Knora Documentation

Release 0.1

Digital Humanities Lab, University of Basel

May 31, 2017

Contents

CHAPTER 1

What Is Knora?

Knora (Knowledge Organization, Representation, and Annotation) is a software framework for storing, sharing, and working with humanities data.

Knora is based on the idea that the continuous availability and reusability of digital qualitative research data in the humanities requires a common, flexible data representation and storage technology capable of performing queries across large quantities of heterogeneous data, organised according to project-specific data structures that cannot be known in advance. It also requires a convenient, storage-independent way for Virtual Research Environments (VREs) and automated data-processing software to access, query, and add to this data.

To solve the data representation and storage problem, Knora represents humanities data as RDF graphs, using OWL ontologies that express abstract, cross-disciplinary commonalities in the structure and semantics of research data. Each project using Knora extends these abstractions by providing its own project-specific ontology, which more specifically describes the structure and semantics of its data. Existing non-RDF repositories can readily be converted to an RDF format based on the proposed abstractions. This design makes it possible to preserve the semantics of data imported from relational databases, XML-based markup systems, and other types of storage, as well as to query, annotate, and link together heterogeneous data in a unified way. By offering a shared, standards-based, extensible infrastructure for diverse humanities projects, Knora also deals with the issue of conversion and migration caused by the obsolescence of file and data formats in an efficient and feasible manner.

To solve the access problem, Knora offers a generic HTTP-based API. In the Knora framework, the standard implementation of this API is a server program called the Knora API Server. The Knora API allows applications to query and work with data in terms of the concepts expressed by the Knora ontologies, without dealing with the complexities of the underlying storage system and its query language (e.g. SPARQL). It also provides features that are not part of SPARQL, such as access control and automatic versioning of data. While the Knora API is best suited to interacting with RDF repositories based on the Knora ontologies, it can also be implemented as a gateway to other sorts of repositories, including non-RDF repositories.

Knora uses a high-performance media server, called Sipi, for serving and converting binary media files such as images and video. Sipi can efficiently convert between many different formats on demand, preserving embedded metadata, and implements the International Image Interoperability Framework (IIIF).

Knora provides a general-purpose, browser-based VRE called SALSAH, which relies on the components described above. Using the Knora API, a project can also create its own VRE or project-specific web site, optionally reusing components from SALSAH.

Knora is thus a set of standard components that can be used separately or together, or extended to meet a project's specific needs. You can learn more about each component:

- *The Knora Ontologies*, a set of OWL ontologies describing a common structure for describing humanities data in RDF.
- *The Knora API Server*, a server program written in Scala that implements an HTTP-based API for accessing and working with data stored in an RDF triplestore according to the structures defined in the Knora ontologies.
- Sipi (to be released soon), a high-performance media server written in C++.

- The SALSAH GUI (to be release soon), a web-based virtual research environment for working with data managed by the Knora API server.

CHAPTER 2

An Example Project: Incunabula

This section introduces some of the basic concepts involved in creating ontologies for Knora projects, by means of a relatively simple example project. Before reading this document, it will be helpful to have some familiarity with the basic concepts explained in *The Knora Base Ontology*.

Knora comes with two example projects, called `incunabula` and `images-demo`. Here we will consider the `incunabula` example, which is a reduced version of a real research project on early printed books. It is designed to store an image of each page of each book, as well as RDF data about books, pages, their contents, and relationships between them. At the moment, only the RDF data is provided in the example project, not the images.

The `incunabula` ontology is in the file `incunabula-onto.ttl`, and its data is in the file `incunabula-demo-data.ttl`. Both these files are in a standard RDF file format called Turtle. The Knora distribution includes sample scripts (in the `webapi/scripts` directory) for importing these files directly into different triplestores. If you are starting a new project from scratch, you can adapt these scripts to import your ontology (and any existing RDF data) into your triplestore for use with Knora.

The syntax of Turtle is fairly simple: it is basically a sequence of triples. We will consider some details of Turtle syntax as we go along.

The Incunabula Ontology

Here we will just focus on some of the main aspects of the ontology. An ontology file typically begins by defining prefixes for the IRIs of other ontologies that will be referred to. First there are some prefixes for ontologies that are very commonly used in RDF:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
```

The `rdf`, `rdfs`, and `owl` ontologies contain basic properties that are used to define ontology entities. The `xsd` ontology contains definitions of literal data types such as `string` and `integer`. (For more information about these ontologies, see the references in *The Knora Base Ontology*.) The `foaf` ontology contains classes and properties for representing people.

Then we define prefixes for Knora ontologies:

```
@prefix knora-base: <http://www.knora.org/ontology/knora-base#> .
@prefix dc: <http://www.knora.org/ontology/dc#> .
@prefix salsah-gui: <http://www.knora.org/ontology/salsah-gui#> .
```

The `knora-base` ontology contains Knora's core abstractions, and is described in *The Knora Base Ontology*. The `dc` ontology is Knora's version of Dublin Core. It is intended to make it possible to define properties in a Knora project in terms of Dublin Core abstractions, to facilitate queries that search for data across multiple projects. The `salsah-gui` ontology includes properties that Knora projects must use to enable SALSAH, Knora's generic virtual research environment.

For convenience, we can use the empty prefix to refer to the `incunabula` ontology itself:

```
@prefix : <http://www.knora.org/ontology/incunabula#> .
```

However, outside the ontology file, it would make more sense to define an `incunabula` prefix to refer to the `incunabula` ontology.

Properties

All the content produced by a Knora project must be stored in Knora resources (see *Resource Classes*). Resources have properties that point to different parts of their contents; for example, the `incunabula` project contains books, which have properties like `title`. Every property that poitns to a Knora value must be a subproperty of `knora-base:hasValue`, and every property that points to another Knora resource must be a subproperty of `knora-base:hasLinkTo`.

Here is the definition of the `incunabula:title` property:

```
:title rdf:type owl:ObjectProperty ;

       rdfs:subPropertyOf dc:title ;

       rdfs:label "Titel"@de ,
           "Titre"@fr ,
           "Titolo"@it ,
           "Title"@en ;

       knora-base:subjectClassConstraint :book ;

       knora-base:objectClassConstraint knora-base:TextValue ;

       salsah-gui:guiOrder "1"^^xsd:integer ;

       salsah-gui:guiElement salsah-gui:SimpleText ;

       salsah-gui:guiAttribute "size=80" ,
           "maxlength=255" .
```

The definition of `incunabula:title` consists of a list of triples, all of which have `:title` as their subject. To avoid repeating `:title` for each triple, Turtle syntax allows us to use a semicolon (`;`) to separate triples that have the same subject. Moreover, some triples also have the same predicate; a comma (`,`) is used to avoid repeating the predicate. The definition of `:title` says:

- `rdf:type owl:ObjectProperty`: It is an `owl:ObjectProperty`. There are two kinds of OWL properties: object properties and datatype properties. Object properties point to objects, which have IRIs and can have their own properties. Datatype properties point to literal values, such as strings and integers.
- `rdfs:subPropertyOf dc:title`: It is a subproperty of `dc:title`, which is a subproperty of `knora-base:hasValue`. It would have been possible to define `incunabula:title` as a direct subproperty of `knora-base:hasValue`, and indeed many properties in Knora projects are defined in that way. The advantage of using `dc:title` is that if you do a search for resources that have a certain `dc:title`, and there is a resource with a matching `incunabula:title`, the search results could include that resource. (This feature is planned but not yet implemented in the Knora API server.)
- `rdfs:label "Titel"@de`, etc.: It has the specified labels in various languages. These are needed, for example, by user interfaces, to prompt the user to enter a value.
- `knora-base:subjectClassConstraint :book`: The subject of the property must be an `incunabula:book`.
- `knora-base:objectClassConstraint knora-base:TextValue`: The object of this property must be a `knora-base:TextValue` (which is a subclass of `knora-base:Value`).

- `salsah-gui:guiOrder "1"^^xsd:integer`: When a resource with this and other properties is displayed in SALSAH, this property will be displayed first. The notation `"1"^^xsd:integer` means that the literal `"1"` is of type `xsd:integer`.
- `salsah-gui:guiElement salsah-gui:SimpleText`: When SALSAH asks a user to enter a value for this property, it should use a simple text field.
- `salsah-gui:guiAttribute "size=80" , "maxlength=255"`: The SALSAH text field for entering a value for this property should be 80 characters wide, and should accept at most 255 characters.

The `incunabula` ontology contains several other property definitions that are basically similar. Note that different subclasses of `Value` are used. For example, `incunabula:pubdate`, which represents the publication date of a book, points to a `knora-base:DateValue`. The `DateValue` class stores a date range, with a specified degree of precision and a preferred calendar system for display.

A property can point to a Knora resource instead of to a Knora value. For example, in the `incunabula` ontology, there are resources representing pages and books, and each page is part of some book. This relationship is expressed using the property `incunabula:partOf`:

```
:partOf rdf:type owl:ObjectProperty ;

        rdfs:subPropertyOf knora-base:isPartOf ;

        rdfs:label "ist ein Teil von"@de ,
                   "est un part de"@fr ,
                   "e una parte di"@it ,
                   "is a part of"@en ;

        rdfs:comment """Diese Property bezeichnet eine Verbindung zu einer anderen Resource, in de

        knora-base:subjectClassConstraint :page ;

        knora-base:objectClassConstraint :book ;

        salsah-gui:guiOrder "2"^^xsd:integer ;

        salsah-gui:guiElement salsah-gui:Searchbox .
```

The key things to notice here are:

- `rdfs:subPropertyOf knora-base:isPartOf`: The Knora base ontology provides a generic `isPartOf` property to express part-whole relationships. Like many properties defined in `knora-base`, a project cannot use `knora-base:isPartOf` directly, but must make a subproperty such as `incunabula:partOf`. It is important to note that `knora-base:isPartOf` is a subproperty of `knora-base:hasLinkTo`. Any property that points to a `knora-base:Resource` must be a subproperty of `knora-base:hasLinkTo`. In Knora terminology, such a property is called a *link property*.
- `knora-base:objectClassConstraint :book`: The object of this property must be a member of the class `incunabula:book`, which, as we will see below, is a subclass of `knora-base:Resource`.
- `salsah-gui:guiElement salsah-gui:Searchbox`: When SALSAH prompts a user to select the book that a page is part of, it should provide a search box enabling the user to find the desired book.

Because `incunabula:partOf` is a link property, it must always accompanied by a *link value property*, which enables Knora to store metadata about each link that is created with the link property. This metadata includes the date and time when the link was created, its owner, the permissions it grants, and whether it has been deleted. Storing this metadata allows Knora to authorise users to see or modify the link, as well as to query a previous state of a repository in which a deleted link had not yet been deleted. (The ability to query previous states of a repository is planned for Knora API version 2.)

The name of a link property and its link value property must be related by the following naming convention: to determine the name of the link value property, add the word `Value` to the name of the link property. Hence, the `incunabula` ontology defines the property `partOfValue`:

```
:partOfValue rdf:type owl:ObjectProperty ;

                rdfs:subPropertyOf knora-base:isPartOfValue ;

                knora-base:subjectClassConstraint :page ;

                knora-base:objectClassConstraint knora-base:LinkValue .
```

As a link value property, `incunabula:partOfValue` must point to a `knora-base:LinkValue`. The `LinkValue` class is an RDF *reification* of a triple (in this case, the triple that links a page to a book). For more details about this, see *LinkValue*.

Note that the property `incunabula:hasAuthor` points to a `knora-base:TextValue`, because the `incunabula` project repåresents authors simply by their names. A more complex project could represent each author as a resource, in which case `incunabula:hasAuthor` would need to be a subproperty of `knora-base:hasLinkTo`.

Resource Classes

The two main resource classes in the `incunabula` ontology are `book` and `page`. Here is `incunabula:book`:

```
:book rdf:type owl:Class ;

      rdfs:subClassOf knora-base:Resource ,
                      [
                         rdf:type owl:Restriction ;
                         owl:onProperty :title ;
                         owl:minCardinality "1"^^xsd:nonNegativeInteger ] ,
                      [
                         rdf:type owl:Restriction ;
                         owl:onProperty :hasAuthor ;
                         owl:minCardinality "0"^^xsd:nonNegativeInteger ] ,
                      [
                         rdf:type owl:Restriction ;
                         owl:onProperty :publisher ;
                         owl:minCardinality "0"^^xsd:nonNegativeInteger ] ,
                      [
                         rdf:type owl:Restriction ;
                         owl:onProperty :publoc ;
                         owl:maxCardinality "1"^^xsd:nonNegativeInteger ] ,
                      [
                         rdf:type owl:Restriction ;
                         owl:onProperty :pubdate ;
                         owl:maxCardinality "1"^^xsd:nonNegativeInteger ] ,
                      [
                         rdf:type owl:Restriction ;
                         owl:onProperty :location ;
                         owl:maxCardinality "1"^^xsd:nonNegativeInteger ] ,
                      [
                         rdf:type owl:Restriction ;
                         owl:onProperty :url ;
                         owl:maxCardinality "1"^^xsd:nonNegativeInteger ] ,
                      [
                         rdf:type owl:Restriction ;
                         owl:onProperty :description ;
                         owl:maxCardinality "1"^^xsd:nonNegativeInteger ] ,
                      [
                         rdf:type owl:Restriction ;
                         owl:onProperty :physical_desc ;
                         owl:maxCardinality "1"^^xsd:nonNegativeInteger ] ,
                      [
```

```
                        rdf:type owl:Restriction ;
                        owl:onProperty :note ;
                        owl:minCardinality "0"^^xsd:nonNegativeInteger ] ,
                    [
                        rdf:type owl:Restriction ;
                        owl:onProperty :citation ;
                        owl:minCardinality "0"^^xsd:nonNegativeInteger ] ,
                    [
                        rdf:type owl:Restriction ;
                        owl:onProperty :book_comment ;
                        owl:minCardinality "0"^^xsd:nonNegativeInteger ] ;

    knora-base:resourceIcon "book.gif" ;

    rdfs:label "Buch"@de ,
               "Livre"@fr ,
               "Libro"@it ,
               "Book"@en ;

    rdfs:comment """Diese Resource-Klasse beschreibt ein Buch"""@de .
```

Like every Knora resource class, `incunabula:book` is a subclass of `knora-base:Resource`. It is also a subclass of a number of other classes of type `owl:Restriction`, which are defined in square brackets, using Turtle's syntax for anonymous blank nodes. Each `owl:Restriction` specifies a cardinality for a property that is allowed in resources of type `incunabula:book`. A cardinality is indeed a kind of restriction: it means that a resource of this type may have, or must have, a certain number of instances of the specified property. For example, `incunabula:book` has cardinalities saying that a book must have at least one title and at most one publication date. In the Knora API version 1, the word 'occurrence' is used instead of 'cardinality'.

As explained in *OWL Cardinalities*, these are the cardinalities supported by Knora:

- `owl:cardinality 1` A resource of this class must have exactly one instance of the specified property (occurrence `1`).
- `owl:minCardinality 1` A resource of this class must have at least one instance of the specified property (occurrence `1-n`).
- `owl:maxCardinality 1` A resource of this class may have zero or one instance of the specified property (occurrence `0-1`).
- `owl:minCardinality 0` A resource of this class may have zero or more instances of the specified property (occurrence `0-n`).

Note that `incunabula:book` specifies a cardinality of `owl:minCardinality 0` on the property `incunabula:hasAuthor`. At first glance, this might seem as if it serves no purpose, since it says that the property is optional and can have any number of instances. You may be wondering whether this cardinality could simply be omitted from the definition of `incunabula:book`. However, Knora requires every property of a resource to have some cardinality in the resource's class. This is because Knora uses the cardinalities to determine which properties are *possible* for instances of the class, and the Knora API relies on this information. If there was no cardinality for `incunabula:hasAuthor`, Knora would not allow a book to have an author.

Here is the definition of `incunabula:page`:

```
:page rdf:type owl:Class ;

    rdfs:subClassOf knora-base:StillImageRepresentation ,
                    [
                        rdf:type owl:Restriction ;
                        owl:onProperty :pagenum ;
                        owl:maxCardinality "1"^^xsd:nonNegativeInteger ] ,
                    [
                        rdf:type owl:Restriction ;
                        owl:onProperty :partOfValue ;
                        owl:cardinality "1"^^xsd:nonNegativeInteger ] ,
```

```
                    [
                       rdf:type owl:Restriction ;
                       owl:onProperty :partOf ;
                       owl:cardinality "1"^^xsd:nonNegativeInteger ] ,
                    [
                       rdf:type owl:Restriction ;
                       owl:onProperty :seqnum ;
                       owl:maxCardinality "1"^^xsd:nonNegativeInteger ] ,
                    [
                       rdf:type owl:Restriction ;
                       owl:onProperty :description ;
                       owl:maxCardinality "1"^^xsd:nonNegativeInteger ] ,
                    [
                       rdf:type owl:Restriction ;
                       owl:onProperty :citation ;
                       owl:minCardinality "0"^^xsd:nonNegativeInteger ] ,
                    [
                       rdf:type owl:Restriction ;
                       owl:onProperty :page_comment ;
                       owl:minCardinality "0"^^xsd:nonNegativeInteger ] ,
                    [
                       rdf:type owl:Restriction ;
                       owl:onProperty :origname ;
                       owl:cardinality "1"^^xsd:nonNegativeInteger ] ,
                    [
                       rdf:type owl:Restriction ;
                       owl:onProperty :hasLeftSidebandValue ;
                       owl:maxCardinality "1"^^xsd:nonNegativeInteger ] ,
                    [
                       rdf:type owl:Restriction ;
                       owl:onProperty :hasLeftSideband ;
                       owl:maxCardinality "1"^^xsd:nonNegativeInteger ] ,
                    [
                       rdf:type owl:Restriction ;
                       owl:onProperty :hasRightSidebandValue ;
                       owl:maxCardinality "1"^^xsd:nonNegativeInteger ] ,
                    [
                       rdf:type owl:Restriction ;
                       owl:onProperty :hasRightSideband ;
                       owl:maxCardinality "1"^^xsd:nonNegativeInteger ] ,
                    [
                       rdf:type owl:Restriction ;
                       owl:onProperty :transcription ;
                       owl:minCardinality "0"^^xsd:nonNegativeInteger ] ;

    knora-base:resourceIcon "page.gif" ;

    rdfs:label "Seite"@de ,
               "Page"@fr ,
               "Page"@en ;

    rdfs:comment """Eine Seite ist ein Teil eines Buchs"""@de ,
                 """Une page est une partie d'un livre"""@fr ,
                 """A page is a part of a book"""@en .
```

The `incunabula:page` class is a subclass of `knora-base:StillImageRepresentation`, which is a subclass of `knora-base:Representation`, which is a subclass of `knora-base:Resource`. The class `knora-base:Representation` is used for resources that contain metadata about files stored by Knora. Each It has different subclasses that can hold different types of files, including still images, audio, and video files. A given `Representation` can store metadata about several different files, as long as they are of the same type and are semantically equivalent, e.g. are different versions of the same image with different colorspaces, so that coordinates in one file will work in the other files.

In Knora, a subclass inherits the cardinalities defined in its superclasses. Let's look at the class hierarchy of `incunabula:page`, starting with `knora-base:Representation`:

```
:Representation rdf:type owl:Class ;

                rdfs:subClassOf :Resource ,
                                [ rdf:type owl:Restriction ;
                                  owl:onProperty :hasFileValue ;
                                  owl:minCardinality "1"^^xsd:nonNegativeInteger
                                ] ;

                rdfs:comment "A resource that can store one or more FileValues"@en .
```

This says that a `Representation` must have at least one instance of the property `hasFileValue`, which is defined like this:

```
:hasFileValue rdf:type owl:ObjectProperty ;

              rdfs:subPropertyOf :hasValue ;

              :subjectClassConstraint :Representation ;

              :objectClassConstraint :FileValue .
```

The subject of `hasFileValue` must be a `Representation`, and its object must be a `FileValue`. There are different subclasses of `FileValue` for different kinds of files, but we'll skip the details here.

This is the definition of `knora-base:StillImageRepresentation`:

```
:StillImageRepresentation rdf:type owl:Class ;

                          rdfs:subClassOf :Representation  ,
                                [ rdf:type owl:Restriction ;
                                  owl:onProperty :hasStillImageFileValue ;
                                  owl:minCardinality "1"^^xsd:nonNegativeInteger
                                ] ;

                          rdfs:comment "A resource that can contain two-dimensional still image fi
```

It must have at least one instance of the property `hasStillImageFileValue`, which is defined as follows:

```
:hasStillImageFileValue rdf:type owl:ObjectProperty ;

              rdfs:subPropertyOf :hasFileValue ;

              :subjectClassConstraint :StillImageRepresentation ;

