Project title: Implications of Medical Low Dose Radiation Exposure
Grant Agreement: 755523
Call identifier: NFRP-2016-2017
Topic: NFRP-9

Documentation of the Semantic Translator software

Lead partner: Inserm
Author(s): Bernard Gibaud
Work Package: WP2
Estimated delivery: 1 November 2020
Actual delivery: 1 November 2020
Type: Report
Dissemination level: Public
Version : V1.0

1. Introduction

This document provides a basic documentation of the Semantic Translator software, a software developed by Inserm LTSI to manage the Semantic database of the Image and Radiation Dose BioBank (IRDBB) system. This semantic database is represented in an RDF graph stored in the Stardog Triple store [1].

For an introduction to the IRDBB semantic database, see the presentation available at: https://eibir.teamwork.com/#/files/4620923 [2].

History of versions

Version	Date	Description
V1.0	1/11/2020	First version corresponding to the distribution of the Semantic translator package SEMANTIC_TRANSLATOR_TAG = 0.0.70 containing the SEMANTICTRANSLATOR_VERSION="0.8.10" based on ONTOLOGY_VERSION="1.3.15"

2. Principle of the Semantic translator

2.1 Why *Semantic Translator* ?

The initial goal of the Semantic Translator was to populate the semantic database. The reason why it was called *Semantic Translator* is that this data is produced by translating metadata related to the files that are imported into the IRDBB system.
It turned out that additional capabilities related to the management of the semantic database were needed. So, the scope of this software was extended to cover them as well, but the name *Semantic Translator* was not changed.
In this document, we describe the capabilities of the Semantic Translator and we explain its architecture.

2.2 Integration of Semantic Translator in the IRDBB architecture

The general architecture of IRDBB is shown Figure 1.

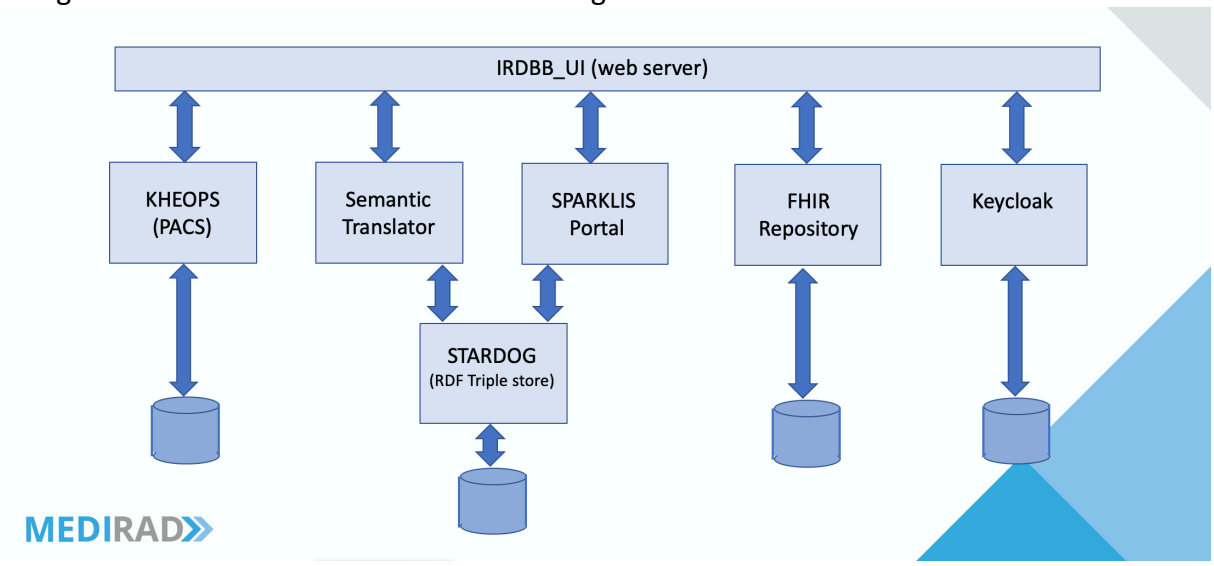

Figure 1: Architecture of the IRDBB system

The Semantic Translator communicates with the IRDBB user interface (IRDBB_UI), which calls its services, and with the Stardog Triple store supporting the semantic database.
In addition, one particular service is called by the KHEOPS component, in order to populate the semantic database with metadata describing the DICOM structured reports that are imported directly to KHEOPS rather then through the IRDBB_UI, as all other kinds of DICOM data.

