection rules and the processing parameters to use;

(2) Queue: The processing queue;

(3) Results: The analysis results.

1.2.6 WAD-Admin

To keep the core of WAD-QC 2.0 (i.e. *Selector*, *Processor* and *IO layers*) compact and manageable, a dedicated administration tool (named *WAD-Admin*) is provided. *WAD-Admin* is a stand-alone web service through which all the essential configurations of WAD-QC should be managed. For example, through *WAD-Admin* new selection rules and changes to analysis settings (run-time parameters) and the PACS connections can be applied. *WAD-Admin* also includes a utility to inspect all tables of the WAD-QC database. *WAD-Admin* should only be accessed by the local administrator of the WAD-QC installation.

main features

- Manage database (upgrade, consistency check, inspect)
- Manage Analysis Modules (import, export, upgrade factory modules)
- Manage Selectors (create, disable, dry-run, backup)
- Manage Processes and Results (delete, redo, view logs)

1.2.7 WAD-Dashboard

For visualization of the analysis results, the stand-alone web service *WAD-Dashboard* is bundled with WAD-QC 2.0. *WAD-Dashboard* is the tool that should be the landing page for a normal user who is only interested in the results of the QC tests. Right now *WAD-Dashboard* is only a simple tool, but it allows for visualization of the results, and also highlights some of the functionalities expected from a final product. In the present implementation, *WAD-Admin* and *WAD-Dashboard* are build with the Flask framework [3], and include their own web servers. Therefore these services can run stand-alone, which is nice for development, but it is recommended to deploy them on a standard web server. Instructions are provided for deploying on Apache 2.0.

main features

- Usermanagement (roles, restrict data shown)
- Grouping Results (join output of different Selectors into one visible Result)
- Show Results (global status, details, graphs, trends, punctuality check of QC frequency)

1.3 Main changes and new features with regard to WAD-QC 1.0

Below is a summary of the main changes of WAD-QC 2.0 compared to WAD-QC 1.0.

1.3.1 Implementation changes

python WAD-QC 2.0 is entirely written in python3 (and fully compatibile with python2) instead of java and php. Just like java, python is cross-platform and widely available. Most people find python easy to learn, and there is a vast community and a wealth of support available for python.

English The default language used throughout WAD-QC is English.

image access DICOM images are accessed through Query/Retrieve only.

external dependencies No hard dependencies on database engine or on PACS type for images.

JSON The main file format for exchanging information is now JSON, while XML was used in WAD-QC 1.0. JSON is a more natural structure for python, and results in more compact files, without loss of readability or flexibility.

1.3.2 Module config changes

Run-time parameters for the analysis modules (the *config.xml* files in WAD-QC 1.0) are now treated as a critical part for the obtained results; as changes in these parameters can lead to different results, those parameters are tracked in the WAD-QC database so that results can be reproduced faithfully. Furthermore, the contents of the earlier *config.xml* files are now split into a part essential for the results to be produced (the *module_config*) and a part useful for the interpretation of the results (the *meta_config*), such as fixed criteria for the values obtained, or the units of those values. Apart from constraints of the outcome values, the *meta_config* contains elements that are only needed when visualizing the data and which are completely ignored by the core of WAD-QC 2.0 and *WAD-Admin*, but can be used by *WAD-Dashboard*.

1.3.3 Module changes

The pyWAD package of analysis modules is split into separate, stand-alone modules, that make use of shared set of IO functions. Modules are stand-alone executables, that expect 3 parameters:

-c `config_file`

-d `study_data_folder`

-r `results_output_file`

The data is always provided to the analysis module in a three level deep folder structure Study/Series/Instances, where one study, but multiple series and instances each are allowed. Just like in WAD-QC 1.0 each analysis module is stored in a separate folder, and only the folder locations are tracked in the WAD-QC database. Although it is possible to store the complete analysis modules as blobs in the database, it was decided not to do so, for flexibility and to keep the database size smaller. To ensure reproducibility of results, it is expected that developers implement an analysis module version number and that this version number is stored as a result item together with the other results of an analysis.