              :objectClassConstraint :StillImageFileValue .
```

Because `hasStillImageFileValue` is a subproperty of `hasFileValue`, the cardinality on `hasStillImageFileValue`, defined in the subclass `StillImageRepresentation`, overrides the cardinality on `hasFileValue`, defined in the superclass `Representation`. In other words, the more general cardinality in the superclass is replaced by a more specific cardinality in the base class. Since `incunabula:page` is a subclass of `StillImageRepresentation`, it inherits the cardinality on `hasStillImageFileValue`. As a result, a page must have at least one image file attached to it.

Here's another example of cardinality inheritance. The class `knora-base:Resource` has a cardinality for `knora-base:seqnum`. The idea is that resources of any type could be arranged in some sort of sequence. As we saw above, `incunabula:page` is a subclass of `knora-base:Resource`. But `incunabula:page` has its own cardinality for `incunabula:seqnum`, which is a subproperty of `knora-base:seqnum`. Once again, the subclass's cardinality on the subproperty replaces the superclass's cardinality on the superproperty: a page is allowed to have an `incunabula:seqnum`, but it is not allowed to have a `knora-base:seqnum`.

CHAPTER 3

The Knora Ontologies

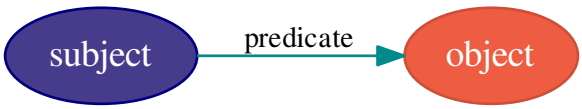
The Knora ontologies provide a generic framework for describing humanities research data, allowing data from different projects to be combined, augmented, and reused.

The Knora Base Ontology

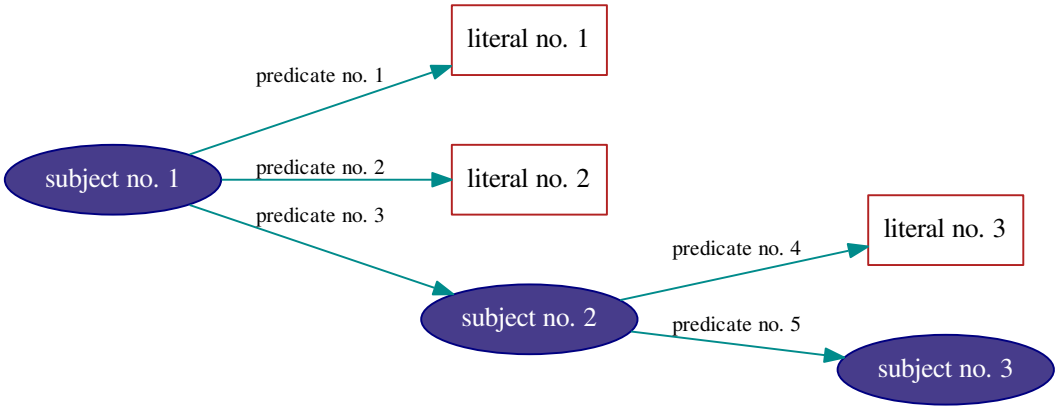
Introduction

Resource Description Framework (RDF)

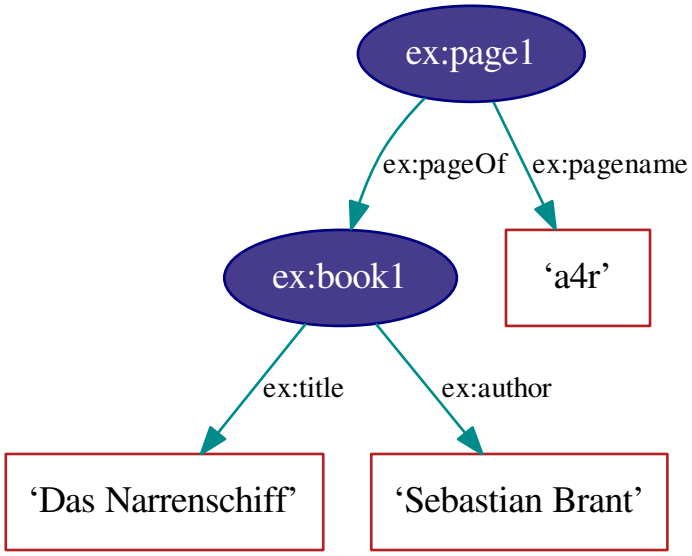
Knora uses a hierarchy of ontologies based on the Resource Description Framework (RDF), RDF Schema (RDFS), and the Web Ontology Language (OWL). Both RDFS and OWL are expressed in RDF. RDF expresses information as a set of statements (called *triples*). A triple consists of a subject, a predicate, and an object:



The object may be either a literal value (such as a name or number) or another subject. Thus it is possible to create complex graphs that connect many subjects, like this:



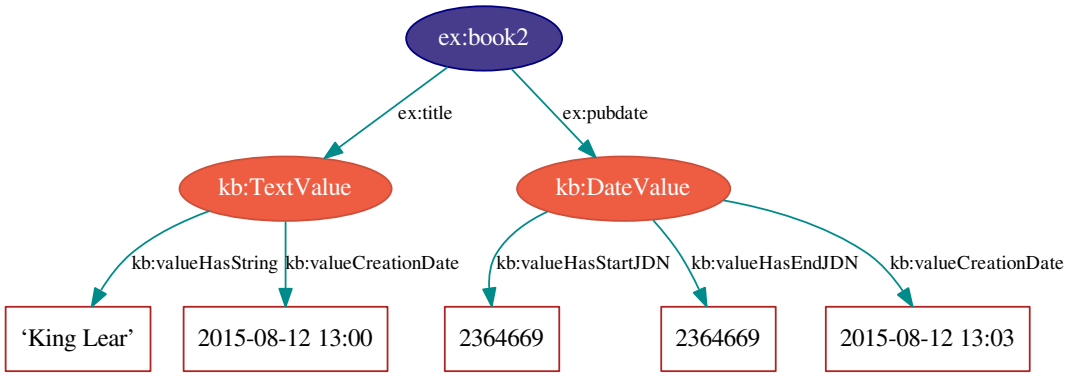
In RDF, each subject and predicate has a unique, URL-like identifier called an Internationalized Resource Identifier (IRI). Within a given project, IRIs typically differ only in their last component (the "local part"), which is often the fragment following a `#` character. Such IRIs share a long "prefix". In Turtle and similar formats for writing RDF, a short prefix label can be defined to represent the long prefix. Then an IRI can be written as a prefix label and a local part, separated by a colon (`:`). For example, if the "example" project's long prefix is `http://www.example.org/rdf#`, and it contains subjects with IRIs like `http://www.example.org/rdf#book`, we can define the prefix label `ex` to represent the prefix label, and write prefixed names for IRIs:



In this document, we use the prefix label `kb` to represent the Knora base ontology, [1] but we usually omit it for brevity.

The Knora Data Model

The Knora data model is based on the observation that, in the humanities, a value or literal is often itself structured and can be highly complex. Moreover, a value may have its own metadata, such as its creation date, information about permissions, and so on. Therefore, the Knora base ontology describes structured value types that can store this type of metadata. In the diagram below, a book (`ex:book2`) has a title (identified by the predicate `ex:title`) and a publication date (`ex:pubdate`), each of which has some metadata.



Projects

In Knora, each item of data belongs to some particular project. Each project using Knora must define a `kb:knoraProject`, which has these properties (cardinalities are indicated in parentheses after each property

[1] `http://www.knora.org/ontology/knora-base#`

name):

shortname (1) A short name that can be used to identify the project in configuration files and the like.

basepath (1) The filesystem path of the directory where the project's files are stored.

foaf:name (0-1) The name of the project.

description (0-1) A description of the project.

belongsTo (0-1) The `kb:Institution` that the project belongs to.

Resources and values are associated with a project by means of the `kb:attachedToProject` property, as described in *The Knora Data Model*. Users are associated with a project by means of the `kb:isInProject` property, as described in *Users and Groups*.

Resources

All the content produced by a project (e.g. digitised primary source materials or research data) must be stored in objects that belong to subclasses of `kb:Resource`, so that the Knora API server can query and update that content. Each project using the Knora base ontology must define its own OWL classes, derived from `kb:Resource`, to represent the types of data it deals with. A subclass of `kb:Resource` may additionally be a subclass of any other class, e.g. an industry-standard class such as `foaf:Person`; this can facilitate searches across projects.

Resources have properties that point to different parts of the content they contain. For example, a resource representing a book could have a property called `hasAuthor`, pointing to the author of the book. There are two possible kinds of content in a Knora resource: Knora values (see *Values*) or links to other resources (see *Links Between Resources*). Properties that point to Knora values must be subproperties of `kb:hasValue`, and properties that point to other resources must be subproperties of `kb:hasLinkTo`. Either of these two types of properties may also be a subproperty of any other property, e.g. an industry-standard property such as `foaf:name`; this can facilitate searches across projects. Each property definition must specify the types that its subjects and objects must belong to (see *Constraints on the Types of Property Subjects and Objects* for details).

Each project-specific resource class definition must use OWL cardinality restrictions to specify the properties that resources of that class can have (see *OWL Cardinalities* for details).

Resources are not versioned; only their values are versioned (see *Values*).

Every resource is required to have an `rdfs:label`. The object of this property is an `xsd:string`, rather than a Knora value; hence it is not versioned. A user who has modify permission on a resource *Authorization* can change its label.

A resource can be marked as deleted; the Knora API server does this by adding the predicate `kb:isDeleted true` to the resource. An optional `kb:deleteComment` may be added to explain why the resource has been marked as deleted. Deleted resources are normally hidden. They cannot be undeleted, because even though resources are not versioned, it is necessary to be able to find out when a resource was deleted. If desired, a new resource can be created by copying data from a deleted resource.

Properties of Resource

creationDate (1) The time when the resource was created.

attachedToUser (1) The user who owns the resource.

attachedToProject (1) The project that the resource is part of.

lastModificationDate (0-1) A timestamp indicating when the resource (or one of its values) was last modified.

seqnum (0-1) The sequence number of the resource, if it is part of an ordered group of resources, such as the pages in a book.

isDeleted (1) Indicates whether the resource has been deleted.

deleteDate (0-1) If the resource has been deleted, indicates when it was deleted.

deleteComment (0-1) If the resource has been deleted, indicates why it was deleted.

Resources can have properties that point to other resources; see *Links Between Resources*. A resource grants permissions to groups of users; see *Authorization*.

Representations

It is not practical to store all data in RDF. In particular, RDF is not a good storage medium for binary data such as images. Therefore, Knora stores such data outside the triplestore, in ordinary files. A resource can have one or more files attached to it. For each file, there is a `kb:FileValue` in the triplestore containing metadata about the file (see *FileValue*). A resource that has file values must belong to one of the subclasses of `kb:Representation`. The base class `Representation`, which is not intended to be used directly, has this property:

hasFileValue (1-n) Points to one or more file values.

Its subclasses, which are intended to be used directly in data, include:

StillImageRepresentation A representation containing still image files.

MovingImageRepresentation A representation containing video files.

AudioRepresentation A representation containing audio files.

DDDrepresentation A representation containing 3D images.

TextRepresentation A representation containing formatted text files, such as XML files.

DocumentRepresentation A representation containing documents (such as PDF files) that are not text files.

There are two ways for a project to design classes for representations. The simpler way is to create a resource class that represents a thing in the world (such as `ex:Painting`) and also belongs to a subclass of `Representation`. This is adequate if the class can have only one type of file attached to it. For example, if paintings are represented only by still images, `ex:Painting` could be a subclass of `StillImageRepresentation`. This is the only approach supported in version 1 of the Knora API.

The more flexible approach, which is allowed by the Knora base ontology and will be supported by version 2 of the Knora API, is for each `ex:Painting` to use the `kb:hasRepresentation` property to point to other resources containing files that represent the painting. Each of these other resources can extend a different subclass of `Representation`. For example, a painting could have a `StillImageRepresentation` as well as a `DDDrepresentation`.

Standard Resource Classes

In general, each project using Knora must define its own subclasses of `kb:Resource`. However, the Knora base ontology provides some standard subclasses of `kb:Resource`, which are intended to be used by any project:

Region Represents a region of a `Representation` (see *Representations*).

Annotation Represents an annotation of a resource. The `hasComment` property points to the text of the annotation, represented as a `kb:TextValue`.

LinkObj Represents a link that connects two or more resources. A `LinkObj` has a `hasLinkTo` property pointing to each resource that it connects, as well as a `hasLinkToValue` property pointing to a reification of each of these direct links (see *Links Between Resources*). A `LinkObj` is more complex (and hence less convenient and readable) than a simple direct link, but it has the advantage that it can be annotated using an `Annotation`. For improved readability, a project can make its own subclasses of `LinkObj` with specific meanings.

Values

The Knora base ontology defines a set of OWL classes that are derived from `kb:Value` and represent different types of structured values found in humanities data. This set of classes may not be extended by project-specific ontologies.

A value is always part of one particular resource, which points to it using some property derived from `hasValue`. For example, a project-specific ontology could specify a `Book` class with a property `hasSummary` (derived from `hasValue`), and that property could have a `knora-base:objectClassConstraint` of `TextValue`. This would mean that the summary of each book is represented as a `TextValue`.

Knora values are versioned. Existing values are not modified. Instead, a new version of an existing value is created. The new version is linked to the old version via the `previousValue` property.

"Deleting" a value means marking it with `kb:isDeleted`. An optional `kb:deleteComment` may be added to explain why the value has been marked as deleted. Deleted values are normally hidden.

Most types of values are marked as deleted without creating a new version of the value. However, link values must be treated as a special case. Before a `LinkValue` can be marked as deleted, its reference count must be decremented to 0. Therefore, a new version of the `LinkValue` is made, with a reference count of 0, and it is this new version that is marked as deleted.

To simplify the enforcement of ontology constraints, and for consistency with resource updates, no new versions of a deleted value can be made; it is not possible to undelete. Instead, if desired, a new value can be created by copying data from a deleted value.

Properties of Value

`valueCreationDate` (1) The date and time when the value was created.

`attachedToUser` (1) The user who owns the value.

`attachedToProject` (0-1) The project that the value is part of. If not specified, defaults to the project of the containing resource.

`valueHasString` (1) A human-readable string representation of the value's contents, which is available to Knora's full-text search index.

`valueHasOrder` (0-1) A resource may have several properties of the same type with different values (which will be of the same class), and it may be necessary to indicate an order in which these values occur. For example, a book may have several authors which should appear in a defined order. Hence, `valueHasOrder`, when present, points to an integer literal indicating the order of a given value relative to the other values of the same property. These integers will not necessarily start at any particular number, and will not necessarily be consecutive.

`previousValue` (0-1) The previous version of the value.

`isDeleted` (1) Indicates whether the value has been deleted.

`deleteDate` (0-1) If the value has been deleted, indicates when it was deleted.

`deleteComment` (0-1) If the value has been deleted, indicates why it was deleted.

Each Knora value can grant permissions (see *Authorization*).

Subclasses of Value

TextValue Represents text, possibly including markup. The text is the object of the `valueHasString` property. A line break is represented as a Unicode line feed character (`U+000A`). The non-printing Unicode character `INFORMATION SEPARATOR TWO (U+001E)` can be used to separate words that are separated only by standoff markup (see below), so they are recognised as separate in a full-text search index.

Markup is stored using this property:

valueHasStandoff (0-n) Points to a standoff markup tag. See *Text with Standoff Markup*.

valueHasMapping (0-1) Points to the mapping used to create the standoff markup and to convert it back to the original XML. See *Mapping to Create Standoff From XML*.

DateValue Humanities data includes many different types of dates. In Knora, a date has a specified calendar, and is always represented as a period with start and end points (which may be equal), each of which has a precision (`DAY`, `MONTH`, or `YEAR`). Internally, the start and end points are stored as two Julian Day Numbers. This calendar-independent representation makes it possible to compare and search for dates regardless of the calendar in which they were entered. Properties:

valueHasCalendar (1) The name of the calendar in which the date should be displayed. Currently `GREGORIAN` and `JULIAN` are supported.

valueHasStartJDN (1) The Julian Day Number of the start of the period (an `xsd:integer`).

valueHasStartPrecision (1) The precision of the start of the period.

valueHasEndJDN (1) The Julian Day Number of the end of the period (an `xsd:integer`).

valueHasEndPrecision (1) The precision of the end of the period.

IntValue Represents an integer. Property:

valueHasInteger (1) An `xsd:integer`.

ColorValue

valueHasColor (1) A string representing a color. The string encodes a color as hexadecimal RGB values, e.g. "#FF0000".

DecimalValue Represents an arbitrary-precision decimal number. Property:

valueHasDecimal (1) An `xsd:decimal`.

UriValue Represents a non-Knora URI. Property:

valueHasUri (1) An `xsd:anyURI`.

BooleanValue Represents a boolean value. Property:

valueHasBoolean (1) An `xsd:boolean`.

GeomValue Represents a geometrical object as a JSON string, using normalized coordinates. Property:

valueHasGeometry (1) A JSON string.

GeonameValue Represents a geolocation, using the identifiers found at GeoNames. Property:

valueHasGeonameCode (1) the identifier of a geographical feature from GeoNames, represented as an `xsd:string`.

IntervalValue Represents a time interval, with precise start and end times on a timeline, e.g. relative to the beginning of an audio or video file. Properties:

valueHasIntervalStart (1) An `xsd:decimal` representing the start of the interval in seconds.

valueHasIntervalEnd (1) An `xsd:decimal` representing the end of the interval in seconds.

ListValue Projects often need to define lists or hierarchies of categories that can be assigned to many different resources. Then, for example, a user interface can provide a drop-down menu to allow the user to assign a category to a resource. The `ListValue` class provides a way to represent these sorts of data structures. It can represent either a flat list or a tree.

A `ListValue` has this property:

`valueHasListNode` (1) Points to a `ListNode`.

Each `ListNode` can have the following properties:

`isRootNode` (0-1) Set to `true` if this is the root node.

`hasSubListNode` (0-n) Points to the node's child nodes, if any.

`hasRootNode` (0-1) Points to the root node of the list (absent if `isRootNode` is `true`).

`listNodePosition` (0-1) An integer indicating the node's position in the list of its siblings (absent if `isRootNode` is `true`).

`listNodeName` (0-1) The node's human-readable name (absent if `isRootNode` is `true`).

FileValue Knora stores certain kinds of data outside the triplestore, in files (see *Representations*). Each digital object that is stored outside the triplestore has associated metadata, which is stored in the triplestore in a `kb:FileValue`. The base class `FileValue`, which is not intended to be used directly, has these properties:

`internalFilename` (1) The name of the file as stored by the Knora API server.

`internalMimeType` (1) The MIME type of the file as stored by the Knora API server.

`originalFilename` (0-1) The original name of the file when it was uploaded to the Knora API server.

`originalMimeType` (0-1) The original MIME type of the file when it was uploaded to the Knora API server.

`isPreview` (0-1) A boolean indicating whether the file is a preview, i.e. a small image representing the contents of the file. A preview is always a `StillImageFileValue`, regardless of the type of the enclosing `Representation`.

The subclasses of `FileValue`, which are intended to be used directly in data, include:

`StillImageFileValue` Contains metadata about a still image file.

`MovingImageFileValue` Contains metadata about a video file.

`AudioFileValue` Contains metadata about an audio file.

`DDDFileValue` Contains metadata about a 3D image file.

`TextFileValue` Contains metadata about a text file.

`DocumentFileValue` Contains metadata about a document (such as PDF) that is not a text file.

Each of these classes contains properties that are specific to the type of file it describes. For example, still image files have dimensions, video files have frame rates, and so on.

The files in a given representation must be semantically equivalent, meaning that coordinates that relate to one file must also be valid for other files in the same representation. Coordinates in Knora are expressed as fractions of the size of the object on some dimension; for example, image coordinates are expressed as fractions of its width and height, rather than in pixels. Therefore, the image files in a `StillImageRepresentation` must have the same aspect ratio, but they need not have the same dimensions in pixels. Similarly, the audio and video files in an `AudioRepresentation` or `MovingImageRepresentation` must have the same length in seconds, but may have different bitrates.

`FileValue` objects are versioned like other values, and the actual files stored by Knora are also versioned. Version 1 of the Knora API does not provide a way to retrieve a previous version of a file, but this feature will be added in a subsequent version of the API.

LinkValue A `LinkValue` is an RDF "reification" containing metadata about a link between two resources. It is therefore a subclass of `rdf:Statement` as well as of `Value`. It has these properties:

`rdf:subject` (1) The resource that is the source of the link.

`rdf:predicate` (1) The link property.

`rdf:object` (1) The resource that is the target of the link.

`valueHasRefCount` (1) The reference count of the link. This is meaningful when the `LinkValue` describes resource references in Standoff text markup (see *StandoffLinkTag*). Otherwise, the reference count will always be 1 (if the link exists) or 0 (if it has been deleted).

For details about how links are created in Knora, see *Links Between Resources*.

ExternalResValue Represents a resource that is not stored in the RDF triplestore managed by the Knora API server, but instead resides in an external repository managed by some other software. The `ExternalResValue` contains the information that the Knora API server needs in order to access the resource, assuming that a suitable gateway plugin is installed.

`extResAccessInfo` (1) The location of the repository containing the external resource (e.g. its URL).

`extResId` (1) The repository-specific ID of the external resource.

`extResProvider` (1) The name of the external provider of the resource.

Links Between Resources

A link between two resources is expressed, first of all, as a triple, in which the subject is the resource that is the source of the link, the predicate is a "link property" (a subproperty of `kb:hasLinkTo`), and the object is the resource that is the target of the link.

It is also useful to store metadata about links. For example, Knora needs to know who owns the link, who has permission to modify it, when it was created, and so on. Such metadata cannot simply describe the link property, because then it would refer to that property in general, not to any particular instance in which that property is used to connect two particular resources. To attach metadata to a specific link in RDF, it is necessary to create an RDF "reification". A reification makes statements about a particular triple (subject, predicate, object), in this case the triple that expresses the link between the resources. Knora uses reifications of type `kb:LinkValue` (described in *LinkValue*) to store metadata about links.

For example, suppose a project describes paintings that belong to collections. The project can define an ontology as follows (expressed here in Turtle format, and simplified for the purposes of illustration):

```
@prefix kb <http://www.knora.org/ontology/knora-base#> .
@prefix : <http://www.knora.org/ontology/paintings#> .

:Painting rdf:type owl:Class ;
    rdfs:subClassOf kb:Resource ,
        [ rdf:type owl:Restriction ;
            owl:onProperty :hasArtist ;
            owl:cardinality 1 ] ,
        [ rdf:type owl:Restriction ;
              owl:onProperty :hasTitle ;
              owl:cardinality 1 ] ;
        [ rdf:type owl:Restriction ;
              owl:onProperty :isInCollection ;
              owl:minCardinality 1 ] ;
        [ rdf:type owl:Restriction ;
              owl:onProperty :isInCollectionValue ;
              owl:minCardinality 1 ] .

:Collection rdf:type owl:Class ;
    rdfs:subClassOf kb:Resource ,
        [ rdf:type owl:Restriction ;
```

```
            owl:onProperty :hasCollectionName ;
            owl:cardinality 1 ] .

:hasArtist rdf:type owl:ObjectProperty ;
    rdfs:label "Name of artist" ;
    kb:subjectClassConstraint :Painting ;
    kb:objectClassConstraint kb:TextValue .

:hasTitle rdf:type owl:ObjectProperty ;
    rdfs:label "Title of painting"
    kb:subjectClassConstraint :Painting ;
    kb:objectClassConstraint kb:TextValue .

:hasCollectionName rdf:type owl:ObjectProperty ;
    rdfs:label "Name of collection" ;
    kb:subjectClassConstraint :Collection ;
    kb:objectClassConstraint kb:TextValue .
```

To link the paintings to the collection, we must add a "link property" to the ontology. In this case, the link property will point from a painting to the collection it belongs to. Every link property must be a subproperty of `kb:hasLinkTo`.

```
:isInCollection rdf:type owl:ObjectProperty ;
    rdfs:subPropertyOf kb:hasLinkTo ;
    kb:subjectClassConstraint :Painting ;
    kb:objectClassConstraint :Collection .
```

We must then add a "link value property", which will point from a painting to a `kb:LinkValue` (described in *LinkValue*), which will contain metadata about the link between the property and the collection. In particular, the link value specifies the creator of the link, the date when it was created, and the permissions that determine who can view or modify it. The name of the link value property is constructed using a simple naming convention: the word `Value` is appended to the name of the link property. In this case, since our link property is called `:isInCollection`, the link value property must be called `:isInCollectionValue`. Every link value property must be a subproperty of `kb:hasLinkToValue`.

```
:isInCollectionValue rdf:type owl:ObjectProperty ;
    rdfs:subPropertyOf kb:hasLinkToValue ;
    kb:subjectClassConstraint :Painting ;
    kb:objectClassConstraint kb:LinkValue .
```

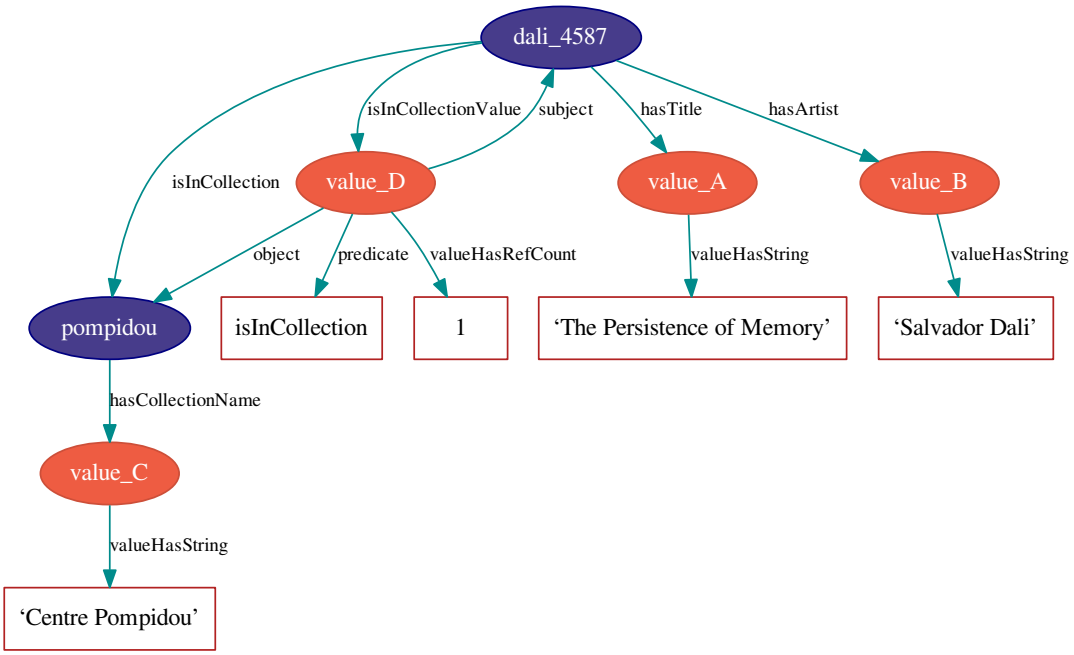
Given this ontology, we can create some RDF data describing a painting and a collection:

```
@prefix paintings <http://www.knora.org/ontology/paintings#> .
@prefix data <http://www.knora.org/ontology/paintings/data#> .

data:dali_4587 rdf:type paintings:Painting ;
    paintings:hasTitle data:value_A ;
    paintings:hasArtist data:value_B .

data:value_A rdf:type kb:TextValue ;
    kb:valueHasString "The Persistence of Memory" .

data:value_B rdf:type kb:TextValue ;
    kb:valueHasString "Salvador Dali" .

data:pompidou rdf:type paintings:Collection ;
    paintings:hasCollectionName data:value_C .

data:value_C rdf:type kb:TextValue ;
    kb:valueHasString "Centre Pompidou, Paris" .
```

We can then state that the painting is in the collection:

```
data:dali_4587 paintings:isInCollection data:pompidou ;
    paintings:isinCollectionValue data:value_D .