2.3 Organization of the software

The Semantic Translator software was designed by Marine Brenet. It is organized in several components as shown on Figure 2.

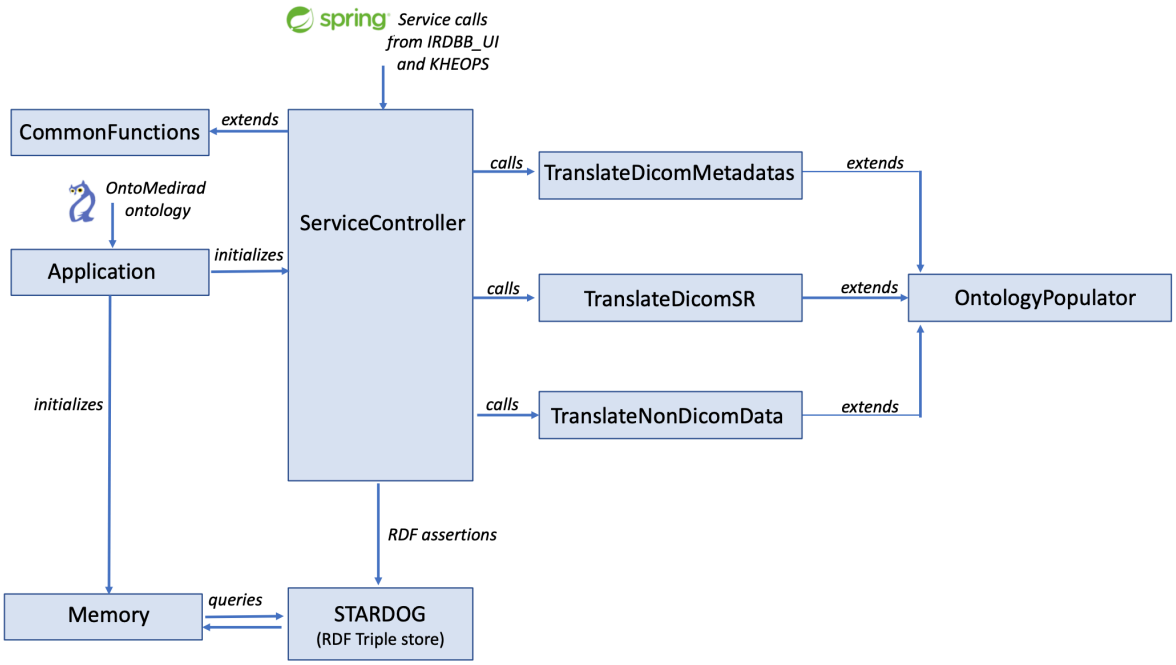

Figure 2: General organization of the Semantic Translator (simplified)

The Semantic Translator uses the Spring framework [3]. It receives requests for services and treats them.
It is activated by the application contained in the `application.Java` function.
The `main()` function:

- creates the resources to manage the predefined SPARQL queries (function `ListQuerries()`, which reads the excel file containing the SPARQL queries
- loads the ontology with the function `LoadOntology()`
- initalizes the memory with the `Memory()` function
- initializes the Spring environment (`SpringApplication.run()`), so that to receive the calls for services.

The reception of the call of services is managed by the `ServiceController()` function.

3. Capabilities of the Semantic Translator

The software is implemented as a set of services, called by IRDBB_UI or KHEOPS.

For each service, we mention the name of the function that receives the call of service in the `ServiceController()` function, and the name of the functions that implement the service itself.
The services are listed in their order of appearance in the `ServiceController()` function.

3.1 Services used in the operational system

Service Import DICOM metadata:

Name of function in the `ServiceController()`: `importDicomMetadata()`

This service is called when a new DICOM series of images or a Radiation Dose Structured Report (RDSR) is imported into IRDBB. This service retrieves from the Key Object Selection file stored in the FHIR repository the references of the DICOM SOP instances. The DICOM SOP instances are then retrieved from KHEOPS and their metadata are analyzed and translated into RDF. All this processing is performed in the `importDicomMetadata()` function of the `ServiceController()`.

