# Project CAE-PA: HAZOP2RDF

Subtitle

Dmytro Kostiuk
Marcus Rothhaupt
Artan Kabashi
Vincenz Forkel

22.22.2022

# Contents

# List of Tables

# List of Figures

# Introduction

- Introduction to
- abstract of the project
- RDF problem description

# Problem Analysis

- Why Hazop 2 RDF?
- When can it help?
- Whom can it help?

# Concept

- How did we solve the problem(framework)
- How did we get to the concept?
- Why did we choose this method of implementing

## Design program components

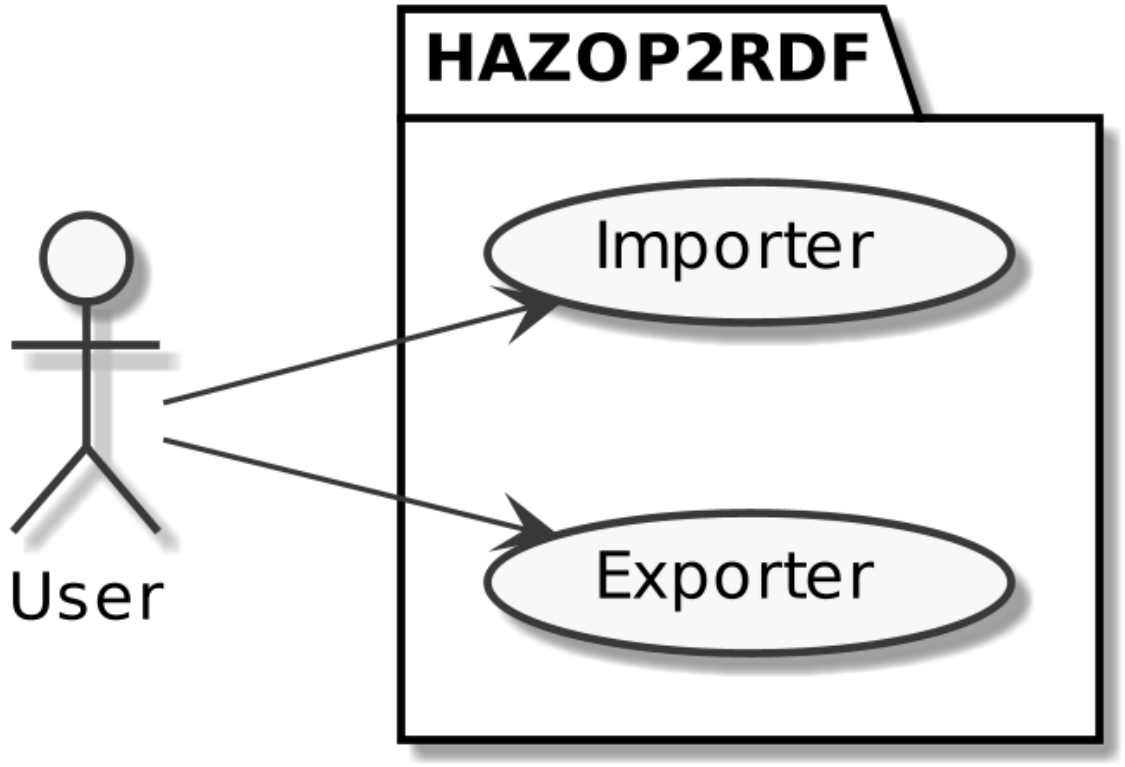
How did we start? What are the main components of the program?

### Diagram


## User interaction scenario

How should users interact with the program?