1.3.4 Results changes

Results are expected to be written as a JSON file. When Objects are produced as results (e.g. images), these are added as blobs in the results database.

1.3.5 Added features

multiple data sources WAD-QC 2.0 allows for multiple data sources at the same time. Multiple PACS systems can be coupled, and other types of data storage systems are feasible as well.

date-time results Date-Time objects are recognized as a result type as well. Use-cases are for example hardware calibration dates, or data acquisition times.

dynamical limits Dynamical limits for results are added. In the produced results file, constraints (limits) to the data can be supplied as well. If static constraints (constraints provided in the *meta_config*) are provided for the for a result, while dynamic constraints are provided too, the static constraints will will take precedence upon storage in the WAD-QC database.

selector logics A *Selector* (i.e. a set of data criteria (*selector_rule*s) and a coupled list of analysis actions and parameters) contains a number of *selector_rule*s that are combined with AND logic. Each *selector_rule* consist of a reference to any valid DICOM header field, a certain comparison logic, and a number of different outcomes that are acceptable. The logics that can be picked are: equal_to, not_equal_to, contains, does_not_contain, starts_with, ends_with, does_not_start_with, does_not_end_with, is_empty, is_not_empty. The latter two take no outcome values. Although it is expected that these possibilities will suffice in most cases, a user is encouraged to consider using multiple selectors instead of one very complex selector; in the reporting tool (*WAD-Dashboard*) the results of these separate selectors should be combined into one test result.

role of *module_config* *module_config*s cannot exist alone. They should always point to an analysis module. If an analysis module can be used for different kinds of analyses (e.g. using different phantoms) the developer is expected to supply a separate *module_config* for each kind of analysis. If the *module_config* supplied with an analysis module adheres to a certain JSON format, then the run-time parameters in that *module_config* can be changed together with all other selector settings in one GUI screen in *WAD-Admin*. Therefore a developer is expected to supply *module_config*s that lists all the possible parameters restricted to that particular analysis, and nothing more. It is also allowed to upload a non-JSON file as *module_config* to be used together with a certain module; in that case the *module_config* is just copied as is, and no tuning of parameters through *WAD-Admin* is possible.

one screen to modify selector In *WAD-Admin*, all selector parameters can be changed in one screen: The *selector_rule*s, the *module_config* used, the run-time parameters, and the constraints (limits) for the results.

import and export An import/export feature is available in *WAD-Admin*. Here *module_config*s together with their analysis modules can be selected for export or import, thus facilitating updates to modules and making the initial setting up of a WAD Server easier. There is a distinction between **factory** and **user** modules and *module_config*s, where the **factory** versions are the original ones bundled in the WAD-QC package, and **user** designates modified factory files or manually uploaded files.

disabled selectors *Selector*s have an additional flag **is_active** that can be used to temporarily enable or disable selectors.

selector dry-run It is possible to dry-run a *Selector* on the data sources. This will show the list of data that would result in a match for that *Selector*, without actually creating new *Process*es. That way, a *Selector* can be adjusted to include or exclude certain data, before going live.

1.4 WAD-QC Database

Figure 1.2 shows the schematic diagram of the database of WAD-QC 2.0. There are a few noteworthy changes with respect to the database of WAD-QC 1.0.

The table *Variables* is added to have a central location for all kinds of variables that are needed for the core of WAD-QC: the locations for temporary files and the modules, the version of the database, and some parameters for the *Processor* (like number of workers available and time between polling the queue).

The table *MetaConfigs* is added and contains all information needed for displaying results (like a description or constraints). This information is stored as a JSON file in the table, to allow for easy extension later on.

The table *ModuleConfigs* has a central position in the scheme. Not only does it contain the *module_config*, but it is also the only table that connects to the tables *Modules*, and *DataTypes*, and *MetaConfigs*. *DataTypes* is a table with the allowed types of data that are available from the various data sources; this information can be used to retrieve the right data from a given data source. For DICOM images in a PACS, recognized data types are dcm_study, dcm_series, and dcm_instance.

