| | | | |
|---|---|---|---|
| | ROC MUsIC Software Quality Analysis report | | |
| Date: | 26/04/2019 | Issue: | 1.0 |
| Reference: | SOLO-GS-RP-2460-CNES | | |
| Custodian: | Dominique Bagot (PAQA CNES) | | |

| Prepared by: | | Date: | Signature: |
|---|---|---|---|
| Dominique Bagot | Software Quality Engineer | | |
| Contributors | | Date: | |
| | | | |
| Approved: | | Date: | Signature: |
| Desi Raulin | Ground Segment Development Manager | | |

| Issue | Date | Page | Description Of Change | Comment |
|---|---|---|---|---|
| 1.0 | 26/04/2019 | all | First version | Report delivered to Lesia laboratory |
| | | | | |

# 1. Table of contents

# 2. Purpose and scope

## 2.1 Purpose

The purpose of this document is to describe the results obtained in the software quality analysis and code inspection of the **ROC MUsIC software** product.

**First objective is to HELP the development team.**
**Please contribute to improve this report.**
**Any comments, ideas are welcome!**

Other objectives are:

- Deliver a *Quality status* on the code;
- *Communicate* it to the code authors, the whole development team and managers;
- Possibly set-up *action plan* for improvement.

For each of the measurements, we cover the following items:

- What is measured and why;
- The measurement tool(s) used;
- The measurement results;
- An analysis of the results and, potentially, actions to be carried out.

The conclusions are derived from good practices and should be taken as a guide instead of a prescription.

*This analysis has been done without knowledge (science, SW implementation…) on this project.*
*Please do not hesitate to mention any error or misunderstanding.*

In the remainder of this report:

*Metrics and their rationale are given in italic blue.*

➔ For each metric, recommended value and applicable vales (from [AD6]) are systematically reminded.

**Proposals for actions** are provided in an orange box.

When a metric is over the applicable value, a red font is used (otherwise the orange font).

The list of the metrics used in this document, with their definitions and thresholds, are in the annex §10 page 37.

The values of these metrics collected by the tools Understand and sonarQube are attached in this file: 
MUsIC_Understand_metrics.xlsx
Other definitions and more details are also in this document.

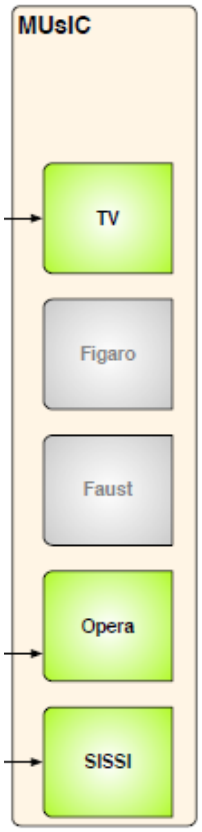
**This is the first quality SW analysis report on ROC ground software.**

A second report (planned on the RSSVC4 milestine, septembre-octobre 2019 TBC), should be produced on more matured source code and covering more functional features (and requirements).

## 2.2 Scope of the analysis

**In this first analysis:**

- Data models are not part of this (software) analysis
- The test folders and test files are not taken into account for this analysis
- **Only Python files have been selected** (*.py) (no analysis done on javascript files)
- the recommandations in this document do not apply on files generated by Django.




The product is introduced in §3 page 7.

This anaysis has been done on the main parts of MusIC:

- TV
- Figaro
- Faust

Opera and SISSI will be analyzed later: the specifications of Opera have to be refined an the SISSI product will be validated on the flight acceptance phase.

## 2.3 Applicable documents

| AD | Title / Author | Document Reference | Issue |
|---|---|---|---|
| 1 | ROC Software Assurance /Product Assurance Plan (SPAP) | ROC-GEN-MGT-QAD00033-LES | 1.2 |
| 2 | Quality Assurance Specification for Software Development with Laboratories | DNO-DA-AQ-2017-0016646 | 1.0 |

## 2.4 Reference documents

### General links

| RD | Description | Adress |
|---|---|---|
| 1 | sonarQube tool: Metrics definitions | https://docs.sonarqube.org/display/SONAR/Metric+Definitions |
| 2 | Understand tool: Metrics & definitions | https://scitools.com/support/metrics_list? |
| 3 | Clean code - A handbook of agile software craftsmanship R. C.Martin | https://sites.google.com/site/unclebobconsultingllc/books |
| 4 | Refactoring techniques | https://refactoring.guru/refactoring |
| 5 | Refactoring – Coupling and Cohesion | M. Fowler. Refactoring. Addison-Wesley, 1999 https://martinfowler.com/books/refactoring.html |
| 6 | Metrics definitions | https://www.ndepend.com/docs/code-metrics |
| 7 | How to save on software maintenance costs | http://asq.org/public/wqm/how-to-save-on-software-maintenance-costs.pdf |
| 8 | Python and Django coverage | ○ https://django-testing-docs.readthedocs.io/en/latest/coverage.html<br>○ https://www.bedjango.com/blog/package-week-coverage-django/<br>○ https://coverage.readthedocs.io/en/coverage-4.4.2/config.html |
| 9 | licences used by the French administrations | https://www.data.gouv.fr/fr/licences |
| 10 | PEP 8 | https://www.python.org/dev/peps/pep-0008/ |

### Project

| RD | Title / Author | Document Reference | Issue |
|---|---|---|---|
| 11 | ROC Glossary of terms | ROC-GEN-OTHNTT-00045-LES | 1.0 |
| 12 | ROC Engineering Guidelines | ROC-GEN-SYSNTT-00008-LE | 1.1 |
| 13 | ROC Project Management Plan | ROC-GEN-MGT-PLN-00013-LES | 1.4 |
| 14 | ROC Software Development Plan | PLN-00015-LES | 2.1 |
| 15 | ROC Concept and Impelement Requirements Document (CIRD) | ROC-GEN-SYS-PLN00002-LES | 1.4 |
| 16 | ROC Software System Design Document (RSSDD) | ROC-GEN-SYS-SPC00036-LES/00 | 1.0 |
| 17 | ROC Software System Specification (RSSS) | ROC-GEN-SYS-SPC00026-LES | 1.0 |
| 18 | ROC Software System User Manual | ROC-GEN-SYS-SUM-XXXX-LES | N/A |

# 3. Information on the project and product analysed

## 3.1.1 Context of the analysis: periodic software quality analysis

This analysis has been done within the frame of periodic software quality analyses, at least one per year, or one per minor version number (m in number version M.m.p).

## 3.1.2 Development team and stakeholders

LESIA is in charge of the global project management, and of operations planning. This includes the definition of interfaces, the writing of the software tools, and their usage.

The table below lists the main stakeholders of the product analysed:

| | |
|---|---|
| **ROC Project manager** | Xavier Bonnin |
| **RPW Project Investigator (PI)** | Milan Maksimovic |
| **ROC Lead software developer** | Sonny Lion |
| **ROC Product PAQA lead** | Stéphane Papais |

More details can be found in the ROC Project Management Plan [RD 13] and ROC Software Development Plan [RD 14].

## 3.1.3 ROC software products overview

The ROC Software System (RSS) definition gathers all of the engineering systems required to reach the ROC functionalities defined in the CIRD [RD 15]. The specification requirements of the RSS can be read in the "ROC Software System Specification" document (RSSS) [RD17], and the RSS design in the "ROC Software System Design Document" (RSSDD) [RD16].

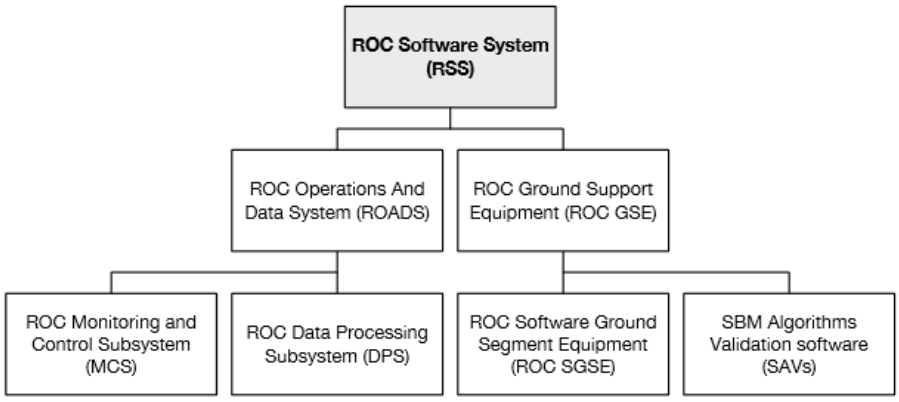



**Figure 1: ROC software System product tree [RD 14]**

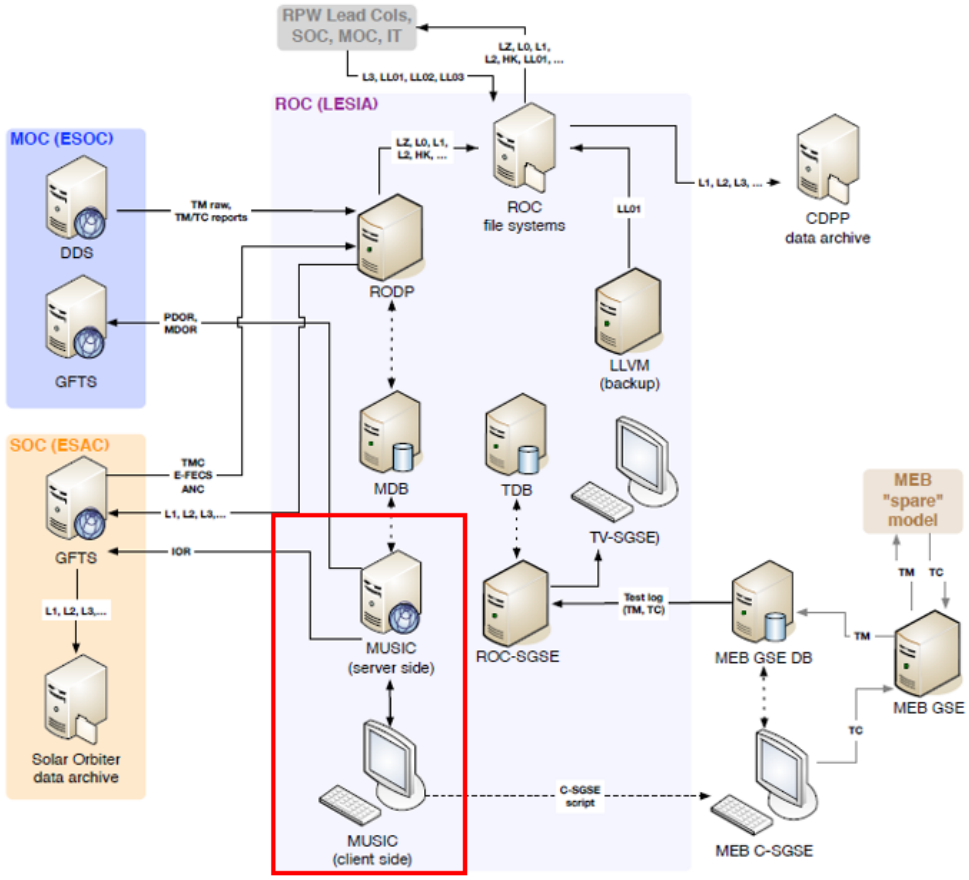



**Figure 2: RSS overall design (MUsIC highlighted in red)**

The ROADS are six main software tools, regrouped into the MCS and DPS sub-systems. One of them is the **MCS User Interface (MUSIC).**

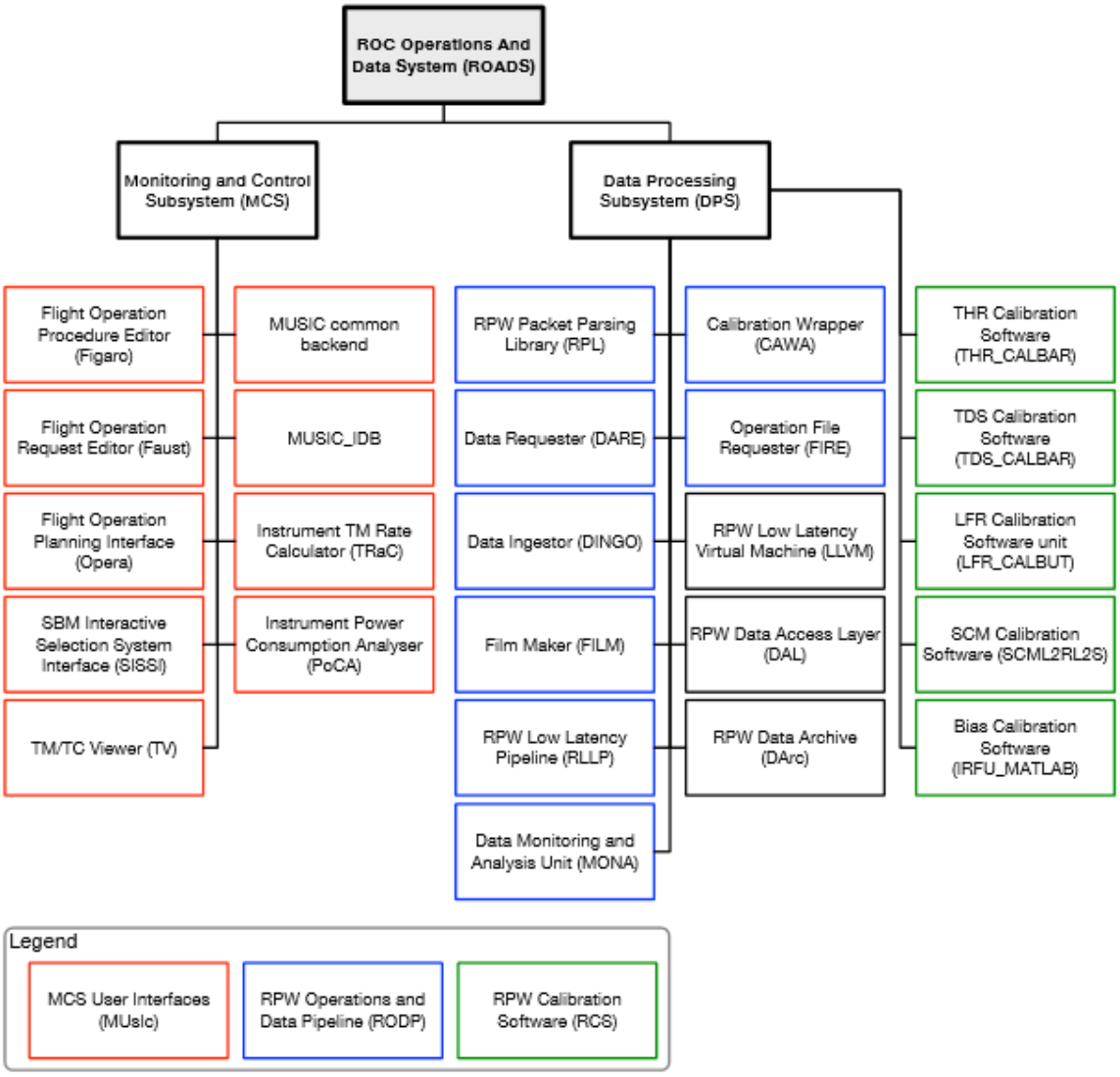



**Figure 3: ROC Operations And Data System (ROADS) software products [RD 14]**

MUSIC is a Web tool allowing ROC operators to view the mission planning, prepare and submit the operations requests, but also monitoring downlink/uplink TM/TC data flows and analysing incoming RPW data.