The translation itself is performed in the functions `TranslateDicomMetadatas.java` and `TranslateSR.java`.

This translation consists in:

1) creating instances of the classes of the ontology,
2) associating to them different attributes represented by, e.g., character strings, integer or floats using data properties of the ontology
3) connecting these instances to other instances using object properties of the ontology.

The resulting RDF data associated to each DICOM series of images or DICOM SR SOP instance are then serialized and added to the semantic graph in the Stardog Triple store.

The Java functions involved are part of the `Repository` package, they are briefly described in Table 1:

Java function	Description
`TranslateDicomMetadatas.java`	This function extends the `OntologyPopulator`. Its main function is `translateDicomMetadata()` that analyzes the main metadata at the Study and Series level then analyzes the kind of DICOM SOP class and image type to select those than need to be translated into RDF.
`TranslateDicomSR.java`	This function extends the `OntologyPopulator`. Its main function is `readingSR()` which calls `translateSR()` that recursively calls `readingSR()`. This processing allows browsing and translating into RDF the content of the SR tree.

Table 1: Functions supporting importation of DICOM data

Note: The function `TranslateDicomCT.java` is deprecated. It is an early version in which DICOM metadata were translated by IRDBB_UI into an XML DICOM File set Descriptor.

Service Import KHEOPS SR:

Name of function in the `ServiceController()`: `importKheopsSR()`

This service is called when a new DICOM SR is imported via the KHEOPS repository. This service retrieves this SR from KHEOPS, translates it into RDF and adds this RDF data to the semantic graph in the Stardog Triple store.
This processing is achieved by the `TranslateDicomMetadatas.translateSRmaienz()` function.

Service Import Non DICOM File Set Descriptor:

Name of function in the `ServiceController()`: `importNonDicomData()`

This service is called when a Non-DICOM File set is imported. This File set includes an XML descriptor called *Non DICOM File Set Descriptor*. This file references all the files that need to be imported, and describes their provenance. It may also contain derived dosimetric data such as the values and units of absorbed doses in organs. This service translates the XML data into RDF and adds this RDF data to the semantic graph in the Stardog Triple store.

This processing is achieved by the `TranslateNonDicomData.translateNonDicomData()` function.
The Java functions involved are part of the `Repository` package, they are described in Table 2:

Java function	Description
`TranslateNonDicomData.java`	This function extends the `OntologyPopulator`. Its main function is `translateNonDicomData()` treats the different workflows, by calling a function dedicated to each workflow, namely: - `retrieveSubtask212()` - `retrieveSPECTCTCalibrationWorkflow()` - `retrieveCTCalibrationWorkflow()` - `retrievePlanarCalibrationWorkflow() (not implemented yet)` - `retrieveThreeDimDosimetrySlide1Workflow()` - `retrieveThreeDimDosimetrySlide2Workflow()`

Table 2: Main functions supporting translation into RDF of Non-DICOM File set Descriptors

Service Validate Non DICOM File Set Descriptor:

Name of function in the `ServiceController()`:
`validateNonDicomFileSetDescriptor()`

This service is called when a Non-DICOM File set is to be imported. It verifies that the XML Non DICOM File Set Descriptor associated to this File Set is valid against the XSD schema XML of the application.

Service Get Version:

Name of function in the `ServiceController()`: `returnVersionNumber()`

Ce service allows extracting the version number of the Semantic Translator from the `pom.xml` file.

Service Download Data From Stardog:

Name of function in the `ServiceController()`: `downloadStarDogDatabase()`

This service allows retrieving the complete RDF graph from the OntoMEDIRAD database stored in the Stardog Triple Store, and copying it into an RDF file.

Service Download Requests:

Name of function in the `ServiceController()`: `downloadRequests()`

It allows retrieving as a CSV file the list of the predefined SPARQL queries available in the application.

Service Request From List:

Name of function in the `ServiceController()`: `requestFromList ()`

This service is called by IRDBB_UI. It triggers the submission to Stardog of the predefined SPARQL query selected interactively by the user, and returns the answer received from Stardog to IRDBB_UI.

Service Get MIME type data format:

Name of function in the `ServiceController()`: `getMimeTypeDataFormat()`

This service is called by IRDBB_UI. It submits to the Stardog system a SPARQL query to retrieve the MIME types associated to the different Non-DICOM file formats. This information is required for a good description of the files in the database of the FHIR repository.