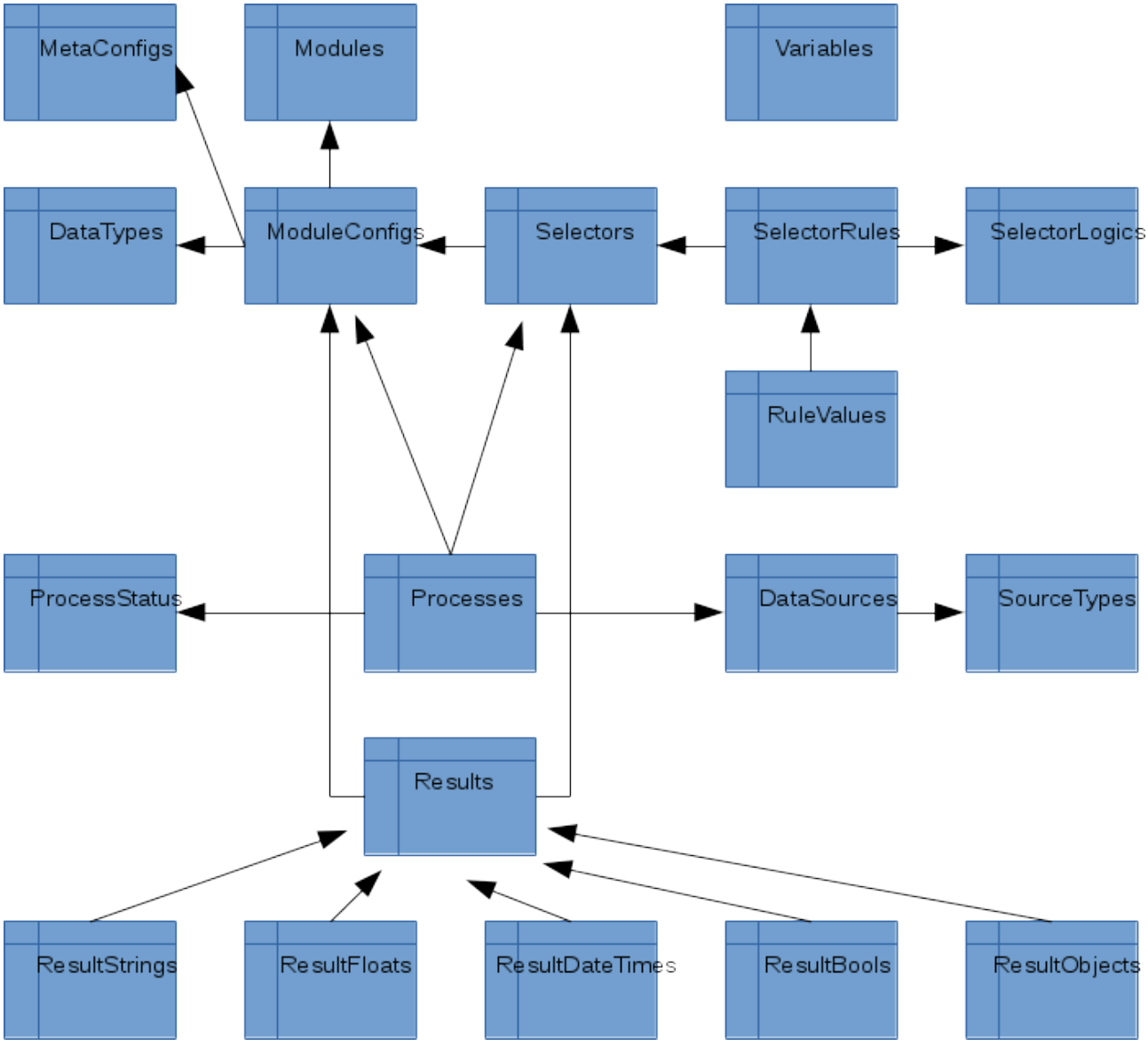

Figure 1.2: Schematic of the WAD-QC database, showing all tables (blue blocks) and the foreign key connections (arrows) between tables.