- The ROC Software Development Plan [RD 14] is clear and describes nearly all the software components.

  This (first) analyses one of the ROC software tools. The next one should embrace all of them.

### 3.1.4 The MCS User Interfaces (MUSIC)

The MCS USer InterfaCes (MUSIC) is a Web interface, dedicated to the preparation of the instrument operations and to the instrument data monitoring.

The **MUSIC frontend** is composed of five tools [RD 14]:

- The RPW TM/TC Viewer (TV), used by ROC operators to promptly visualize the instrument status, TM/TC history and statistics, as well as the HK/science data.
- The RPW Flight Operation Procedure Editor (FIGARO), to create the RPW flight procedures (RFP) in the expected format.
- The RPW Flight Operation Request Editor (FAUST), to prepare and submit to the SOC/MOC the Instrument Operations Requests (IOR) in the expected format, and in accordance with the mission planning constraints.
- The RPW Operation Planning Interface (Opera), to visualize the mission and instrument planning and constraints (i.e., allocation resources) and prepare the operations timeline.
- The SBM Interactive Selection System Interface (SISSI), to manage and select the SBM1/SBM2 event data to downlink.

The **MUSIC backend** is composed of the following components [RD 14]:

- MUSIC common backend; the main backend of the MUSIC Web tool, which relies on the Django framework architecture.
- MUSIC_IDB; a module providing a database model to the other MUSIC backend components, in order to access the RPW instrument Database (IDB) in a standard way.
- The IDB used by the ROADS is stored in the ROC MDB. The database model is the same than for MUSIC (i.e., Django database model).
- Instrument TM RAte Calculator (TRAC); a module dedicated to the TM data rate computation for a given instrument state. Especially, this module serves to compare the instrument states against the Telemetry Corridors (TMC) provided by the SOC.
- Instrument POwer Consumption Analyser (POCA); a module to check the instrument power consumption.
- INstrument Commanding Automaton (INCA); a module in charge of managing the instrument state model (ISM) of MUSIC.

The architecture of the MUSIC backend also has an interface with the MDB to retrieve/store related data and meta-data.

# 4. tools and source code inputs

## 4.1 Environnent

The following table shows the environment and tools used for the analysis of the code.

| Name | Version |
|---|---|
| Understand | 3.1 (2014) |
| pylint | 1.6.5 |
| sonarQube | 7.4 (with the CNES applicable configuration) |

The SW quality tools below are not in the project framework (including sonarQube). Their results are complementary to sonarQube results.

- *Understand* [2] has been used to analyse the design (dependencies between files) and to get detailed metrics (down to the method level).

The definitions of the metrics of both *sonarQube* and *Understand* are provided in annex §10 page 37.

## 4.2 Code analysed

The analysis has been carried out over the source code in the GitLab repository. The following table shows the repository information at the time of the analysis.

| | |
|---|---|
| **Location in CM tool** | https://gitlab.obspm.fr/ROC/MUsIC/-/archive/develop/MUsIC-develop.zip |
| **Location in sonarQube** | N/A |
| **Release major changes** | under current development phase (no official release)<br><br>Note: a Software Configuration File (or Software Release Note) [AD 2] is expected for the next official delivery (RSSVC4 milestone) |

This analysis is mainly based on metrics. The advantage is to cover large number of lines of code. In order to be close to the "real" source code, a (too) short analysis has been done on a piece of code: see section §8 Inspection of pieces of code page 32.

## 4.3 Code top-level structure

The ROC Software System Design [RD 2] summarizes in §5.1 the main components and data composing the system:

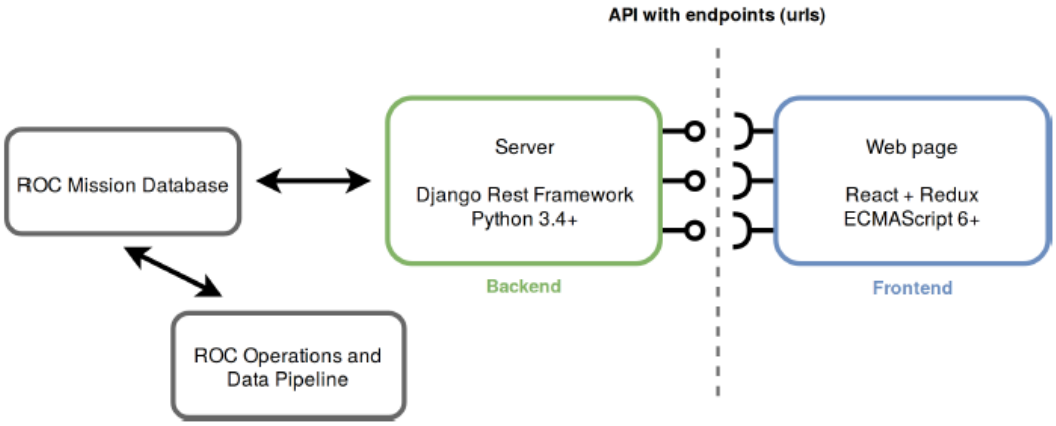


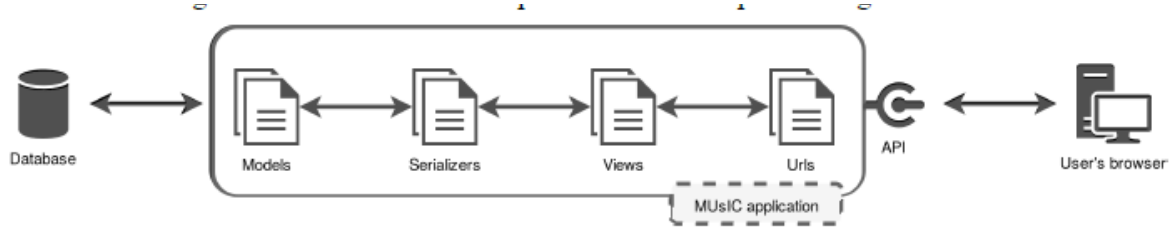

Figure 4: MUsIC architecture overview

**Backend software:**

It is composed of these folders:

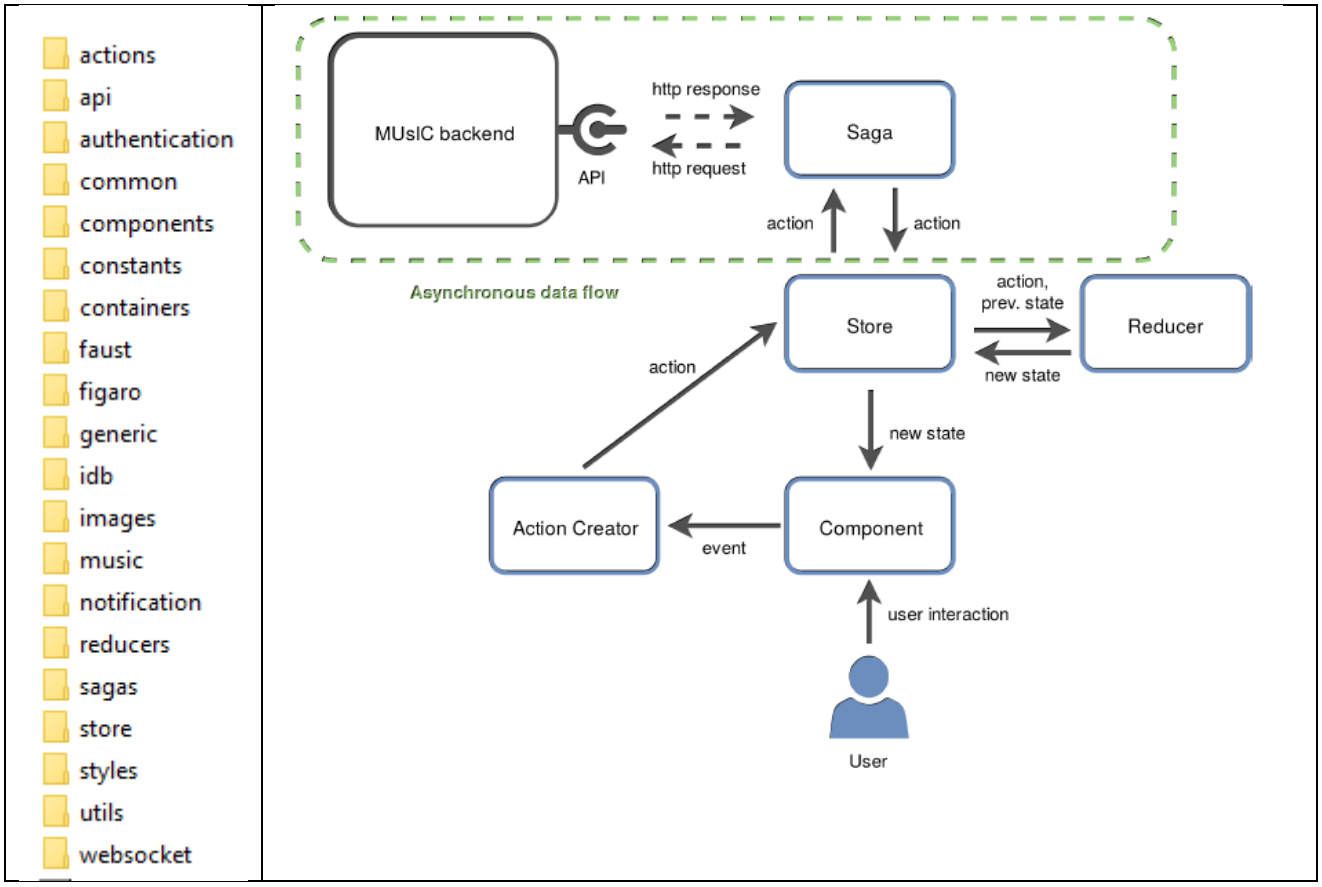
- accounts
- faust
- figaro
- lib
- music
- plots_static
- templates
- tv
- tv_plot

The "dynamic" of the inernal parts is represneted as below:



**FrontEnd software:**

The frontend is based on Reactjs (javascript library) and Redux (for organizing data).
It is composed of these folders:



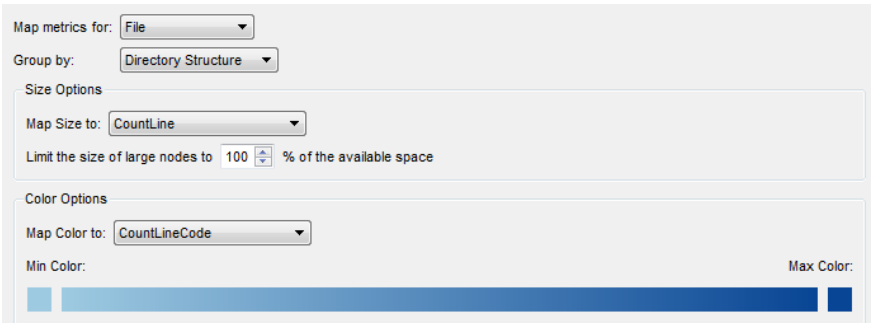
**Full contents of the analyzed folders:**

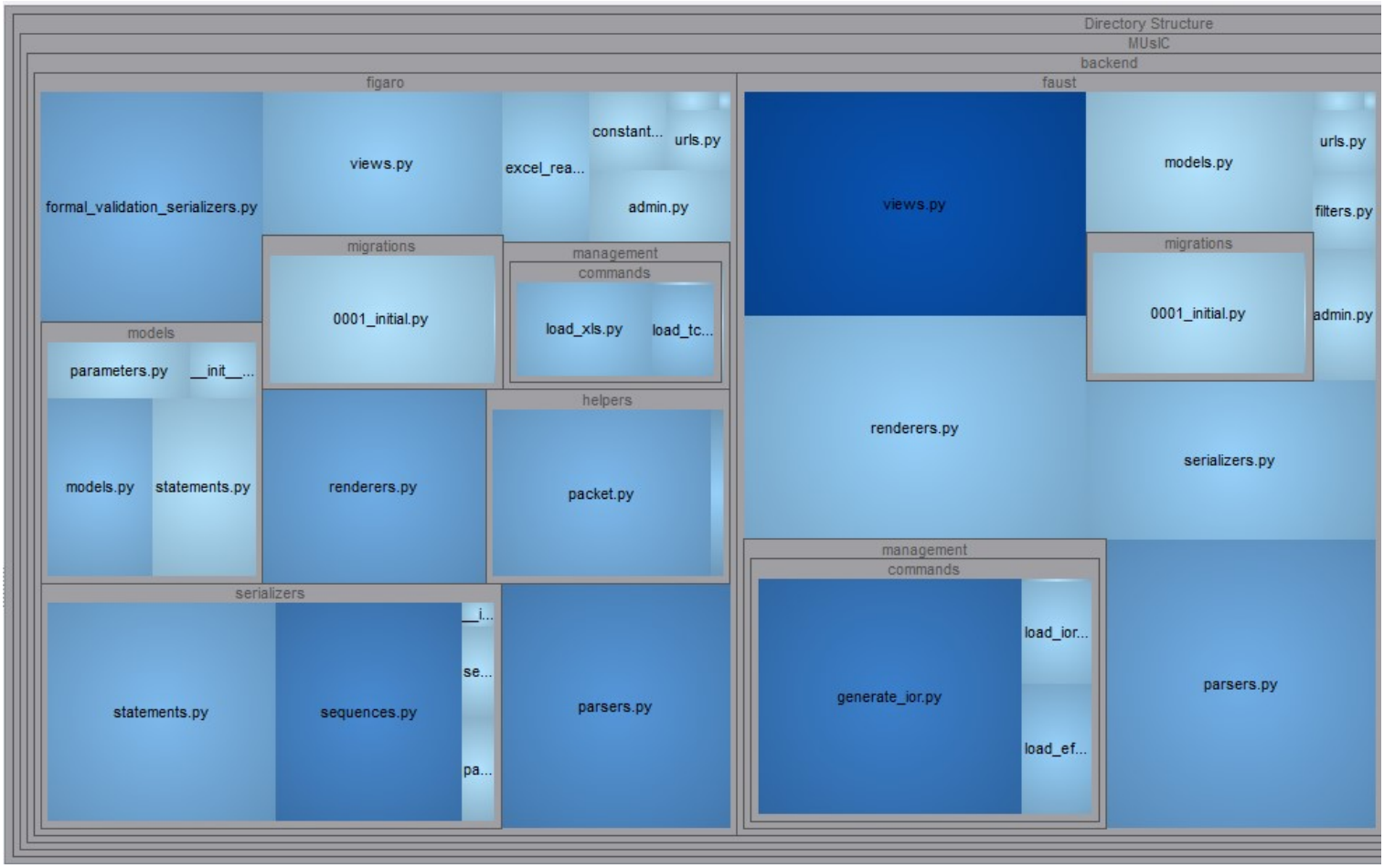
Here are the elements analyzed in this report:

```
.
|-- AUTHORS.rst
|-- backend
|   |-- accounts
|   |-- faust
|   |-- figaro
|   |-- lib
|   |-- manage.py
|   |-- music
|   |-- plots_static
|   |-- templates
|   |-- tests
|   |-- tv
|   `-- tv_plot
|-- docker
|   |-- backend
|   |-- base
|   |-- config.ini
|   |-- docker_pipeline.info
|   |-- frontend
|   |-- pipeline
|   |-- postgres
|   `-- ssh_key
|-- docker-compose.yml
|-- docs
|   |-- make.bat
|   |-- Makefile
|   `-- source
```

```
|-- frontend
|   |-- actions
|   |-- api
|   |-- authentication
|   |-- common
|   |-- components
|   |-- constants
|   |-- containers
|   |-- faust
|   |-- figaro
|   |-- generic
|   |-- history.js
|   |-- idb
|   |-- images
|   |-- index.js
|   |-- music
|   |-- notification
|   |-- reducers
|   |-- sagas
|   |-- store
|   |-- styles
|   |-- tests
|   |-- utils
|   `-- websocket
|-- js_dependencies.py
|-- libs
|   `-- install_cdf.sh
|-- LICENSE
|-- package.json
|-- py_dependencies.py
|-- README.md
|-- requirements
|   |-- base.txt
|   |-- dev.txt
|   |-- prod.txt
|   `-- test.txt
|-- requirements.txt
|-- sonar-project.properties
|-- tox.ini
`-- webpack
    |-- common.config.js
    |-- dev.config.js
    `-- prod.config.js
```