Service Get Research studies:

Name of function in the `ServiceController()`: `getResearchStudies()`

This service is called by IRDBB_UI at initialization. It submits to the Stardog system a SPARQL query to retrieve the MEDIRAD Clinical research studies, so that the user can select to which one the imported data should be related.

Service Get XSD Files Name:

Name of function in the `ServiceController()`: `getXSDfilesName()`

This service is called by IRDBB_UI. It returns the names of the XSD (XML Schema Definition) files against which the XML File set descriptors must be valid.

Service Get XSD:

Name of function in the `ServiceController()`: `getXSD()`

This service is called by IRDBB_UI. It returns the XSD file requested by the user.

Service Get Request List:

Name of function in the `ServiceController()`: `getRequestList()`

This service is called by IRDBB_UI. It allows providing IRDBB_UI with the list of the predefined SPARQL queries available in the application. The list of such predefined SPARQL queries is store in the `RequestList.csv` file located in the `../metadata-repository/src/main/resources` directory.

3.2 Services used for testing and development only

Service Test XML:

Name of function in the `ServiceController()`: `testXML()`

This service translates into RDF assertions the XML data sent in input to the service. It is used to test the translation of XML File set descriptors of new workflows. It uses the `TranslateNonDicomData.translateNonDicomData()` function.

Service Test DICOM uids:

Name of function in the `ServiceController()`: `testDICOMuids()`

This service was developed to test the existence of a DICOM entity in the semantic graph.
This service is NOT functional, yet.

Service Test SR:

Name of function in the `ServiceController()`: `testSR()`

This service translates into RDF assertions the content of DICOM Radiation Dose Structured Reports (RDSR), whose file names are stored in the source code.
It uses the `TranslateDicomSR.readingSR()` function, that recursively analyzes the content of the SR tree.

Service Test SR KHEOPS:

Name of function in the `ServiceController()`: `testSRkheops()`

This service allows testing the translation into RDF assertions of the basic metadata of a DICOM Structured Report (SR) originally created by the MRRT SR creating system developed in Mainz, and whose file name is stored in the source code.

It uses the `TranslateDicomMetadatas.translateSRmaienz()` function, that translates into RDF the basic metadata of the SR.

Service Test Metadatas:

Name of function in the `ServiceController():testMetadatas()`

This service translates into RDF assertions the content of DICOM files, whose file names are stored in the source code in the `listeRDF` variable.
It uses the `TranslateDicomMetadatas.translateDicomMetaData()` function, that translates into RDF the basic metadata of the DICOM files.

4. Instance identifiers and management of cache memory

4.1 What is the problem ?

Most of the entities that populate the graph are particulars (i.e. instances). They are created when they are met for the first time when new data are imported. In most cases, the instances are assigned an IRI which includes a uuid, which guarantees unicity.
However, there are many cases in which it is important to correctly identify each instance in order not to create duplicates, e.g. for humans that participate in MEDIRAD clinical research studies. Therefore, a mechanism is needed in order to store in the memory of the Semantic translator software the instances already created, and to check whether they already exist.
The list of the classes of entity to which this principle currently applies is given in Table 3.

Entity	Class in OntoMEDIRAD ontology
Software	ontomedirad:software
MC Method	ontomedirad:Monte_Carlo_CT_dosimetry_method
Institution	ontomedirad:institution
DICOM Dataset	instance bearing the ontomedirad:has_IRDBB_WADO_handle object property
Human	ontomedirad:human
Template of SR	ontomedirad:template_of_structured_report
Study Instance UID	ontomedirad:imaging_study
Internal Radiotherapy	ontomedirad:internal_radiotherapy
TimePoint	subClassOf ontomedirad:timepoint_of_internal_radiotherapy
SPECT calibration	ontomedirad:SPECT_CT_calibration
CT calibration	ontomedirad:CT_calibration

Table 3: Entities concerned by the memory mechanism

Remark: Ideally, more entities should be concerned by this memory mechanism. However, the way metadata is collected (either through DICOM metadata, or through XML data in File set descriptors) does not provide enough information to reliably identify the entities in the real world, so a conservative approach was used, consisting in ignoring that these instances might be duplicates.

4.2 Implementation of the memory mechanism

The initialization of the memory is implemented by the `Memory()` method of `Memory.java` function of the `repository` package.