Another central position is held by the table *Selectors*. In the first place it is used as the linking pin between the data selection criteria in table *SelectorRules* and the analysis to perform through table *ModuleConfigs*. Later it is used to identify the origin of the results produced. Note that multiple entries of table *SelectorRules* are combined with logical AND in one entry of table *Selectors*.

The data selection criteria are now split into three different tables. Table *SelectorRules* lists the different selection criteria, each consisting of a data tag and a given outcome value and the logical relation between the two. The allowed logical relations are stored in the table *SelectorLogics*. Multiple outcome values are allowed for each rule; these outcomes are stored in the table *RuleValues* and

they are treated as a list of logical OR values.

The table *Processes* is in fact the *Processor* queue. When the *Selector* finds a match between a new dataset and all the *SelectorRules* of a given entry in *Selectors*, a new item is added to the *Processes* table, stating:

(1) for which entry of *Selectors* this match was found;

(2) what *ModuleConfigs* entry should be used for analysis;

(3) the origin of the data (*DataSources* entry);

(4) the unique ID needed to retrieve the data from given data source;

(5) the actual status of the *Processes* entry. The allowed states of the latter are defined in the table *ProcessStatus* (e.g. 'new', 'busy', 'error').

The table *Results* is new, and is very similar to the *Processes* table. After a *Processes* entry is picked up by the *Processor* and successfully analyzed, it is removed from the *Processes* table and a new *Results* entry is created, containing the same information as the *Processes* entry on how the results were obtained. With the information in the *Results* entry, the original *Processes* entry can be recreated, so the results can be reproduced.

The values of the analysis results themselves are stored in the tables *ResultStrings*, *ResultFloats*, *ResultBools*, *ResultObjects*, *ResultDateTimes*, depending on the types of results. The *ResultDateTimes* table is new, and is used for storage of DateTime elements. Each of these five tables of types of results has entries for the names of the values, the values themselves and values for all dynamic constraints allowed for that particular type of result. For example a *ResultStrings* entry has only an 'equal to' constraint, while a *ResultFloats* entry has constraints for 'equal to', 'minimum', 'maximum', 'acceptable lower limit' and 'acceptable higher limit'. A *ResultObjects* entry does not have any constraints. Also note that all *ResultObjects* are stored as blobs in the database.

Chapter 2

Implementation details

2.1 Specific implementation details

WAD-QC 2.0 expects certain input files in JSON format, and other configuration files at certain locations, as will be explained below.

2.1.1 module_config.json

If the *WAD-Admin* interface is to be used to manage the run-time parameter settings of an analysis module, these parameters should be stored in a prescribed manner so that *WAD-Admin* can access them. In WAD-QC 2.0 this information is stored in the *module_config* JSON blob. A blob was chosen instead of a new table of parameters, as it is unclear what kind of parameters and what else is to be stored in that table. It also keeps it flexible to other kinds of *module_config* files, that might be needed for a specific analysis module. However, only if it is a JSON file with a certain structure, *WAD-Admin* can be used to make modifications to that file. Examples of the proposed JSON file structure can be found in the distributed factory *module_config*s, which can be viewed and downloaded through *WAD-Admin*.

2.1.2 meta_config.json

The *meta_config* contains all the information that helps in the interpretation and visualization of the results themselves, but which is not needed for the actual generation of those results. As it is unclear what information is to be expected there, it was opted to store this information as a JSON blob. The contents of this blob can be managed separately from the WAD-QC core, for example by *WAD-Dashboard*. In the current implementation, the *meta_config* is a JSON file containing the fixed constraints for analysis results (as opposed to the dynamic constraints that can be supplied by together with the results by the analysis module), and settings for visualization according to a certain structure. Examples of the proposed JSON file structure can be found in the distributed factory *meta_config*s, which can be viewed and downloaded through *WAD-Admin*.