The figure below is a graphical representation of the MUsIC source code (Python source code only):




Directory Structure
MUsIC
backend
figaro
formal_validation_serializers.py
views.py
excel_rea...
constant...
urls.py
admin.py
migrations
0001_initial.py
management
commands
load_xls.py
load_tc...
models
parameters.py
__init__...
models.py
statements.py
renderers.py
helpers
packet.py
serializers
statements.py
sequences.py
__i...
se...
pa...
parsers.py
faust
views.py
models.py
urls.py
filters.py
migrations
0001_initial.py
admin.py
renderers.py
serializers.py
management
commands
generate_ior.py
load_ior...
load_ef...
parsers.py

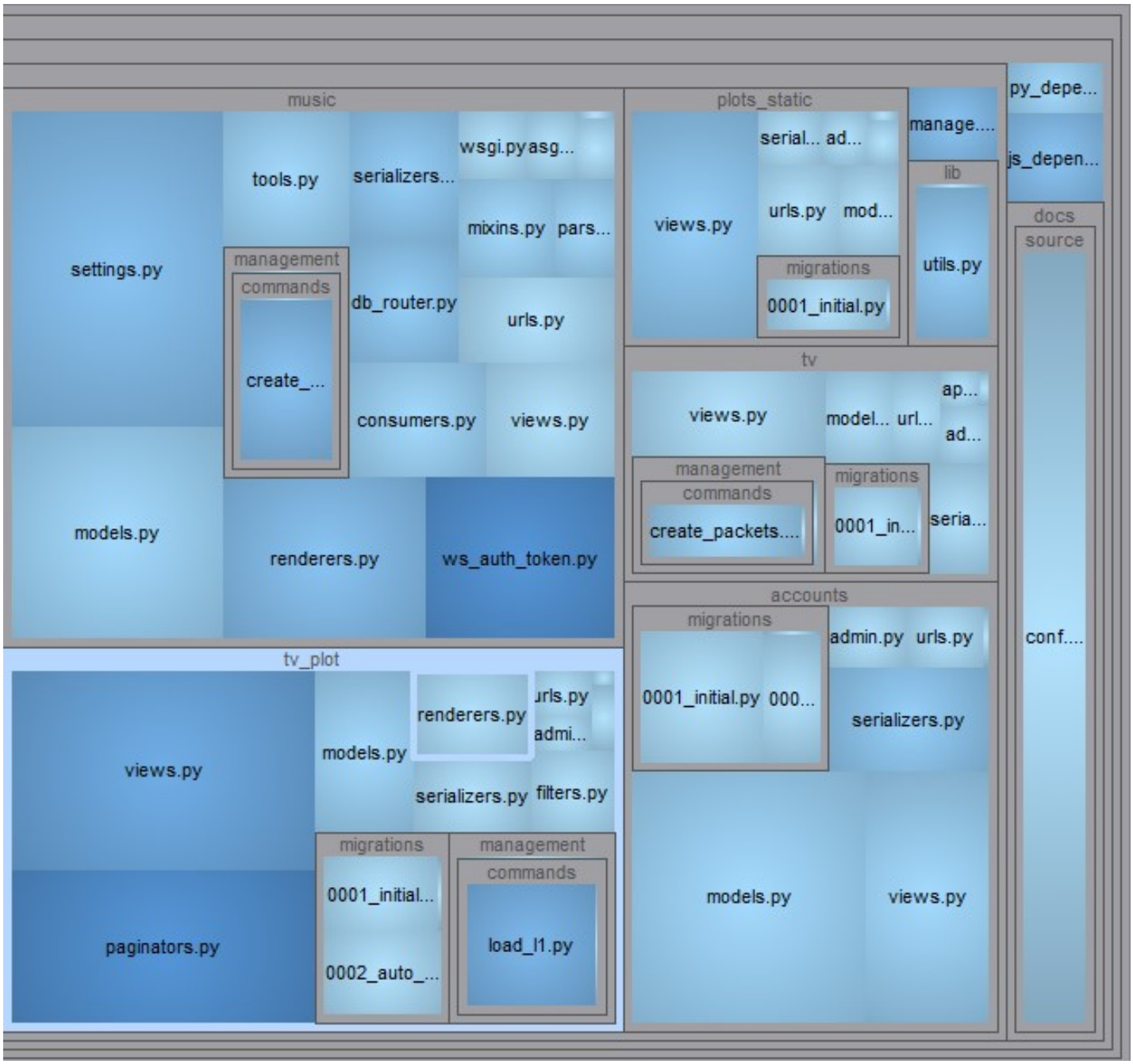



**Figure 5: Treemap view of the MUsIC source code (Python only) (see legend above)**

- The ROC Software System Design Document [RD 16] is clear and describes nearly all the software components.

  An effort could be done on:

- Sections with TBC/TBD, particularly adding **static diagrams (eg class) and dynamic diagrams** (eg sequential).

- the "left to be done", i.e. add more details or quantitative values on the work to be done.

## 4.4 Product size and category

*Some key values give a good indication on the effort to be invested to maintain the project. In the frame of science source code, where projects range from around 10 to $10^5$ lines of code, let us introduce the following categories:*

- ***Small** project: Less than **1,000** lines of code;*
- ***Medium** project: 1,000 to **10,000** lines of code;*
- ***Large** project: **More than 10,000** lines of code.*

*In any case, the famous "rule of 30" is a good guideline to ensure that the maintenance will be reasonable. In terms of metrics, this rule states that:*

a) ***Methods** should not have more than **30 code lines** (not blank counting lines and comments).*
b) *A **class** should contain less than 30 methods, resulting in up to **900 lines** of code.*
c) *A **package** shouldn't contain more than 30 classes, thus comprising up to **27,000 code lines**.*

The table below is extracted from Understand metrics (see annex §10.2 page 38):

**Table 1: Sizing metrics (understand)**

| Item | Count | Average sub-element count |
|---|---|---|
| Python Modules (files) | 115 | |
| Classes | 276 | ~ 2,4 classes per module |
| Methods | 280 | ~ 1 method per class |
| Lines of code | 5138 | 18 lines of code per method |

The project is medium-sized i.e. categorized as a **medium project**.

**The breakdown in directories, files, classes, methods and statements seems globally reasonable at *this level of details*, with respect to the rule of 30.**

The high value of count of classes (and low value of count of methods per class) is due to the Django "usage".
For information 8 python files (.py) contain the header "Generated by Django".
This is only an overview, as introduction: the sections below will provide some details on these values.

# 5. Software engineering compliance

This section provides a status on the compliance of the source code analysed with standard software engineering rules.

## 5.1 Configuration management (GitLab)

**Table 2: Checks on Configuration Management**

| Checks | Results / Comments |
|---|---|
| *The project should be hosted on the project GitLab repository to benefit from continuous integration and deployment;* | Yes: GitLab fully used |
| *Master and develop branches exist (or equivalent).* | Yes: branches exist and used<br>Use tags for official deliveries |
| *Data management: there are no big data files managed under CM* | OK. None file over 1 Mo. |

## 5.2 Product documentation

*Each product version should have developer and user documentation, in order to ease its understanding and future maintenance.*

**Table 3: Checks on Documentation**

| Checks: the product is… | Results / Comments |
|---|---|
| *is described in a specification or/and design document* | Yes document ROC Software System Design Document (RSSDD) [RD 16] |
| *has a Software Configuration File (SCF) or a Software Release Note (SRN)* | NOK, SRN to be initialized<br><br>There is (updated) information in the gitlab website (changelog) |
| *has a Software User Manual (SUM)* | NOK, , SUM to be initialized [RD 18]<br><br>There is (updated) information in the gitlab website |
| *has a managed list of issues (Software Problem Reports)* | OK (in Gitlab) |

- To be discussed with the overall team: Initialize or not these documents:
  - **SRN** (Software Release Note)
  - and **SUM** (Software User Manual)

## 5.3 Generation

*The product should be generated and installed easily and terminated with success.*

**Table 4: Checks on generation tasks**

| Objective: check the execution of these jenkins executions phases.... | Results / Comments |
|---|---|
| *binaries generation (build step)* | OK<br>Documentation in the GitLab site clear and complete<br><br>Note: prerequisistes to detailed |
| *tests execution (After build step)* | NOK |
| *quality tools execution (Quality Analysis step)* | Partially OK (no coverage performed or documented) |

The figure below is a snapshot of the web application.

- Set-up unit tests in order to run them with a unique command.
- Then, **ensure that the structural coverage is measured in sonarQube dashboard**.

## 6. Maintainability

Maintainable software allows to quickly and easily:

- Fix a bug, without introducing a new bug
- Add new features, without introducing bugs
- Improve usability
- Increase performance
- Make a fix that prevents a bug from occurring in future
- Make changes to support new environments, operating systems or tools
- Bring new developers on board the project

The sub-sections intend to check maintainability characteristics from metrics values.

## 6.1 Dependencies

The goals of this verification are the following:
- Help the reader to understand the "dynamic" organization: what calls what?
- Identify packages which depend on many others,
- List packages with cyclic dependencies (package A calls B, which itself calls A).

The tool Understand V3.1 has been used in this section for its results on the dependencies between files: *calls, includes/imports, inherits, implements, inits, overrides, modifies, sets, throws, uses…*

The top-level level dependency graph is the following (divided in 2 parts, for convenience):

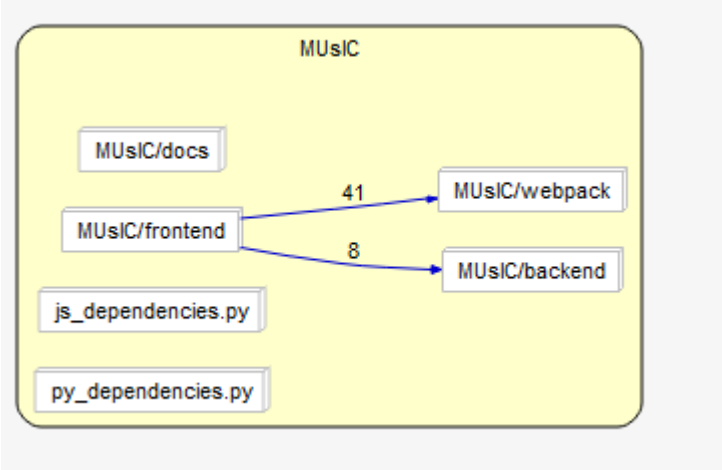



**Figure 6: Top-level dependency graph**

The tool highlighted cyclic dependencies (see the red arrows in the graph above) between:
- components
- modules (see example of dependancy in §10.3 page 40).

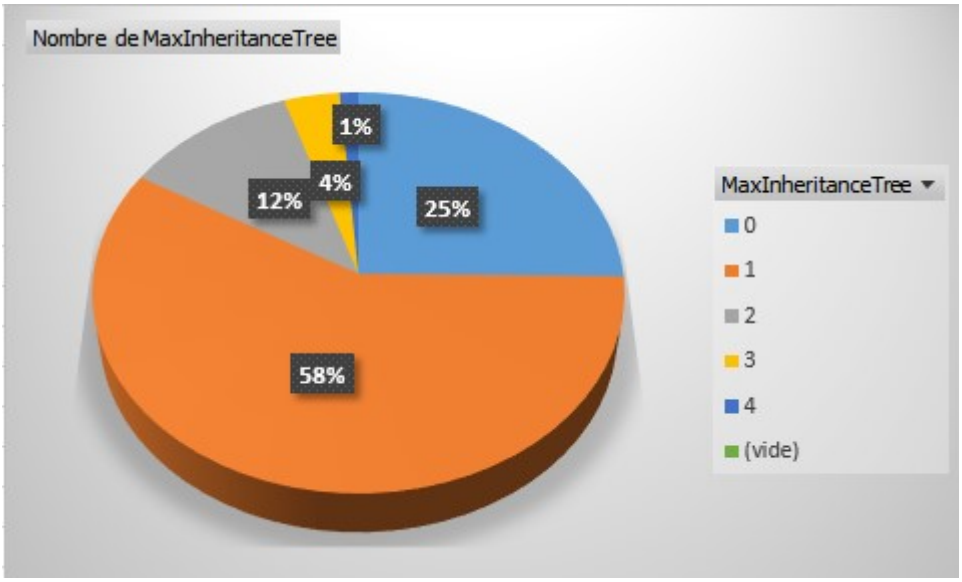
The cyclic dependencies between these methods have not been identified in this report (lack of time).

- Possibly identify the cyclic dependencies, between:
  - **Methods (critical)**
  - **Modules (major)**
- Redesign the classes if necessary (a lot of books and internet sites offer recipes to fix the cyclic dependencies)

## 6.2 Design analysis

Coupling and cohesion are both indications of the quality of the design.
They have not been analysed in this report (no tool available for Python code).
A quick look on the inheritance tree has been performed.

There are **242 classes** : 58 % of the classes have a inheritance tree level (or depth) at 1, level 2: 12%, level 3: 4% and level 4: 25%.