It is composed of a set of functions listed in Table 4. Each of them executes a SPARQL query so that to retrieve from the semantic graph the instances that need to be present in the cache memory.

Entity	Function retrieving the instances
Software	`requestSoftware()`
MC Method	`requestMCMethod()`
Institution	`requestInstit()`
DICOM Dataset	`requestDicomDatasets()`
Human	`requestHuman()`
Template of SR	`requestTemplateOfSR()`
Study Instance UID	`requestStudyInstanceUID()`
Internal Radiotherapy	`requestInternalRadiotherapy()`
TimePoint	`requestTimePoint()`
SPECT calibration	`requestSPECTCalibration()`
CT calibration	`requestCTCalibration()`

Table 4: Functions retrieving the instances from the semantic graph

These functions store the IRI of the instances as well as identifying attributes in dedicated lists (`LinkedList<String>` or `Hashtable<String, Individual>`), from which they can be retrieved thanks to dedicated functions. These lists are filled at initialization (i.e. when the Semantic Translator is started) and when new instances are created.

Marine Brenet created functions to retrieve entities from the memory, but some of them of not used in the software, yet.

5. Management of XML File set descriptors

The Semantic Translator software is dependent on the structure of the XML schema against which the XML File set descriptors provided in the users' non-DICOM file sets must be valid.

The `translateNonDicomData()` function reads the XML elements thanks to a set of automatically generated Java functions stored in the `../metadata-repository/src/main/java/javaXSDclass` directory. They are produced by the reference implementation JavaTM Architecture for XML Binding (JAXB).

This XML schema exists in two forms in the implementation:

- a *comprehensive* one, `nonDicomFileSetDescriptor.xsd`, which is the one actually used by the software to check the validity of the XML File set descriptors; it is stored is the `../metadata-repository/src/main/resources/xsd` directory

- *partial* ones, (simple xsd files), which focus on each individual workflow. They allow users to ensure that their XML File set descriptors pertaining to each particular workflow are valid without having to consider the whole domain of the MEDIRAD application; they are stored in the following directory:
- `../metadata-repository/src/main/resources/xsdSimpledirectory`.
- The current list of simple XSD files is as follows:
    - `2D-DosimetryWorkflow.xsd`
    - `3D-DosimetrySlide2Workflow.xsd`
    - `WP2subtask212WorkflowData.xsd`
    - `3D-DosimetrySlide1Workflow.xsd`
    - `Hybrid-DosimetryWorkflow.xsd`
    - `calibrationWorkflow.xsd`

In order to facilitate the creation and management of XML schemas, and guarantee their internal and external consistency (by external consistency we mean consistency between the elements shared by the different workflows), the creation of actual XSD files was automated. The environment for producing XSD files is located in the `workflowDescriptor2XSD` directory.

- The user edit text versions of the workflows, which are located in the `workflowDescriptor2XSD/txt` directory
- the current list of the text versions of the workflows is as follows:
    - `2D-DosimetryWorkflow.txt`
    - `3D-DosimetrySlide2Workflow.txt`
    - `WP2subtask212WorkflowData.txt`
    - `3D-DosimetrySlide1Workflow.txt`
    - `Hybrid-DosimetryWorkflow.txt`
    - `calibrationWorkflow.txt`
- the generation of the xsd is performed by the execution of the `conversionNonDicom.command` program, located in the `workflowDescriptor2XSD` directory; this script uses several programs written in Python that analyse the txt files, control their consistency and perform the translation into XSD)
- this execution creates both the comprehensive and the partial XSD files that are stored in the `workflowDescriptor2XSD/xsd` and `workflowDescriptor2XSD/xsdSimple` directories.
- To be taken into account in the Semantic Translator, these directories need to be copied to the `../metadata-repository/src/main/resources/xsd` and `../metadata-repository/src/main/resources/xsdSimple` directories
- it also creates graphical representation of the XSD files in Scalable Vector Graphics format (.svg format), located in the `workflowDescriptor2XSD/schemas` directory; an example is provided Figure 3.