2.1.3 wadconfig.ini, wadsetup.ini, and orthanc.json

For the WAD-QC framework to run correctly, several files are needed. These files are all created from templates during installation by *wad_setup*.

WAD-QC needs an initialized database for storing results and information regarding analysis modules, but also with access credentials for the data sources. This database is created automatically by *wad_setup*, and filled with the credentials from the file `wadsetup.ini`, which will be located at

`$WADROOT/WAD_QC/wadsetup.ini` after installation. This ini file is needed only for the initial creation, and for recreation after database truncation.

The Orthanc server needs a configuration file (detailing access rights and what plugins are available), which will be located at `$WADROOT/orthanc/config/orthanc.json` after installation. This file is needed every time the Orthanc server is restarted.

wad_setup will create the folder `$WADROOT/pgsql/data` for PostgreSQL to store its data and access configuration. The location of this folder is needed every time the PostgreSQL database is restarted. If for whatever reason the user has decided to store the PostgreSQL data in a different folder, the user should make a symbolic link `$WADROOT/pgsql/data` to the real location; without that link `wadservices` and *WAD-Admin* cannot access the WAD-QC database.

Finally, all elements of WAD-QC obtain access to the WAD-QC database through the credentials in the file `wadconfig.ini`, which will be located at `$WADROOT/WAD_QC/wadconfig.ini`.

2.1.4 separation between WAD-Admin and WAD-Dashboard

By intention, *WAD-Admin* and *WAD-Dashboard* are separated services. While *WAD-Admin* is an essential part of WAD-QC 2.0, *WAD-Dashboard* is meant only as a demonstration of a possible reporting tool for the analysis results, so that developers are provided with a way to visualize the results.

2.1.5 integrity testing

For the core of WAD-QC 2.0, a whole set of unit tests for the separate components as well as an end-test for the whole package is available. Such test do not exist yet for *WAD-Admin*, *WAD-Dashboard*, and *wad_setup*.

2.1.6 SQLite

Although SQLite is available as a tested option for the WAD-QC database, its usage is not recommended. SQLite works best if only one user (e.g. Orthanc) can access the database at any one time; simultaneous multi-user access to SQLite requires a lot of concurrency checks.

2.1.7 DICOM-tags

Using *WAD-Admin* to add selector rules to a selector, the list of tags to choose from seems only small. Through the menu item 'tags for selector rules' under 'Resources', any DICOM tag can be added to the list of options to pick. Note the special tag 'RemoteAET', which is not a real DICOM tag but is very useful for *selector_rule*s.

2.1.8 features not yet implemented

dcm4chee For now, only the specific IO class for Orthanc has been written, using the REST api of Orthanc. For dcm4chee a bit more work is needed, as there is no REST api available. A good solution would be to write a generic PACS IO class for non-Orthanc PACS, which makes use of a separately running DICOM receiver and supports standard Query/Retrieve tools, and to have a separate service that queries the generic PACS for new studies. This will have an impact on performance and should only be pursued if there is enough demand.

MySQL As only Orthanc is completely implemented in the WAD-QC 2.0 framework, and since Orthanc needs PostgreSQL as a database engine (or SQLite), PostgreSQL was picked as the database of choice for WAD-QC. Although using different databases engines for WAD-QC and for Orthanc is possible, it would add another service to the lists of running services, and thus another process to manage. There are some plans by the developer of Orthanc to implement a MySQL back-end. When that is available, MySQL could be implemented completely in WAD-QC 2.0 as well.

Wait for user input For some quality tests it can be necessary that the user supplies some extra information, in addition to a new data set, before a test can be performed. In those cases the *Processor* should not start processing the job before all information is present. Because of a lack of proper uses-cases for these kinds of tests at this time, it was not meaningful to implement this feature yet.