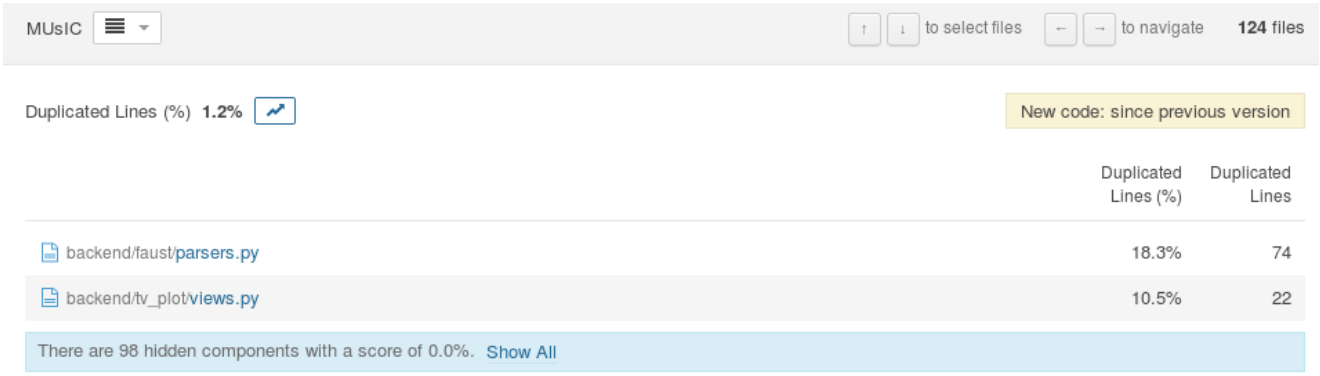

**Comments:**

- When possible, use the Object Programming Concepts (here use inheritance)
- Stick nevertheless to the "good" practices in term of architecture: inheritance has to be implemented only if the subclass is an extension of the superclass, not in order to combine common code (e.g. A new subclass should not violate the Liskov substitution principle [RD 4], [RD 5]).

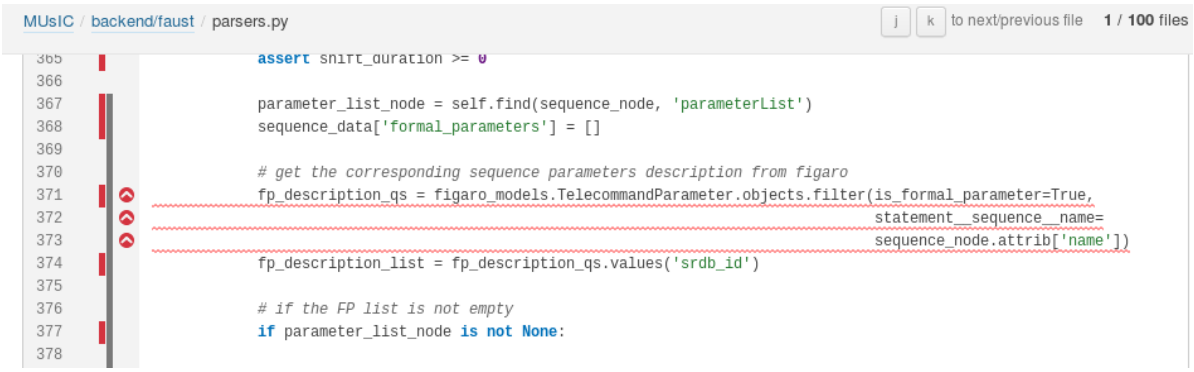
## 6.3 Duplications

*Code duplication is a very important measurement from the maintenance point of view. Indeed, the effort to modify duplicated code might become prohibitive if one has to ensure that duplicated lines should remain the same.* ***Duplication rate should therefore be exactly 0%.***

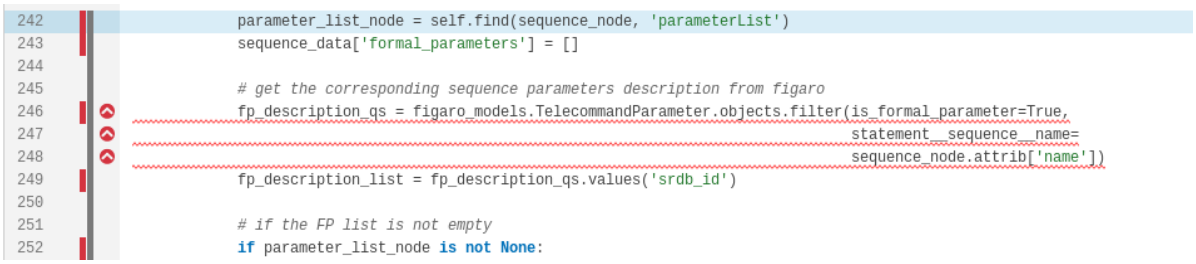
sonarQube is able to detect the number of duplicated blocks of lines (see definition in §10.1 page 37).




Example of duplication block found:




This block is similar as this one, in the same file:




We can consider that this status does not present a risk for the maintenance, considering the Django specificities and “current usage” by developers.

- The count of duplicated lines is not important and considered as acceptable

  Analyse nevertheless each duplicated block of code and, if the duplication is considered by the team as a risk for the maintenance, **try to reduce.**

## 6.4 Sizes and complexities

### 6.4.1 File and class sizes

*As stated in §4.4 page 17, **a class** should contain less than 30 methods, resulting in up to **1000** lines of code.*

*Another close and interesting metric is the number of classes in a source file.*

*Placing each class in an individual file promotes reuse by making classes easier to see when browsing the source code: a reasonable value is consequently 1: a source file should contain only one class.*

The figure below shows the distribution of the number of source lines of code per source code file.

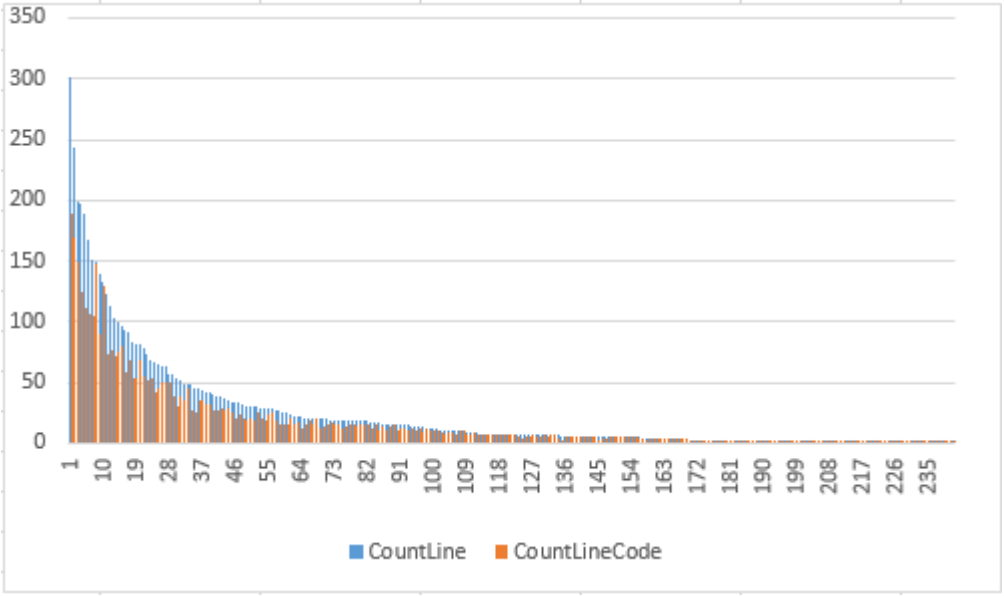
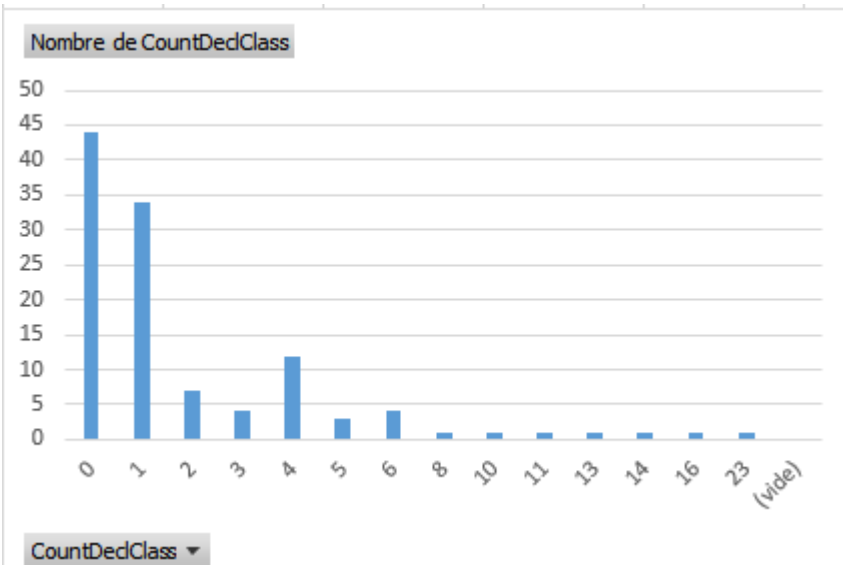



**Figure 7: Number of source lines of code par class (Understand)**

The table below shows the largest files. No one overpassess the recommended count of lines of code (1000 lines).

| Kind | Name | CountDeclClass | CountLine | CountLineCode |
|---|---|---|---|---|
| **File** | MUsIC\backend\faust\renderers.py | 5 | 397 | 267 |
| **File** | MUsIC\backend\faust\parsers.py | 4 | 403 | 264 |
| **File** | MUsIC\backend\faust\views.py | 6 | 398 | 256 |
| **File** | MUsIC\backend\faust\management\commands\generate_ior.py | 2 | 340 | 217 |
| **File** | MUsIC\backend\figaro\parsers.py | 2 | 290 | 204 |
| **File** | MUsIC\backend\figaro\formal_validation_serializers.py | 13 | 266 | 180 |
| **File** | MUsIC\backend\figaro\serializers\statements.py | 14 | 268 | 170 |

The figure below show the distribution of the count of classes by file. For example, 44 files do have any class. 34 files have one class, 7 files have 3 classes etc.




The table below shows the files having the most count of classes. As shown above, a lot of files (37 classes ie 32% of the python files analyzed ) have more than the expected count of classes, i.e 1 class only.

| Name | CountDeclClass | CountLine | CountLineCode |
|---|---|---|---|
| MUsIC\backend\faust\serializers.py | 23 | 238 | 162 |
| MUsIC\backend\faust\models.py | 16 | 166 | 122 |
| MUsIC\backend\figaro\serializers\statements.py | 14 | 268 | 170 |
| MUsIC\backend\figaro\formal_validation_serializers.py | 13 | 266 | 180 |
| MUsIC\backend\figaro\models\statements.py | 11 | 97 | 67 |
| MUsIC\backend\figaro\admin.py | 10 | 53 | 32 |

As stated above, this status is not considered as a risk because due to the Django specificities. There are maybe possibilities of improvement (generic declarations and initializations…) but we leave here the development team to choose their best way to manage their database.
Let's focus on the first file : *backend/faust/serializer.py*. The snapshot below can be considered as a typical way to serialize Django data:

(Django guide: "Serializers allow complex data such as querysets and model instances to be converted to native Python datatypes that can then be easily rendered into `JSON`, `XML` or other content types.")

```
49  class SeqFpSerializer(serializers.ModelSerializer):
50      """
51      SeqFp Serializer
52      """
53
54      class Meta:
55          model = models.SeqFp
56          fields = '__all__'
57          read_only_fields = ('sequence',)
58
59
60  class SequenceSerializer(serializers.ModelSerializer):
61      duration = serializers.IntegerField(read_only=True)
62
63      class Meta:
64          model = figaro_models.Sequence
65          fields = ('duration', 'name', 'description')
66
67
68  class ScenarioSeqSerializer(WritableNestedModelSerializer):
69      """
70      ScenarioSeq Serializer
71      """
72      formal_parameters = SeqFpSerializer(many=True)
```

In summary the distribution of classes in files, due to Django here, seems acceptable.

### 6.4.2 Class contents

*As stated above, the rule of 30 holds for classes, which means that there should be no more than 30 member variables and no more than 30 methods in a class.*

The histogram of variables and methods per class built from Understand's outputs is following:

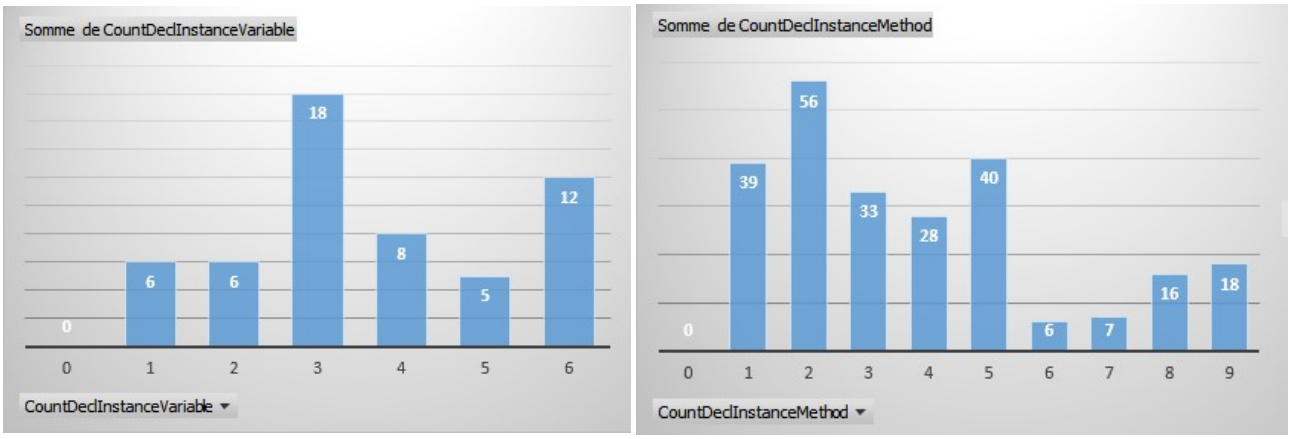



**Figure 8: Number of variables and methods par class**

- 18 classes have 3 variables

- 56 classes have 2 methods

The results are fully compatible with the rule of 30:

- All the classes have less than 30 methods
- All the methods have less than 30 instance variables

A large number of classes show "well-balanced" classes in terms on contents and none class exceeds the limit number of methods or variables (Note: the threshold of 30 is very high: it is recommended in Clean Code [RD 3] *to not exceed 14 methods*).
The table below shows these values and the Maximum cyclomatic complexity of all nested functions or methods (per class).

### 6.4.3 Class complexities

The complexity of a class or method may be measured by different means. The sections below are based on the simplest metrics: lines of Code and cyclomatic Number.

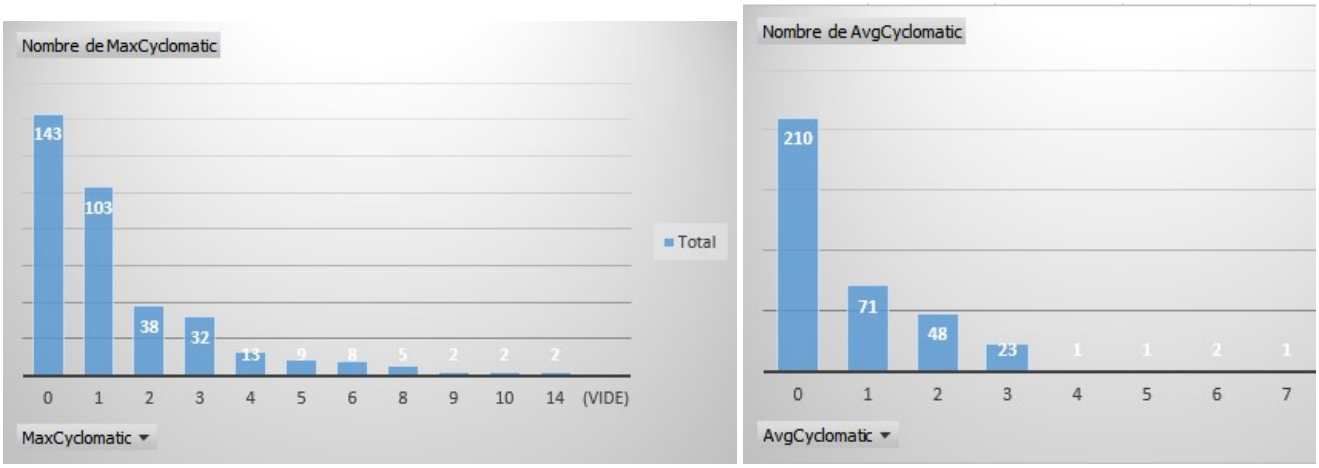



**Figure 9: Class complexities**

The figures above show that all most of the classes have an average cyclomatic value compliant with the max expected (20). and none MaxCyclomatic number is beyond the this recommended value.