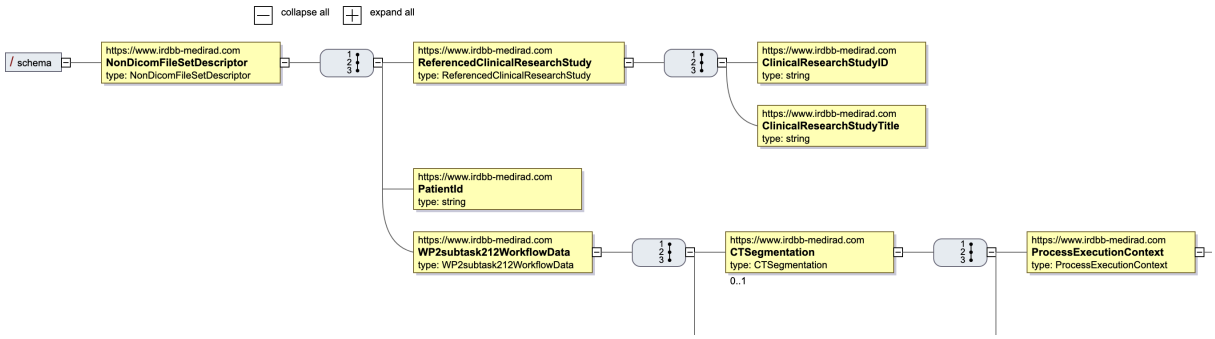

Figure 1: Extract of the display of the WP2subtask212WorkflowData.svg file (with Firefox)

6. Main limitations of the software

6.1 General issues

The Semantic Translator software was developed in an iterative way, without full understanding of the needs and constraints at the beginning, and with limited development resources. As a result, the implementation choices were made progressively, based on successive assessments of the situation. Partial refactoring of the code was made at several periods of the development, but always with limited ambition, due to limited development resources.

The most important refactoring concerned abandoning the use of DICOM File set descriptors as containers of the DICOM metadata to be translated into RDF assertions. The reasons for this important change were:

- the frequent need to extend it
- the too complex management of this intermediary data structure
- the need to involve b<>com in its implementation and management.

6.2 Limitations of `TranslateNonDicomData.java`

There are two workflows that are still missing:

- the 2D dosimetry workflow (`nonDicomFileSetDescriptor.getTwoDimDosimetryworkflow()`)
- the Hybrid dosimetry workflow (`nonDicomFileSetDescriptor.getHybridDosimetryworkflow()`)

In the `retrieveThreeDimDosimetrySlide2Workflow()` function, the `getKernelLimitForConvolutionsUsed` is not delt with, because the class is still missing in the ontology.

Moreover, the part of code dedicated to calibration workflows was only partially tested, using ad-hoc fake data. The testing with real calibration data will be done once this data is provided by Alex Vergara Gil (Inserm CRCT, Toulouse).

Limitations of `Memory.java`

No particular limitation was noted.

Limitations of `OntologyPopulator.java`

No particular limitation was noted.

Limitations of `ServiceController.java`

No particular limitation was noted.

Limitations of `TranslateDicomSR.java`

All the uses of the object property 'has_measure' should be changed, because this object property was removed from the ontology.

Limitations of `TranslateDicomMetadatas.java`

The logics of the processing of the numerous kinds of DICOM objects (especially the tests on the SOP Class UID and the ImageType DICOM tag) is extremely complex and hard to follow (due to the extensive use of if {} else if {}). The whole structure of this quite long program (more than 2000 lines of code) should be deeply refactored to increase readability and enable extension to other DICOM objects (especially Projection radiography images and MR images).

Limitations of `TranslateDicomCT.java`

This function is deprecated.

Limitations of `CommonFunctions.java`

No particular limitation was noted.

Limitations of `Application.java`

No particular limitation was noted.

Limitations of `FilterLog.java`

No particular limitation was noted.

Limitations of `WebMvcConfigurerAdapterExtension.java`

No particular limitation was noted.

Limitations of `Import.java`

No particular limitation was noted.

Limitations of `SwaggerConfig.java`

No particular limitation was noted.

Limitations of `ValidationReport.java`

No particular limitation was noted.

Limitations of `ListQuerries.java`

No particular limitation was noted.

Limitations of `Querry.java`

No particular limitation was noted.

7. References

[1] Stardog : https://www.stardog.com/
[2] Gibaud B, Spaltenstein J, IRDBB database to incorporate dosimetry data, MEDIRAD Plenary Meeting, September 21th, 2020, https://eibir.teamwork.com/#/files/4620923
[3] Spring framework: https://spring.io/