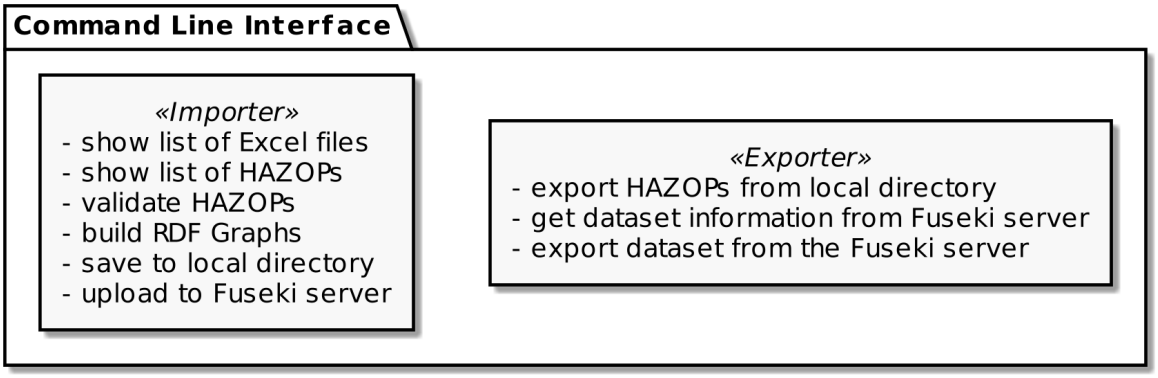
### Diagram




## Design Command Line Interface

What would the command line interface structure look like?

## Diagram


Command Line Interface
«Importer»
- show list of Excel files
- show list of HAZOPs
- validate HAZOPs
- build RDF Graphs
- save to local directory
- upload to Fuseki server
«Exporter»
- export HAZOPs from local directory
- get dataset information from Fuseki server
- export dataset from the Fuseki server

# Implementation

We implemented the Command Line Interface using “Click 8.0” (2021) package. It is highly configurable and can build very complex applications. The ComplexCLI utility, we used in our project, combines multiple interfaces in a single Command Line Interface.

The following diagram shows the structure of the Command Line Interface. It contains Importer and Exporter interface, which use services. The services contain utilities needed for the interfaces to perform lower level actions.