Fill SelectorRules from DICOM It would be nice to have an option when building *SelectorRules* to pick DICOM tags and fill their accepted contents by selecting a DICOM image that should be picked up by the selector.

Filters in config.json Selectors expect either a dcm_study, or a dcm_series, or a dcm_instance. If a dcm_study is used as input, it might be nice to use filters in the *module_config* to indicate what dcm_series of the dcm_study should be used for a specific analysis. Although these filters are implemented, they have not been tested yet because of a lack of proper uses-cases.

Chapter 3

Module guidelines

3.1 Analysis module

An analysis module is a stand-alone executable, that accept 3 parameters:

-c `config_file`

-d `study_data_folder`

-r `results_output_file`

The `study_data_folder` should be a three level deep folder structure Study/Series/Instances, where one study, but multiple series with multiple instances each are allowed.

The `results_output_file` is the name of the JSON file containing all the results of the analysis.

3.2 JSON files

WAD-QC 2.0 has adopted the JSON standard for files. In python terminology, JSON files are plain text files to store lists, and dictionaries (and combinations thereof) of key-value pairs.

A few points of note for constructing a JSON file:

- The start and end of a list are marked with square brackets.
- The start and end of a dictionary are marked with curly brackets.
- All strings (including keys) are given as double quoted strings, regardless of the type of value they represent, be it floating point numbers or strings or whatever.
- List elements and dictionary elements are separated by a comma, where a comma indicates that a new element will follow. Therefore, do not leave a trailing comma after the last element of a list or dictionary.

3.2.1 results.json

Figure 3.1 shows an example of a results.json file.

A results.json file contains a list of results. Each result is a dictionary, stating the **name**, the **category** and the **val** of the result.

The **name** of a result is the identifier of the result, and can be picked at will.

The **category** of a result indicates in which table the result should be stored, and thus what kind of constraints can be applied to that result. Valid categories are 'float', 'string', 'bool', 'object', and 'datetime'.

The **val** of a result gives the value of named result.

```
[
  {
    "name": "MeanCenter",
    "category": "float",
    "val": "2.3433962264150945"
  },
  {
    "name": "CTslice",
    "category": "object",
    "val": "/full/path/to/test.jpg"
  },
  {
    "name": "pluginversion",
    "category": "string",
    "val": "20160902"
  },
  {
    "name": "AcquisitionDateTime",
    "category": "datetime",
    "val": "2015-01-13 10:53:54"
  },
  {
    "name": "CurrentModulation",
    "category": "bool",
    "val": "True",
    "constraint_equals": "True"
  }
]
```

Figure 3.1: Example of a results.json file, showing the different types of results. Note the usage of double quotes, brackets and commas.

3.2.2 *module_config*

See Figure 3.2 for a *module_config* example. A *module_config* is a JSON file containing a dictionary of keys 'actions' and 'comments'. The values corresponding to these keys are again dictionaries.

actions A analysis module will choose the analysis function to apply (a so-called **action**) to a provided dataset. An **action** is defined in the *module_config* by a key in the actions dictionary. The corresponding value is a dictionary of two dictionaries 'filters' and 'params'. Multiple **action**s can be applied to the same data, which will produce one results.json with all results combined.

In the 'filters' dictionary different DICOM tags with values can be supplied to select a subset of the dataset as input for the **action**. For example, if a multi-series DICOM study is used as input for an analysis module, a specific DICOM series can be picked from the DICOM study for a specific **action**, based on the Series Description. WARNING: the usage of these filters has not been thoroughly tested yet!

In the 'params' dictionary, values for run-time parameters can be supplied. If each **param** is defined as a single value (i.e. not a list or dictionary) *WAD-Admin* can be used to modify the values of those parameters. This means that it is up to the developer to provide a way to enter lists of values as a single value. A possible solution is shown in Figure 3.2 line 13, where the x and y coordinates of a point are provided as a single string containing a semi colon to separate the two values.

comments The developer is free to put whatever desired information about the analysis module and this specific *module_config* in a key-value format in the 'comments' dictionary. Please note that the idea is that this information is somehow relevant for reproducing results, like the author of the file. Other kinds of comments should be in the companion *meta_config* file.