### 6.4.4 Method sizes

*As stated in §4.4 page 17, methods should not have more than 30 code lines.*
*The max mandatory value is 100.*

None method overpasses the max value, i.e. has a count of lines of code less than 100.

### 6.4.5 Method complexities

*Functions with too high complexity are error-prone. Functions should be as simple as possible in order to ensure smooth testing and maintenance. To this end, two metrics are evaluated:*

- *The cyclomatic complexity is the number of decision points ("while", "for", "foreach",*
  *"continue", "if","case", "goto", "try" and "catch"...) plus one;*
  *It should be as low as possible, and certainly not higher than 10.*
- *The nesting level is the number of nested blocks (conditions and loops); It is 0 or 1 in ideal cases, and should definitely not be higher than 5.*

*When complexity is too high, a simple solution is to split the method in submethods.*

None method overpasses the max value, i.e. has a complexity higher than 25. In addtion the nesting level is always less t egual to 5.

## 6.5 Headers and comments in the source code

### 6.5.1 Metrics on API headers

*Documenting the API of the project inside the source code is of utmost importance because this is generally the most up-to-date documentation. Specifically, public items should absolutely be documented.*
*Check the sonarQube metric "Density of public documented API", which threshold expected value is 100%.*

Sonar reports (see §11 page 49) show that ...the tool has not been able to collect metrics on API items (type: file/class/method,...).

Another tool has been used: pylint, which output on docstrings is following:

| type | number | % |
|---|---|---|
| code | 4539 | 55.87 |
| docstring | 1122 | 13.81 |
| comment | 750 | 9.23 |
| empty | 1713 | 21.09 |

About 13% of the source code has docstrings and about 9 % has comments.

This quick and straight analyse show an important lack of commenting in the code, on public methods.

**In conclusion, we consider globally that the API headers are missing in the global source code.**

- Major recommendation: add docstrings on the public methods.
- Deploy the expected headers to the whole code (files, classes, methods).
- Be compliant with the Python docstrings format in the Coding Standards [RD 10], i.e. Use NumPy Style for Python Docstrings

### 6.5.2 Global metrics on comments

*Density of comment lines is a degree of commenting within the source code. It measures the care taken by programmers to make the source code and algorithms understandable. Poorly commented code makes the maintenance activities an extremely expensive. Applicable minimum is 30% in the Coding Standards [AD 6].*

***Important note**: this metric has to be balanced with the metric Density of public documented API. It is reasonable to get a low density of comment lines under the expected value in (small) methods which header is complete.*

The average comment density measured by pylint is around 9 % (see above).

The figures below illustrate this statement (e.g. 205 methods have 0 or 1 line of comment)

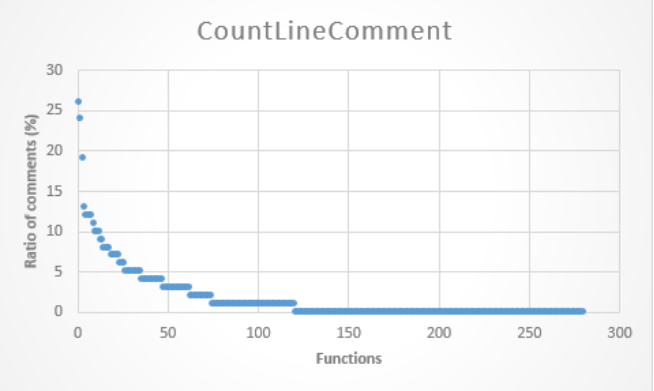



**Figure 10: distribution of comment density**

- As stated above, put the effort on the headers (almost at method level). When done, add comments if necessary.

# 7. Reliability

*Some issues in the code might prevent it to run smoothly (e.g., memory leaks). They should be solved.*

There are only 9 critical issues and 618 major issues reported by sonarQube.
The cost of accumulating technical debt is around 7 days .

## 7.1 Critical Issues

These issues are seen as critical by Pylint and current CNES rule profile file:

| | |
|---|---|
| (Python) Ungrouped imports | 7 |
| (Python) Wrong import order | 2 |

- recommandation on style: possibly re-arrange the oder of the import operations

## 7.2 Major issues

These issues are seen as major by Pylint and current CNES rule profile file:

| | |
|---|---|
| (Python) Docstrings should be defined | 316 |
| (Python) Lines should not be too long | 193 |
| (Python) Source files should have a suf... | 60 |
| (Python) Functions should not contain t... | 27 |
| (Python) Sections of code should not b... | 12 |
| (Python) Undefined variable | 5 |
| (Python) "\" should only be used as an e... | 3 |
| (Python) Function names should comply... | 2 |

We consider in this first analysis that **there are no bugs that could alter the realibility.**

We strongly recommend nevertheless to treat these messages, having an impact on the maintenance cost.

**Reminder**: these recommandations do not apply on files generated by Django. The count of messages (by issue) have been reminded par parenthesis below:

1. Issue “Add a docstring to …” (316) : see recommandation above
2. Issue “The line contains … characters which is greater than 100 authorized.” (193) : use the continuation line recommandations in PEP8
3. Issue “more comment lines need to be written to reach the minimum threshold of 20.0% comment density.” (60) : add comment in targeted methods
4. Issue “This function has 2 returns or yields, which is more than the 1 allowed” (27) : factorize “Return” statement in targeted methods
5. Issue “Remove this commented out code.” (12) : Once the source code will be ready for production, delete the commented out code or replace them by “human” comment (and not statements)
6. Issue “Undefined variable ‘…’” (5): It seems that this is a false error message: to investigated and possibly fixed.
7. Issue “remove the ‘\’ …” (2) : a pattern is used in the targeted methods: no fix recommended.
8. Issue “rename function … to match the regular expression”: follow the naming rule.

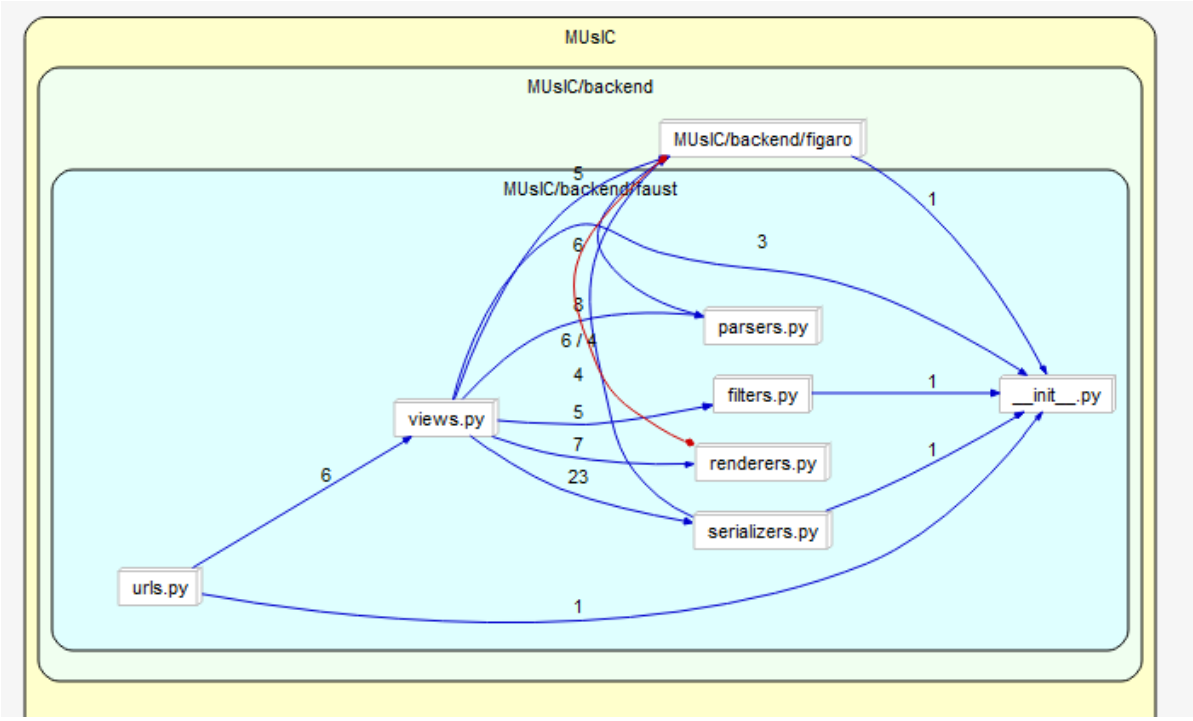
# 8. Inspection of pieces of code

In order to analyse (quickly) a portion of code, let's take an example with the file MusIC/backend/faust/views.py

## 8.1 Introduction

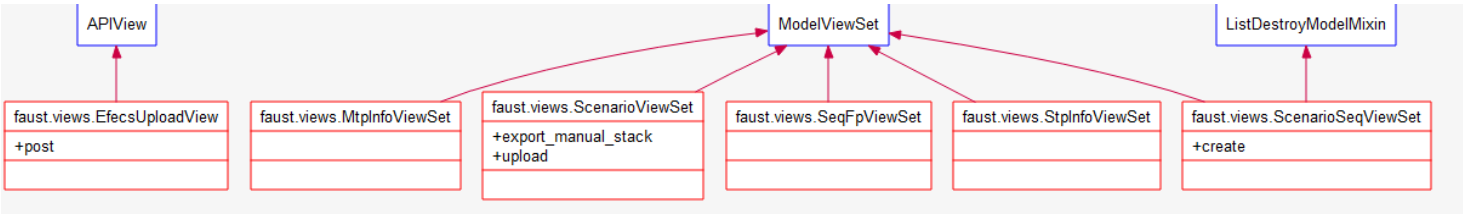
## 8.2 Dependancies

All he possible links around the file are represented here:




*The file contains 6 classes, derived from APIView, ModelViewSet and ListDestroyModelMixin:*

| Name | CountLine | CountLineCode |
|---|---|---|
| faust.views.EfecsUploadView | 26 | 16 |
| faust.views.MtpInfoViewSet | 6 | 6 |
| faust.views.ScenarioSeqViewSet | 45 | 26 |
| faust.views.ScenarioViewSet | 243 | 169 |
| faust.views.SeqFpViewSet | 6 | 6 |
| faust.views.StpInfoViewSet | 6 | 6 |




## 8.3 Headers

The **file header** is not documented and does not include Copyrights . See the link [RD 9]: licences used by the French administrations.

```
1  import io
2  import zipfile
3  from datetime import datetime, timedelta
4
5  from dateutil.parser import parse as parse_datetime
6  # from django.db.models import F
7  from django_filters.rest_framework import DjangoFilterBackend
8  from faust.parsers import EfecsParser
9  from figaro import models as figaro_models
```

There is a (short) description of the **class** and none API description in all the public methods:

When a method has parameters, there are no comments on them (type and description):

```
378     def post(self, request, filename, format=None):
379         xml_data = request.data
380         print('filename:', filename, 'efecs_xml_data', xml_data)
381         print('The following MTPs have been parsed:')
```

The reviewer might have difficulties to distinguish/identify the type of the parameters. It seems clear that priority has to be done on adding headers on methods, like done in numpy math library:

```
def function_with_pep484_type_annotations(param1: int, param2: str) -> bool:
    """Example function with PEP 484 type annotations.

    The return type must be duplicated in the docstring to comply
    with the NumPy docstring style.

    Parameters
    ----------
    param1
        The first parameter.
    param2
        The second parameter.

    Returns
    -------
    bool
        True if successful, False otherwise.

    """
```

## 8.4 Lines of comments

There is only a few comments in this file. As stated above, it seems that 9% of the source code has comments.

**This "numeric" statement should nevertheless been mitigated because the source code is generally very clear and easy to understand (i.e. line by line in a method).**

```
250     def export(self, request, pk=None):
251         """
252         Define specific URL for C-SGSE/IOR exports
253         """
254         # get the scenario instance
255         instance = self.get_object()
256         scenario_type = request.accepted_renderer.format
257
258         # select the appropriated serializer
259         if scenario_type == 'csgse':
260             serializer = serializers.CsgseScenarioSerializer(instance,
261             response = Response(serializer.data)
262         elif scenario_type == 'pdor':
```

### 8.5 Global remarks (on the whole file):

- Class documentation: more information on this class could be useful: prerequisites, limitations, TODO…

- The code is easy to read, i.e. do not have "technical" python lines difficult to understand.
  The difficulty is to disentangle the entities used and also the level of tasks implemented.
  …it is true that the "features" to be implemented are not easy to code: it's not as "structured" as a scientific algorithm, or pure IT topic (as code a linked chain).

  - Methods: they are small (exept *ScenarioViewSet*) and easy to read.
  - They are focused on a single task and globally well named.
  - Unit-testing of these methods *should be easy to set-up*. If tests are easier to write for independent methods, then split the big method up

  - A lot of constructors and public methods do not have checks on the parameters validity (see the Python 3.x, function annotations, variables checks with isinstance(obj, type), issubclass(obj, class), hasattr.. or also Type Enforcement accept/returns decorators from PythonDecoratorLibrary).

    If the development team want to be strictly "pythonic", fulfil then the headers with doctrings and add unit tests with specific input parameters.

  - There are not globally hard coded values

  - Log: use Python libraries (avoid 'print')

  - Exceptions: they are not used in this file et globally in the full code.

## 9. Conclusions and recommendations

This product is the result of an important work done and represents a critical added value for the ROC project (source code, architecture, production environment, documentation…).

After analysis, we have the feeling that each line of the code is the result of both a global analysis (the Specification and design documentation [RD 16] is clear) on the features to be implemented and a response to concrete and "daily" challenges /tasks to implement.

Each line is thus written for an operational goal, contributing to the software performance.

We consider that a major action, even if it already started, has to be continued and even strengthened:

**The source code does not have risks for the realibility.**
**Continue to improve the maintainability**.

This would greatly help future maintainers: recent studies show that some 40-60% of the maintenance effort is devoted to …only understanding the source code [RD 8]

The surveys confirmed also that source code and comments (including headers in classes and methods) are the most important artefacts for understanding a system to be maintained.

The following section lists conclusions and recommendations derived from the analysis described in the remainder of the document.

> As stated in the introduction of this report, these proposals for action are derived from good practices and should be taken as a guide instead of a prescription.
>
> The developers should feel free to implement or not the proposed changes.

## 9.1 Top-priority

1. An important effort should be put in providing more details in the public API documentation: add headers (i.e; docstrings) in the source code, at least on the classes and public methods. This is crucial for understanding the code. And continue to improve it later.