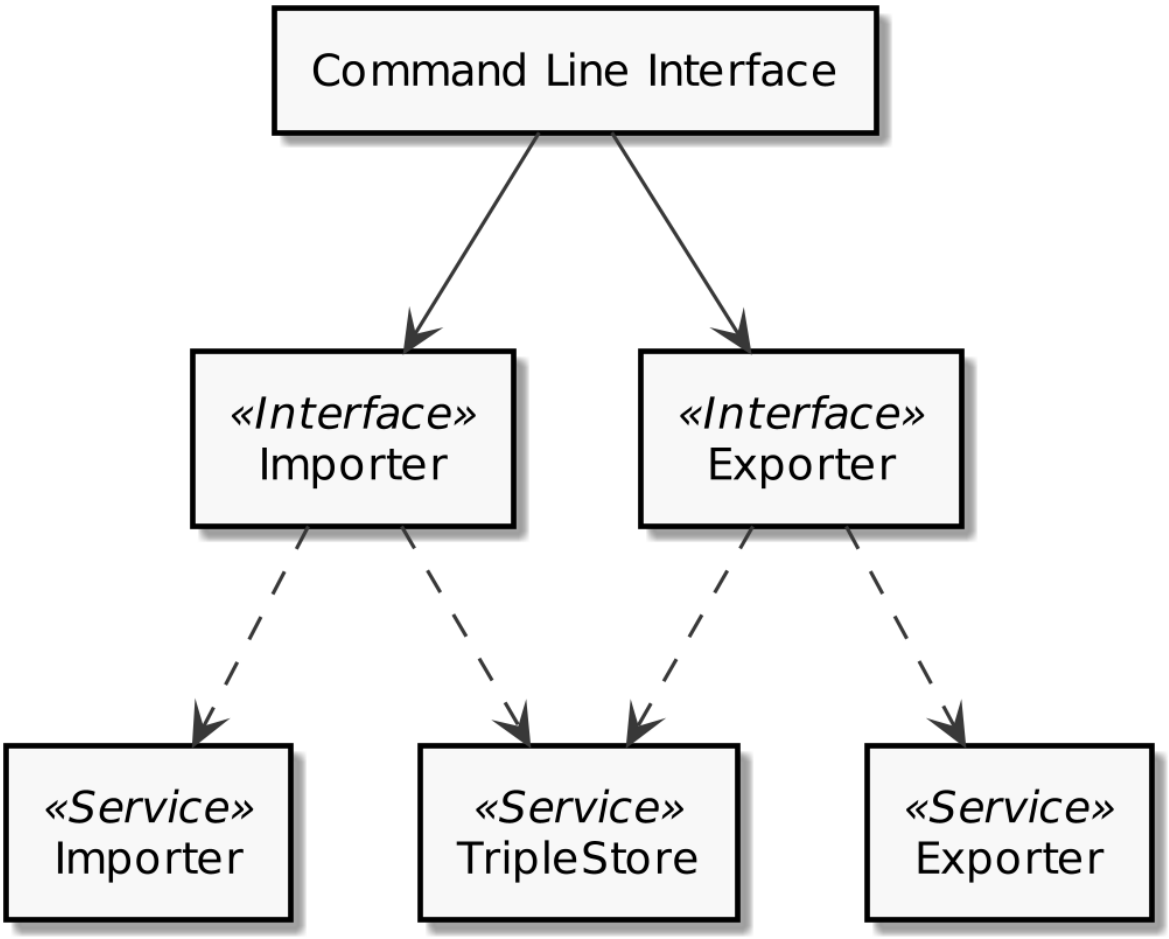



**Figure 0.1:** Structure diagram: Command Line Interface

Using the Command Line Interface users can interact with our software.

Using our Importer interface the user can import and validate incoming HAZOP data in Excel format and generate RDF graphs from it. They can be locally stored or uploaded to a Fuseki server.

Using our Exporter interface the user can export RDF graphs containing HAZOP data to Excel format. The source for the Exporter interface can either be a locally stored RDF file or an RDF file stored on a Fuseki server.

## Importer interface

The main purpose of the Importer interface is to build an RDF graph from incoming Excel data. To build an RDF graph, we carefully read the incoming HAZOP Data and validate it. To validate the data correctly we implemented a config file, which stores all the metadata needed to describe the importing and validating process.

The main command of the Importer interface is cmd-build-hazop-graphs, which reads the HAZOP data stored in a local directory and transforms it to an RDF graph. The graph can be consequentially stored locally or uploaded to a Fuseki server. The two other commands cmd-read-excel-data and cmd-read-hazop-data make it possible for the user to perform steps of the main importer command individually.

The installation of a Fuseki server is optional. If the server is offline, the files cannot be uploaded to the server resulting in an error message which is displayed to the user.