If a dictionary 'params' is supplied here, *WAD-Admin* will use that to display that info next to the parameter, so that a user has some information on the meaning or allowed values of a **param**.

```
1  {
2      "actions": {
3          "acqdatetime": {
4              "filters": {},
5              "params": {}
6          },
7          "qc_series": {
8              "filters": {},
9              "params": {
10                 "auto_suffix": true,
11                 "linepair_type": "RXT02",
12                 "pidmm": 70,
13                 "xymm0.6": "-83.0;-25.0",
14             }
15         }
16     },
17     "comments": {
18         "author": "Arnold Schilham, UMCU",
19         "creator": "generate_config_json.py version 20160822",
20         "description": "DX/Normi13 module for Philips Digital Diagnost R3",
21         "params": {
22             "auto_suffix": "add suffix based on detector name to results",
23             "linepair_type": "must be RXT02 or typ38",
24             "pidmm": "distance between phantom and image detector",
25             "xymm0.6": "position of the dot in 0.6"
26         },
27         "version": "20160825"
28     }
29 }
```

Figure 3.2: Example of a *module_config* JSON file, showing the usage of 'actions' and 'comments'. Note the usage of double quotes, brackets and commas.

3.2.3 *meta_config*

The *meta_config* is meant to be used by the reporting tool only, and should not contain any information needed by the *Processor*. If *WAD-Dashboard* is used as the reporting tool, the format of Figure 3.3 is used. Please note that only if this format is used, *WAD-Admin* can be used to change the values

of the constraints. For all other changes *WAD-Dashboard* should be used. The reason to make this mixed-up use of tools, is to provide a single page for altering the run-time parameters, action limits, and *selection_rule*s of a selector; as most of those settings belong to the domain of *WAD-Admin*, access to the action limits was put there as well.

A *meta_config* is a JSON file containing a dictionary of keys 'results' and 'comments'. The values for 'comments' and 'results' are again dictionaries.

comments The developer is free to put whatever desired information about this specific *meta_config* in a key-value format in the 'comments' dictionary.

results Each **result** is entered as a dictionary of information to be used when visualizing the data. By default *WAD-Dashboard* only displays results that are mentioned by name in this dictionary. By logging in as 'root' to *WAD-Dashboard*, results that are not named in this dictionary can be added to the dictionary. Right now, the following information is supported:

name This is a required field and should match the **name** of a result as contained in the result.json.

display_level This is a required field to indicate what user should be able to see this result: 0=admin, 1=key-user, 2=normal user.

display_name An optional field to alter the name of the result when displaying.

display_position An optional field to fix a certain order when displaying results. If not provided, alphabetical ordering is used.

description An optional field to provide a more complete description of the result.

units An optional field to provide the units of the measurement value.

constraint_equals An optional action limit to this result, meaning the measurement value should be equal to this constraint value. Is applicable to all types of results, except objects.

constraint_period An optional action limit to this result, meaning that this measurement should be done once every x days, with x the constraint value. Is only applicable to DateTime results.

constraint_minlowhighmax An optional list of action limits, stating the minimally allowed, the minimally acceptable, the maximally acceptable and the maximally allowed value for the measurement value. Is only applicable to Float results.

constraint_refminlowhighmax An optional list of action limits, stating the reference value, and the relative values for the minimally allowed, the minimally acceptable, the maximally acceptable and the maximally allowed value for the measurement value. Is only applicable to Float results.

WAD-Dashboard can be used to change the information of 'display_level', 'display_name', 'display_position', 'description', and 'units'. Furthermore, *WAD-Dashboard* can be used to (temporarily) disable a constraint. The values of the constraints themselves can only be changed through *WAD-Admin*.