2. Set-up unit tests in order to run them with a unique command (possibly using a simple command)

3. Try to decrease the issues raised in sonarQube and possibly follow the recommandations stated in §7 page 30)

4. Improve the sonarQube configuration: ensure that the structural coverage is measured and reinforce python rules in SonarQube.

5. Avoid cyclic dependencies, at least at module and method level

## 9.2 Other recommendations

a. The ROC Software System Specification [RD 16] is clear and describes nearly all the software components.

   An effort could be done on the “left to be done”, i.e. add more details or quantitative values on the work to be done (ex: add new column in table in §4.3 page 10 and add details as “IORs generation: 40% left to be done…”)

b. A few files and classes are relatively important (lines of code, complexity): pay attention not grow again these entities.

c. Initialize Software Release Note and Software User Manual documents (planned on the RSSVC4 TBC)

d. The file headers is not documented and does not include Copyrights. See the link [RD 9]: licences used by the French administrations

# 10. Annex 1: metrics definition

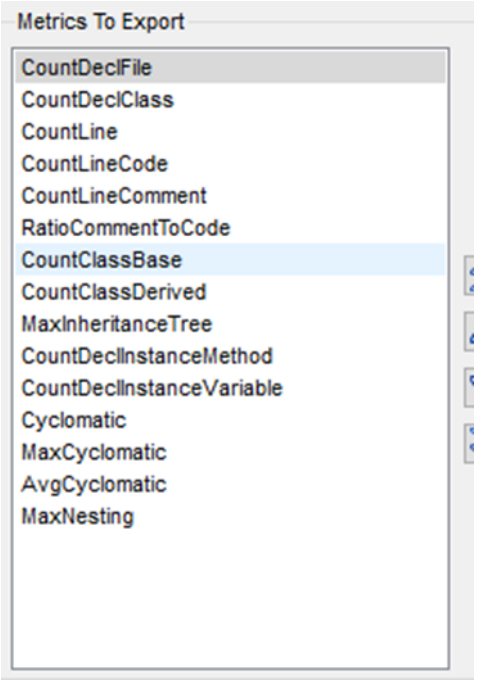
Refer to the Excel file joined for more details (« Metrics definitions » table).

## 10.1 sonarQube

| Category | Metric | Threshold | SonarQube https://docs.sonarqube.org/display/SONAR/Metric+Definitions | |
|---|---|---|---|---|
| | | | Metric | Definition |
| *Documentation* | **Number of comment lines** | | comment_lines | Number of lines containing either comment or commented-o |
| | **Density of comment lines % (Min)** | **0.3** | Density of comment lines | Number of comment lines with respect to total Lines Of Code : Comment lines / (Lines of code + Comment lines) * 100 |
| | **Public documented API % (Min)** | **100** | Density of public documented API | Number of public API comment lines with respect to total Lines Of Code |
| *Complexity* | **Complexity / function (Max)** | **10** | Complexity | It is the complexity calculated based on the number of paths through the code. Whenever the control flow of a method splits, the complexity counter gets incremented by one. Each method has a minimum complexity of 1.<br>This calculation varies slightly by language because keywords and methodalities do. |
| *Design* | **Parameters/function(Max)** | **7** | python: pylint:arguments for function / method python: | max-args=5 |
| *Issues* | **Blocker issues (Max)** | **0** | Blocker issues | Blocker severity : Operational/security risk: This issue might make the whole application unstable in production. Ex: calling garbage collector, not closing a socket, etc. |
| | **Critical issues (Max)** | **0** | Critical issues | Operational/security risk: This issue might lead to an unexpected behavior in production without impacting the integrity of the whole application. Ex: NullPointerException, badly caught exceptions, lack of unit tests, etc. |
| | **Major issues (Max)** | **0** | Major issues | This issue might have a substantial impact on productivity. Ex: too complex methods, package cycles, etc. |
| *Tests* | **Unit tests (Min)** | **1** | Unit tests (Min) | Note:The same kinds of metrics exist for Integration tests coverage and Overall tests coverage (Units tests + Integration tests). |
| | **Unit tests Success % (Min)** | **100** | Unit test success density (%) | test_success_density:Test success density = (Unit tests - (Unit test errors + Unit test failures)) / Unit tests * 100 |
| | **Line coverage % (Min)** | **70** | Line coverage | On a given line of code, Line coverage simply answers the following question: Has this line of code been executed during the execution of the unit tests?. It is the density of covered lines by unit tests:<br><br>Line coverage = LC / EL<br><br>where<br><br>LC = covered lines (lines_to_cover - uncovered_lines)<br>EL = total number of executable lines (lines_to_cover) |

## 10.2 Understand

The following metrics have been exported in the Excel file attached:




| Category | Metric | Threshold | Understand | |
|---|---|---|---|---|
| | | | https://scitools.com/support/metrics_list? | |
| | | | Metric | Definition |
| *Documentation* | **Number of comment lines** | | CountLineComment | Number of lines containing comment. [aka CLOC]<br><br>This can overlap with other code counting metrics. For instance int j = 1; // comment has a comment, is a source line, is an executable source line, and a declarative source line.<br><br>Example: https://scitools.com/documents/imagesMetrics/CountLineCommentC.png |
| | **Density of comment lines % (Min)** | **0.3** | RatioCommentToCode | Ratio of number of comment lines to number of code lines. Note that because some lines are both code and comment, this could easily yield percentages higher than 100<br><br>Exemple: https://scitools.com/documents/imagesMetrics/RatioCommentToCodeC.png |
| *Complexity* | **Complexity / function (Max)** | **10** | AvgCyclomatic | **Average Cyclomatic Complexity**<br>Description: Average cyclomatic complexity for all nested methods or methods<br>Example: https://scitools.com/documents/imagesMetrics/AvgCyclomaticC.png<br><br>**Max Cyclomatic Complexity**<br>Description: Maximum cyclomatic complexity of all nested methods or methods.<br>Example: https://scitools.com/documents/imagesMetrics/MaxCyclomaticC.png |
| | **Nested loops / function (Max)** | **5** | MaxNesting | Description: Maximum nesting level of control constructs (if, while, for, switch, etc.) in the method.<br>Detailed Example:https://scitools.com/documents/imagesMetrics/MaxNestingC.png |
| ***Design*** | **Lack Of Cohesion % (Max)** | **50** | PercentLackOfCohesion | Research: Chidamber & Kemerer – Lack of Cohesion in Methods (LCOM/LOCM)<br><br>Description: 100% minus average cohesion for class data members. Calculates what percentage of class methods use a given class instance variable. To calculate, average percentages for all of that class'es instance variables and subtract from 100%. A lower percentage means higher cohesion between class data and methods. |

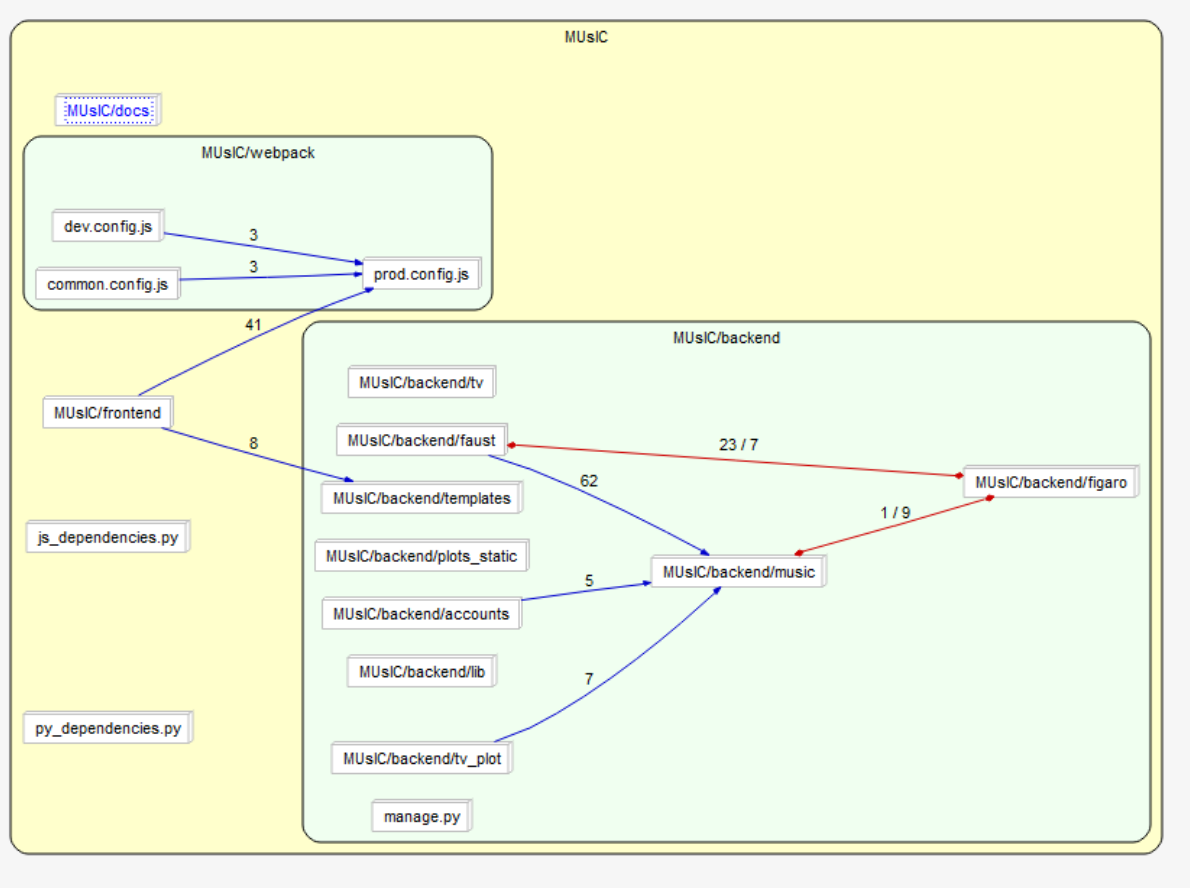
| | | | | |
|---|---|---|---|---|
| | **Coupling Between Objects (Max)** | **10** | CountClassCoupled | Description: The Coupling Between Object Classes (CBO) measure for a class is a count of the number of other classes to which it is coupled. Class A is coupled to class B if class A uses a type, data, or member from class B. This metric is also referred to as Efferent Coupling (Ce). Any number of couplings to a given class counts as 1 towards the metric total<br><br>Example: https://scitools.com/documents/imagesMetrics/CountClassCoupledC.png |
| *Size* | **# classes by file** | **1** | CountDeclClass | Number of classes |
| | **# methods/class (Max)** | **30** | CountDeclInstanceMethod | Number of instance methods – methods defined in a class that are only accessable through an object of that class<br>Ex: https://scitools.com/documents/imagesMetrics/CountDeclInstanceMethodC.png |
| | **# variables/class (Max)** | **30** | CountDeclInstanceVariable | Number of instance variables – variables defined in a class that are only accessable through an object of that class<br><br>Ex: https://scitools.com/documents/imagesMetrics/CountDeclInstanceVariableC.png |
| | **# lines of code/class (Max)** | **900** | AltAvgLineCode (class level) | Average number of lines containing source code for all nested functions or methods, including inactive regions.<br>Ex: https://scitools.com/documents/imagesMetrics/AltAvgLineCodeC.png |
| | **# lines of code/method (Max)** | **30** | AltAvgLineCode (method lev | Average number of lines containing source code for all nested functions or methods, including inactive regions.<br>Ex: https://scitools.com/documents/imagesMetrics/AltAvgLineCodeC.png |
| *Issues* | **# duplicated lines (Max)** | **0** | duplicated_lines_density | Density of duplication = Duplicated lines / Lines * 100<br>Duplicated lines = Number of lines involved in duplications. |

## 10.3 Annex 3: Dependency graphs by main folder

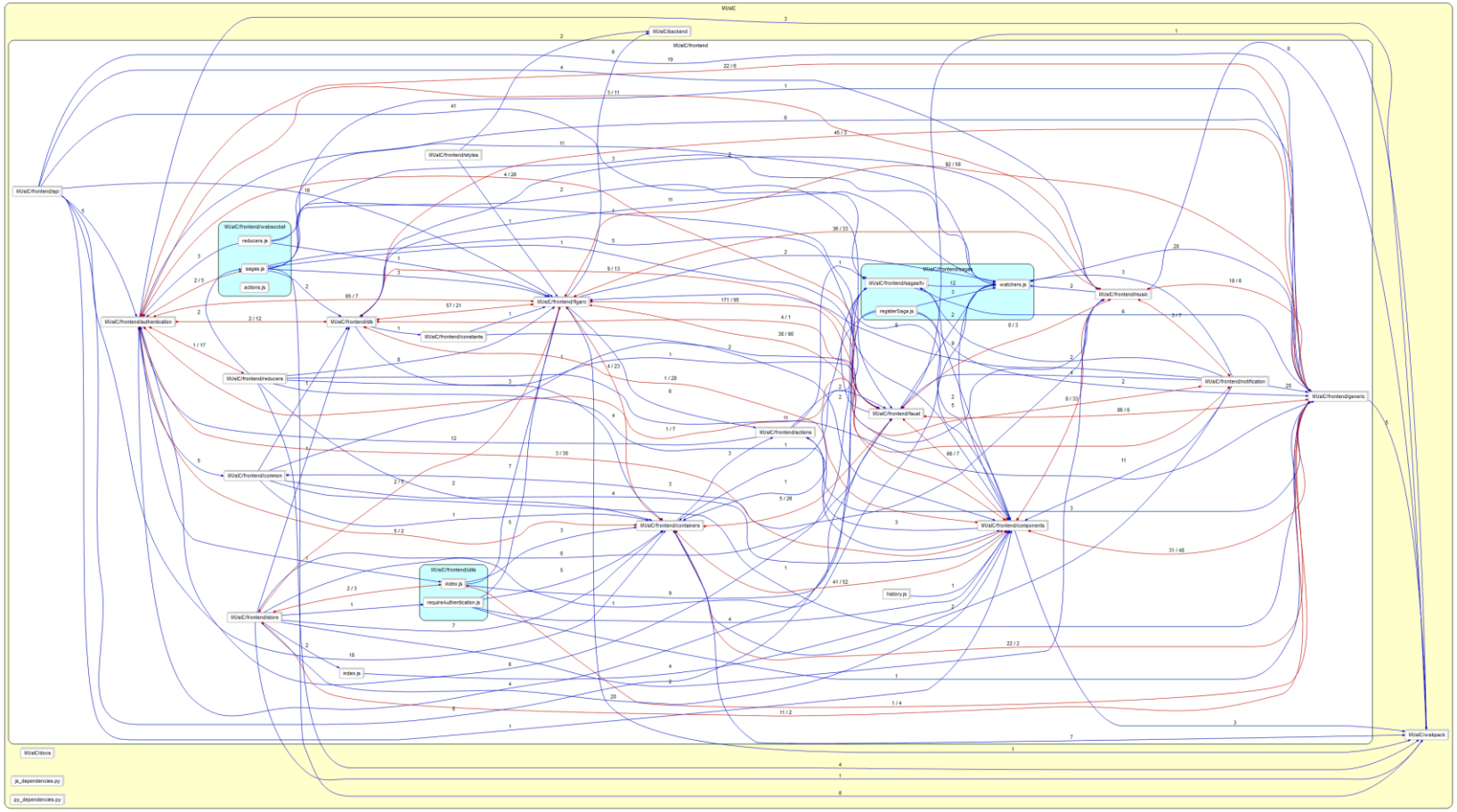
The figures below are provided only for information. Please contact the reviewer for more details (Such pictures could be added values in the Software Design Document).