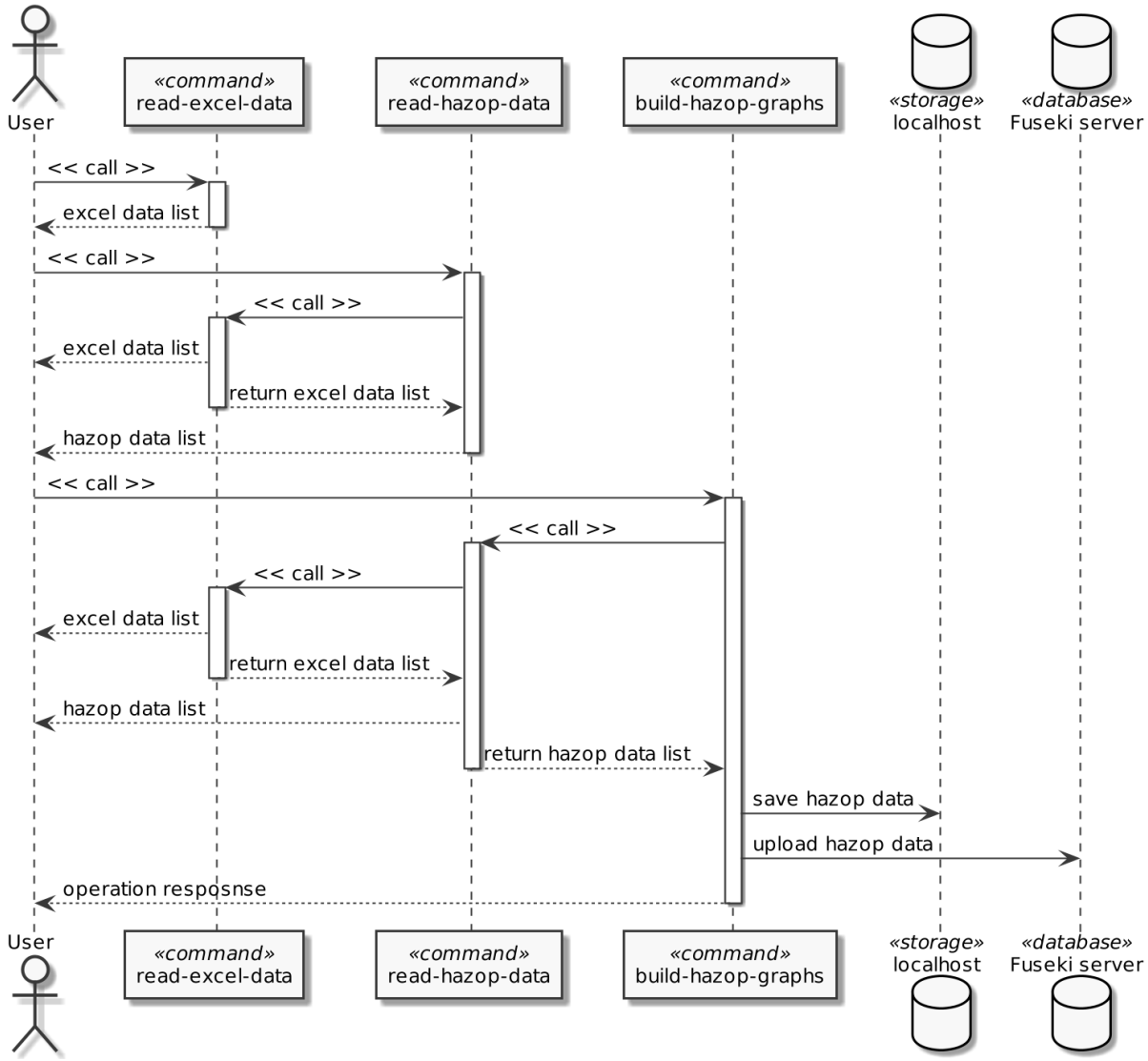



**Figure 0.2:** Sequence diagram: Importer interface

## Exporter interface

After the HAZOP data was successfully imported and stored, the user can convert the RDF graph to Excel format again.

There are two main commands in the Exporter interface for the users to interact with. The user can either export data from an RDF file located in a local directory or from a file located online on a Fuseki server. For the successful export from the Fuseki Server the server needs to

be running.

As a result, the RDF graphs will be stored locally in Excel format again.