Note the constraints provided in the *meta_config* are treated as static. Dynamic constraints should be provided with the results themselves (as for example in Figure 3.1 line 26). If a constraint is given both in *meta_config* and in the results.json, the *meta_config* values will be used.

```
{
    "comments": {
        "author": "Arnold Schilham, UMCU",
        "creator": "generate_config_json.py version 20160822",
        "description": "CT/QCCT_wadwrapper (QuickIQ) module for Philips iCT 256"
    },
    "results": {
        "AcquisitionDateTime": {
            "constraint_period": 14,
            "description": "date and time of acquisition",
            "display_level": 2,
            "display_name": "DateTime",
            "units": ""
        },
        "Patient Position": {
            "constraint_equals": "HFS",
            "display_level": 1
        },
        "Protocol Name": {
            "display_level": 1,
            "display_name": "Protocol HEAD"
        },
        "unif": {
            "constraint_minlowhighmax": [
                -3.5,
                -2.5,
                -0.5,
                0.5
            ],
            "description": "Non-uniformity of phantom",
            "display_level": 2,
            "display_name": "Non-uniformity HEAD",
            "units": "HU"
        }
    }
}
```

Figure 3.3: Example of a *meta_config* JSON file, showing the different types of results. Note the usage of double quotes, brackets and commas.

Chapter 4

API Layers

4.1 DBIO

To make WAD-QC 2.0 independent of a specific database engine, a *dbIO layer* was developed. A minimal example for accessing the WAD-QC database is shown in Figure 4.1.

4.2 PACSIO

To make WAD-QC 2.0 independent of a specific PACS, a *PACSIO layer* was developed. For each PACS type a wrapper providing the following functions must be provided:

- getSharedStudyHeaders(studyid)
- getSharedSeriesHeaders(seriesid)
- getInstanceHeaders(instanceid)
- getSeriesIds(studyid)
- getStudyIds(patientid=None)
- getPatientIds()
- getInstancesIds(seriesid)
- getStudyId(seriesid=None, instanceid=None)
- getData(data_id, data_type, dcmfolder)
- uploadDicomFile(filename)
- uploadDicomFolder(folder)
- deleteData(data_id, data_type)

Right now, this wrapper only exists for Orthanc.

```
from __future__ import print_function # python2 compatibility in python3
from os import path
from wad_qc.connection import dbio

def test(selname=None):
    """
    dump some results values of a selector
    """
    # get a selector
    if selname is None:
        selname = dbio.DBSelectors.get().name # get the name of the first one
    print("test for {}".format(selname))

    # find the selector in the dbase as the first match with this name
    sel = dbio.DBSelectors.get(dbio.DBSelectors.name==selname)

    # get all results for this selector
    results = sel.results

    # dump the name and the value of all float results
    print("res_i, float_i, name, value")
    for i,res in enumerate(results): # loop over all results
        for f,val in enumerate(res.floats): # loop over all floatresults
            print(i, f, val.name, val.val)


# __main__
    # construct the full path to the INIFILE needed for WAD-QC db access
    wadqcfolder = path.expanduser(path.join("~","WADROOT","WAD_QC"))
    inifile = path.join(wadqcfolder,"wadconfig.ini")

    # open connection to db
    try:
        dbio.db_connect(inifile)
    except Exception as e:
        raise RuntimeError("Cannot connect to WAD-QC database")

    test(selname="Normi13")
```

Figure 4.1: Example of a accessing the WAD-QC database through the *dbIO layer*.

Bibliography

[1] S. Jodogne, *Orthanc: open-source lightweight DICOM server*, `http://www.orthanc-server.com`

[2] C. Leifer, *peewee: a simple and small ORM*, `http://docs.peewee-orm.com`

[3] A. Ronacher, *Flask: a microframework for Python based on Werkzeug, Jinja2 and good intentions*, `http://flask.pocoo.org`

[4] Continuum Analytics, *conda: open source cross-platform packaging system* `http://conda.pydata.org`

[5] BigSQL Corp, *BigSQL: an open-source developer-friendly distribution of PostgreSQL binaries*, `http://bigsql.org`