## 10.4 Graphs with Python and Javascript languages

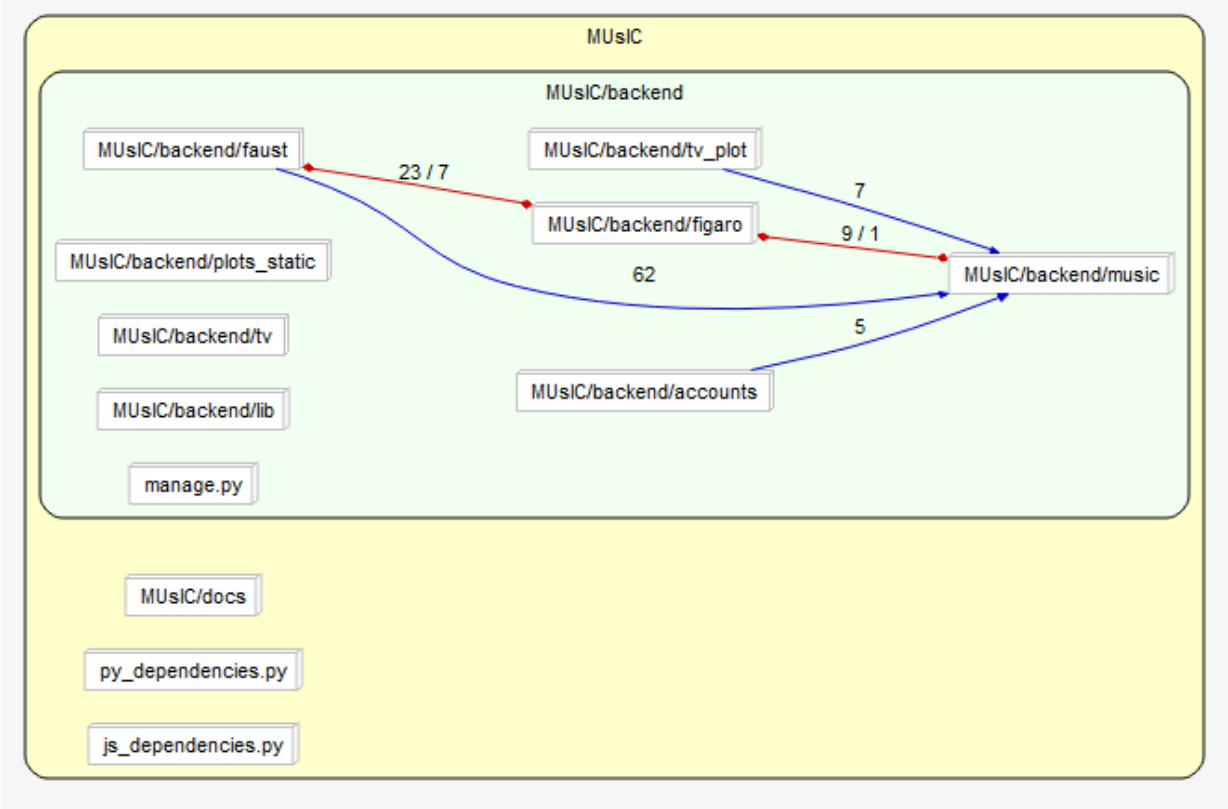
**Webpack and Backend directories:**



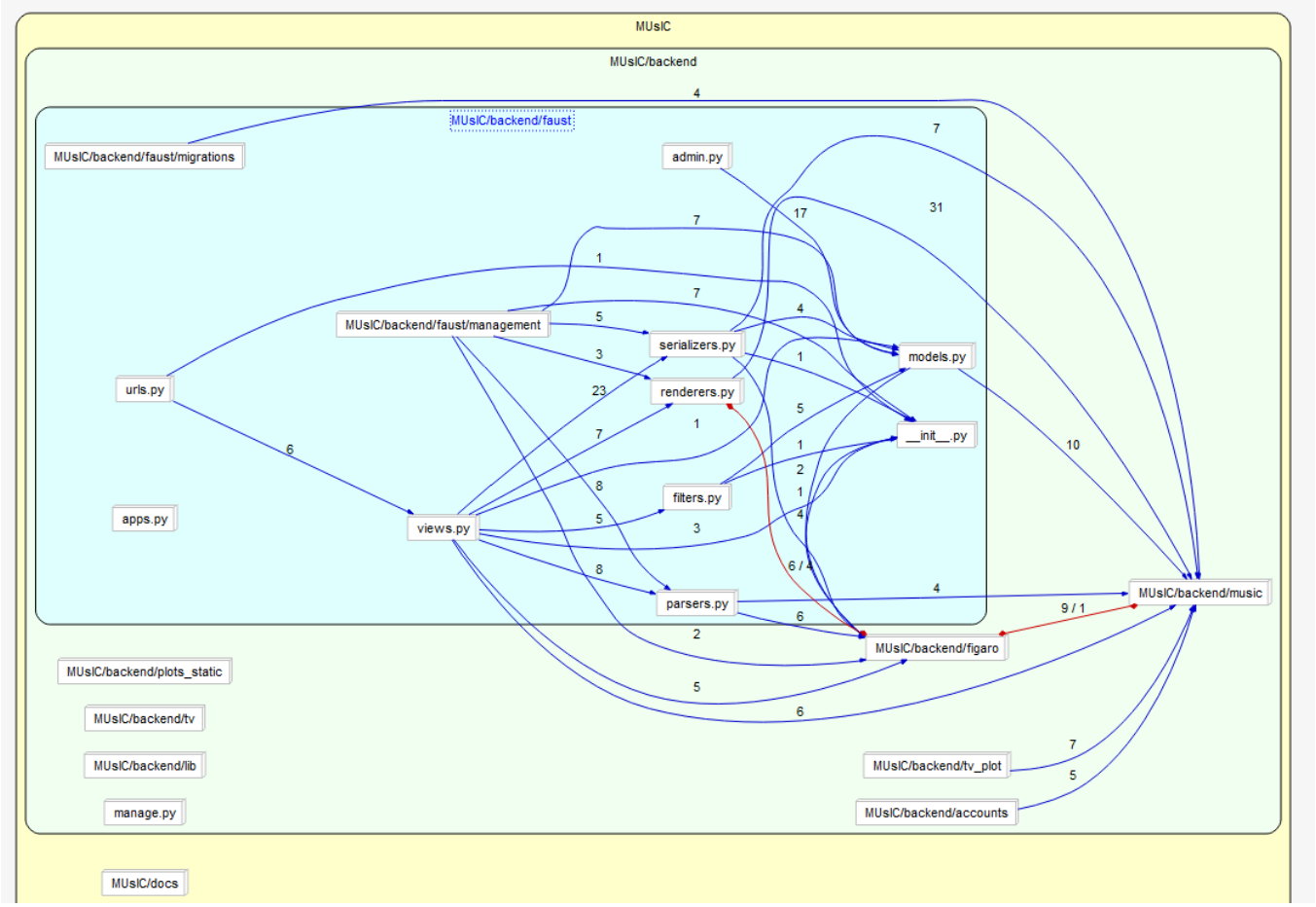
**FrontEnd directory :**



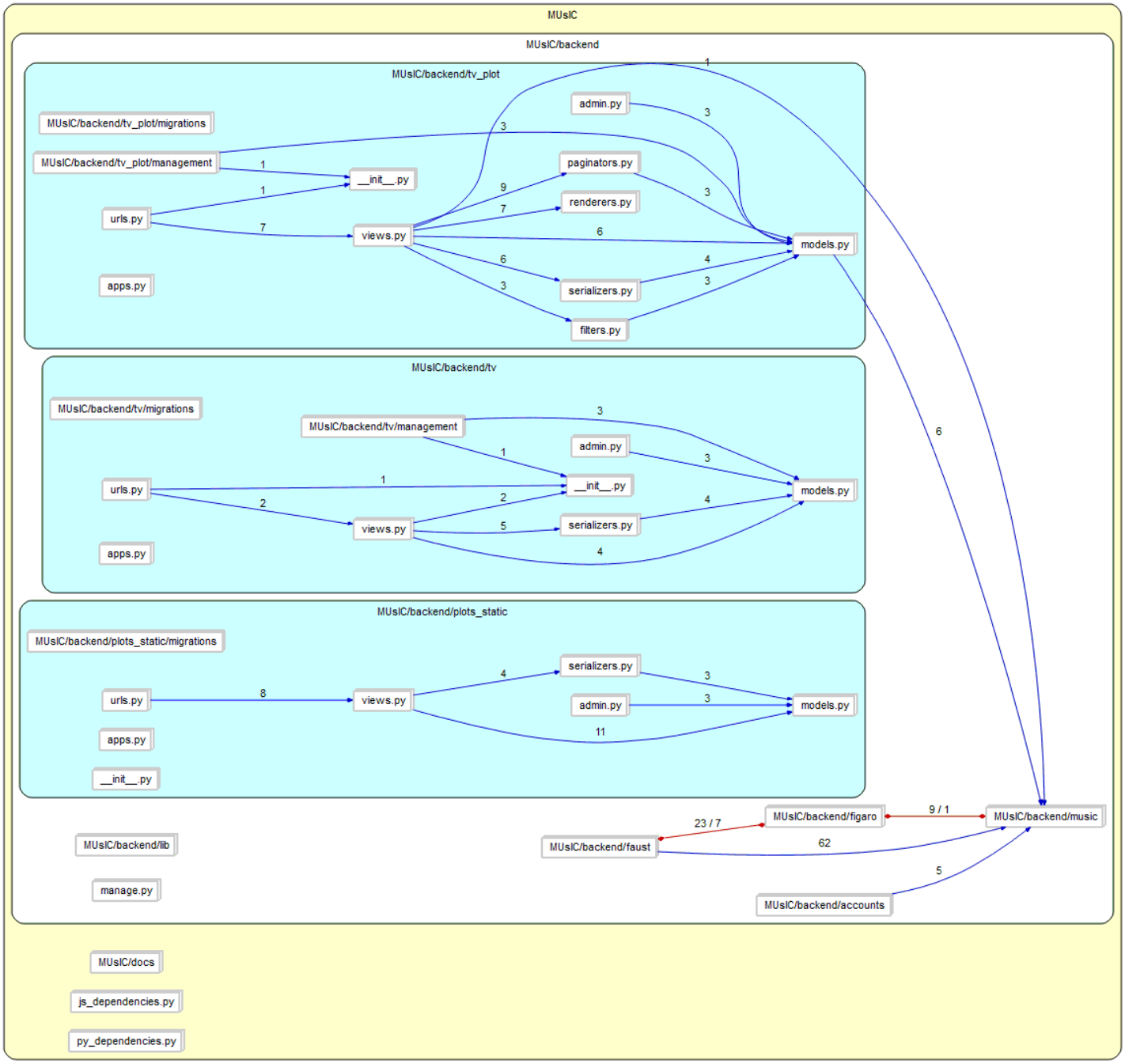
## 10.1 Graphs with Python language only