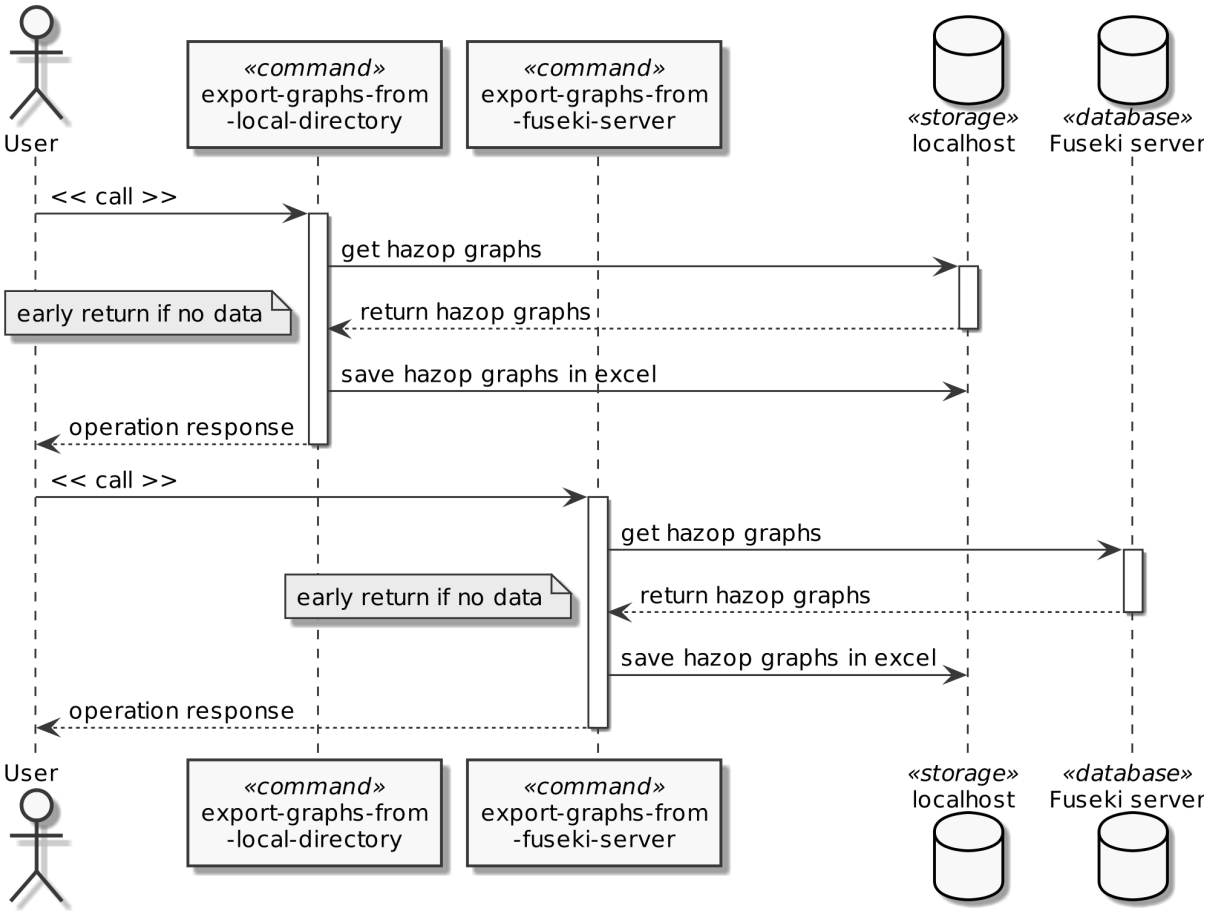



**Figure 0.3:** Sequence diagram: Exporter interface

## Remarks

We developed the HAZOP2RDF Project with version control on GitHub. The program is available for Windows and macOS. We also included a detailed installation guide in the documentation.

# Verification

The important part of the project is validation. We designed a test pattern to validate the results of the program execution. This pattern is simple and extendable and covers the main parts of the program.

In the tests' directory you can find a list of the following files:

- `test_cli.py` - test command line interface initialization
- `test_cmd_importer.py` - test importer interface
- `test_cmd_exporter.py` - test exporter interface

The prime objective of the pattern is consistency. It allows us to apply this pattern to every test we want to implement.

The pattern covers the following test cases:

- execution errors
- execution exceptions
- output validation

The code coverage value varies around 95 ± 2%. The value depends on the state of the Fuseki server. It increases if the Fuseki server is running and there is pre-uploaded data on the server.

The coverage report below shows the detailed information about the tests results.

```
1 ---------- coverage: platform darwin, python 3.8.2-final-0
     -----------
2 Name                                Stmts   Miss  Cover   Missing
3 -------------------------------------------------------------------
4 src/__init__.py                         0      0   100%
5 src/cli.py                             20      3    85%   34-35,
     43
6 src/commands/__init__.py                0      0   100%
```

```
src/commands/cmd_exporter.py          50      6    88%   37, 43, 48-50, 65
src/commands/cmd_importer.py          66      2    97%   38, 80
src/config/__init__.py                 0      0   100%
src/config/config.py                   3      0   100%
src/services/__init__.py               0      0   100%
src/services/svc_exporter.py          28      0   100%
src/services/svc_importer.py          97      0   100%
src/services/svc_triplestore.py       16      3    81%   37-40
-----------------------------------------------------------------
TOTAL                                280     14    95%
```

The user can although generate a coverage report in HTML format and easily discover the missing statements.

There are predefined commands in the Makefile for quick access.

# Summary

- Discussion
- Future Work
- Future Projects
- Problems to be solved

PROBLEM: We use a fixed schema to validate the input data, by pattern mismatch the data will be skipped.
TODO: Dynamic schema for accepting dynamic changes.
NEED: Dynamic changes constrains.

PROBLEM: We don’t validate the logic of the HAZOP data.
TODO: Validation schema.
NEED: Rules, what are acceptable in HAZOP and what are not.

IMPROVEMENT: TripleStore SOH API (SPARQL over HTML) to an API using Requests or SPARQL-Wrapper packages.
ADVANTAGES: better control over request/response operations, customization.

# Appendix

- Refernezimplementationen beachten und zitieren
- HAZOP source
- additional graphs, pictures…
- README.md

# References

“Click 8.0.” 2021. https://click.palletsprojects.com/en/8.0.x/.