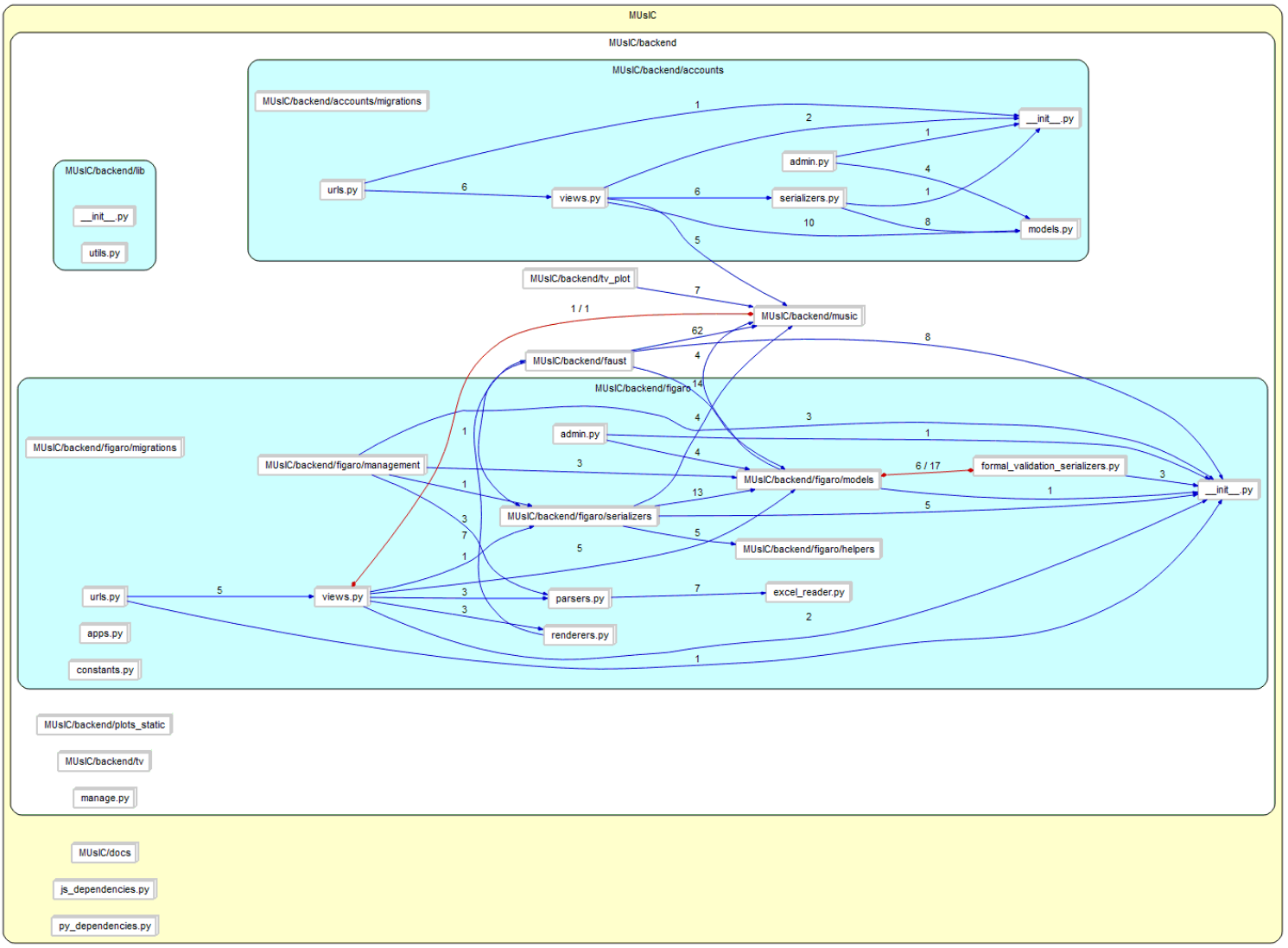
### 10.1.1 backend/faust



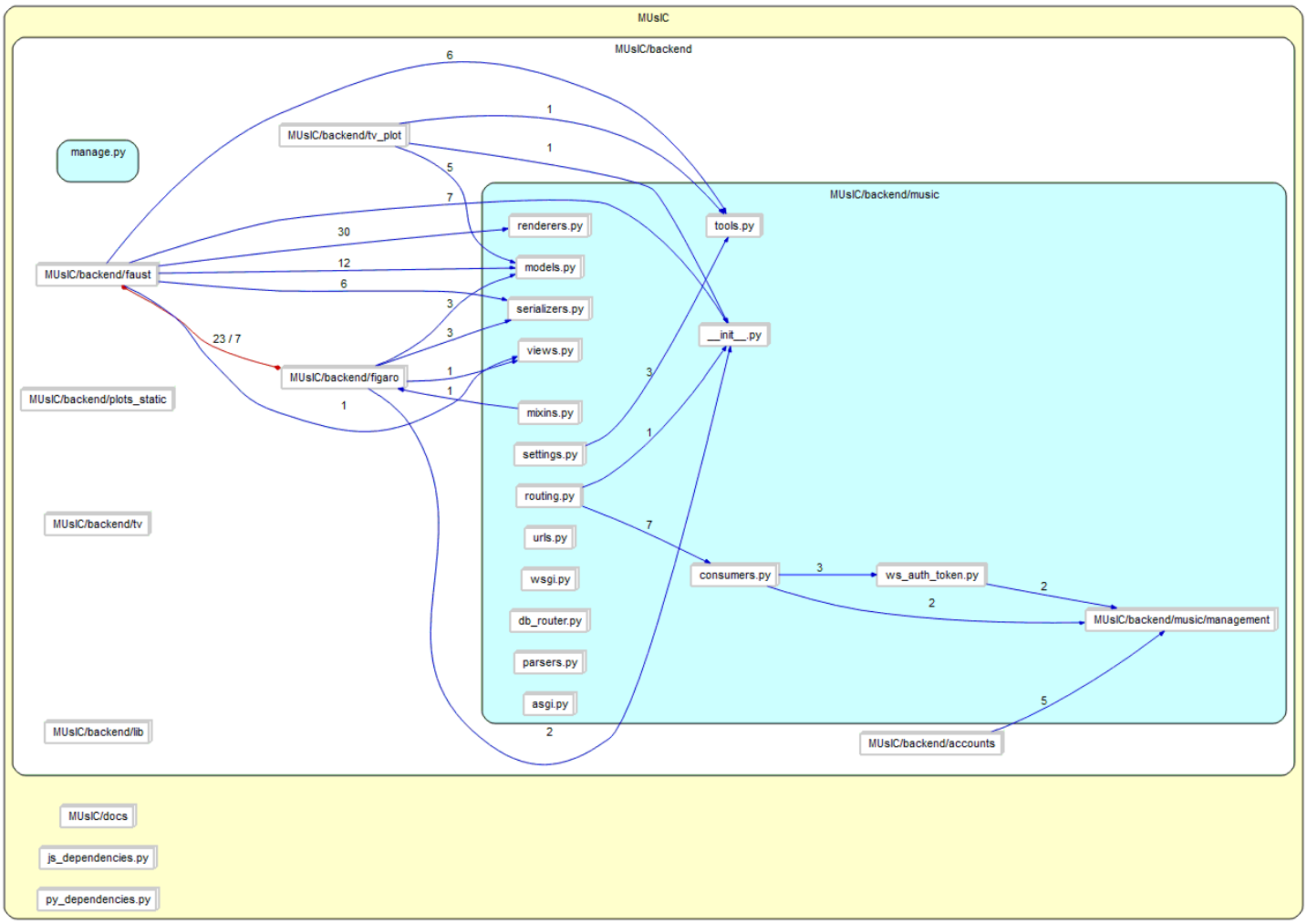
### 10.1.2 backend/tv_plot,tv,plots_static



### 10.1.3 backend/accounts, lib, figaro

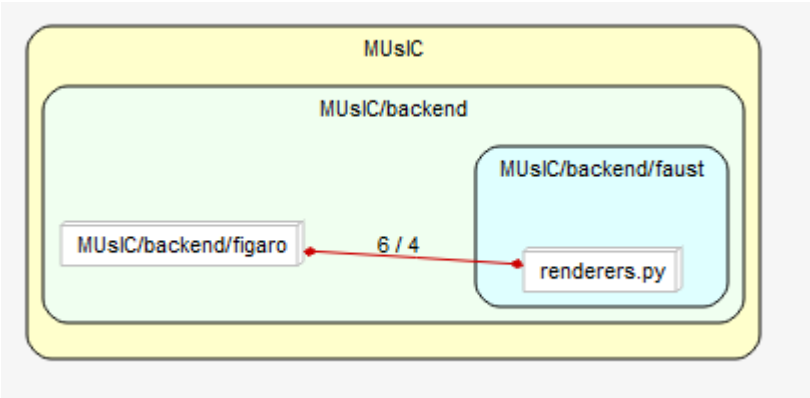


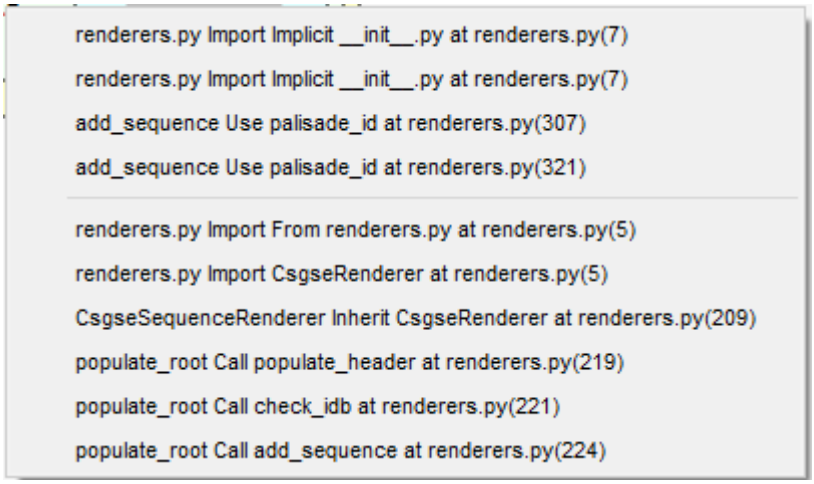
### 10.1.4 backend/music



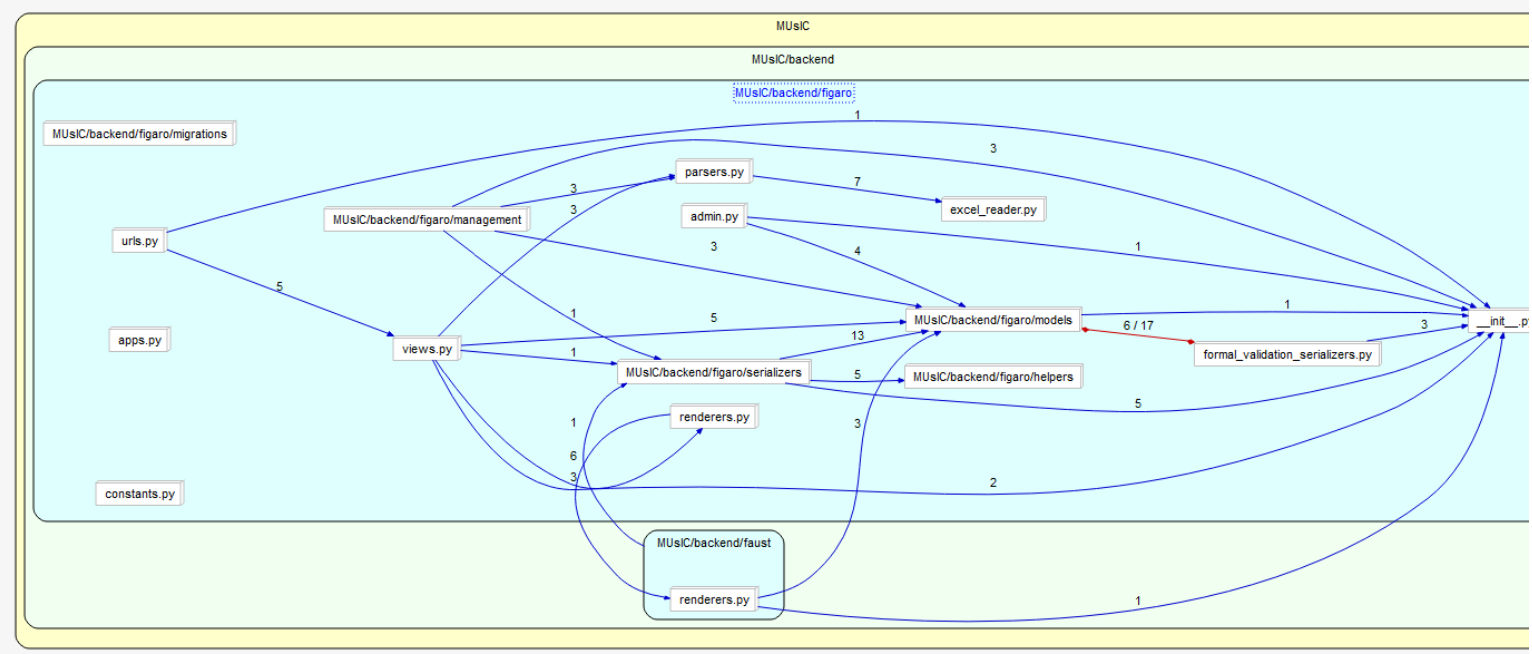
## 10.2 Focus on a dependancy (example)

Let's focus on one of these red lines, supposed to identify a mutual or cyclic dependancy:

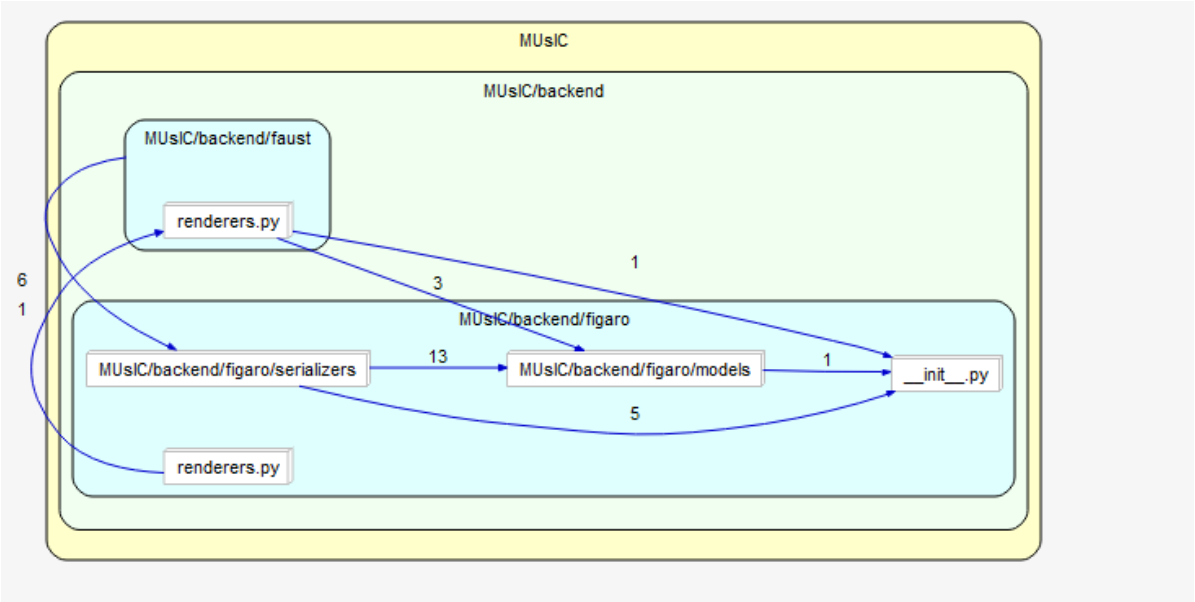


The detailed graphs show the input and ouput links:




This one shows only the cyclic links between *MusIC/backend/faust* and *MusIC/backend/figaro*:




*MusIC/backend/figaro/renders.py import code from MusIC/backend/faust:*

```
1  import datetime
2  import io
3
4  import xlwt
5  from faust.renderers import CsgseRenderer
6  from rest_framework.renderers import BaseRenderer
7
```

And *MusIC/backend/faust/renderers.py import code from MusIC/backend/figaro:*

```
1  import datetime
2  import uuid
3  from xml.etree.ElementTree import SubElement
4
5  from dateutil.parser import parse as parse_datetime
6  from django.utils.timezone import now
7  from figaro.models import Telecommand
8  from music.renderers import XmlRenderer
```

## 11. Annex 4 : pylint report

```
Report
======
Raw metrics
-----------

+----------+-------+------+---------+-----------+
|type      |number |%     |previous |difference |
+==========+=======+======+=========+===========+
|code      |4539   |55.87 |NC       |NC         |
+----------+-------+------+---------+-----------+
|docstring |1122   |13.81 |NC       |NC         |
+----------+-------+------+---------+-----------+
|comment   |750    |9.23  |NC       |NC         |
+----------+-------+------+---------+-----------+
|empty     |1713   |21.09 |NC       |NC         |
+----------+-------+------+---------+-----------+



Duplication
-----------

+-------------------------+------+---------+-----------+
|                         |now   |previous |difference |
```

| | | | |
|---|---|---|---|
| nb duplicated lines | 121 | 0 | +121.00 |
| percent duplicated lines | 1.536 | 0.000 | +1.54 |

Messages by category

| type | number | previous | difference |
|---|---|---|---|
| convention | 833 | 8 | +825.00 |
| refactor | 281 | 0 | +281.00 |
| warning | 167 | 0 | +167.00 |
| error | 291 | 0 | +291.00 |

Messages

| message id | occurrences |
|---|---|
| missing-docstring | 432 |
| import-error | 216 |
| line-too-long | 205 |
| too-few-public-methods | 187 |
| invalid-name | 141 |
| no-member | 65 |
| no-self-use | 62 |
| unused-argument | 57 |
| bad-continuation | 35 |
| attribute-defined-outside-init | 27 |
| protected-access | 16 |
| unused-import | 14 |
| unused-variable | 13 |

| | |
|---|---|
| too-many-arguments | 10 |
| too-many-locals | 9 |
| redefined-builtin | 9 |
| duplicate-code | 9 |
| ungrouped-imports | 7 |
| fixme | 7 |
| wildcard-import | 6 |
| undefined-variable | 5 |
| trailing-newlines | 4 |
| logging-not-lazy | 4 |
| broad-except | 4 |
| superfluous-parens | 3 |
| redefined-variable-type | 3 |
| no-name-in-module | 3 |
| lost-exception | 3 |
| anomalous-backslash-in-string | 3 |
| wrong-import-order | 2 |
| dangerous-default-value | 2 |
| bad-whitespace | 2 |
| bad-indentation | 2 |
| unidiomatic-typecheck | 1 |
| simplifiable-if-statement | 1 |
| not-callable | 1 |
| consider-iterating-dictionary | 1 |
| bad-super-call | 1 |

.

## 12. Annex 5: sonarQube dashboard

Source code analyzed from