data:value_D rdf:type kb:LinkValue ;
    rdf:subject data:dali_4587 ;
    rdf:predicate paintings:isInCollection ;
    rdf:object data:pompidou ;
    kb:valueHasRefCount 1 .
```

This creates a link (`paintings:isInCollection`) between the painting and the collection, along with a reification containing metadata about the link. We can visualise the result as the following graph:



The Knora API server allows a user to see a link if the requesting user has permission to see the source and target resources as well as the `kb:LinkValue`.

Text with Standoff Markup

Knora is designed to be able to store text with markup, which can indicate formatting and structure, as well as the complex observations involved in transcribing handwritten manuscripts. One popular way of representing text in the humanities is to encode it in XML using the Text Encoding Initiative (TEI) guidelines. [2] In Knora, a TEI/XML document can be stored as a file with attached metadata, but this is not recommended, because it does not allow Knora to perform searches across multiple documents.

The recommended way to store text with markup in Knora is to use Knora's built-in support for "standoff" markup, which is stored separately from the text. This has some advantages over embedded markup such as XML. [3] While XML requires markup to have a hierarchical structure, and does not allow overlapping tags, standoff nodes do not have these limitations (*Schmidt2016*). A standoff tag can be attached to any substring in the text by giving its start and end positions. [4] For example, suppose we have the following text:

[2] TEI refers both to an organization and an XML-based markup language (or more precisely: a set of grammar modules – XML schemas – that can be combined to define a markup language). For reasons of clarity, we use the term TEI/XML to refer to the markup language.

[3] It is also possible to encode standoff markup using XML. For example, the TEI guidelines discuss standoff markup. However, standoff markup is not widely applied in the TEI community. TEI's main focus is on encoding a hierarchy of elements.

[4] Unlike in corpus linguistics, we do not use any tokenization resulting in a form of predefined segmentation that would limit the user's possibility to freely annotate any ranges in the text.

This would require just two standoff tags: `(italic, start=5, end=29)` and `(bold, start=14, end=36)`.

Moreover, standoff makes it possible to mark up the same text in different, possibly incompatible ways, allowing for different interpretations without making redundant copies of the text. In the Knora base ontology, any text value can have standoff tags.

By representing standoff as RDF triples, Knora makes markup searchable across multiple text documents in a repository. For example, if a repository contains documents in which references to persons are indicated in standoff, it is straightforward to find all the documents mentioning a particular person. Knora's standoff support is intended to make it possible to convert documents with embedded, hierarchical markup, such as TEI/XML, into RDF standoff and back again, with no data loss, thus bringing the benefits of RDF to existing TEI-encoded documents.

In the Knora base ontology, a `TextValue` can have one or more standoff tags. Each standoff tag indicates the start and end positions of a substring in the text that has a particular attribute. The OWL class `kb:StandoffTag`, which is the base class of all standoff node classes, has these properties:

`standoffTagHasStart` (1) The index of the first character in the text that has the attribute.

`standoffTagHasEnd` (1) The index of the last character in the text that has the attribute, plus 1.

`standoffTagHasUUID` (1) A UUID identifying this instance and those corresponding to it in later versions of the `TextValue` it belongs to. The UUID is a means to maintain a reference to a particular range of a text also when new versions are made and standoff tag IRIs change.

`standoffTagHasOriginalXMLID` (0-1) The original id of the XML element that the standoff tag represents, if any.

`standoffTagHasStartIndex` (1) The start index of the standoff tag. Start indexes are numbered from 0 within the context of a particular text. When several standoff tags share the same start position, they can be nested correctly with this information when transforming them to XML.

`standoffTagHasEndIndex` (1) The end index of the standoff tag. Start indexes are numbered from 0 within the context of a particular text. When several standoff tags share the same end position, they can be nested correctly with this information when transforming them to XML.

`standoffTagHasStartParent` (0-1) Points to the parent standoff tag. This corresponds to the original nesting of tags in XML. If a standoff tag has no parent, it represents the XML root element. If the original XML element is a CLIX tag, it represents the start of a virtual (non syntactical) hierarchy.

`standoffTagHasEndParent` (0-1) Points to the parent standoff tag if the original XML element is a CLIX tag and represents the end of a virtual (non syntactical) hierarchy.

The `StandoffTag` class is not used directly in RDF data; instead, its subclasses are used. A few subclasses are currently provided in `standoff-onto.ttl`, and more will be added to support TEI semantics. Projects are able to define their own custom standoff tag classes (direct subclasses of `StandoffTag` or one of the standoff data type classes or subclasses of one of the standoff classes defined in `standoff-onto.ttl`).

Subclasses of StandoffTag

Standoff Data Type Tags Associates data in some Knora value type with a substring in a text. Standoff data type tags are subclasses of `ValueBase` classes.

- **`StandoffLinkTag` Indicates that a substring refers to another `kb:Resource`.** See *StandoffLinkTag*.
- **`StandoffInternalReferenceTag` Indicates that a substring refers to another standoff tag in the same text** value. See *Internal Links in a TextValue*.
- **`StandoffUriTag` Indicates that a substring is associated with a** URI, which is stored in the same form that is used for `kb:UriValue`. See *UriValue*.
- **`StandoffDateTag` Indicates that a substring represents a** date, which is stored in the same form that is used for `kb:DateValue`. See *DateValue*.

- **`StandoffColorTag` Indicates that a substring represents a color,** which is stored in the same form that is used for `kb:ColorValue`. See *ColorValue*.
- **`StandoffIntegerTag` Indicates that a substring represents an integer,** which is stored in the same form that is used for `kb:IntegerValue`. See *IntValue*.
- **`StandoffDecimalTag` Indicates that a substring represents a number with fractions,** which is stored in the same form that is used for `kb:DecimalValue`. See *DecimalValue*.
- **`StandoffIntervalTag` Indicates that a substring represents an interval,** which is stored in the same form that is used for `kb:IntervalValue`. See *IntervalValue*.
- **`StandoffBooleanTag` Indicates that a substring represents a Boolean,** which is stored in the same form that is used for `kb:BooleanValue`. See *BooleanValue*.

StandoffLinkTag A `StandoffLinkTag` Indicates that a substring is associated with a Knora resource. For example, if a repository contains resources representing persons, a text could be marked up so that each time a person's name is mentioned, a `StandoffLinkTag` connects the name to the Knora resource describing that person. Property:

`standoffTagHasLink` (1) The IRI of the resource that is referred to.

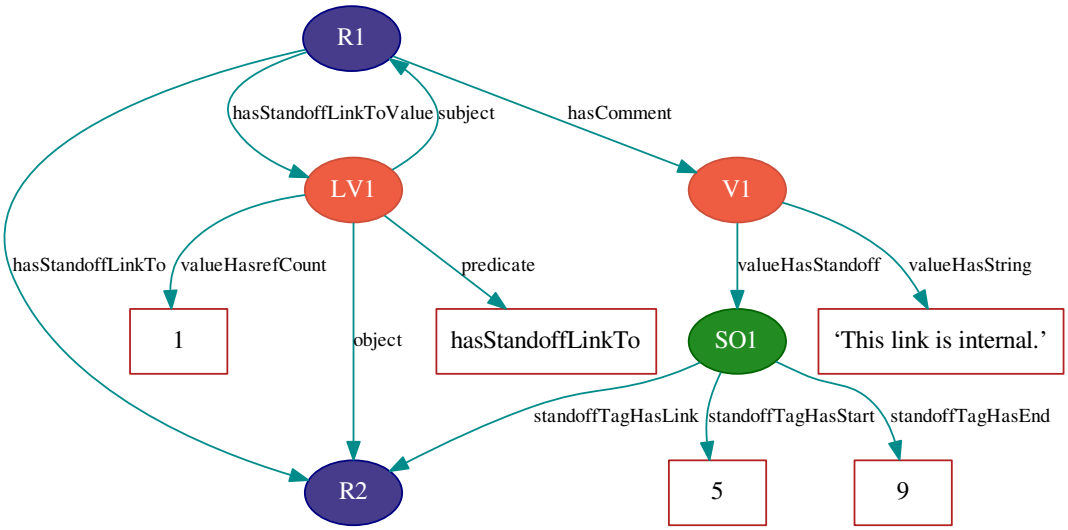
One of the design goals of the Knora ontology is to make it easy and efficient to find out which resources contain references to a given resource. Direct links are easier and more efficient to query than indirect links. Therefore, when a text value contains a resource reference in its standoff nodes, the Knora API server automatically creates a direct link between the containing resource and the target resource, along with an RDF reification (a `kb:LinkValue`) describing the link, as discussed in *Links Between Resources*. In this case, the link property is always `kb:hasStandoffLinkTo`, and the link value property (which points to the `LinkValue`) is always `kb:hasStandoffLinkToValue`.

The Knora API server automatically updates direct links and reifications for standoff resource references when text values are updated. To do this, it keeps track of the number of text values in each resource that contain at least one standoff reference to a given target resource. It stores this number as the reference count of the `LinkValue` (see *LinkValue*) describing the direct link. Each time this number changes, it makes a new version of the `LinkValue`, with an updated reference count. When the reference count reaches zero, it removes the direct link and makes a new version of the `LinkValue`, marked with `kb:isDeleted`.

For example, if `data:R1` is a resource with a text value in which the resource `data:R2` is referenced, the repository could contain the following triples:

```
data:R1 ex:hasComment data:V1 .

data:V1 rdf:type kb:TextValue ;
    kb:valueHasString "This link is internal." ;
    kb:valueHasStandoff data:SO1 .

data:SO1 rdf:type kb:StandoffLinkTag ;
    kb:standoffTagHasStart: 5 ;
    kb:standoffTagHasEnd: 9 ;
    kb:standoffTagHasLink data:R2 .

data:R1 kb:hasStandoffLinkTo data:R2 .
data:R1 kb:hasStandoffLinkToValue data:LV1 .

data:LV1 rdf:type kb:LinkValue ;
    rdf:subject data:R1 ;
    rdf:predicate kb:hasStandoffLinkTo ;
    rdf:object data:R2 ;
    kb:valueHasRefCount 1 .
```

The result can be visualized like this:



Link values created automatically for resource references in standoff are visible to all users, and the creator of these link values is always `kb:SystemUser` (see *Users and Groups*). The Knora API server allows a user to see a standoff link if the user has permission to see the source and target resources.

Internal Links in a TextValue

Internal links in a `TextValue` can be using the data type standoff class `StandoffInternalReferenceTag` or a subclass of it. It has the following property:

`standoffTagHasInternalReference` (1) Points to a `StandoffTag` that belongs to the same `TextValue`. It has an `objectClassConstraint` of `StandoffTag`.

For links to a `kb:Resource`, see *StandoffLinkTag*.

Mapping to Create Standoff From XML

A mapping allows for the conversion of an XML document to RDF-standoff and back. A mapping defines one-to-one relations between XML elements (with or without a class) and attributes and standoff classes and properties (see *XML to Standoff Mapping*).

A mapping is represented by a `kb:XMLToStandoffMapping` which contains one or more `kb:MappingElement`. A `kb:MappingElement` maps an XML element (including attributes) to a standoff class and standoff properties. It has the following properties:

`mappingHasXMLTagname` (1) The name of the XML element that is mapped to a standoff class.

`mappingHasXMLNamespace` (1) The XML namespace of the XML element that is mapped to a standoff class. If no namespace is given, `noNamespace` is used.

`mappingHasXMLClass` (1) The name of the class of the XML element. If it has no class, `noClass` is used.

`mappingHasStandoffClass` (1) The standoff class the XML element is mapped to.

`mappingHasXMLAttribute` (0-n) Maps XML attributes to standoff properties using `MappingXMLAttribute`. See below.

`mappingHasStandoffDataTypeClass` (0-1) Indicates the standoff data type class of the standoff class the XML element is mapped to.

mappingElementRequiresSeparator (1) Indicates if there should be an invisible word separator inserted after the XML element in the RDF-standoff representation. Once the markup is stripped, text segments that belonged to different elements may be concatenated.

A `MappingXMLAttribute` has the following properties:

mappingHasXMLAttributename The name of the XML attribute that is mapped to a standoff property.

mappingHasXMLNamespace The namespace of the XML attribute that is mapped to a standoff property. If no namespace is given, `noNamespace` is used.

mappingHasStandoffProperty The standoff property the XML attribute is mapped to.

Knora includes a standard mapping used by the SALSAH GUI. It has the IRI `http://data.knora.org/projects/standoff/mappings/StandardMapping` and defines mappings for a few elements used to write texts with simple markup (see *XML to Standoff Mapping*).

Standoff in Digital Editions

Knora's standoff is designed to make it possible to convert XML documents to standoff and back. One application for this feature is an editing workflow in which an editor works in an XML editor, and the resulting XML documents are converted to standoff and stored in Knora, where they can be searched and annotated.

If an editor wants to correct text that has been imported from XML into standoff, the text can be exported as XML, edited, and imported again. To preserve annotations on standoff tags across edits, each tag can automatically be given a UUID. In a future version of the Knora base ontology, it will be possible to create annotations that point to UUIDs rather than to IRIs. When a text is exported to XML, the UUIDs can be included in the XML. When the edited XML is imported again, it can be converted to new standoff tags with the same UUIDs. Annotations that applied to standoff tags in the previous version of the text will therefore also apply to equivalent tags in the new version.

When text is converted from XML into standoff, tags are also given indexes, which are numbered from 0 within the context of a particular text. This makes it possible to order tags that share the same position, and to preserve the hierarchy of the original XML document. An ordinary, hierarchical XML tag is converted to a standoff tag that has one index, as well as the index of its parent tag, if any. The Knora base ontology also supports non-hierarchical markup such as CLIX, which enables overlapping markup to be represented in XML. When non-hierarchical markup is converted to standoff, both the start position and the end position of the standoff tag have indexes and parent indexes.

To support these features, a standoff tag can have these additional properties:

standoffTagHasStartIndex (0-1) The index of the start position.

standoffTagHasEndIndex (0-1) The index of the end position, if this is a non-hierarchical tag.

standoffTagHasStartParent (0-1) The IRI of the tag, if any, that contains the start position.

standoffTagHasEndParent (0-1) The IRI of the tag, if any, that contains the end position, if this is a non-hierarchical tag.

standoffTagHasUUID (0-1) A UUID that can be used to annotate a standoff tag that may be present in different versions of a text, or in different layers of a text (such as a diplomatic transcription and an edited critical text).

Querying Standoff in SPARQL

A future version of the Knora API server will provide an API for querying standoff markup. In the meantime, it is possible to query it directly in SPARQL. For example, here is a SPARQL query (using RDFS inference) that finds all the text values texts that have a standoff date tag referring to Christmas Eve 2016, contained in a `StandoffItalicTag`:

```
PREFIX knora-base: <http://www.knora.org/ontology/knora-base#>
PREFIX standoff: <http://www.knora.org/ontology/standoff#>

select * where {
    ?standoffTag a knora-base:StandoffDateTag  .

    ?standoffTag knora-base:valueHasStartJDN ?dateStart .
    ?standoffTag knora-base:valueHasEndJDN ?dateEnd .

    FILTER (2457747  <= ?dateEnd && 2457747  >= ?dateStart)

    ?standoffTag knora-base:standoffTagHasStartParent ?parent .
    ?parent a standoff:StandoffItalicTag .

    ?textValue knora-base:valueHasStandoff ?standoffTag .
    ?textValue knora-base:valueHasString ?string .

    ?standoffTag knora-base:standoffTagHasStart ?startPos .
    ?standoffTag knora-base:standoffTagHasEnd ?endPos .
}
```

Authorization

Users and Groups

Each Knora user is represented by an object belonging to the class `kb:User`, which is a subclass of `foaf:Person`, and has the following properties:

`userid` (1) A unique identifier that the user must provide when logging in.

`password` (1) A cryptographic hash of the user's password.

`email` (0-n) Email addresses belonging to the user.

`isInProject` (0-n) Projects that the user is a member of.

`isInGroup` (0-n) Project-specific groups that the user is a member of.

`foaf:familyName` (1) The user's family name.

`foaf:givenName` (1) The user's given name.

Knora's concept of access control is that an object (a resource or value) can grant permissions to groups of users (but not to individual users). There are four built-in groups:

`UnknownUser` Any user who has not logged into the Knora API server is automatically assigned to this group.

`KnownUser` Any user who has logged into the Knora API server is automatically assigned to this group.

`ProjectMember` When checking a user's permissions on an object, the user is automatically assigned to this group if she is a member of the project that the object belongs to.

`Creator` When checking a user's permissions on an object, the user is automatically assigned to this group if he is the creator of the object.

A project-specific ontology can define additional groups, which must belong to the OWL class `kb:UserGroup`.

There is one built-in `SystemUser`, which is the creator of link values created automatically for resource references in standoff markup (see *StandoffLinkTag*).

Permissions

An object can grant the following permissions, which are stored in a compact format in a single string. This string is the object of the predicate `kb:hasPermissions`, which is required on every `kb:Resource` and

`kb:Value`.

1. **Restricted view permission (RV)** Allows a restricted view of the object, e.g. a view of an image with a watermark.
2. **View permission (V)** Allows an unrestricted view of the object. Having view permission on a resource only affects the user's ability to view information about the resource other than its values. To view a value, she must have view permission on the value itself.
3. **Modify permission (M)** For values, this permission allows a new version of a value to be created. For resources, this allows the user to create a new value (as opposed to a new version of an existing value), or to change information about the resource other than its values. When he wants to make a new version of a value, his permissions on the containing resource are not relevant. However, when he wants to change the target of a link, the old link must be deleted and a new one created, so he needs modify permission on the resource.
4. **Delete permission (D)** Allows the item to be marked as deleted.
5. **Change rights permission (CR)** Allows the permissions granted by the object to be changed.

Each permission in the above list implies all lower-numbered permissions. A user's permission level on a particular object is calculated in the following way:

1. Make a list of the groups that the user belongs to, including `Creator` and/or `ProjectMember` if applicable.
2. Make a list of the permissions that she can obtain on the object, by iterating over the permissions that the object grants. For each permission, if she is in the specified group, add the specified permission to the list of permissions she can obtain.
3. From the resulting list, select the highest-level permission.
4. If the result is that she would have no permissions, give her whatever permission `UnknownUser` would have.

To view a link between resources, a user needs permission to view the source and target resources. He also needs permission to view the `LinkValue` representing the link, unless the link property is `hasStandoffLinkTo` (see *StandoffLinkTag*).

The format of the object of `kb:hasPermissions` is as follows:

- Each permission is represented by the one-letter or two-letter abbreviation given above.
- Each permission abbreviation is followed by a space, then a comma-separated list of groups that the permission is granted to.
- The IRIs of built-in groups are shortened using the `knora-base` prefix.
- Multiple permissions are separated by a vertical bar (`|`).

For example, if an object grants view permission to unknown and known users, and modify permission to project members, the resulting permission literal would be:

```
V knora-base:UnknownUser,knora-base:KnownUser|M knora-base:ProjectMember
```

Consistency Checking

Knora tries to enforce repository consistency by checking constraints that are specified in the Knora base ontology and in project-specific ontologies. Three types of consistency rules are enforced:

- Cardinalities in OWL class definitions must be satisfied.
- Constraints on the types of the subjects and objects of OWL object properties must be satisfied.
- A datatype property may not have an empty string as an object.

The implementation of consistency checking is partly triplestore-dependent; Knora may be able to provide stricter checks with some triplestores than with others.

OWL Cardinalities

As noted in *Resources*, each subclass of `Resource` must use OWL cardinality restrictions to specify the properties it can have. More specifically, a resource is allowed to have a property that is a subproperty of `kb:hasValue` or `kb:hasLinkTo` only if the resource's class has some cardinality for that property. Similarly, a value is allowed to have a subproperty of `kb:valueHas` only if the value's class has some cardinality for that property.

Knora supports, and attempts to enforce, the following cardinality constraints:

`owl:cardinality 1` A resource of this class must have exactly one instance of the specified property.

`owl:minCardinality 1` A resource of this class must have at least one instance of the specified property.

`owl:maxCardinality 1` A resource of this class may have zero or one instance of the specified property.

`owl:minCardinality 0` A resource of this class may have zero or more instances of the specified property.

Knora requires cardinalities to be defined using blank nodes, as in the following example from `knora-base`:

```
:Representation rdf:type owl:Class ;
    rdfs:subClassOf :Resource ,
        [ rdf:type owl:Restriction ;
          owl:onProperty :hasFileValue ;
          owl:minCardinality "1"^^xsd:nonNegativeInteger ] .

:StillImageRepresentation rdf:type owl:Class ;
    rdfs:subClassOf :Representation ,
        [ rdf:type owl:Restriction ;
          owl:onProperty :hasStillImageFileValue ;
          owl:minCardinality "1"^^xsd:nonNegativeInteger ] .
```

A resource class inherits cardinalities from its superclasses. This follows from the rules of RDFS inference. Also, in Knora, cardinalities in the subclass can override cardinalities that would otherwise be inherited from the superclass. Specifically, if a superclass has a cardinality on a property P, and a subclass has a cardinality on a subproperty of P, the subclass's cardinality overrides the superclass's cardinality. In the example above, `hasStillImageFileValue` is a subproperty of `hasFileValue`. Therefore, the cardinality on `hasStillImageFileValue` overrides (i.e. replaces) the one on `hasFileValue`.

Note that, unlike cardinalities, predicates of properties are not inherited. If `:foo rdfs:subPropertyOf :bar`, this does not mean that `:foo` inherits anything from `:bar`. Any predicates of `:foo` that are also needed by `:bar` must be defined explicitly on `:bar`. This design decision was made because property predicate inheritance is not provided by RDFS inference, and would make it more difficult to check the correctness of ontologies, while providing little practical benefit.

For more information about OWL cardinalities, see the OWL 2 Primer.

Constraints on the Types of Property Subjects and Objects

When a project-specific ontology defines a property, it must indicate the types that are allowed as objects (and, if possible, as subjects) of the property. This is done using the following Knora-specific properties:

`subjectClassConstraint` Specifies the class that subjects of the property must belong to. This constraint is recommended but not required. Knora will attempt to enforce this constraint.

`objectClassConstraint` If the property is an object property, specifies the class that objects of the property must belong to. Every subproperty of `kb:hasValue` or a `kb:hasLinkTo` (i.e. every property of a resource that points to a `kb:Value` or to another resource) is required to have this constraint, because the Knora API server relies on it to know what type of object to expect for the property. Knora will attempt to enforce this constraint.

`objectDatatypeConstraint` If the property is a datatype property, specifies the type of literals that can be objects of the property. Knora will not attempt to enforce this constraint, but it is useful for documentation purposes.

Consistency Constraint Example

A project-specific ontology could define consistency constraints as in this simplified example:

```
:book rdf:type owl:Class ;
    rdfs:subClassOf knora-base:Resource ,
        [ rdf:type owl:Restriction ;
          owl:onProperty :hasTitle ;
          owl:cardinality "1"^^xsd:nonNegativeInteger ] ,
        [ rdf:type owl:Restriction ;
          owl:onProperty :hasAuthor ;
          owl:minCardinality "0"^^xsd:nonNegativeInteger ] .

:hasTitle rdf:type owl:ObjectProperty ;
    knora-base:subjectClassConstraint :book ;
    knora-base:objectClassConstraint knora-base:TextValue .

:hasAuthor rdf:type owl:ObjectProperty ;
    knora-base:subjectClassConstraint :book ;
    knora-base:objectClassConstraint knora-base:TextValue .
```

Open Questions

Extending Existing Resource Definitions

How should extensions of existing resources be handled? Project B extends a resource defined in the project A ontology, by adding new properties/values which are interesting for project B.

Notes

References

CHAPTER 4

The Knora API Server

The Knora API server implements Knora's HTTP-based API, and manages data stored in an RDF triplestore and in files. It is designed to work with any standards-compliant RDF triplestore, and is configured to work out of the box with Ontotext GraphDB and Apache Jena.

Deploying the Knora API Server

Getting Started with the Knora API Server

Choosing and Setting Up a Triplestore

The Knora API server requires a standards-compliant RDF triplestore. A number of triplestore implementations are available, including free software as well as proprietary options. The Knora API server is tested and configured to work out of the box with the following triplestores:

- Ontotext GraphDB, a high-performance, proprietary triplestore. The Knora API server is tested with GraphDB Standard Edition and GraphDB Free (which is proprietary but available free of charge).
- Apache Jena, which is free software. Knora comes bundled with Jena and with its standalone SPARQL server, Fuseki.

See the chapters on *Starting Fuseki 3* and *Starting GraphDB-SE* for more details.

Creating Repositories and Loading Test Data

To create a test repository called `knora-test` and load test data into it, go to `webapi/scripts` and run the script for the triplestore you have chosen.

- For Fuseki, run `fuseki-load-test-data.sh`.
- **For GraphDB:**
  - If you are running GraphDB directly from its installation directory (using its `graphdb` script), run `graphdb-se-local-init-knora-test.sh`.
  - If you are running GraphDB from a Docker image, run `graphdb-se-docker-init-knora-test.sh`.

You can create your own scripts based on these scripts, to create new repositories and optionally to load existing Knora-compliant RDF data into them.

If you are using GraphDB, you must create your repository using a repository configuration file that specifies the file `KnoraRules.pie` as its `owlim:ruleset`. This enables RDFS inference and Knora-specific consistency rules. When using GraphDB, the Knora API server uses RDFS inference to improve query performance. The Knora-specific consistency rules help ensure that your data is internally consistent and conforms to the Knora ontologies.

When testing with GraphDB, you may sometimes get an error when loading the test data that says that there are multiple IDs for the same repository `knora-test`. In that case, something went wrong when dropping and recreating the repository. You can solve this by deleting the repository manually and starting over. **Make sure you don't delete important data.** To delete the repository, stop GraphDB, delete the `data` directory in your GraphDB installation, and restart GraphDB.

Creating a Test Installation

TODO: write subsections like this:

- Download the Knora API Server and Sipi from GitHub
- Configure
- Run

Transforming Data When Ontologies Change

When there is a change in Knora's ontologies or in a project-specific ontology, it may be necessary to update existing data to conform to the new ontology. This can be done directly in SPARQL, but for simple transformations, Knora includes a command-line program that works on RDF data files in Turtle format. You can run it from SBT:

```
> run-main org.knora.webapi.util.TransformData --help
[info] Running org.knora.webapi.util.TransformData --help
[info]
[info] Updates the structure of Knora repository data to accommodate changes in Knora.
[info]
[info] Usage: org.knora.webapi.util.TransformData -t [deleted|permissions|strings|standoff|all] in
[info]
[info]   -t, --transform  <arg>   Selects a transformation. Available transformations:
[info]                            'deleted' (adds missing 'knora-base:isDeleted'
[info]                            statements), 'permissions' (combines old-style
[info]                            multiple permission statements into single permission
[info]                            statements), 'strings' (adds missing valueHasString),
[info]                            'standoff' (transforms old-style standoff into
[info]                            new-style standoff), 'creator' (transforms existing
[info]                            'knora-base:Owner' group inside permissions to
[info]                            'knora-base:Creator'), 'owner' (gives
[info]                            'knora-base:Creator' CR permissions to correspond to
[info]                            the previous behaviour for owners - use with care as
[info]                            it will add permissions that where not there before),
[info]                            'all' (all of the above minus 'owner')
[info]       --help               Show help message
[info]
[info]  trailing arguments:
[info]   input (required)    Input Turtle file
[info]   output (required)   Output Turtle file
```

The currently available transformations are:

deleted Adds `knora-base:isDeleted false` to resources and values that don't have a `knora-base:isDeleted` predicate.

permissions Combines old-style permission statements (`hasViewPermission`, `hasModifyPermission`, etc.) into one `hasPermissions` statement per resource or value, as described in the section **Permissions** in *The Knora Ontologies*.

strings Adds missing `valueHasString` statements to Knora value objects.

standoff Transforms old-style standoff markup (containing tag names as strings) to new-style standoff markup (using different OWL class names for different tags).

creator Transforms existing `knora-base:Owner` group inside permissions to `knora-base:Creator`.

owner Gives `knora-base:Creator` **CR permissions** to correspond to the previous behaviour for owners. Use with care as it will add permissions that where not there before.

all Runs all of the above transformations.

Transformations that are not needed have no effect, so it is safe to use `-t all`.

The program uses the Turtle parsing and formatting library from RDF4J. Additional transformations can be implemented as subclasses of `org.eclipse.rdf4j.rio.RDFHandler`.

Using HTTPS in the Knora API Server

Enabling HTTPS

The Knora API server can be configured to accept requests over HTTP, HTTPS, or both. In the `app.http` section of `application.conf`, the relevant configuration options look like this by default:

```
https {
   keystore = "https/localhost.jks"
   keystore-password = "test keystore password"
}

knora-api {
   host = "localhost"
   http-port = 3333
   https-port = 3334
   use-http = true
   use-https = false
}
```

On a production system, you should enable HTTPS and disable HTTP, to protect passwords and other private data from being intercepted in transit.

To enable HTTPS, you will need an SSL/TLS certificate, signed by a certificate authority (CA) and stored in a Java KeyStore (JKS) file. For information on storing a certificate in a JKS file, see the Oracle keytool documentation. Once you have a JKS, you can configure the Knora API Server to load it by changing the `https` configuration in `application.conf`. You can then set `use-https` to `true`. The HTTP and HTTPS ports can be any ports you choose.

Creating a Self-Signed Certificate for Testing

For testing purposes, you can create your own CA and self-signed certificate. Open a terminal in the Knora source directory `webapi/src/main/resources/https`, and type:

```
$ ./generate-test-ca.sh
$ ./generate-test-cert.sh
```

This will create a CA, then create an SSL/TLS certificate signed by the CA, in the file `localhost.jks`, matching the `https` configuration in `application.conf` shown above. You can then set `use-https` to `true`.

Configuring A Web Browser to Accept a Self-Signed Certificate

If you are using a self-signed certificate, you must configure your web browser to accept it.

Chrome To configure the Chrome browser to accept self-signed certificates for `localhost`, type this in the location bar:

```
chrome://flags/#allow-insecure-localhost
```

Click on **Enable** to enable the option, then restart the browser.

Firefox Make a request to the API server over HTTPS by typing a Knora API URL into the browser's location bar, e.g.:

```
https://localhost:3334/v1/resources/http%3A%2F%2Fdata.knora.org%2Fc5058f3a
```

Firefox will say that your connection is not secure. Click **Advanced**, then **Add Exception**, then **Confirm Security Exception**.

Configuring the SALSAH GUI to Connect to the Knora API Server over HTTPS

In the file `salsah/src/public/js/00_init_javascript.js`, change the value of the variable `API_URL` to specify `https` instead of `http`, along with the HTTPS port configured in the Knora API server's `application.conf`. For example:

```
var API_URL = 'https://localhost:3334';
```

Note that this only affects the communication between the SALSAH GUI and the Knora API server. On a production system, you should also use a web server that serves the SALSAH GUI itself over HTTPS, to protect private data from being intercepted in transit. You must then set `http.salsah.base-url` in `application.conf` to the base HTTPS URL of the SALSAH GUI.

Running the Knora API Server on a Production System

This section describes possible ways of running the Knora API server in an production environment. The description should only be taken as a first short introduction to this topic. Further reading of the referenced materials is advised.

Note: Our platform of choice is Linux CentOS 7 and is thus assumed in the description. The general idea should be usable on all platforms with small changes.

To run the Knora API server, we have two main components. First, the zipped distribution of the server and second a supported triplestore.

Creating and running the distribution package

Inside the *knora/webapi* folder run the following *sbt* commands:

```
` $ sbt packageBin `
```

This will create a *zip* file inside the *knora/webapi/target/universal* folder. To run the Knora API Server, unzip this package, and execute the *webapi* script inside the *bin* folder.

Alternatively, the command *sbt stage* will create a folder with the same content as before, but will skip the zipping step.

Running a supported triplestore

See the chapters on *Starting Fuseki 3* and *Starting GraphDB-SE* on how to start a supported triplestore.

Knora API Server Design Documentation

Knora API Server Design Overview

Introduction

The Knora API server implements Knora's web-based Application Programming Interface (API). It is responsible for receiving HTTP requests from clients (which may be web browsers or other software), performing authentication and authorisation, querying and updating the RDF triplestore, transforming the results of SPARQL queries into Knora API responses, and returning these responses to the client. It is written in Scala, using the Akka framework for message-based concurrency. It is designed to work with any standards-compliant triplestore. It can communicate with triplestores either via the SPARQL 1.1 Protocol or by embedding the triplestore in the API server as a library.

Design Diagram

Modules

HTTP Module

- org.knora.webapi.http
- org.knora.webapi.routes

Responders Module

- org.knora.webapi.responders

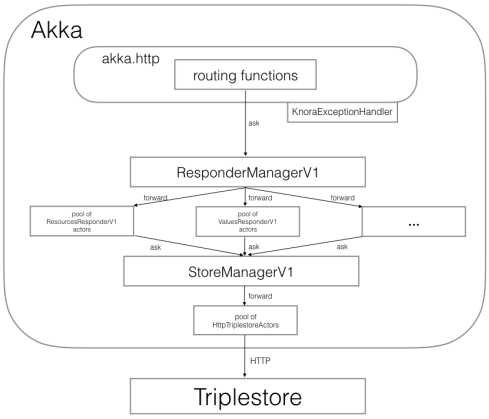


Fig. 4.1: A high-level diagram of the Knora API server.

Store Module

- org.knora.store

Shared Between Modules

- org.knora.webapi
- org.knora.webapi.util
- org.knora.webapi.messages

Actor Supervision and Creation

At system start, the supervisor actors are created in `KnoraService.scala`:

```
val responderManager = system.actorOf(Props(new ResponderManagerV1 with LiveActorMaker), name = "
val storeManager = system.actorOf(Props(new StoreManager with LiveActorMaker), name = "storeManage
```

Each supervisor creates and maintains a pool of workers, with an Akka router that dispatches messages to the workers according to some strategy. For now, all the pools use the 'round-robin' strategy. The pools and routers are configured in `application.conf`:

```
actor {
    deployment {
        user/storeManager/triplestoreRouter {
            router = round-robin-pool
            nr-of-instances = 50
        }

        user/responderManager/resourcesRouter {
            router = round-robin-pool
            nr-of-instances = 20
        }

        user/responderManager/valuesRouter {
            router = round-robin-pool
            nr-of-instances = 20
        }

        user/responderManager/representationsRouter {
            router = round-robin-pool
```

```
            nr-of-instances = 20
        }

        user/responderManager/usersRouter {
            router = round-robin-pool
            nr-of-instances = 20
        }
    }
}
```

Additionally, in `KnoraService` also the `akka-http` layer is started:

```
val host = settings.httpInterface
val port = settings.httpPort
val bindingFuture: Future[ServerBinding] = Http().bindAndHandle(Route.handlerFlow(apiRoutes), host
println(s"Knora API Server started. You can access it on http://${settings.httpInterface}:${settin

bindingFuture.onFailure {
    case ex: Exception =>
    log.error(ex, s"Failed to bind to ${settings.httpInterface}:${settings.httpPort}!")
}
```

Concurrency

Except for a bit of caching, the Knora API server is written in a purely functional style and has no mutable state, shared or otherwise, not even within actors. This makes it easier to reason about concurrency, and eliminates an important potential source of bugs (see Out of the Tar Pit).

There is a pool of HTTP workers that handle HTTP requests concurrently using the spray routes in the `routing` package. Each spray route constructs a request message and sends it to `ResponderManagerV1`, which forwards it to a worker actor in one of its pools. So the size of the HTTP worker pool sets the maximum number of concurrent HTTP requests, and the size of the worker pool for each responder sets the maximum number of concurrent messages for that responder. Whenever a responder needs to do a SPARQL query, it sends a message to the store manager, which forwards it to a triplestore actor. The size of the pool(s) of triplestore actors sets the maximum number of concurrent SPARQL queries.

The routes and actors in the Knora API server uses Akka's `ask` pattern, rather than the `tell` pattern, to send messages and receive responses, because this simplifies the code considerably (using `tell` would require actors to maintain complex mutable state), with no apparent reduction in performance.

To manage asynchronous communication between actors, the Knora API server uses Scala's `Future` monad extensively. See *Futures with Akka* for details.

We use Akka's asynchronous logging interface (see Akka Logging).

What the Responders Do

In the Knora API server, a 'responder' is an actor that receives a request message (a Scala case class) in the `ask` pattern, gets data from the triplestore, and turns that data into a reply message (another case class). These reply messages are are defined in the `schemas` package. A responder can produce a reply representing a complete API response, or part of a response that will be used by another responder. If it's a complete API response, it will extend `KnoraJsonResponse`, which can be converted directly into JSON by calling its `toJsValue` method (see the section on JSON below).

All messages to responders go through the responder supervisor actor (`ResponderManagerV1`).

Store Module (org.knora.webapi.store package)

The Store module is used for accessing the triplestore and other external storage providers.

All access to the Store module goes through the `StoreManager` supervisor actor. The `StoreManager` creates pools of actors, such as `HttpTriplestoreActor`, that interface with the storage providers.

The contents of the `store` package are not used directly by other packages, which interact with the `store` package only by sending messages to `StoreManager`.

Generation and parsing of SPARQL are handled by this module.

See *Store Module* for a deeper discussion.

Triplestore Access

SPARQL queries are generated from templates, using the Twirl template engine. For example, if we're querying a resource, the template will contain a placeholder for the resource's IRI. The templates can be found under `src/main/twirl/queries/sparql/v1`. So far we have been able to avoid generating different SPARQL for different triplestores.

The `org.knora.webapi.store` package contains actors for communicating with triplestores in different ways: a triplestore can be accessed over HTTP via the SPARQL 1.1 Protocol, or it can be embedded in the Knora API server. However, a responder is not expected to know which triplestore is being used or how the triplestore is accessed. To perform a SPARQL query, a responder sends a message to the `storeManager` actor, like this:

```
private val storeManager = context.actorSelection("/user/storeManager")

// ...

private def getSomeValue(resourceIri: IRI): Future[String] = {
    for {
        sparqlQuery <- Future(queries.sparql.v1.txt.someTemplate(resourceIri).toString())
        queryResponse <- (storeManager ? SparqlSelectRequest(sparqlQuery)).mapTo[SparqlSelectResp
        someValue = // get some value from the query response
    } yield someValue
}
```

Error Handling

The error-handling design has these aims:

1. Simplify the error-handling code in actors as much as possible.
2. Produce error messages that clearly indicate the context in which the error occurred (i.e. what the application was trying to do).
3. Ensure that clients receive an appropriate error message when an error occurs.
4. Ensure that `ask` requests are properly terminated with an `akka.actor.Status.Failure` message in the event of an error, without which they will simply time out (see Send-And-Receive-Future).
5. When a actor encounters an error that isn't the client's fault (e.g. a triplestore failure), log it, but don't do this with errors caused by bad input.
6. When logging errors, include the full JVM stack trace.

The design does not yet include, but could easily accommodate, translations of error messages into different languages.

A hierarchy of exception classes is defined in `Exceptions.scala`, representing different sorts of errors that could occur. The hierarchy has two main branches:

- `RequestRejectedException`, an abstract class for errors that are the client's fault. These errors are not logged.
- `InternalServerException`, an abstract class for errors that are not the client's fault. These errors are logged.

Exception classes in this hierarchy can be defined to include a wrapped `cause` exception. When an exception is logged, its stack trace will be logged along with the stack trace of its `cause`. It is therefore recommended that low-level code should catch low-level exceptions, and wrap them in one of our higher-level exceptions, in order to clarify the context in which the error occurred.

To simplify error-handling in responders, a utility method called `future2Message` is provided in `ActorUtils`. It is intended to be used in an actor's `receive` method to respond to messages in the `ask` pattern. If the responder's computation is successful, it is sent to the requesting actor as a response to the `ask`. If the computation fails, the exception representing the failure is wrapped in a `Status.Failure`, which is sent as a response to the `ask`. If the error is a subclass of `RequestRejectedException`, only the sender is notified of the error; otherwise, the error is also logged and rethrown (so that the `KnoraExceptionHandler` can handle the exception).

In many cases, we transform data from the triplestore into a `Map` object. To simplify checking for required values in these collections, the class `ErrorHandlingMap` is provided. You can wrap any `Map` in an `ErrorHandlingMap`. You must provide a function that will generate an error message when a required value is missing, and optionally a function that throws a particular exception. Rows of SPARQL query results are already returned in `ErrorHandlingMap` objects.

If you want to add a new exception class, see the comments in `Exceptions.scala` for instructions.

Transformation of Exception to Client Responses

The `org.knora.webapi.KnoraExceptionHandler` is brought implicitly into scope of `akka-http`, and by doing so registered and used to handle the transformation of all `KnoraExceptions` into `HttpResponses`. This handler handles only exceptions thrown inside the route and not the actors. However, the design of reply message passing from actors (by using `future2Message`), makes sure that any exceptions thrown inside actors, will reach the route, where they will be handled.

See also *Futures with Akka*.

API Routing

The API routes in the `routing` package are defined using the DSL provided by the akka-http library. A routing function has to do the following:

1. Authenticate the client.
2. Figure out what the client is asking for.
3. Construct an appropriate request message and send it to `ResponderManagerV1`, using the `ask` pattern.
4. Return a result to the client.

To simplify the coding of routing functions, they are contained in objects that extend `org.knora.webapi.routing.Authenticator`. Each routing function performs the following operations:

1. `Authenticator.getUserProfileV1` is called to authenticate the user.
2. The request parameters are interpreted and validated, and a request message is constructed to send to the responder. If the request is invalid, `BadRequestException` is thrown. If the request message is requesting an update operation, it must include a UUID generated by `UUID.randomUUID`, so the responder can obtain a write lock on the resource being updated.

The routing function then passes the message to `org.knora.webapi.routing.RouteUtils.runJsonRoute()`, which takes care of sending the message to `ResponderManagerV1` and returning a response to the client. Any exceptions thrown befor calling `org.knora.webapi.routing.RouteUtils.runJsonRoute()` are handled by the `KnoraExceptionHandler`.

See *How to Add an API Route* for an example.

JSON

The Knora API server parses and generate JSON using the spray-json library.

The triplestore returns results in JSON, and these are parsed into `SparqlSelectResponse` objects in the `store` package (by `SparqlUtils`, which can be used by any actor in that package). A `SparqlSelectResponse` has a structure that's very close to the JSON returned by a triplestore via the SPARQL 1.1 Protocol: it contains a header (listing the variables that were used in the query) and a body (containing rows of query results). Each row of query results is represented by a `VariableResultsRow`, which contains a `Map[String, String]` of variable names to values.

The `Jsonable` trait marks classes that can convert themselves into spray-json AST objects when you call their `toJsValue` method; it returns a `JsValue` object, which can then be converted to text by calling its `prettyPrint` or `compactPrint` methods. Case classes representing complete API responses extend the `KnoraResponseV1` trait, which extends `Jsonable`. Case classes representing Knora values extend the `ApiValueV1` trait, which also extends `Jsonable`. To make the responders reusable, the JSON for API responses is generated only at the last moment, by the `RouteUtils.runJsonRoute()` function.

Futures with Akka

Introduction

Scala's documentation on futures introduces them in this way:

> Futures provide a nice way to reason about performing many operations in parallel – in an efficient and non-blocking way. The idea is simple, a Future is a sort of a placeholder object that you can create for a result that does not yet exist. Generally, the result of the Future is computed concurrently and can be later collected. Composing concurrent tasks in this way tends to result in faster, asynchronous, non-blocking parallel code.

The rest of that page is well worth reading to get an overview of how futures work and what you can do with them.

In Akka, one of the standard patterns for communication between actors is the ask pattern, in which you send a message to an actor and you expect a reply. When you call the `ask` function (which can be written as a question mark, `?`, which acts as an infix operator), it immediately returns a `Future`, which will complete when the reply is sent. As the Akka documentation explains in Use with Actors, it is possible to block the calling thread until the future completes, using `Await.result`. However, they say: 'Blocking is discouraged though as it will cause performance problems.' In particular, by not blocking, you can do several `ask` requests in parallel.

One way to avoid blocking is to register a callback on the future, which will be called when it completes (perhaps by another thread), like this:

```
future.onComplete {
    case Success(result) => println(result)
    case Failure(ex) => ex.printStackTrace()
}
```

But this won't work if you're writing a method that needs return a value based on the result of a future. In this case, you can register a callback that transforms the result of a future into another future:

```
val newFuture = future.map(x => x + 1)
```

However, registering callbacks explicitly gets cumbersome when you need to work with several futures together. In this case, the most convenient alternative to blocking is to use `Future` as a monad. The links above explain what this means in detail, but the basic idea is that a special syntax, called a `for`-comprehension, allows you to write code that uses futures as if they were complete, without blocking. In reality, a `for`-comprehension is syntactic sugar for calling methods like `map`, but it's much easier to write and to read. You can do things like this:

```
val fooFuture = (fooActor ? GetFoo("foo")).mapTo[Foo]
val barFuture = (barActor ? GetBar("bar")).mapTo[Bar]

val totalFuture = for {
    foo: Foo <- fooFuture
    bar: Bar <- barFuture

    total = foo.getCount + bar.getCount
} yield total
```

Here the messages to `fooActor` and `barActor` are sent and processed in parallel, but you're guaranteed that `total` won't be calculated until the values it needs are available. Note that if you construct `fooFuture` and `barFuture` inside the `for` comprehension, they won't be run in parallel (see Scala for-comprehension with concurrently running futures).

With one line of code, you can even make a list of messages to be sent to actors, send them all in parallel, get back a list of futures, and convert it to a single future which will complete when all the results are available; see `org.knora.webapi.util.ActorUtils.parallelAsk`.

Handling Errors with Futures

The constructors and methods of `Future` (like those of `Try`) catch exceptions, which cause the future to fail. This very useful property of futures means that you usually don't need `try-catch` blocks when using the `Future` monad (although it is sometimes helpful to include them, in order to catch low-level exceptions and wrap them in higher-level ones). Any exception thrown in code that's being run asynchronously by `Future` (including in the `yield` expression of a `for` comprehension) will be caught, and the result will be a `Future` containing a `Failure`. Also, in the previous example, if `fooActor` or `barActor` returns a `Status.Failure` message, the `for`-comprehension will also yield a failed future.

However, you need to be careful with *the first line* of the `for`-comprehension. For example, this code doesn't handle exceptions correctly:

```
private def doFooQuery(iri: IRI): Future[String] = {
    for {
        queryResponse <- (storeManager ? SparqlSelectRequest(queries.sparql.v1.txt.getFoo(iri).to
        ...
   } yield ...
}
```

The `getFoo()` method calls a Twirl template function to generate SPARQL. The `?` operator returns a `Future`. However, the template function *is not run asynchronously*, because it is called before the `Future` constructor is called. So if the template function throws an exception, it won't be caught here. Instead, you can do this:

```
private def doFooQuery(iri: IRI): Future[String] = {
    for {
        queryString <- Future(queries.sparql.v1.txt.getFoo(iri).toString())
        queryResponse <- (storeManager ? SparqlSelectRequest(queryString)).mapTo[SparqlSelectResp
        ...
   } yield ...
}
```

Here the `Future` constructor will call the template function asynchronously, and catch any exceptions it throws. This is only necessary if you need to call the template function at the *very beginning* of a `for`-comprehension. In the rest of the `for` comprehension, you'll already implicitly have a `Future` object.

Using `recover` on Futures

By using `recover` on a `Future`, an apt error message can be thrown if the `Future` fails. This is particularly useful when an an error message should be made more clear depending on the context the `Future` is used in.

For example, we are asking the resources responder to query for a certain resource in order to process it in a special way. However, the client does not know that the resources responder is sent a request and in case the resource cannot be found, the message sent back from the resources responder (`NotFoundException`) would not make sense to it. Instead, we would like to handle the message in a way so that it makes sense for the operation the client actually executed. We can do this by calling `recover` on a `Future`.

```
private def mySpecialResourceRequest(iri: IRI, userProfile: UserProfileV1): Future[...] = {

    val resourceRequestFuture = for {
        resResponse: ResourceFullResponseV1 <- (responderManager ? ResourceFullGetRequestV1(iri =
    } yield resResponse

    val resourceRequestFutureRecovered = resourceRequestFuture.recover {
        case notFound: NotFoundException => throw BadRequestException(s"Special resource handling
    }

    for {

        res <- resourceRequestFutureRecovered

        ...

    } yield ...

}
```

Please note that the content of the `Future` has to be accessed using `<-` to make this work correctly. Otherwise the content will never be looked at.

Designing with Futures

In the current design, the Knora API Server almost never blocks to wait for a future to complete. The normal flow of control works like this:

1. Incoming HTTP requests are handled by an actor called `KnoraHttpService`, which delegates them to routing functions (in the `routing` package).
2. For each request, a routing function gets a `spray-http RequestContext`, and calls `RouteUtils.runJsonRoute` to send a message to a supervisor actor to fulfil the request. Having sent the message, the `runJsonRoute` gets a future in return. It does not block to wait for the future to complete, but instead registers a callback to process the result of the future when it becomes available.
3. The supervisor forwards the message to be handled by the next available actor in a pool of responder actors that are able to handle that type of message.
4. The responder's `receive` method receives the message, and calls some private method that produces a reply message inside a future. This usually involves sending messages to other actors using `ask`, getting futures back, and combining them into a single future containing the reply message.
5. The responder passes that future to `ActorUtils.future2Message`, which registers a callback on it. When the future completes (perhaps in another thread), the callback sends the reply message. In the meantime, the responder doesn't block, so it can start handling the next request.
6. When the responder's reply becomes available (causing the future created by `RouteUtils.runJsonRoute` to complete), the callback registered in (2) calls `complete` on the `RequestContext`, which sends an HTTP response to the client.

The basic rule of thumb is this: if you're writing a method in an actor, and anything in the method needs to come from a future (e.g. because you need to use `ask` to get some information from another actor), have the method return a future.

Mixing Futures with non-Futures

If you have a `match ... case` or `if` expression, and one branch obtains some data in a future, but another branch can produce the data immediately, you can wrap the result of the latter branch in a future, so that both branches have the same type:

```
def getTotalOfFooAndBar(howToGetFoo: String): Future[Int] = {
    for {
        foo <- howToGetFoo match {
            case "askForIt" => (fooActor ? GetFoo("foo")).mapTo[Foo]
            case "createIt" => Future(new Foo())
        }

        bar <- (barActor ? GetBar("bar")).mapTo[Bar]

        total = foo.getCount + bar.getCount
    } yield total
}
```

How to Write For-Comprehensions

Here are some basic rules for writing `for`-comprehensions:

1. The first line of a `for`-comprehension has to be a "generator", i.e. it has to use the <− operator. If you want to write an assignment (using =) as the first line, the workaround is to wrap the right-hand side in a monad (like `Future`) and use <− instead.
2. Assignments (using =) are written without `val`.
3. You're not allowed to write statements that throw away their return values, so if you want to call something like `println` that returns `Unit`, you have to assign its return value to _.

The `yield` returns an object of the same type as the generators, which all have to produce the same type (e.g. `Future`).

Execution Contexts

Whenever you use a future, there has to be an implicit 'execution context' in scope. Scala's documentation on futures says, 'you can think of execution contexts as thread pools'.

If you don't have an execution context in scope, you'll get a compile error asking you to include one, and suggesting that you could use `import scala.concurrent.ExecutionContext.Implicits.global`. Don't do this, because the global Scala execution context is not the most efficient option. Instead, you can use the one provided by the Akka `ActorSystem`:

```
implicit val executionContext = system.dispatcher
```

Akka's execution contexts can be configured (see Dispatchers). You can see a Listing of the Reference Configuration.

HTTP Module

The http module holds only a convenience method for adding CORS support to api routes. The CORS implementation uses the akka-http-cors directives implementation.

Responders Module

Version 1.0 Responders

ResponderManagerV1

CkanResponderV1

HierarchicalListsResponderV1

OntologyResponderV1

The ontology responder provides information derived from all the ontologies in the repository, including Knora ontologies as well as project-specific ontologies. Most importantly, it provides information about resource classes and properties. This includes the cardinalities defined on each resource class, and takes into account the rules of cardinality inheritance, as described in the section **OWL Cardinalities** in *The Knora Ontologies*.

For performance reasons, all ontology data is loaded and cached at application startup. Currently, to refresh the cache, you must restart the application. To maximise performance, the SPARQL queries used by the ontology responder are very simple, and the responder calculates class hierarchies and cardinality inheritance in Scala.

ProjectsResponderV1

RepresentationsResponderV1

ResourcesResponderV1

SearchResponderV1

UsersResponderV1

ValuesResponderV1

Shared

- ResponderV1
- ValueUtilV1

Store Module

Overview

The store module houses the different types of data stores supported by the Knora API server. At the moment, only triplestores are supported. The triplestore support is implemented in the `org.knora.webapi.store.triplestore` package.

Lifecycle

At the top level, the store package houses the `StoreManager`-Actor which is started when the Knora API server starts. The `StoreManager` then starts the `TripleStoreManagerActor` which in turn starts the correct actor implementation (e.g., GraphDB, Fuseki, embedded Jena, etc.).

HTTP-based Triplestores

HTTP-based triplestore support is implemented in the `org.knora.webapi.triplestore.http` package.

An HTTP-based triplestore is one that is accessed remotly over the HTTP protocol. We have implemented support for the following triplestores:

- Ontotext GraphDB
- Fuseki 2

GraphDB

Fuseki 2

Embedded Triplestores

Embedded triplestores is implemented in the `org.knora.webapi.triplestore.embedded` package.

An embedded triplestore is one that runs in the same JVM as the Knora API server.

Apache Jena TDB

Note: The support for embedded Jena TDB is currently dropped. The documentation and the code will remain in the repository. You can use it at your own risk.

The support for the embedded Jena-TDB triplestore is implemented in `org.knora.webapi.triplestore.embedded.JenaTDBActor`.

The relevant Jena libraries that are used are the following:

- Jena API - The library used to work programmatically with RDF data
- Jena TDB - Their implementation of a triple store

Concurrency Jena provides concurrency on different levels.

On the Jena TDB level there is the `Dataset` object, representing the triple store. On every access, a transaction (read or write) can be started.

On the Jena API level there is a `Model` object, which is equivalent to an RDF `Graph`. Here we can lock the model, so that MRSW (Multiple Reader Single Writer) access is allowed.

- https://jena.apache.org/documentation/tdb/tdb_transactions.html
- https://jena.apache.org/documentation/notes/concurrency-howto.html

Implementation We employ transactions on the `Dataset` level. This means that every thread that accesses the triplestore, starts a read or write enabled transaction.

The transaction mechanism in TDB is based on write-ahead-logging. All changes made inside a write-transaction are written to journals, then propagated to the main database at a suitable moment. This design allows for read-transactions to proceed without locking or other overhead over the base database.

Transactional TDB supports one active write transaction, and multiple read transactions at the same time. Read-transactions started before a write-transaction commits see the database in a state without any changes visible. Any transaction starting after a write-transaction commits sees the database with the changes visible, whether fully propagates back to the database or not. There can be active read transactions seeing the state of the database before the updates, and read transactions seeing the state of the database after the updates running at the same time.

Configuration In `application.conf` set to use the embedded triplestore:

```
triplestore {
    dbtype = "embedded-jena-tdb"

    embedded-jena-tdb {
        persisted = true // "false" -> memory, "true" -> disk
        loadExistingData = false // "false" -> use data if exists, "false" -> create a fresh store
        storage-path = "_TMP" // ignored if "memory"
    }

    reload-on-start = false // ignored if "memory" as it will always reload

    rdf-data = [
        {
            path = "../knora-ontologies/knora-base.ttl"
            name = "http://www.knora.org/ontology/knora-base"
        }
        {
            path = "../knora-ontologies/knora-dc.ttl"
            name = "http://www.knora.org/ontology/dc"
        }
        {
            path = "../knora-ontologies/salsah-gui.ttl"
            name = "http://www.knora.org/ontology/salsah-gui"
        }
        {
            path = "_test_data/ontologies/incunabula-onto.ttl"
            name = "http://www.knora.org/ontology/incunabula"
        }
        {
            path = "_test_data/demo_data/incunabula-demo-data.ttl"
            name = "http://www.knora.org/data/incunabula"
        }
        {
            path = "_test_data/ontologies/images-onto.ttl"
            name = "http://www.knora.org/ontology/dokubib"
        }
        {
            path = "_test_data/demo_data/images-demo-data.ttl"
            name = "http://www.knora.org/data/dokubib"
        }
    ]
}
```

Here the storage is set to `persistent`, meaning that a Jena TDB store will be created under the defined `tdb-storage-path`. The `reload-on-start` flag, if set to `true` would reload the triplestore with the data referenced in `rdf-data`.

TDB Disk Persisted Store

Note: Make sure to set `reload-on-start` to `true` if run for the first time. This will create a TDB store and load the data.

If only *read access* is performed, then Knora can be run once with reloading enabled. After that, reloading can be turned off, and the persisted TDB store can be reused, as any data found under the `tdb-storage-path` will be reused.

If the TDB storage files get corrupted, then just delete the folder and reload the data anew.

Actor Messages

- `ResetTripleStoreContent(rdfDataObjects: List[RdfDataObject])`
- `ResetTripleStoreContentACK()`

The embedded Jena TDB can receive reset messages, and will ACK when reloading of the data is finished. `RdfDataObject` is a simple case class, containing the path and name (the same as `rdf-data` in the config file)

As an example, to use it inside a test you could write something like:

```
val rdfDataObjects = List (
       RdfDataObject(path = "../knora-ontologies/knora-base.ttl",
                     name = "http://www.knora.org/ontology/knora-base"),
       RdfDataObject(path = "../knora-ontologies/knora-dc.ttl",
                     name = "http://www.knora.org/ontology/dc"),
       RdfDataObject(path = "../knora-ontologies/salsah-gui.ttl",
                     name = "http://www.knora.org/ontology/salsah-gui"),
       RdfDataObject(path = "_test_data/ontologies/incunabula-onto.ttl",
                     name = "http://www.knora.org/ontology/incunabula"),
       RdfDataObject(path = "_test_data/all_data/incunabula-data.ttl",
                     name = "http://www.knora.org/data/incunabula")
)

"Reload data " in {
    storeManager ! ResetTripleStoreContent(rdfDataObjects)
    expectMsg(300.seconds, ResetTripleStoreContentACK())
}
```

Shared Packages

TODO

How to Add an API Route

Write SPARQL templates

Add any SPARQL templates you need to `src/main/twirl/queries/sparql/v1`, using the Twirl template engine.

Write Responder Request and Response Messages

Add a file to the `org.knora.webapi.messages.v1respondermessages` package, containing case classes for your responder's request and response messages. Add a trait that the responder's request messages extend. Each request message type should contain a `UserProfileV1`.

Response message classes that represent a complete API response must extend `KnoraResponseV1`, and must therefore have a `toJsValue` method that converts the response message to a JSON AST using spray-json.

Write a Responder

Write an Akka actor class that extends `ResponderV1`, and add it to the `org.knora.webapi.responders.v1` package.

Give your responder a `receive()` method that handles each of your request message types by generating a `Future` containing a response message, and passing the `Future` to `ActorUtils.futureToMessage()`. See *Futures with Akka* and *Error Handling* for details.

See *Triplestore Access* for details of how to access the triplestore in your responder.

Add an actor pool for your responder to `application.conf`, under `actor.deployment`.

In `ResponderManagerV1`, add a reference to your actor pool. Then add a `case` to the `receive()` method in `ResponderManagerV1`, to match messages that extend your request message trait, and forward them to that pool.

Write a Route

Add an object to the `org.knora.webapi.routing.v1` package for your route. Your object should look something like this:

```
import akka.actor.ActorSystem
import akka.event.LoggingAdapter
import org.knora.webapi.SettingsImpl
import org.knora.webapi.messages.v1respondermessages.SampleGetRequestV1
import org.knora.webapi.routing.RouteUtils
import spray.routing.Directives._
import spray.routing._
import org.knora.webapi.util.StringConversions
import org.knora.webapi.BadRequestException

object SampleRouteV1 extends Authenticator {

    def knoraApiPath(_system: ActorSystem, settings: SettingsImpl, log: LoggingAdapter): Route =
        implicit val system: ActorSystem = _system
        implicit val executionContext = system.dispatcher
        implicit val timeout = settings.defaultTimeout
        val responderManager = system.actorSelection("/user/responderManager")

        path("sample" / Segment) { iri =>
            get { requestContext =>
                val userProfile = getUserProfileV1(requestContext)
                val requestMessage = makeRequestMessage(iri, userProfile)

                RouteUtils.runJsonRoute(
                    requestMessage,
                    requestContext,
                    settings,
                    responderManager,
                    log
                )
            }
```

```
        }
    }

    private def makeRequestMessage(iriStr: String, userProfile: UserProfileV1): SampleGetRequestV
        val iri = StringConversions.toIri(iriStr, () => throw BadRequestException(s"Invalid IRI:
        SampleGetRequestV1(iri, userProfile)
    }
}
```

Finally, add your `knoraApiPath()` function to the `apiRoutes` member variable in `KnoraService`. Any exception thrown inside the route (e.g., input validation, getUserProfile, etc.) will be handled by the `KnoraExceptionHandler`, so that the correct client response (status code, format) will be returned.

Triplestore Updates

Requirements

General

The supported update operations are:

- Create a new resource with its initial values.
- Add a new value.
- Change a value.
- Delete a value (i.e. mark it as deleted).
- Delete a resource (i.e. mark it as deleted).

Users must be able to edit the same data concurrently.

Each update must be atomic and leave the database in a consistent, meaningful state, respecting ontology constraints and permissions.

The application must not use any sort of long-lived locks, because they tend to hinder concurrent edits, and it is difficult to ensure that they are released when they are no longer needed. Instead, if a user requests an update based on outdated information (because another user has just changed something, and the first user has not found out yet), the update must be not performed, and the application must notify the user who requested it, suggesting that the user should check the relevant data and try again if necessary. (We may eventually provide functionality to

help users merge edits in such a situation. The application can also encourage users to coordinate with one another when they are working on the same data, and may eventually provide functionality to facilitate this coordination.)

We can assume that each SPARQL update operation will run in its own database transaction with an isolation level of 'read committed'. This is what GraphDB does when it receives a SPARQL update over HTTP (see GraphDB SE Transactions). We cannot assume that it is possible to run more than one SPARQL update in a single database transaction. (The SPARQL 1.1 Protocol does not provide a way to do this, and currently it can be done only by embedding the triplestore in the application and using a vendor-specific API, but we cannot require this in Knora.)

Permissions

To create a new value (as opposed to a new version of an existing value), the user must have `knora-base:hasModifyPermission` on the containing resource.

To create a new version of an existing value, the user needs only to have `knora-base:hasModifyPermission` on the current version of the value; no permissions on the resource are needed.

Since changing a link requires deleting the old link and creating a new one (as described in *Linking*), a user wishing to change a link must have modify permission on both the containing resource and the `knora-base:LinkValue` for the existing link.

When a new value is created, it is given the default permissions specified in the definition of its property. These are subproperties of `knora-base:hasDefaultPermission`, and are converted into the corresponding subproperties of `knora-base:hasPermission`. Similarly, when a new resource is created, it is given the default permissions specified in the definition of its OWL class.

Ontology Constraints

Knora must not allow an update that would violate an ontology constraint.

When creating a new value (as opposed to adding a new version of an existing value), Knora must not allow the update if the containing resource's OWL class does not contain a cardinality restriction for the submitted property, or if the new value would violate the cardinality restriction.

It must also not allow the update if the type of the submitted value does not match the `knora-base:objectClassConstraint` of the property, or if the property has no `knora-base:objectClassConstraint`. In the case of a property that points to a resource, Knora must ensure that the target resource belongs to the OWL class specified in the property's `knora-base:objectClassConstraint`, or to a subclass of that class.

Duplicate and Redundant Values

When creating a new value, or changing an existing value, Knora checks whether the submitted value would duplicate an existing value for the same property in the resource. The definition of 'duplicate' depends on the type of value; it does not necessarily mean that the two values are strictly equal. For example, if two text values contain the same Unicode string, they are considered duplicates, even if they have different Standoff markup. If resource `R` has property `P` with value `V1`, and `V1` is a duplicate of `V2`, the API server must not add another instance of property `P` with value `V2`. However, if the requesting user does not have permission to see `V2`, the duplicate is allowed, because forbidding it would reveal the contents of `V2` to the user.

When creating a new version of a value, Knora also checks whether the new version is redundant, given the existing value. It is possible for the definition of 'redundant' can depend on the type of value, but in practice, it means that the values are strictly equal: any change, however trivial, is allowed.

Versioning

Each Knora value (i.e. something belonging to an OWL class derived from `knora-base:Value`) is versioned. This means that once created, a value is never modified. Instead, 'changing' a value means creating a new version of the value — actually a new value — that points to the previous version using `knora-base:previousValue`. The versions of a value are a singly- linked list, pointing backwards into the past. When a new version of a value is made, the triple that points from the resource to the old version (using a subproperty of `knora-base:hasValue`) is removed, and a triple is added to point from the resource to the new version. Thus the resource always points only to the current version of the value, and the older versions are available only via the current version's `knora-base:previousValue` predicate.

Unlike values, resources (members of OWL classes derived from `knora-base:Resource`) are not versioned. The data that is attached to a resource, other than its values, can be modified.

Deleting Knora does not actually delete resources or values; it only marks them as deleted. Deleted data is normally hidden. All resources and values must have the predicate `knora- base:isDeleted`, whose object is a boolean. If a resource or value has been marked as deleted, it has `knora-base:isDeleted true` and has a `knora-base:deleteDate`. An optional `knora-base:deleteComment` may be added to explain why the resource or value has been marked as deleted.

Normally, a value is marked as deleted without creating a new version of it. However, link values must be treated as a special case. Before a `LinkValue` can be marked as deleted, its reference count must be decremented to 0. Therefore, a new version of the `LinkValue` is made, with a reference count of 0, and it is this new version that is marked as deleted.

Since it is necessary to be able to find out when a resource was deleted, it is not possible to undelete a resource. Moreover, to simplify the checking of cardinality constraints, and for consistency with resources, it is not possible to undelete a value, and no new versions of a deleted value can be made. Instead, if desired, a new resource or value can be created by copying data from a deleted resource or value.

Linking

Knora API v1 treats a link between two resources as a value, but in RDF, links must be treated differently to other types of values. Knora needs to maintain information about the link, including permissions and a version history. Since the link does not have a unique IRI of its own, Knora uses RDF reifications for this purpose. Each link between two resources has exactly one (non-deleted) `knora-base:LinkValue`. The resource itself has a predicate that points to the `LinkValue`, using a naming convention in which the word `Value` is appended to the name of the link predicate to produce the link value predicate. For example, if a resource representing a book has a predicate called `hasAuthor` that points to another resource, it must also have a predicate called `hasAuthorValue` that points to the `LinkValue` in which information about the link is stored. To find a particular `LinkValue`, one can query it either by using its IRI (if known), or by using its `rdf:subject`, `rdf:predicate`, and `rdf:object` (and excluding link values that are marked as deleted).

Like other values, link values are versioned. The link value predicate always points from the resource to the current version of the link value, and previous versions are available only via the current version's `knora-base:previousValue` predicate. Deleting a link means deleting the triple that links the two resources, and making a new version of the link value, marked with `knora-base:isDeleted`. A triple then points from the resource to this new, deleted version (using the link value property).

The API allows a link to be 'changed' so that it points to a different target resource. This is implemented as follows: the existing triple connecting the two resources is removed, and a new triple is added using the same link property and pointing to the new target resource. A new version of the old link's `LinkValue` is made, marked with `knora-base:isDeleted`. A new `LinkValue` is made for the new link. The new `LinkValue` has no connection to the old one.

When a resource contains `knora-base:TextValue` with Standoff markup that includes a reference to another resource, this reference is materialised as a direct link between the two resources, to make it easier to query. A special link property, `knora-base:hasStandoffLinkTo`, is used for this purpose. The corresponding link value property, `knora-base:hasStandoffLinkToValue`, points to a `LinkValue`. This `LinkValue`

contains a reference count, indicated by `knora-base:valueHasRefCount`, that represents the number of text values in the containing resource that include one or more Standoff references to the specified target resource. Each time this number changes, a new version of this `LinkValue` is made. When the reference count reaches zero, the triple with `knora-base:hasStandoffLinkTo` is removed, and a new version of the `LinkValue` is made and marked with `knora-base:isDeleted`. If the same resource reference later appears again in a text value, a new triple is added using `knora-base:hasStandoffLinkTo`, and a new `LinkValue` is made, with no connection to the old one.

For consistency, every `LinkValue` contains a reference count. If the link property is not `knora-base:hasStandoffLinkTo`, the reference count will always be either 1 (if the link exists) or 0 (if it has been deleted, in which case the link value will also be marked with `knora-base:isDeleted`).

When a `LinkValue` is created for a standoff resource reference, it is given the same permissions as the text value containing the reference.

Design

Responsibilities of Responders

`ResourcesResponderV1` has sole responsibility for generating SPARQL to create and updating resources, and `ValuesResponderV1` has sole responsibility for generating SPARQL to create and update values. When a new resource is created with its values, `ValuesResponderV1` generates SPARQL statements that can be included in the `WHERE` and `INSERT` clauses of a SPARQL update to create the values, and `ResourcesResponderV1` adds these statements to the SPARQL update that creates the resource. This ensures that the resource and its values are created in a single SPARQL update operation, and hence in a single triplestore transaction.

Application-level Locking

The 'read committed' isolation level cannot prevent a scenario where two users want to add the same data at the same time. It is possible that both requests would do pre-update checks and simultaneously find that it is OK to add the data, and that both updates would then succeed, inserting redundant data and possibly violating ontology constraints. Therefore, Knora uses short-lived, application-level write locks on resources, to ensure that only one request at a time can update a given resource. Before each update, the application acquires a resource lock. It then does the pre-update checks and the update, then releases the lock. The lock implementation (in `ResourceLocker`) requires each API request message to include a random UUID, which is generated in the *API Routing* package. Using application-level locks allows us to do pre-update checks in their own transactions, and finally to do the SPARQL update in its own transaction.

Ensuring Data Consistency

Knora enforces consistency constraints using three redundant mechanisms:

1. By doing pre-update checks in SPARQL SELECT queries.
2. By doing checks in the `WHERE` clauses of SPARQL updates.
3. By using GraphDB's built-in consistency checker (see *Consistency Checking*).

We take the view that redundant consistency checks are a good thing.

Pre-update checks are SPARQL `SELECT` queries that are executed while holding an application-level lock on the resource to be updated. These checks should work with any triplestore, and can return helpful, Knora-specific error messages to the client if the request would violate a consistency constraint.

However, the SPARQL update itself is our only chance to do pre-update checks in the same transaction that will perform the update. The design of the SPARQL 1.1 Update standard makes it possible to ensure that if certain conditions are not met, the update will not be performed. In our SPARQL update code, each update contains a `WHERE` clause, possibly a `DELETE` clause, and an `INSERT` clause. The `WHERE` clause is executed first. It

performs consistency checks and provides values for variables that are used in the `DELETE` and/or `INSERT` clauses. In our updates, if the expectations of the `WHERE` clause are not met (e.g. because the data to be updated does not exist), the `WHERE` clause should return no results; as a result, the update will not be performed.

Regardless of whether the update succeeds or not, it returns nothing. So the only way to find out whether it was successful is to do a `SELECT` afterwards. Moreover, if the update failed, there is no straightforward way to find out why. This is one reason why Knora does pre-update checks by means of separate `SELECT` queries, *before* performing the update. This makes it possible to return specific error messages to the user to indicate why an update cannot be performed.

Moreover, while some checks are easy to do in a SPARQL update, others are difficult, impractical, or impossible. Easy checks include checking whether a resource or value exists or is deleted, and checking that the `knora-base:objectClassConstraint` of a predicate matches the `rdf:type` of its intended object. Cardinality checks are not very difficult, but they perform poorly on Jena. Knora does not do permission checks in SPARQL, because its permission-checking algorithm is too complex to be implemented in SPARQL. For this reason, Knora's check for duplicate values cannot be done in SPARQL update code, because it relies on permission checks.

SPARQL Update Examples

The following sample SPARQL update code is simpler than what Knora actually does. It is included here to illustrate the way Knora's SPARQL updates are structured and how concurrent updates are handled.

Finding a value IRI in a value's version history

We will need this query below. If a value is present in a resource property's version history, the query returns everything known about the value, or nothing otherwise:

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix knora-base: <http://www.knora.org/ontology/knora-base#>

SELECT ?p ?o
WHERE {
    BIND(IRI("http://data.knora.org/c5058f3a") as ?resource)
    BIND(IRI("http://www.knora.org/ontology/incunabula#book_comment") as ?property)
    BIND(IRI("http://data.knora.org/c5058f3a/values/testComment002") as ?searchValue)

    ?resource ?property ?currentValue .
    ?currentValue knora-base:previousValue* ?searchValue .
    ?searchValue ?p ?o .
}
```

Creating the initial version of a value

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix knora-base: <http://www.knora.org/ontology/knora-base#>

WITH <http://www.knora.org/ontology/incunabula>
INSERT {
    ?newValue rdf:type ?valueType ;
              knora-base:valueHasString """Comment 1""" ;
              knora-base:attachedToUser <http://data.knora.org/users/91e19f1e01> ;
              knora-base:attachedToProject <http://data.knora.org/projects/77275339> ;
              knora-base:hasPermissions "V knora-admin:KnownUser,knora-admin:UnknownUser|M knora-
              knora-base:valueTimestamp ?currentTime .

    ?resource ?property ?newValue .
```

```
} WHERE {
    BIND(IRI("http://data.knora.org/c5058f3a") as ?resource)
    BIND(IRI("http://www.knora.org/ontology/incunabula#book_comment") as ?property)
    BIND(IRI("http://data.knora.org/c5058f3a/values/testComment001") AS ?newValue)
    BIND(IRI("http://www.knora.org/ontology/knora-base#TextValue") AS ?valueType)
    BIND(NOW() AS ?currentTime)

    # Do nothing if the resource doesn't exist.
    ?resource rdf:type ?resourceClass .

    # Do nothing if the submitted value has the wrong type.
    ?property knora-base:objectClassConstraint ?valueType .
}
```

To find out whether the insert succeeded, the application can use the query in *Finding a value IRI in a value's version history* to look for the new IRI in the property's version history.

Adding a new version of a value

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix knora-base: <http://www.knora.org/ontology/knora-base#>

WITH <http://www.knora.org/ontology/incunabula>
DELETE {
    ?resource ?property ?currentValue .
} INSERT {
    ?newValue rdf:type ?valueType ;
              knora-base:valueHasString """Comment 2""" ;
              knora-base:previousValue ?currentValue ;
              knora-base:attachedToUser <http://data.knora.org/users/91e19f1e01> ;
              knora-base:attachedToProject <http://data.knora.org/projects/77275339> ;
              knora-base:hasPermissions "V knora-admin:KnownUser,knora-admin:UnknownUser|M knora-
              knora-base:valueTimestamp ?currentTime .

    ?resource ?property ?newValue .
} WHERE {
    BIND(IRI("http://data.knora.org/c5058f3a") as ?resource)
    BIND(IRI("http://data.knora.org/c5058f3a/values/testComment001") AS ?currentValue)
    BIND(IRI("http://data.knora.org/c5058f3a/values/testComment002") AS ?newValue)
    BIND(IRI("http://www.knora.org/ontology/knora-base#TextValue") AS ?valueType)
    BIND(NOW() AS ?currentTime)

    ?resource ?property ?currentValue .
    ?property knora-base:objectClassConstraint ?valueType .
}
```

The update request must contain the IRI of the most recent version of the value (`http://data.knora.org/c5058f3a/values/c3295339`). If this is not in fact the most recent version (because someone else has done an update), this operation will do nothing (because the `WHERE` clause will return no rows). To find out whether the update succeeded, the application will then need to do a SELECT query using the query in *Finding a value IRI in a value's version history*. In the case of concurrent updates, there are two possibilities:

1. Users A and B are looking at version 1. User A submits an update and it succeeds, creating version 2, which user A verifies using a SELECT. User B then submits an update to version 1 but it fails, because version 1 is no longer the latest version. User B's SELECT will find that user B's new value IRI is absent from the value's version history.
2. Users A and B are looking at version 1. User A submits an update and it succeeds, creating version 2. Before User A has time to do a SELECT, user B reads the new value and updates it again. Both users then

do a SELECT, and find that both their new value IRIs are present in the value's version history.

Getting all versions of a value

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix knora-base: <http://www.knora.org/ontology/knora-base#>

SELECT ?value ?valueTimestamp ?previousValue
WHERE {
    BIND(IRI("http://data.knora.org/c5058f3a") as ?resource)
    BIND(IRI("http://www.knora.org/ontology/incunabula#book_comment") as ?property)
    BIND(IRI("http://data.knora.org/c5058f3a/values/testComment002") AS ?currentValue)

    ?resource ?property ?currentValue .
    ?currentValue knora-base:previousValue* ?value .

    OPTIONAL {
        ?value knora-base:valueTimestamp ?valueTimestamp .
    }

    OPTIONAL {
        ?value knora-base:previousValue ?previousValue .
    }
}
```

This assumes that we know the current version of the value. If the version we have is not actually the current version, this query will return no rows.

Consistency Checking

Ontotext GraphDB provides a mechanism for checking the consistency of data in a repository each time an update transaction is committed. Knora provides GraphDB-specific consistency rules that take advantage of this feature to provide an extra layer of consistency checks, in addition to the checks that are implemented in the Knora API server.

Requirements

The Knora API server is designed to prevent inconsistencies in RDF data, as far as is practical, in a triplestore-independent way (see *Triplestore Updates*). However, it is also useful to enforce consistency constraints in the triplestore itself, for two reasons:

1. To prevent inconsistencies resulting from bugs in the Knora API server.
2. To prevent users from inserting inconsistent data directly into the triplestore, bypassing the Knora API server.

The design of the `knora-base` ontology supports two ways of specifying constraints on data (see *The Knora Ontologies* for details):

1. A property definition should specify the types that are allowed as subjects and objects of the property, using `knora-base:subjectClassConstraint` and (if it is an object property) `knora-base:objectClassConstraint`. Every subproperty of `knora-base:hasValue` or a `knora-base:hasLinkTo` (i.e. every property of a resource that points to a `knora-base:Value` or to another resource) is required have this constraint, because the Knora API server relies on it to know what type of object to expect for the property. Use of `knora-base:subjectClassConstraint` is recommended but not required.
2. A class definition should use OWL cardinalities (see OWL 2 Quick Reference Guide) to indicate the properties that instances of the class are allowed to have, and to constrain the number of objects that each property

can have. Subclasses of `knora-base:Resource` are required to have a cardinality for each subproperty of `knora-base:hasValue` or a `knora-base:hasLinkTo` that resources of that class can have.

Specifically, consistency checking should prevent the following:

- An object property or datatype property has a subject of the wrong class, or an object property has an object of the wrong class (GraphDB's consistency checke cannot check the types of literals).
- An object property has an object that does not exist (i.e. the object is an IRI that is not used as the subject of any statements in the repository). This can be treated as if the object is of the wrong type (i.e. it can cause a violation of `knora-base:objectClassConstraint`, because there is no compatible `rdf:type` statement for the object).
- A class has `owl:cardinality 1` or `owl:minCardinality 1` on an object property or datatype property, and an instance of the class does not have that property.
- A class has `owl:cardinality 1` or `owl:maxCardinality 1` on an object property or datatype property, and an instance of the class has more than one object for that property.
- An instance of `knora-base:Resource` has an object property pointing to a `knora-base:Value` or to another `Resource`, and its class has no cardinality for that property.
- An instance of `knora-base:Value` has a subproperty of `knora-base:valueHas`, and its class has no cardinality for that property.
- A datatype property has an empty string as an object.

Cardinalities in base classes are inherited by derived classes. Derived classes can override inherited cardinalities by making them more restrictive, i.e. by specifying a subproperty of the one specified in the original cardinality.

Instances of `Resource` and `Value` can be marked as deleted, using the property `isDeleted`. This must be taken into account as follows:

- With `owl:cardinality 1` or `owl:maxCardinality 1`, if the object of the property can be marked as deleted, the property must not have more than one object that has not been marked as deleted. In other words, it's OK if there is more than one object, as long only one of them has `knora-base:isDeleted false`.
- With `owl:cardinality 1` or `owl:minCardinality 1`, the property must have an object, but it's OK if the property's only object is marked as deleted. We allow this because the subject and object may have different owners, and it may not be feasible for them to coordinate their work. The owner of the object should always be able to mark it as deleted. (It could be useful to notify the owner of the subject when this happens, but that is beyond the scope of consistency checking.)

Design

When a repository is created in GraphDB, a set of consistency rules can be provided, and GraphDB's consistency checker can be turned on to ensure that each update transaction respects these rules, as described in the section Reasoning of the GraphDB documentation. Like custom inference rules, consistency rules are defined in files with the `.pie` filename extension, in a GraphDB-specific syntax.

We have added rules to the standard RDFS inference rules file `builtin_RdfsRules.pie`, to create the file `KnoraRules.pie`. The `.ttl` configuration file that is used to create the repository must contain these settings:

```
owlim:ruleset "/path/to/KnoraRules.pie" ;
owlim:check-for-inconsistencies "true" ;
```

The path to `KnoraRules.pie` must be an absolute path. The scripts provided with Knora to create test repositories set this path automatically.

Consistency checking in GraphDB relies on reasoning. GraphDB's reasoning is Forward-chaining, which means that reasoning is applied to the contents of each update, before the update transaction is committed, and the inferred statements are added to the repository.

A GraphDB rules file can contain two types of rules: inference rules and consistency rules. Before committing an update transaction, GraphDB applies inference rules, then consistency rules. If any of the consistency rules are violated, the transaction is rolled back.

An inference rule has this form:

```
Id: <rule_name>
    <premises> <optional_constraints>
    ------------------------------
    <consequences> <optional_constraints>
```

The premises are a pattern that tries to match statements found in the data. Optional constraints, which are enclosed in square brackets, make it possible to specify the premises more precisely, or to specify a named graph (see examples below). Consequences are the statements that will be inferred if the premises match. A line of hyphens separates premises from consequences.

A GraphDB consistency rule has a similar form:

```
Consistency: <rule_name>
    <premises> <optional_constraints>
    ------------------------------
    <consequences> <optional_constraints>
```

The differences between inference rules and consistency rules are:

- A consistency rule begins with `Consistency` instead of `Id`.
- In a consistency rule, the consequences are optional. Instead of representing statements to be inferred, they represent statements that must exist if the premises are satisfied. In other words, if the premises are satisfied and the consequences are not found, the rule is violated.
- If a consistency rule doesn't specify any consequences, and the premises are satisfied, the rule is violated.

Rules use variable names for subjects, predicates, and objects, and they can use actual property names.

Empty string as object

If subject `i` has a predicate `p` whose object is an empty string, the constraint is violated:

```
Consistency: empty_string
    i p ""
    -------------------------------------
```

Subject and object class constraints

If subject `i` has a predicate `p` that requires a subject of type `t`, and `i` is not a `t`, the constraint is violated:

```
Consistency: subject_class_constraint
    p <knora-base:subjectClassConstraint> t
    i p j
    -------------------------------------
    i <rdf:type> t
```

If subject `i` has a predicate `p` that requires an object of type `t`, and the object of `p` is not a `t`, the constraint is violated:

```
Consistency: object_class_constraint
    p <knora-base:objectClassConstraint> t
    i p j
    -------------------------------------
    j <rdf:type> t
```

Cardinality constraints

A simple implementation of a consistency rule to check `owl:maxCardinality 1`, for objects that can be marked as deleted, could look like this:

```
Consistency: max_cardinality_1_with_deletion_flag
    i <rdf:type> r
    r <owl:maxCardinality> "1"^^xsd:nonNegativeInteger
    r <owl:onProperty> p
    i p j
    i p k [Constraint j != k]
    j <knora-base:isDeleted> "false"^^xsd:boolean
    k <knora-base:isDeleted> "false"^^xsd:boolean
    ------------------------------------
```

This means: if resource `i` is a subclass of an `owl:Restriction r` with `owl:maxCardinality 1` on property `p`, and the resource has two different objects for that property, neither of which is marked as deleted, the rule is violated. Note that this takes advantage of the fact that `Resource` and `Value` have `owl:cardinality 1` on `isDeleted` (`isDeleted` must be present even if false), so we do not need to check whether `i` is actually something that can be marked as deleted.

However, this implementation would be much too slow. We therefore use two optimisations suggested by Ontotext:

1. Add custom inference rules to make tables (i.e. named graphs) of pre-calculated information about the cardinalities on properties of subjects, and use those tables to simplify the consistency rules.
2. Use the `[Cut]` constraint to avoid generating certain redundant compiled rules (see Entailment rules).

For example, to construct a table of subjects belonging to classes that have `owl:maxCardinality 1` on some property `p`, we use the following custom inference rule:

```
Id: maxCardinality_1_table
    i <rdf:type> r
    r <owl:maxCardinality> "1"^^xsd:nonNegativeInteger
    r <owl:onProperty> p
    ------------------------------------
    i p r [Context <onto:_maxCardinality_1_table>]
```

The constraint `[Context <onto:_maxCardinality_1_table>]` means that the inferred triples are added to the context (i.e. the named graph) `http://www.ontotext.com/_maxCardinality_1_table`. (Note that we have defined the prefix `onto` as `http://www.ontotext.com/` in the `Prefices` section of the rules file.) As the GraphDB documentation on Rules explains:

> If the context is provided, the statements produced as rule consequences are not 'visible' during normal query answering. Instead, they can only be used as input to this or other rules and only when the rule premise explicitly uses the given context.

Now, to find out whether a subject belongs to a class with that cardinality on a given property, we only need to match one triple. The revised implementation of the rule `max_cardinality_1_with_deletion_flag` is as follows:

```
Consistency: max_cardinality_1_with_deletion_flag
    i p r [Context <onto:_maxCardinality_1_table>]
    i p j [Constraint j != k]
    i p k [Cut]
    j <knora-base:isDeleted> "false"^^xsd:boolean
    k <knora-base:isDeleted> "false"^^xsd:boolean
    ------------------------------------
```

The constraint `[Constraint j != k]` means that the premises will be satisfied only if the variables `j` and `k` do not refer to the same thing.

With these optimisations, the rule is faster by several orders of magnitude.

Since properties whose objects can be marked as deleted must be handled differently to properties whose objects cannot be marked as deleted, the `knora-base` ontology provides a property called `objectCannotBeMarkedAsDeleted`. All properties in `knora-base` whose objects cannot take the `isDeleted` flag (including datatype properties) should be derived from this property. This is how it is used to check `owl:maxCardinality 1` for objects that cannot be marked as deleted:

```
Consistency: max_cardinality_1_without_deletion_flag
    i p r [Context <onto:_maxCardinality_1_table>]
    p <rdfs:subPropertyOf> <knora-base:objectCannotBeMarkedAsDeleted>
    i p j [Constraint j != k]
    i p k [Cut]
    -----------------------------------
```

To check `owl:minCardinality 1`, we do not care whether the object can be marked as deleted, so we can use this simple rule:

```
Consistency: min_cardinality_1_any_object
    i p r [Context <onto:_minCardinality_1_table>]
    -----------------------------------
    i p j
```

This means: if a subject `i` belongs to a class that has `owl:minCardinality 1` on property `p`, and `i` has no object for `p`, the rule is violated.

To check `owl:cardinality 1`, we need two rules: one that checks whether there are too few objects, and one that checks whether there are too many. To check whether there are too few objects, we don't care whether the objects can be marked as deleted, so the rule is the same as `min_cardinality_1_any_object`, except for the cardinality:

```
Consistency: cardinality_1_not_less_any_object
    i p r [Context <onto:_cardinality_1_table>]
    -----------------------------------
    i p j
```

To check whether there are too many objects, we need to know whether the objects can be marked as deleted or not. In the case where the objects can be marked as deleted, the rule is the same as `max_cardinality_1_with_deletion_flag`, except for the cardinality:

```
Consistency: cardinality_1_not_greater_with_deletion_flag
    i p r [Context <onto:_cardinality_1_table>]
    i p j [Constraint j != k]
    i p k [Cut]
    j <knora-base:isDeleted> "false"^^xsd:boolean
    k <knora-base:isDeleted> "false"^^xsd:boolean
    -----------------------------------
```

In the case where the objects cannot be marked as deleted, the rule is the same as `max_cardinality_1_without_deletion_flag`, except for the cardinality:

```
Consistency: cardinality_1_not_less_any_object
    i p r [Context <onto:_cardinality_1_table>]
    -----------------------------------
    i p j
```

Knora allows a subproperty of `knora-base:hasValue` or `knora-base:hasLinkTo` to be a predicate of a resource only if the resource's class has some cardinality for the property. For convenience, `knora-base:hasValue` and `knora-base:hasLinkTo` are subproperties of `knora-base:resourceProperty`, which is used to check this constraint in the following rule:

```
Consistency: resource_prop_cardinality_any
    i <knora-base:resourceProperty> j
    -----------------------------------
    i p j
```

```
    i <rdf:type> r
    r <owl:onProperty> p
```

If resource `i` has a subproperty of `knora-base:resourceProperty`, and `i` is not a member of a subclass of an `owl:Restriction` `r` with a cardinality on that property (or on one of its base properties), the rule is violated.

A similar rule, `value_prop_cardinality_any`, ensures that if a value has a subproperty of `knora-base:valueHas`, the value's class has some cardinality for that property.

Authentication in the Knora API Server

Scope

Authentication is the process of making sure that if someone is accessing something then this someone is actually also the someone he pretends to be. The process of making sure that someone is authorized, i.e. has the permission to access something, is handled as described in the section on authorization in the Knora base ontology document. TODO: add a link to this.

Implementation

The authentication in Knora is based on Basic Auth HTTP basic authentication, URL parameters, and cookies. This means that on every request (to any of the routes), an authentication header, URL parameters or cookie need to be sent.

All routes are always accessible and if there are no credentials provided, a default user is assumed. If credentials are sent and they are not correct (e.g., wrong username, password incorrect), then the request will end in an error message. This is not true for a cookie containing an expired session id. In this case, the default user is assumed.

Usage

Checking Credentials

To check the credentials and create a 'session', e.g., by a login window in the client, there is a special route called **/v1/authentication**, which returns following for each case:

Credentials correct:

```
{
  "status": 0,
  "message": "credentials are OK".
  "sid": "1437643844783"
}
```

In this case, the found user profile is written to a cache and stored under the ''sid" key. Also a header requesting to store the ''sid" in a cookie is sent. On subsequent access to all the other routes, the ''sid" is used to retrieve the cached user profile. If successful, the user is deemed authenticated.

Email wrong:

```
{
  "status": 2,
  "message": "bad credentials: user not found"
}
```

Password wrong:

```
{
  "status": 2,
  "message": "bad credentials: user found, but password did not match"
}
```

No credentials:

```
{
  "status": 999,
  "message": "no credentials found"
}
```

Web client (Login/Logout)

When a web client accesses the **/v1/authentication** route successfully, it gets back a cookie. To **logout** the client can call the same route and provide the logout parameter **/v1/authenticate?logout**. This will delete the cache entry and remove the session id from the cookie on the client.

Submitting Credentials in the URL or in the HTTP Authentication Header

As an alternative to creating a session, the client may also submit the credentials:

- in the URL (when doing a HTTP-GET request) submitting the parameters `email` and `password` (e.g. http://knora-host/v1/resources/resIri?email=userUrlEncodedEmail&password=pw)
- in the HTTP header (HTTP basic authentication) when doing a HTTP request to the API (all methods). When using Python's module `requests`, the credentials can simply be submitted as a tuple with each request using the param `auth` (python requests).

Workflow

1. The login form on the client can use */v1/authentication* to check if the email/password combination provided by the user is correct. The email and password can be provided as URL parameters (see above).
2. on the server, this gets checked and a corresponding result as described will be returned
3. all subsequent calls can then send these credentials as authentication header or URL parameters (email / password), and in the case of a web client just the cookie.

Step 1 and 2 are optional, and can be skipped, if prior checking of the credentials is not needed. Naturally, this won't work for a web client using cookies for authentication.

Skipping Authentication

There is the possibility to turn skipping authentication on and use a hardcoded user (Test User). In **application.conf** set the `skip-authentication = true` and Test User will be always assumed.

Sipi (Media Server)

For authentication to work with the media server, we need to add support for cookies. At the moment the SALSAH-App would set BasicAuth heathers, but this only works for AJAX requests using `SALSAH.ApiGet` (`Put`, etc.). Since the medias are embedded as source tags, the browser would get them on his own, and doesn't know anything about the needed AuthHeathers. With cookies, the browser would send those automatically with every request. The media server can use the credentials of the user requesting something for accessing the RepresentationsRouteV1, i.e. make this request in the name of the user so to speak, then the RepresentationResponderV1 should have all the information it needs to filter the result based on the users permissions.

Improving Security

In the first iteration, the email/password would be sent in clear text. Since we will use HTTPS this shouldn't be a problem. The second iteration, could encrypt the email/password.

Administration (Users, Projects, Groups, Institutions, Permissions)

Scope

This Section includes management (creation, updating, deletion) of *Users*, *Projects*, *Groups*, *Institutions*, and *Permissions*.

Implementation

All administration functions will be implemented as part of the Knora API in the `webapi` codebase. There is also a separate web-application as part of the `salsah` codebase using this API, allowing basic management operations.

Overview

During the initial deployment of a Knora server, the main administration user (*root*) is created. This *root* user has the right to do anything.

Knora's concept of access control is that permissions can only be granted to groups (or the whole project, i.e. all members of a project) and not to individual users. There are two distinct ways of granting permission. Firstly, an object (a resource or value) can grant permissions to groups of users, and secondly, permissions can be granted directly to a group of users (not bound to a specific object). There are six built-in groups: *UnknownUser*, *KnownUser*, *Creator*, *ProjectMember*, *ProjectAdmin*, and *SystemAdmin*. These groups can be used in the same way as normal user created groups for permission management, i.e. can be used to give certain groups of users, certain permissions, without the need to explicitly create them.

A user becomes implicitly a member of such a group by satisfying certain conditions:

knora-base:UnknownUser: Any user who has not logged into the Knora API server is automatically assigned to this group.

knora-base:KnownUser: Any user who has logged into the Knora API server is automatically assigned to this group.

knora-base:Creator: When checking a user's permissions on an object, the user is automatically assigned to this group if he is the creator of the object.

knora-base:ProjectMember: When checking a user's permissions, the user is automatically assigned to this group by being a member of a project designated by the `knora-base:isInProject` property.

knora-base:ProjectAdmin: When checking a user's permission, the user is automatically assigned to this group through the `knora-base:isInProjectAdminGroup` property, which points to the project in question.

knora-base:SystemAdmin: Membership is received by setting the property `knora-base:isInSystemAdminGroup` to `true` on a `knora-base:User`.

To use these build-in groups as values for properties (Object Access and Default Permissions), the IRI is constructed by appending the name of the built-in group to `knora-base`, e.g., `knora-base:KnownUser` where `knora-base` corresponds to `http://www.knora.org/ontology/knora-base#`.

Permissions

Up until know, we have mentioned two groups of permissions. The first called *object access permissions*, which contains permissions that point from explicit **objects** (resources/values) to groups. The second group of permissions called *administrative permissions*, and which contains permissions that are put on instances of **knora-base:Permission** objects directly affecting groups. There is another, third group of permissions, called *default object access permissions* which is also put on instances of *knora-base:Permission*, and wich also directly affect groups.

Object Access Permissions

An object (resource / value) can grant the following permissions, which are stored in a compact format in a single string, which is the object of the predicate `knora-base:hasPermissions`:

1. **Restricted view permission (RV)**: Allows a restricted view of the object, e.g. a view of an image with a watermark.
2. **View permission (V)**: Allows an unrestricted view of the object. Having view permission on a resource only affects the user's ability to view information about the resource other than its values. To view a value, she must have view permission on the value itself.
3. **Modify permission (M)**: For values, this permission allows a new version of a value to be created. For resources, this allows the user to create a new value (as opposed to a new version of an existing value), or to change information about the resource other than its values. When he wants to make a new version of a value, his permissions on the containing resource are not relevant. However, when he wants to change the target of a link, the old link must be deleted and a new one created, so he needs modify permission on the resource.
4. **Delete permission (D)**: Allows the item to be marked as deleted.
5. **Change rights permission (CR)**: Allows the permissions granted by the object to be changed.

Each permission in the above list implies all lower-numbered permissions.

A user's permission level on a particular object is calculated in the following way:

1. Make a list of the groups that the user belongs to, including Creator and/or ProjectMember and/or ProjectAdmin if applicable.
2. Make a list of the permissions that she can obtain on the object, by iterating over the permissions that the object grants. For each permission, if she is in the specified group, add the specified permission to the list of permissions she can obtain.
3. From the resulting list, select the highest-level permission.
4. If the result is that she would have no permissions, give her whatever permission *UnknownUser* would have.

The format of the object of `knora-base:hasPermissions` is as follows:

- Each permission is represented by the one-letter or two-letter abbreviation given above.
- Each permission abbreviation is followed by a space, then a comma-separated list of groups that the permission is granted to.
- The IRIs of built-in groups are shortened using the knora-base prefix.
- Multiple permissions are separated by a vertical bar (|).

For example, if an object grants view permission to *unknown* and *known users*, and modify permission to *project members*, the resulting permission literal would be:

```
V knora-base:UnknownUser,knora-base:KnownUser|M knora-base:ProjectMember
```

Administrative Permissions

The following permissions can be set via instances of `knora-base:AdministrativePermission` on any group belonging to a project. For users that are members of a number of groups with administrative permissions attached, the final set of permissions is additive and most permissive. The administrative permissions are stored in a compact format in a single string, which is the object of the predicate `knora-base:hasPermissions` attached to an instance of the `knora-base:AdministrativePermission` class. The following permission values can be used:

1. Resource / Value Creation Permissions:
    (a) **ProjectResourceCreateAllPermission**:
    - description: gives the permission to create resources inside the project.
    - usage: used as a value for *knora-base:hasPermissions*.

    (a) **ProjectResourceCreateRestrictedPermission**:
    - description: gives restricted resource creation permission inside the project.
    - usage: used as a value for *knora-base:hasPermissions*.
    - value: `RestrictedProjectResourceCreatePermission` followed by a comma-separated list of *ResourceClasses* the user should only be able to create instances of.
2. Project Administration Permissions:
    (a) **ProjectAdminAllPermission**:
    - description: gives the user the permission to do anything on project level, i.e. create new groups, modify all existing groups (*group info*, *group membership*, *resource creation permissions*, *project administration permissions*, and *default permissions*).
    - usage: used as a value for *knora-base:hasPermissions*.

    (a) **ProjectAdminGroupAllPermission**:
    - description: gives the user the permission to modify *group info* and *group membership* on *all* groups belonging to the project.
    - usage: used as a value for the *knora-base:hasPermissions* property.

    (a) **ProjectAdminGroupRestrictedPermission**:
    - description: gives the user the permission to modify *group info* and *group membership* on *certain* groups belonging to the project.
    - usage: used as a value for *knora-base:hasPermissions*
    - value: `ProjectGroupAdminRestrictedPermission` followed by a comma-separated list of `knora-base:UserGroup`.

    (a) **ProjectAdminRightsAllPermission**:
    - description: gives the user the permission to change the *permissions* on all objects belonging to the project (e.g., default permissions attached to groups and permissions on objects).
    - usage: used as a value for the *knora-base:hasPermissions* property.
3. Ontology Administration Permissions:
    (a) **ProjectAdminOntologyAllPermission**:
    - description: gives the user the permission to administer the project ontologies

- usage: used as a value for the *knora-base:hasPermissions* property.

The administrative permissions are stored in a compact format in a single string, which is the object of the predicate `knora-base:hasPermissions` attached to an instance of the `knora-base:AdministrativePermission` class.

The format of the object of `knora-base:hasPermissions` is as follows:

- Each permission is represented by the name given above.
- Each permission is followed by a space, then if applicable, by a comma separated list of IRIs, as defined above.
- The IRIs of built-in values (e.g., built-in groups, resource classes, etc.) are shortened using the knora-base prefix `knora-base:`.
- Multiple permissions are separated by a vertical bar (|).

For example, if an administrative permission grants the `knora-base:ProjectMember` group the permission to create all resources (*ProjectResourceCreateAllPermission*), the resulting administrative permission object with the compact form literal would be:

```
<http://data.knora.org/permissions/001>
        rdf:type knora-base:AdministrativePermission ;
        knora-base:forProject <http://data.knora.org/projects/images> ;
        knora-base:forGroup knora-base:ProjectMember ;
        knora-base:hasPermissions "ProjectResourceCreateAllPermission"^^xsd:string .
```

Default Object Access Permissions

Default Object Access Permissions are used when new objects (resources and/or values) are created. They represent object access permissions with which the new object will be initially outfitted. As with administrative permissions, these default object access permissions can be defined for any number of groups. Additionally, they can be also defined for resource classes and properties.

The following default object access permissions can be attached to groups, resource classes and/or properties via instances of *knora-base:DefaultObjectAccessPermission* (described further bellow). The default object access permissions correspond to the earlier described object access permission:

1. **Default Restricted View Permission (RV)**:
   - description: any object, created by a user inside a group holding this permission, is restricted to carry this permission
   - value: `RV` followed by a comma-separated list of `knora-base:UserGroup`
2. **Default View Permission (V)**:
   - description: any object, created by a user inside a group holding this permission, is restricted to carry this permission
   - value: `V` followed by a comma-separated list of `knora-base:UserGroup`
3. **Default Modify Permission (M)** accompanied by a list of groups.
   - description: any object, created by a user inside a group holding this permission, is restricted to carry this permission
   - value: `M` followed by a comma-separated list of `knora-base:UserGroup`
4. **Default Delete Permission (D)** accompanied by a list of groups.
   - description: any object, created by a user inside a group holding this permission, is restricted to carry this permission
   - value: `D` followed by a comma-separated list of `knora-base:UserGroup`
5. **Default Change Rights Permission (CR)** accompanied by a list of groups.

- description: any object, created by a user inside a group holding this permission, is restricted to carry this permission
- value: `CR` followed by a comma-separated list of `knora-base:UserGroup`

A single instance of `knora-base:DefaultObjectAccessPermission` must always reference a project, but can only reference **either** a group (`knora-base:forGroup` property), a resource class (`knora-base:forResourceClass`), a property (`knora-base:forProperty`), or a combination of resource class **and** property.

Example default object access permission instance:

```
<http://data.knora.org/permissions/002>
        rdf:type knora-base:DefaultObjectAccessPermission ;
        knora-base:forProject <http://data.knora.org/projects/images> ;
        knora-base:forGroup knora-base:ProjectMember ;
        knora-base:hasPermissions "CR knora-base:Creator|M knora-base:ProjectMember|V knora-base:
```

This instance is setting default object access permissions to the project member group of a project, giving change right permission to the creator, modify permission to all project members, and view permission to known users. Further, this **implicitly** applies to all resource classes and all their properties inside the project.

Permission Precedence Rules

For both administrative permissions and default object access permissions, the resulting permissions are derived by applying precedence rules, for the case that the user is member of more than one group.

The following list is sorted by the permission precedence level in descending order:

- permissions on `knora-base:ProjectAdmin` (highest level)
- permissions on resource classes and property combination (own project)
- permissions on resource classes and property combination (`knora-base:SystemProject`)
- permissions on resource classes / properties (own project)
- permissions on resource classes / properties (`knora-base:SystemProject`)
- permissions on custom groups
- permissions on `knora-base:ProjectMember`
- permissions on `knora-base:KnownUser` (lowest level)

The permissions on resource classes / properties are only relevant for default object access permissions.

Administrative Permissions: When a user performs an operation requiring administrative permissions, then **only** the permissions from the **highest level** are taken into account. If a user is a member of more than one group on the same level (only possible for custom groups) then the defined permissions are summed up and all are taken into account.

Default Object Access Permissions: When a user creates a resource or value, then **only** the default object permissions from the **highest level** are applied. If a user is a member of more than one group on the same level (only possible for custom groups) then the defined permissions are summed up and the most permissive are applied.

In the case of users belonging to the **SystemAdmin** group, but which are not members of a project and thus no group belonging to the project, the *default object access permissions* from the **highest defined level** will apply.

Implicit Permissions

The `knora-base:SystemAdmin` group receives implicitly the following permissions:

- receives implicitly *ProjectAllAdminPermission* for all projects.
- receives implicitly *ProjectResourceCreateAllPermission* for all projects.

- receives implicitly *CR* on all objects from all projects.

Theses permissions are baked into the system, and cannot be changed.

Permission Templates

The permission capabilities of Knora are very large, as it needs to be able to satisfy a broad set of requirements. To simplify permission management for the users, we provide permission templates, which can be used during creation of new projects, or applied to existing projects. A permission template defines a set of administrative and default object access permission. Currently, two different templates will be defined *OPEN*, *CLOSED*.

Template: OPEN

The *OPEN* template, defines the following permissions:

- **The `knora-base:ProjectAdmin` group:**
  - receives explicitly *ProjectResourceCreateAllPermission*.
  - receives explicitly *ProjectAllAdminPermission*.
- **The `knora-base:ProjectMember` group:**
  - receives explicitly *ProjectResourceCreateAllPermission*.
  - receives explicitly *CR* for the *knora-base:Creator* and *knora-base:ProjectAdmin* group.
  - receives explicitly *M* for the *ProjectMember* group.
  - receives explicitly *V* for the *knora-base:KnownUser* group.

Template: CLOSED

The *CLOSED* template, defined the following permissions:

- **The `knora-base:ProjectAdmin` group:**
  - receives explicitly *ProjectResourceCreateAllPermission*.
  - receives explicitly *ProjectAllAdminPermission*.
- **The `knora-base:ProjectMember` group:**
  - receives explicitly *ProjectResourceCreateAllPermission*.
  - receives explicitly *CR* for the *knora-base:ProjectAdmin* group.
  - receives explicitly *M* for the *ProjectMember* group.

Default Permissions Matrix for new Projects

The access control matrix defines what are the default operations a *subject* (i.e. User), being a member of a built-in group (represented by row headers), is permitted to perform on an *object* (represented by column headers). The different operation abbreviations used are defined as follows:

***C*:** *Create* - the subject inside the group is allowed to *create* the object.

***U*:** *Update* - the subject inside the group is allowed to *update* the object.

***R*:** *Read* - the subject inside the group is allowed to *read* **all** information about the object.

***D*:** *Delete* - the subject inside the group is allowed to *delete* the object.

***P*:** *Permission* - the subject inside the group is allowed to change the *permissions* on the object.

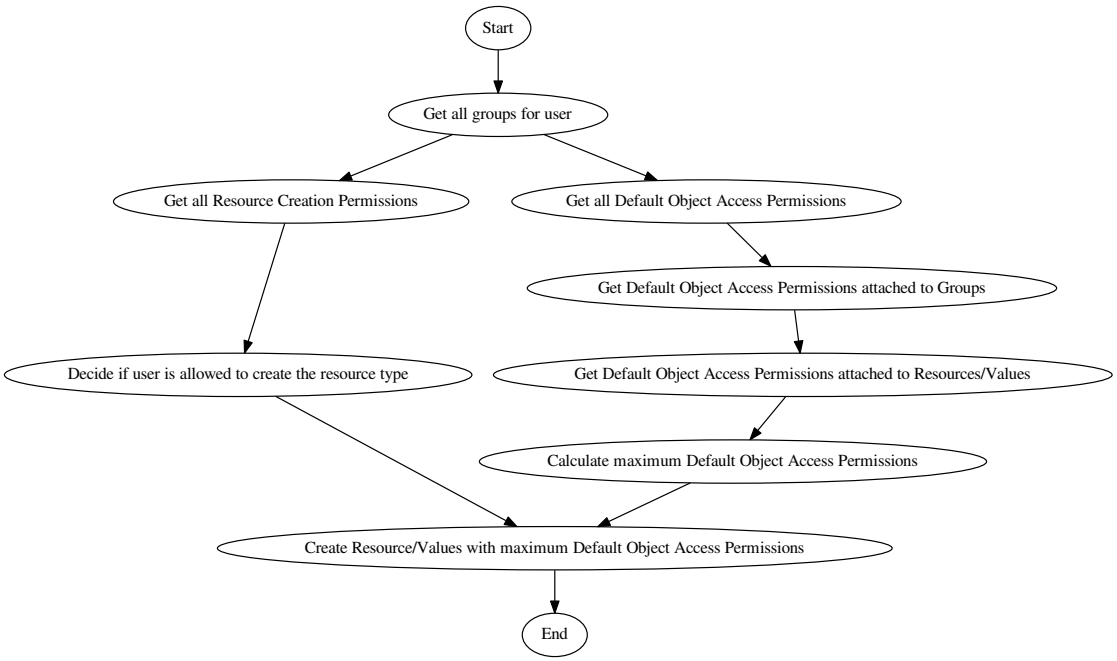
-: *none* - none or not applicable

Table 4.1: Default Permissions Matrix for new Projects

Built-In Group	Project	Group	User	Resource	Value
SystemAdmin	CRUD	CRUDP	CRUDP all	CRUDP all	CRUDP all
ProjectAdmin	-RUD	CRUDP	CRUDP +/- project	CRUDP (in project)	CRUDP (in project)
ProjectMember	----	-----	-----	CRUD- (in project)	----- (in project)
Creator	----	-----	-----	-RUDP (his resource)	----- (his value)
KnownUser	C---	C----	CRUD- himself	R---- (in project)	R---- (in project)

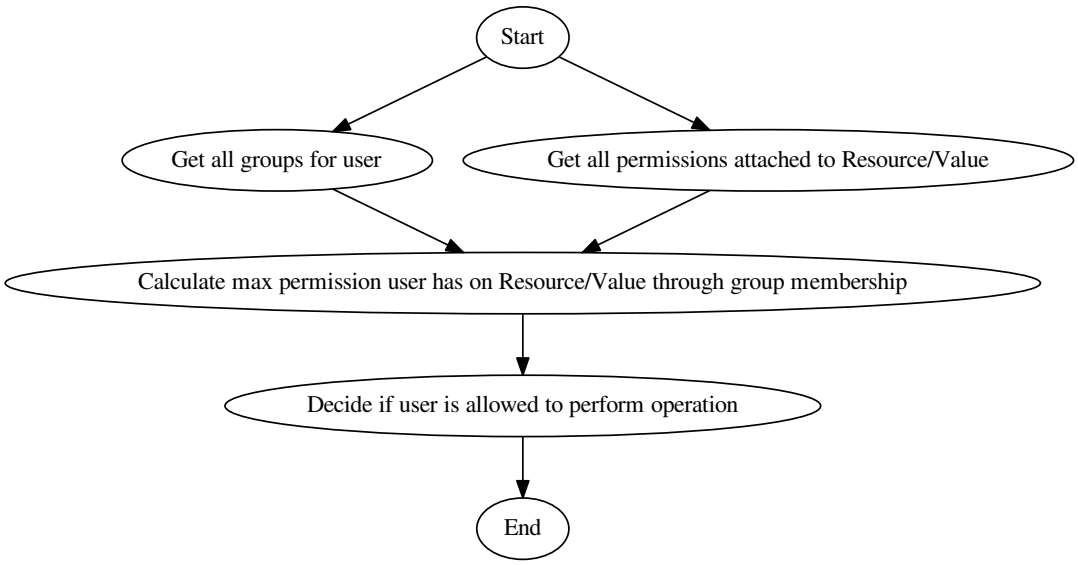
Basic Workflows involving Permissions

Creating a new Resource



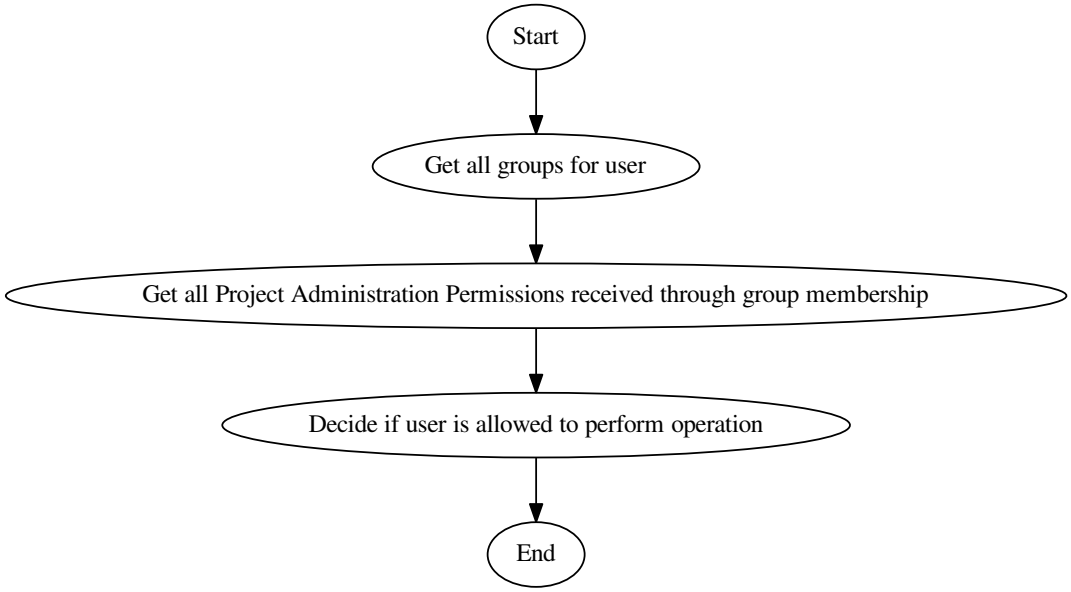
TODO: Text describing the graph.

Accessing a Resource/Value



TODO: Text describing the graph.

Project / Group Administration



TODO: Text describing the graph.

Implementation

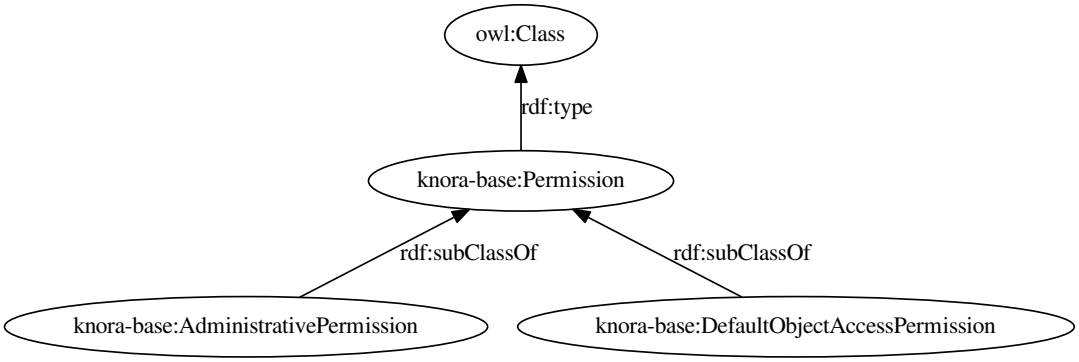
The requirements for defining default permissions imposed by all the different use cases are very broad. Potentially, we need to be able to define default permissions per project, per group, per resource class, per resource property, and all their possible combinations.

For this reason, we introduce the *knora-base:Permission* class with two sub-classes, namely *knora-base:AdministrativePermission* and *knora-base:DefaultObjectAccessPermission*, which instances will carry all the necessary information.

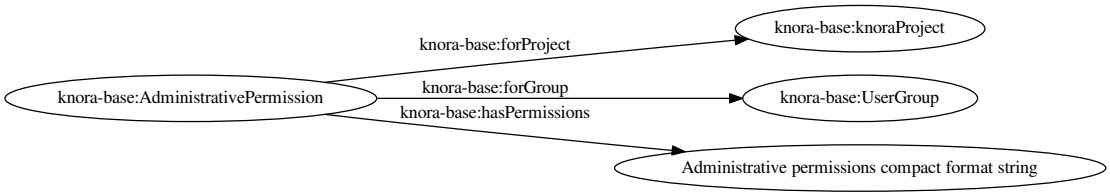
Permission Class Hierarchy and Structure

The following graphs show the class hierarchy and the structure of each permission class.

Permission Class Hierarchy



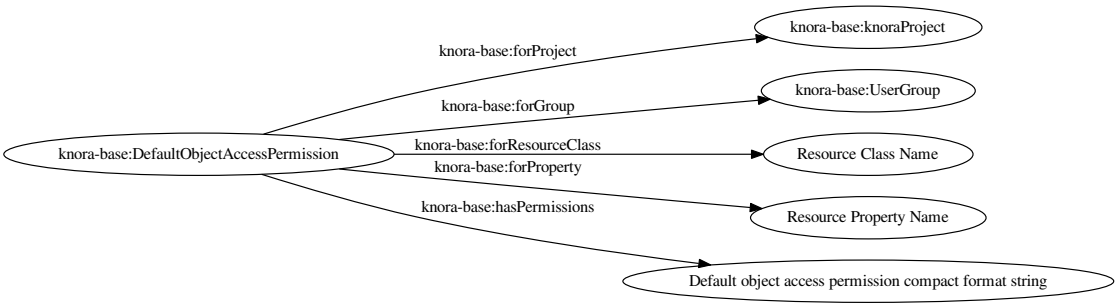
Administrative Permission Structure:



and the same as RDF:

```
<http://data.knora.org/permissions/[UUID]> rdf:type knora-base:AdministrativePermission ;
     knora-base:forProject <http://data.knora.org/projects/[UUID]> ;
     knora-base:forGroup <http://data.knora.org/groups/[UUID]> ;
     knora-base:hasPermissions "ProjectResourceCreateAllPermission|
                                ProjectResourceCreateRestrictedPermission "<Resource Class IRI>"|
                                ProjectAdminAllPermission|
                                ProjectAdminGroupAllPermission|
                                ProjectAdminGroupRestrictedPermission "<http://data.knora.org/grou
                                ProjectAdminRightsAllPermission|
                                ProjectAdminOntologyAllPermission"^^xsd:string .
```

Default Object Access Permission Structure:



and the same as RDF:

```
<http://data.knora.org/permissions/[UUID]> rdf:type knora-base:DefaultObjectAccessPermission ;
     knora-base:forProject <http://data.knora.org/projects/[UUID]> ;
     knora-base:forGroup <http://data.knora.org/groups/[UUID]> ;
     knora-base:forResourceClass "Resource Class Name" ;
     knora-base:forProperty "Resource Property Name" ;
     knora-base:hasPermissions "RV <http://data.knora.org/groups/[UUID]>|
                                 V <http://data.knora.org/groups/[UUID]>|
                                 M <http://data.knora.org/groups/[UUID]>|
                                 D <http://data.knora.org/groups/[UUID]>|
                                CR <http://data.knora.org/groups/[UUID]>"^^xsd:string .
```

Querying Permission Instances

The properties **forProject** and either of **forGroup**, **forResourceClass**, and **forProperty** form together a *compound key*, allowing finding existing permission instances, that address the same set of Project / Group / Resource-Class / Property combination, thus making it possible to extend or change the attached permissions.

Administrative Permission Instances: For each group inside the project, there can be **zero** or **one** instance holding administrative permission information. Querying is straitforward by using the *knora-base:forProject* and *knora-base:forGroup* properties as the compound key.

Default Object Access Permission Instances: For each group, resource class, or property inside the project, there can be **zero** or **one** instances holding default object access permission informations. Querying is straitforward by using the *knora-base:forProject* and either *knora-base:forGroup*, *knora-base:forResourceClass*, or *knora-base:forProperty* properties as part of the compound key.

Example Data stored in the permissions graph

Administrative permissions on a 'ProjectAdmin' group:

```
<http://data.knora.org/permissions/[UUID]> rdf:type knora-base:AdministrativePermission ;
     knora-base:forProject <http://data.knora.org/projects/images> ;
     knora-base:forGroup knora-base:ProjectAdmin ;
     knora-base:hasPermissions "ProjectResourceCreateAllPermission|
                                ProjectAdminAllPermission"^^xsd:string .
```

Administrative permissions on a 'ProjectMember' group:

```
<http://data.knora.org/permissions/[UUID]> rdf:type knora-base:AdministrativePermission ;
     knora-base:forProject <http://data.knora.org/projects/images> ;
     knora-base:forGroup knora-base:ProjectMember ;
     knora-base:hasPermissions "ProjectResourceCreateAllPermission"^^xsd:string .
```

Administrative permission restricting project admin permission on a group:

```
<http://data.knora.org/permissions/[UUID]> rdf:type knora-base:Permission ;
     knora-base:forProject <http://data.knora.org/projects/[UUID]> ;
     knora-base:forGroup <http://data.knora.org/groups/[UUID]> ;
     knora-base:hasPermissions "ProjectGroupAdminRestrictedPermission <http://data.knora.org/group
```

Administrative permission restricting resource creation for a group:

```
<http://data.knora.org/permissions/[UUID]> rdf:type knora-base:AdministrativePermission ;
     knora-base:forProject <http://data.knora.org/projects/[UUID]> ;
     knora-base:forGroup <http://data.knora.org/groups/[UUID]> ;
     knora-base:hasPermissions "ProjectResourceCreateRestrictedPermission <http://www.knora.org/o
```

Default object access permission on a 'ProjectMember' group:

```
<http://data.knora.org/permissions/[UUID]> rdf:type knora-base:DefaultObjectAccessPermission ;
     knora-base:forProject <http://data.knora.org/projects/images> ;
     knora-base:forGroup knora-base:ProjectMember ;
     knora-base:hasPermissions "CR knora-base:Creator|
                                M <http://data.knora.org/groups/[UUID]>|
                                V knora-base:KnownUser"^^xsd:string .
```

Default object access permission on a resource class:

```
<http://data.knora.org/permissions/[UUID]> rdf:type knora-base:DefaultObjectAccessPermission ;
     knora-base:forProject <http://data.knora.org/projects/[UUID]> ;
     knora-base:forResourceClass <http://www.knora.org/ontology/images#person> ;
     knora-base:hasPermissions "CR knora-base:Creator,knora-base:ProjectMember|
                                V knora-base:KnownUser,knora-base:UnknownUser"^^xsd:string .
```

Default object access permission on a resource property:

```
<http://data.knora.org/permissions/[UUID]> rdf:type knora-base:DefaultObjectAccessPermission ;
     knora-base:forProject <http://data.knora.org/projects/[UUID]> ;
     knora-base:forProperty <http://www.knora.org/ontology/images#lastname> ;
     knora-base:hasPermissions "D knora-base:ProjectMember,knora-base:Creator|
                               V knora-base:KnownUser,knora-base:UnknownUser"^^ .
```

Default object access permission on a resource class and property:

```
<http://data.knora.org/permissions/[UUID]> rdf:type knora-base:DefaultObjectAccessPermission ;
     knora-base:forProject <http://data.knora.org/projects/[UUID]> ;
     knora-base:forResourceClass <http://www.knora.org/ontology/images#person> ;
     knora-base:forProperty <http://www.knora.org/ontology/images#lastname> ;
     knora-base:hasPermissions "CR knora-base:Creator,knora-base:ProjectMember|
                                V knora-base:KnownUser,knora-base:UnknownUser"^^xsd:string .
```

Default object access permission on a knora-base property:

```
<http://data.knora.org/permissions/[UUID]> rdf:type knora-base:DefaultObjectAccessPermission ;
     knora-base:forProject knora-base:SystemProject ;
     knora-base:forProperty <http://www.knora.org/ontology/knora-base#hasStillImageFileValue> ;
     knora-base:hasPermissions "RV knora-base:UnknownUser|
                                V knora-base:KnownUser|
                                M knora-base:ProjectMember,knora-base:Creator"^^xsd:string .
```

A the time the user's `UserProfile` is queried, all permissions for all projects and groups the user is a member of are also queried. This information is then stored as an easy accessible object inside the `UserProfile`, being readily available wherever needed. As this is a somewhat expensive operation, built-in caching mechanism at different levels (e.g., UsersResponder, PermissionsResponder), will be applied.

Use Cases

UC01: Teaching a Class

Description: I'm teaching a class and I have the names and email addresses of all the students. I want to create a project, divide the students into groups (which will only be relevant to this project, e.g. one group for each section of the class), and put some students in each group. I don't want people to be able to join the project or the group by themselves.

Solution: The teacher creates different groups and adds users to those groups. Additionally, the teacher can give TA's *GroupAdmin* privileges, and let the TA's add students to the different groups.

UC02: Unibas Librarian

Description: I'm a Unibas librarian managing several archiving projects. I need to give everyone at the university permission to view all these projects. I want to create a group called *UnibasUser* that everyone with a Unibas email address will automatically belong to. Most of the resources in those projects can then grant view permission to *UnibasUser*. Or perhaps the group will be *SwitchUser*, for anyone at a Swiss university. Or something even broader.

Solution: These can be solved by creating *Smart Groups*, where the user can define what properties need to be set, so that an user is automatically part of this group. This will be implemented at a later time, as it is not trivial and should also include all special groups (e.g., KnownUser, ProjectMember, ProjectAdmin, etc.) that are currently hard-coded inside the system.

UC03: Crowdsourcing Project

Description: I'm doing a crowdsourcing project, which involves several different groups that work on different tasks. I'm hoping for thousands of users, and I'd like anyone to be able to join the project and add themselves to any group they want (as long as Knora verifies their email address), without needing approval from me.

Solution: This can be solved by allowing self-assignment to a group.

UC04: User "left" Knora

Description: An user who was an active collaborator, decides to "quit", and wants to delete his user.

Solution: The user's IRI is saved on each value change as part of the versioning mechanism. Exchanging the user's IRI in those places would count as 'rewriting history'. So deleting a user will not be possible, instead the user will be set as `not active`.

Webapi Components

For the management of *users*, *projects*, and *groups*, the Knora API following a resource centric approach, provides three endpoints corresponding to the three classes of objects that they have an effect on, namely:

- Users Endpoint: `http://server:port/v1/users` -> `knora-base:User`
- Projects Endpoint: `http://server:port/v1/projects` -> `knora-base:knoraProject`
- Groups Endpoint: `http://server:port/v1/groups` -> `knora-base:UserGroup`

All information regarding users, projects and groups is stored in the `http://www.knora.org/admin` named graph.

Users Endpoint

Create user:

- Required permission: none, self-registration is allowed

- Required information: username, given name, family name, email, password
- Optional information: phone
- Returns IRI of newly created user

Update user:

- Required permission: SystemAdmin / User
- Changeable information: username, given name, family name, email, password, phone

Delete user (-> update user):

- Required permission: SystemAdmin / User
- Effects property: `knora-base:isActiveUser` with value `true` or `false`

Example User Information stored in admin graph:

```
<http://data.knora.org/users/91e19f1e01> rdf:type knora-base:User ;
     knora-base:email "test@test.ch"^^xsd:string ;
     knora-base:givenName "Administrator"^^xsd:string ;
     knora-base:familyName "Admin"^^xsd:string ;
     knora-base:password "$2a$10$fTEr/xVjPq7UBAy1O6KWKOM1scLhKGeRQdR4GTA997QPqHzXv0MnW"^^xsd:stri
     knora-base:phone "123456" ;
     knora-base:preferredLanguage "de"^^xsd:string ;
     knora-base:status "true"^^xsd:boolean ;
     knora-base:isInProject <http://data.knora.org/projects/[UUID]> ;
     knora-base:isInSystemAdminGroup "true"^^xsd:boolean ;
     knora-base:isInProjectAdminGroup <http://data.knora.org/projects/[UUID]> ;
     knora-base:isInGroup <http://data.knora.org/groups/[UUID]> .
```

Projects Endpoint

Create project:

- Required permission: SystemAdmin
- Required information: projectShortname (unique; used for named graphs), projectBasepath
- Optional information: projectLongname, projectDescription, projectKeyword, projectLogo
- Returns IRI of newly created project
- **Effects:**
  - create project
  - create group named *ProjectAdmin*, give group *hasProjectAllAdminPermission* and *hasProjectResourceCreateAllPermission*
  - create group named *ProjectMember*, give group *hasProjectResourceCreateAllPermission*, *knora-base:hasDefaultChangeRightsPermission* for *knora-base:Creator*, *knora-base:hasDefaultModifyPermission* for this *ProjectMember* group, and *knora-base:hasDefaultViewPermission* for *knora-base:KnownUser*

Update project information:

- Required permission: SystemAdmin / ProjectAdmin
- Changeable information: longname, description
- Effects property: `knora-base:projectLongname`, `knora-base:description`

Add/remove user to/from project:

- Required permission: SystemAdmin / ProjectAdmin / User (if project self-assignment is enabled)
- Required information: project IRI, user IRI

- Optional information: admin status
- Effects: `knora-base:isInProject` and `knora-base:isInGroup` named `ProjectMember` of current project

Delete/Un-Delete project (-> update project):

- Required permission: SystemAdmin / ProjectAdmin
- Effects property: `knora-base:isActiveProject` with value `true` or `false`

Enable/disable self-join:

- Required permission: SystemAdmin / ProjectAdmin
- Effects property: `knora-base:hasSelfAssignmentEnabled` with value `true` or `false`

Example Project Information stored in admin named graph:

```
<http://data.knora.org/projects/[UUID]>
     rdf:type knora-base:knoraProject ;
     knora-base:projectBasepath "/imldata/SALSAH-TEST-01/images" ;
     knora-base:projectShortname "images" ;
     knora-base:projectLongname "Images Collection Demo" ;
     knora-base:projectOntologyGraph "http://www.knora.org/ontology/images" ;
     knora-base:projectDataGraph "http://www.knora.org/data/images" ;
     knora-base:isActiveProject "true"^^xsd:boolean ;
     knora-base:hasSelfJoinEnabled "false"^^xsd:boolean .
```

Groups Endpoint

Create group:

- Required permission: SystemAdmin / hasProjectAllAdminPermission / hasProjectAllGroupAdminPermission
- Required information: group name (unique inside project), project IRI
- Optional information: group description
- Returns IRI of newly created group

Update group information:

- Required permission: SystemAdmin / hasProjectAllAdminPermission / hasProjectAllGroupAdminPermission / hasProjectRestrictedGroupAdminPermission (for this group)
- Changeable information: name, group description
- Effects property: `<http://xmlns.com/foaf/0.1/name>`, `knora-base:groupDescription`

Add/remove user to/from 'normal' group (not *SystemAdmin* or *ProjectAdmin*):

- Required permission: SystemAdmin / hasProjectAllAdminPermission / hasProjectAllGroupAdminPermission / hasProjectRestrictedGroupAdminPermission (for this group) / User (if group self-assignment is enabled)
- Required information: group IRI, user IRI
- Effects: `knora-base:isInGroup`

Add/remove user to/from SystemAdmin group:

- Required permission: SystemAdmin
- Required information: group IRI (http://www.knora.org/ontology/knora-base#SystemAdmin), user IRI
- Effects: `knora-base:isInGroup`

Add/remove user to/from ProjectAdmin group:

- Required permission: SystemAdmin, ProjectAdmin
- Required information: project IRI, group IRI, user IRI
- Effects: `knora-base:isInGroup`

Enable/disable self-join:

- Required permission: SystemAdmin / hasProjectAllAdminPermission / hasProjectAllGroupAdminPermission / hasProjectRestrictedGroupAdminPermission (for this group)
- Effects property: `knora-base:hasSelfAssignmentEnabled` with value `true` or `false`

Add/change administrative permissions to a group:

- Required permission: SystemAdmin / hasProjectAllAdminPermission / hasProjectRightsAdminPermission
- Effects property: resource creation permissions, project administration permissions, default permissions

Delete group:

- Required permission: SystemAdmin / hasProjectAllAdminPermission
- Effect: `knora-base:isInGroup` / removes group from any object permissions

Example Group Information stored in admin named graph:

```
<http://data.knora.org/groups/[UUID]>
     rdf:type knora-base:UserGroup ;
     knora-base:groupName "Name of the group" ;
     knora-base:groupDescription "A description of the group" ;
     knora-base:belongsToProject <http://data.knora.org/projects/[UUID]> ;
     knora-base:status "true"^^xsd:boolean ;
     knora-base:hasSelfJoinEnabled "false"^^xsd:boolean .
```

Redesign / Questions June 2016

Permissions constrained to groups

- Why this constraint?
- => This is just the way we are doing it. Makes it a bit simpler.

Resource owner permission to disruptive

- knora-base:attachedToUser gives owner status to the person who created the resource.
- **Proposed change:** remove this altogether or make institution/project owner of the resource.
- Should hiwis be "owners" of resources they create on behalf of their professor?
- If the creator should have max permission, then give it explicitly.
- => Owner will be renamed to creator. We need this for provenance. Does not give any permissions automatically. The permissions depend on what is defined for the project and the *creator* smart group.

Resource creation permission to course

- being part of a projects gives resource creation permission. What if some project members are not allowed to create new resources (or only certain types; Lumiere Lausanne requirement), but are only allowed to change existing resources?
- => These kind of permissions can be set on groups. A project can have different groups, giving different kind of permissions.

Support Default Permissions

- Allow for a project to define permissions that a newly created resource inside a project should receive (current Salsah behavior)
- Lumiere Lausanne requirement
- => Will be allowed.

Groups

- Do groups belong to projects, i.e. are they seen as extensions to projects?
- Does someone need to be part of a project to belong to a group of that project?
- => Every group needs to belong to a project. No GroupAdmins. ProjectAdmins with additional GroupAdmin permissions.

root

- Should the 'root' / SystemAdmin user have 'implicitly' or 'explicitly' all permissions?
- => Has implicitly all permissions.
- Does the has all permissions also extend to projects? Is the root user going to be part of every project? If yes, then again implicitly or explicitly?
- => Since 'root' / SystemAdmin already has all permissions, doesn't realy matter if part of a project or group

Ivan's Use Case

- The system administrator creates the project and sets Ivan as the project administrator. As the project administrator, I have all permissions on all objects (Resources/Values; Project Groups) belonging to the project (knora-base:attachedToProject). Nobody outside of the project should be allowed to see anything that is created as part of Ivan's project. He wants to be able to create two groups: *Reviewer*, *Creator*. The *Reviewer* group should only give *read-access* to someone inside the group to resources pointing to this group, but allow the creation of annotations. Further, annotations should only be readable by users inside the *Reviewer* group. The *Creator* group should give a user create permission and modify permission on the objects the user has created. Any resources created belong to the project. The *Creator* group is meant for contributors helping out with the project, e.g., Hiwis.
- => Covered

Lausanne Projects

- A project wants to restrict the permissions of newly created resources to a fixed set
- => Covered. Will be able do define 'default permissions' and restrict the creation of new resources to these permissions
- This means for the current implementation, that any permissions supplied during the resource creation request need to be checked and if needed overriden.
- => Covered. Also in the new design, the backend will need to always check the suplied permissions for newly created resources as we cannot ve sure that the GUI will behave correctly (e.g., many different "Salsah" implementations)
- Restrict creation/access of certain classes of resources to certain groups, e.g., group A is able to create/access resources of class A but not of class B.
- => Covered. Will be able to give a certain group only create permission for specific classes

Results

- *Owner* renamed to *Creator*
- Some permissions are attached to groups (e.g., Add Resource (Class), Modify Ontology, etc.), and some are attached to resources (e.g., this group has read/modify permission, etc.)
- Ontologien Benutzung einschränken (nur auf bestimmte Gruppen, oder frei zur Verfügung)
- System Admin Rechte implizit
- Gruppen immer an Projekt gebunden

- Keine Gruppen-Admins. Soll über Rollen vom Projekt-Admin geregelt werden können.

Plans for Knora API v2

Naming

In API v1, the same data types are named inconsistently (`resinfo`, `res_info`) or unclearly (`value_restype` is actually a label). Version 2 should adopt a clear, consistent naming convention.

Structure

API v1 sometimes uses parallel array structures to represent multiple complex objects, e.g. the values of resource properties or the items in a resource's context. Version 2 should use nested structures instead.

Redundancy

Some information in API v1 is presented redundantly, e.g. `resinfo` and `resdata`, and the `__location__` property. This should be cleaned up.

Efficiency

Some queries, like the resource context query, produce so much data that they cannot be made efficient. We should consider breaking up these API calls into smaller chunks.

It should be possible to customise GET requests so that they return only as much data as the user wants. For example, in Fedora 4's equivalent of the `resources` route with `reqtype=full`, you can specify whether you want child resources and incoming references. This would enable clients to request only the information they actually need, improving performance and reducing server load.

Suitability for non-GUI applications

When returning the 'full' information about a resource, the API currently includes valueless properties to reflect the possible properties in the resource type (if a property has no value, or only has values that the user isn't allowed to see), *unless* a property already has a value that the user isn't allowed to see, and its cardinality is `MustHaveOne` or `MayHaveOne`. This makes sense from the point of a GUI: the valueless properties are there to indicate that the user could add values for those properties. If a property already has a value and its cardinality is `MustHaveOne` or `MayHaveOne`, the user can't add a value for it, so there is no reason to include it.

In version 2, it might make more sense to separate information about resource types from information about resources (rather than mixing these two kinds of information together in one API response), and to separate *displaying* a resource from indicating which properties a particular user can add.

Working with multiple projects

The user will be able to choose which project to use for an update.

We will handle the case where Project A defines a resource class X, and Project B declares a resource class Y with additional properties, asserting that X is a subclass of Y, so that users in Project B can add these extra properties to resources that already exist in Project A. Users in project A will want to be able to ignore the extra properties from Y, or optionally see and use them.

The ontology responder will distinguish between definitions in the active project's named graph and definitions added elsewhere, so the user can choose to see just what's defined in their own project or to include definitions from elsewhere.

Annotating values

In API v1, only resources can be annotated. In v2, it will also be possible to annotate values.

Typing

Each data item should have a consistent data type. In an JSON object, the same name should always contain a value of the same type. Numbers should be represented as numbers rather than as strings.

JSON-LD

Consider using JSON-LD to specify data types and semantics within API responses, instead of providing separate JSON schemas.

The basic idea is just that your API can return JSON like this:

```
{
  "book": {
    "id": "http://data.knora.org/c5058f3a"
    "title": "Zeitglöcklein des Lebens und Leidens Christi"
  }
}
```

and it can also include a "context" (which can be embedded in the same JSON, or provided as the URL of a separate JSON document) specifying that `book` is an `incunabula:book`, and that `title` means `dc:title`. So everything in an API response can have semantics and type information specified. The idea is that the keys in the JSON stay short and readable, so someone writing a simple browser-based client can write `book.title` in JavaScript and it will work. At the same time, a more complex, automated client can easily get the semantic and type information.

Developing the Knora API Server

Overview

Developing for the Knora API server requires a complete local installation of Knora. The different parts are:

1. The cloned Knora Github repository

2. One of the supplied triple stores in the Knora Github repository (GraphDB-SE 8 or Fuseki 3).
3. SIPI by building from source or using the docker image

Knora Github Repository

```
$ git clone https://github.com/dhlab-basel/Knora
```

Triple Store

A number of triplestore implementations are available, including free software as well as proprietary options. The Knora API server is tested and configured to work out of the box with the following triplestores:

- Ontotext GraphDB, a high-performance, proprietary triplestore. The Knora API server is tested with GraphDB Standard Edition and GraphDB Free (which is proprietary but available free of charge).
- Apache Jena, which is free software. Knora comes bundled with Jena and with its standalone SPARQL server, Fuseki.

See the chapters on *Starting Fuseki 3* and *Starting GraphDB-SE* for more details.

SIPI

Build SIPI Docker Image

The Sipi docker image needs to be build by hand, as it requires the Kakadu distribution.

To build the image, and push it to the docker hub, follow the following steps:

```
$ git clone https://github.com/dhlab-basel/docker-sipi
(copy the Kakadu distribution ``v7_8-01382N.zip`` to the ``docker-sipi`` directory)
$ docker build -t dhlabbasel/sipi
$ docker run --name sipi --rm -it -p 1024:1024 dhlabbasel/sipi
(Ctrl-c out of terminal will stop and delete container)
$ docker push dhlabbasel/sipi
```

Pushing the image to the docker hub requires prior authentication with `$ docker login`. The user needs to be registered on `hub.docker.com`. Also, the user needs to be allowed to push to the `dblabbasel` organisation.

Running SIPI

To use the docker image stored locally or on the docker hub repository type:

```
$ docker run --name sipi -d -p 1024:1024 dhlabbasel/sipi
```

This will create and start a docker container with the `dhlabbasel/sipi` image in the background. The default behaviour is to start Sipi by calling the following command:

```
$ /sipi/local/bin/sipi -config /sipi/config/sipi.knora-test-config.lua
```

To override this default behaviour, start the container by supplying another config file:

```
$ docker run --name sipi \
             -d \
             -p 1024:1024 \
             dhlabbasel/sipi \
             /sipi/local/bin/sipi -config /sipi/config/sipi.config.lua
```

You can also mount a directory (the local directory in this example), and use a config file that is outside of the docker container:

```
$ docker run --name sipi \
             -d \
             -p 1024:1024 \
             -v $PWD:/localdir \
             dhlabbasel/sipi \
             /sipi/local/bin/sipi -config /localdir/sipi.knora-test-config.lua
```

Starting Fuseki 3

Locally

Inside the Knora API server git repository, there is a folder called `triplestores/fuseki` containing a script named `fuseki-server`. All needed configuration files are in place. To start Fuseki 3, just run this script:

```
$ ./fuseki-server
```

Inside Docker

We can use the `dhlabbasel:fuseki` docker image from docker hub:

```
$ docker run --rm -it -p 3030:3030 dhlabbasel/fuseki
```

Starting GraphDB-SE

Inside the Knora API server git repository, there is a folder called `/triplestores/graphdb-se` containing the latest supported version of the GraphDB-SE distribution archive.

Running Locally

Unzip `graphdb-se-x.x.x-dist.zip` to a place of your choosing and run the following:

```
$ cd /to/unziped/location
$ ./bin/graphdb -Dgraphdb.license.file=/path/to/GRAPHDB_SE.license
```

Running inside Docker

Important Steps

To be able to successfully run GraphDB inside docker two important steps need to be done beforhand:

1. Install Docker from http://docker.com.
2. Copy the GraphDB-SE license file into a folder of you choosing and name it `GRAPHDB_SE.license`. We will mount this folder into the docker container, so that the license can be used by GraphDB running inside the container.

Usage

```
 $ docker run --rm -it -v /path/to/license/folder:/external -p 7200:7200 dhlabbasel/graphdb


- ``--rm`` removes the container as soon as you stop it
- ``-p`` forwards the exposed port to your host (or if you use boot2docker to this IP)
- ``-it`` allows interactive mode, so you see if something get's deployed
```

After the GraphDB inside the docker container has started, you can find the GraphDB workbench here: http://localhost:7200

Above, we create and start a transient container (`--rm` flag). To create a container that we can stop and start again at a later time, follow the following steps:

```
 $ docker run --name graphdb -d -t -v /path/to/license/folder:/external -p 7200:7200 dhlabbasel/gr

 (to see the console output, attach to the container; to detach press Ctrl-c)
 $ docker attach graphdb

 (to stop the container)
 $ docker stop graphdb

 (to start the container again)
 $ docker start graphdb

 (to remove the container; needs to be stopped)
 $ docker rm graphdb

- ``--name`` give the container a name
- ``-d`` run container in background and print container ID
- ``-t`` allocate a pseudo TTY, so you see the console output
- ``-p`` forwards the exposed port to your host
```

Build Process

TODO: complete this file.

- SBT
- Using GraphDB for development and how to initializing the 'knora-test-unit' repository
- Using Fuseki for development

Building and Running

Using Fuseki

Start the provided Fuseki triplestore:

```
$ cd KNORA_PROJECT_DIRECTORY/triplestores/fuseki
$ ./fuseki-server
```

Then in another terminal, load some test data into the triplestore:

```
$ cd KNORA_PROJECT_DIRECTORY/webapi/scripts
$ ./fuseki-load-test-data.sh
```

Then go back to the webapi root directory and use SBT to start the API server:

```
$ cd KNORA_PROJECT_DIRECTORY/webapi
$ sbt
> compile
> re-start
```

To shut down the Knora API server:

```
> re-stop
```

Using GraphDB

The archive with the newest supported version of the GraphDB-SE triplestore is provided under `'triplestores/graphdb-se'`. Please keep in mind, that GraphDB-SE must be licensed separately by the user, and that no license file is provided in the repository. GraphDB-SE will not run without a license file.

Unzip `graphdb-se-x.x.x-dist.zip` to a place of your choosing and run the following, to start graphdb:

```
$ cd /to/unziped/location
$ ./bin/graphdb -Dgraphdb.license.file=/path/to/GRAPHDB_SE.license
```

After the GraphDB inside the docker container has started, you can find the GraphDB workbench here: http://localhost:7200

Then in another terminal, load some test data into the triplestore:

```
$ cd KNORA_PROJECT_DIRECTORY/webapi/scripts
$ ./graphdb-se-local-init-knora-test.sh
```

Then go back to the webapi root directory and use SBT to start the API server:

```
$ cd KNORA_PROJECT_DIRECTORY/webapi
$ sbt
> compile
> re-start
```

To shut down the Knora API server:

```
> re-stop
```

Running the automated tests

Running Tests with Fuseki

Make sure you've started Fuseki as shown above. Then at the SBT prompt:

```
> fuseki:test
```

Running Tests with GraphDB

Make sure GraphDB is running (as described earlier).

Then in another terminal, initialise the repository used for automated testing:

```
$ cd KNORA_PROJECT_DIRECTORY/webapi/scripts
$ ./graphdb-se-local-init-knora-test-unit.sh
```

Run the automated tests from sbt:

```
> graphdb:test
```

Load Testing on Mac OS X

To test the Knora API server with many concurrent connections on Mac OS X, you will need to adjust some kernel parameters to allow more open connections, to recycle ephemeral ports more quickly, and to use a wider range of ephemeral port numbers. The script `webapi/scripts/os-x-kernel-test-config.sh` will do this.

Continuous Integration

For continuous integration testing, we use Travis-CI. Every commit pushed to the git repository or every pull request, triggers the build. Additionaly, in Github there is a litle checkmark beside every commit, signaling the status of the build (successful, unsucessful, ongoing).

The build that is executed on Travis-CI is defined in `.travis.yml` situated in the root folder of the project, and looks like this:

```
1  dist: trusty
2  sudo: required
3  git:
4      depth: 1
5  language: scala
6  scala:
7  - 2.12.1
8  jdk:
9  - oraclejdk8
10 cache:
11   directories:
12     - $HOME/.ivy2
13 script:
14 - cd triplestores/fuseki/ && ./fuseki-server &
15 - cd webapi/ && sbt test
16 notifications:
17   slack:
18     secure: AJZARDC7P6bwjFwk6gpe+p2ozLj+bH3h83PapfCTL0xi7frHd4y6/jXOs9ac+m7ia5FlnzgBxrf0lmaE+IkqlI
```

It basically means:

- use the virtual machine based environment (line 1)
- checkout git with a shorter history (lines 2-3)
- add scala libraries (lines 4-6)
- add oracle jdk version 8 (lines 7-8)
- cache some directories between builds to make it faster (line 9-11)
- start fuseki and afterwards start all tests (lines 12-14)
- send notification to our slack channel (lines 15-17)

SBT Build Configuration

```
import sbt._
import sbt.Keys._
import spray.revolver.RevolverPlugin._
import NativePackagerHelper._

connectInput in run := true
```

```
// Bring the sbt-aspectj settings into this build
//aspectjSettings

lazy val webapi = (project in file(".")).
        configs(
            FusekiTest,
            FusekiTomcatTest,
            GraphDBTest,
            GraphDBFreeTest,
            SesameTest,
            EmbeddedJenaTDBTest,
            IntegrationTest
        ).
        settings(webApiCommonSettings:  _*).
        settings(inConfig(FusekiTest)(
            Defaults.testTasks ++ Seq(
                fork := true,
                javaOptions ++= javaFusekiTestOptions,
                testOptions += Tests.Argument("-oDF") // show full stack traces and test case dura
            )
        ): _*).
        settings(inConfig(FusekiTomcatTest)(
            Defaults.testTasks ++ Seq(
                fork := true,
                javaOptions ++= javaFusekiTomcatTestOptions,
                testOptions += Tests.Argument("-oDF") // show full stack traces and test case dura
            )
        ): _*).
        settings(inConfig(GraphDBTest)(
            Defaults.testTasks ++ Seq(
                fork := true,
                javaOptions ++= javaGraphDBTestOptions,
                testOptions += Tests.Argument("-oDF") // show full stack traces and test case dura
            )
        ): _*).
        settings(inConfig(GraphDBFreeTest)(
            Defaults.testTasks ++ Seq(
                fork := true,
                javaOptions ++= javaGraphDBFreeTestOptions,
                testOptions += Tests.Argument("-oDF") // show full stack traces and test case dura
            )
        ): _*).
        settings(inConfig(SesameTest)(
            Defaults.testTasks ++ Seq(
                fork := true,
                javaOptions ++= javaSesameTestOptions,
                testOptions += Tests.Argument("-oDF") // show full stack traces and test case dura
            )
        ): _*).
        settings(inConfig(EmbeddedJenaTDBTest)(
            Defaults.testTasks ++ Seq(
                fork := true,
                javaOptions ++= javaEmbeddedJenaTDBTestOptions,
                testOptions += Tests.Argument("-oDF") // show full stack traces and test case dura
            )
        ): _*).
        settings(inConfig(IntegrationTest)(
            Defaults.itSettings ++ Seq(
                fork := true,
                javaOptions ++= javaIntegrationTestOptions,
                testOptions += Tests.Argument("-oDF") // show full stack traces and test case dura
            )
```

```
        ): _*).
        settings(
            libraryDependencies ++= webApiLibs,
            scalacOptions ++= Seq("-feature", "-unchecked", "-deprecation", "-Yresolve-term-confli
            logLevel := Level.Info,
            fork in run := true,
            javaOptions in run ++= javaRunOptions,
            //javaOptions in run <++= AspectjKeys.weaverOptions in Aspectj,
            //javaOptions in Revolver.reStart <++= AspectjKeys.weaverOptions in Aspectj,
            mainClass in (Compile, run) := Some("org.knora.webapi.Main"),
            fork in Test := true,
            javaOptions in Test ++= javaTestOptions,
            parallelExecution in Test := false,
            // enable publishing the jar produced by `sbt it:package`
            publishArtifact in (IntegrationTest, packageBin) := true
        ).
        settings( // enable deployment staging with `sbt stage`
          mappings in Universal ++= {
            // copy the scripts folder
            directory("scripts") ++
            // copy configuration files to config directory
            contentOf("src/main/resources").toMap.mapValues("config/" + _)
          },
          // add 'config' directory first in the classpath of the start script,
          scriptClasspath := Seq("../config/") ++ scriptClasspath.value,
          // add license
          licenses := Seq(("GNU AGPL", url("https://www.gnu.org/licenses/agpl-3.0"))),
          // need this here, but why?
          mainClass in Compile := Some("org.knora.webapi.Main")
        ).
        settings(Revolver.settings: _*).
        enablePlugins(SbtTwirl). // Enable the SbtTwirl plugin
        enablePlugins(JavaAppPackaging) // Enable the sbt-native-packager plugin

lazy val webApiCommonSettings = Seq(
    organization := "org.knora",
    name := "webapi",
    version := "0.1.0-beta",
    ivyScala := ivyScala.value map { _.copy(overrideScalaVersion = true) },
    scalaVersion := "2.12.1"
)

lazy val akkaVersion = "2.4.16"
lazy val akkaHttpVersion = "10.0.3"

lazy val webApiLibs = Seq(
    // akka
    "com.typesafe.akka" %% "akka-actor" % akkaVersion,
    "com.typesafe.akka" %% "akka-agent" % akkaVersion,
    "com.typesafe.akka" %% "akka-stream" % akkaVersion,
    "com.typesafe.akka" %% "akka-slf4j" % akkaVersion,
    "com.typesafe.akka" %% "akka-http" % akkaHttpVersion,
    "com.typesafe.akka" %% "akka-http-xml" % akkaHttpVersion,
    "com.typesafe.akka" %% "akka-http-spray-json" % akkaHttpVersion,

    "org.scala-lang.modules" %% "scala-xml" % "1.0.6",

    // testing
    "org.scalatest" %% "scalatest" % "3.0.0" % "test",
    //CORS support
    "ch.megard" %% "akka-http-cors" % "0.1.10",
    // jena
    "org.apache.jena" % "apache-jena-libs" % "3.0.0" exclude("org.slf4j", "slf4j-log4j12"),
```

```
    "org.apache.jena" % "jena-text" % "3.0.0" exclude("org.slf4j", "slf4j-log4j12"),
    // http client
    // "net.databinder.dispatch" %% "dispatch-core" % "0.11.2",
    // logging
    "com.typesafe.scala-logging" %% "scala-logging" % "3.5.0",
    "ch.qos.logback" % "logback-classic" % "1.1.7",
    // input validation
    "commons-validator" % "commons-validator" % "1.4.1",
    // authentication
    "org.bouncycastle" % "bcprov-jdk15on" % "1.56",
    "org.springframework.security" % "spring-security-core" % "4.2.1.RELEASE",
    // caching
    "net.sf.ehcache" % "ehcache" % "2.10.0",
    // monitoring - disabled for now
    //"org.aspectj" % "aspectjweaver" % "1.8.7",
    //"org.aspectj" % "aspectjrt" % "1.8.7",
    //"io.kamon" %% "kamon-core" % "0.5.2",
    //"io.kamon" %% "kamon-spray" % "0.5.2",
    //"io.kamon" %% "kamon-statsd" % "0.5.2",
    //"io.kamon" %% "kamon-log-reporter" % "0.5.2",
    //"io.kamon" %% "kamon-system-metrics" % "0.5.2",
    //"io.kamon" %% "kamon-newrelic" % "0.5.2",
    // other
    //"javax.transaction" % "transaction-api" % "1.1-rev-1",
    "org.apache.commons" % "commons-lang3" % "3.4",
    "commons-io" % "commons-io" % "2.4",
    "commons-beanutils" % "commons-beanutils" % "1.9.2", // not used by us, but need newest versio
    "org.jodd" % "jodd" % "3.2.6",
    "joda-time" % "joda-time" % "2.9.1",
    "org.joda" % "joda-convert" % "1.8",
    "com.sksamuel.diff" % "diff" % "1.1.11",
    "org.xmlunit" % "xmlunit-core" % "2.1.1",
    // testing
    "com.typesafe.akka" %% "akka-testkit" % akkaVersion % "test, fuseki, fuseki-tomcat, graphdb, t
    "com.typesafe.akka" %% "akka-http-testkit" % akkaHttpVersion % "test, fuseki, fuseki-tomcat, g
    "com.typesafe.akka" %% "akka-stream-testkit" % akkaVersion % "test, fuseki, fuseki-tomcat, gra
    "org.scalatest" %% "scalatest" % "3.0.0" % "test, fuseki, fuseki-tomcat, graphdb, tdb, it",
    "org.eclipse.rdf4j" % "rdf4j-rio-turtle" % "2.0M3",
    "org.rogach" %% "scallop" % "2.0.5",
    "com.google.gwt" % "gwt-servlet" % "2.8.0",
    "net.sf.saxon" % "Saxon-HE" % "9.7.0-14"
)

lazy val javaRunOptions = Seq(
    // "-showversion",
    "-Xms2048m",
    "-Xmx4096m"
    // "-verbose:gc",
    //"-XX:+UseG1GC",
    //"-XX:MaxGCPauseMillis=500"
)

lazy val javaTestOptions = Seq(
    // "-showversion",
    "-Xms2048m",
    "-Xmx4096m"
    // "-verbose:gc",
    //"-XX:+UseG1GC",
    //"-XX:MaxGCPauseMillis=500",
    //"-XX:MaxMetaspaceSize=4096m"
)

lazy val FusekiTest = config("fuseki") extend(Test)
```

```
lazy val javaFusekiTestOptions = Seq(
    "-Dconfig.resource=fuseki.conf"
) ++ javaTestOptions

lazy val FusekiTomcatTest = config("fuseki-tomcat") extend(Test)
lazy val javaFusekiTomcatTestOptions = Seq(
    "-Dconfig.resource=fuseki-tomcat.conf"
) ++ javaTestOptions

lazy val GraphDBTest = config("graphdb") extend(Test)
lazy val javaGraphDBTestOptions = Seq(
    "-Dconfig.resource=graphdb.conf"
) ++ javaTestOptions

lazy val GraphDBFreeTest = config("graphdb-free") extend(Test)
lazy val javaGraphDBFreeTestOptions = Seq(
    "-Dconfig.resource=graphdb-free.conf"
) ++ javaTestOptions

lazy val SesameTest = config("sesame") extend(Test)
lazy val javaSesameTestOptions = Seq(
    "-Dconfig.resource=sesame.conf"
) ++ javaTestOptions

lazy val EmbeddedJenaTDBTest = config("tdb") extend(Test)
lazy val javaEmbeddedJenaTDBTestOptions = Seq(
    "-Dconfig.resource=jenatdb.conf"
) ++ javaTestOptions

// The 'IntegrationTest' config does not need to be created here, as it is a built-in config!
// The standard testing tasks are available, but must be prefixed with 'it:', e.g., 'it:test'
// The test need to be stored in the 'it' (and not 'test') folder. The standard source hierarchy
lazy val javaIntegrationTestOptions = Seq(
    "-Dconfig.resource=graphdb.conf"
) ++ javaTestOptions
```

Webapi Server Startup-Flags

The Webapi-Server can be started with a number of flags. These flags can be supplied either to the `reStart` or the `run` command in sbt, e.g.,:

```
$ sbt
> reStart flag
```

or

```
$sbt
> run flag
```

`loadDemoData` - Flag

When the webapi-server is started with the `loadDemoData` flag, then at startup, the data which is configured in `application.conf` under the `app.triplestore.rdf-data` key is loaded into the triplestore, and any data in the triplestore is removed beforehand.

Usage:

```
$ sbt
> reStart loadDemoData
```

`allowResetTriplestoreContentOperationOverHTTP` - Flag

When the webapi.server is started with the `allowResetTriplestoreContentOperationOverHTTP` flag, then the `v1/store/ResetTriplestoreContent` route is activated. This route accepts a `POST` request, with a json payload consisting of the following exemplary content:

```
[
  {
    "path": "../knora-ontologies/knora-base.ttl",
    "name": "http://www.knora.org/ontology/knora-base"
  },
  {
    "path": "../knora-ontologies/knora-dc.ttl",
    "name": "http://www.knora.org/ontology/dc"
  },
  {
    "path": "../knora-ontologies/salsah-gui.ttl",
    "name": "http://www.knora.org/ontology/salsah-gui"
  },
  {
    "path": "_test_data/ontologies/incunabula-onto.ttl",
    "name": "http://www.knora.org/ontology/incunabula"
  },
  {
    "path": "_test_data/all_data/incunabula-data.ttl",
    "name": "http://www.knora.org/data/incunabula"
  }
]
```

This content corresponds to the payload sent with the `ResetTriplestoreContent` message, defined inside the `org.knora.webapi.messages.v1.store.triplestoremessages` package. The `path` being the relative path to the `ttl` file which will be loaded into a named graph by the name of `name`.

Usage:

```
$ sbt
> reStart allowResetTriplestoreContentOperationOverHTTP
```

Setup IntelliJ for development of Knora

Create an IntelliJ Project for the Knora API Server

- Download and install IntelliJ IDEA.
- Follow the installation procedure and install the `Scala plugin`
- Import the `webapi` directory in the Knora source tree: `Import Project` -> Choose the option `module SBT`
- make sure that the tab size is set correctly to **4 spaces** (so you can use automatic code reformatting): `Preferences -> Code Style -> Scala`:

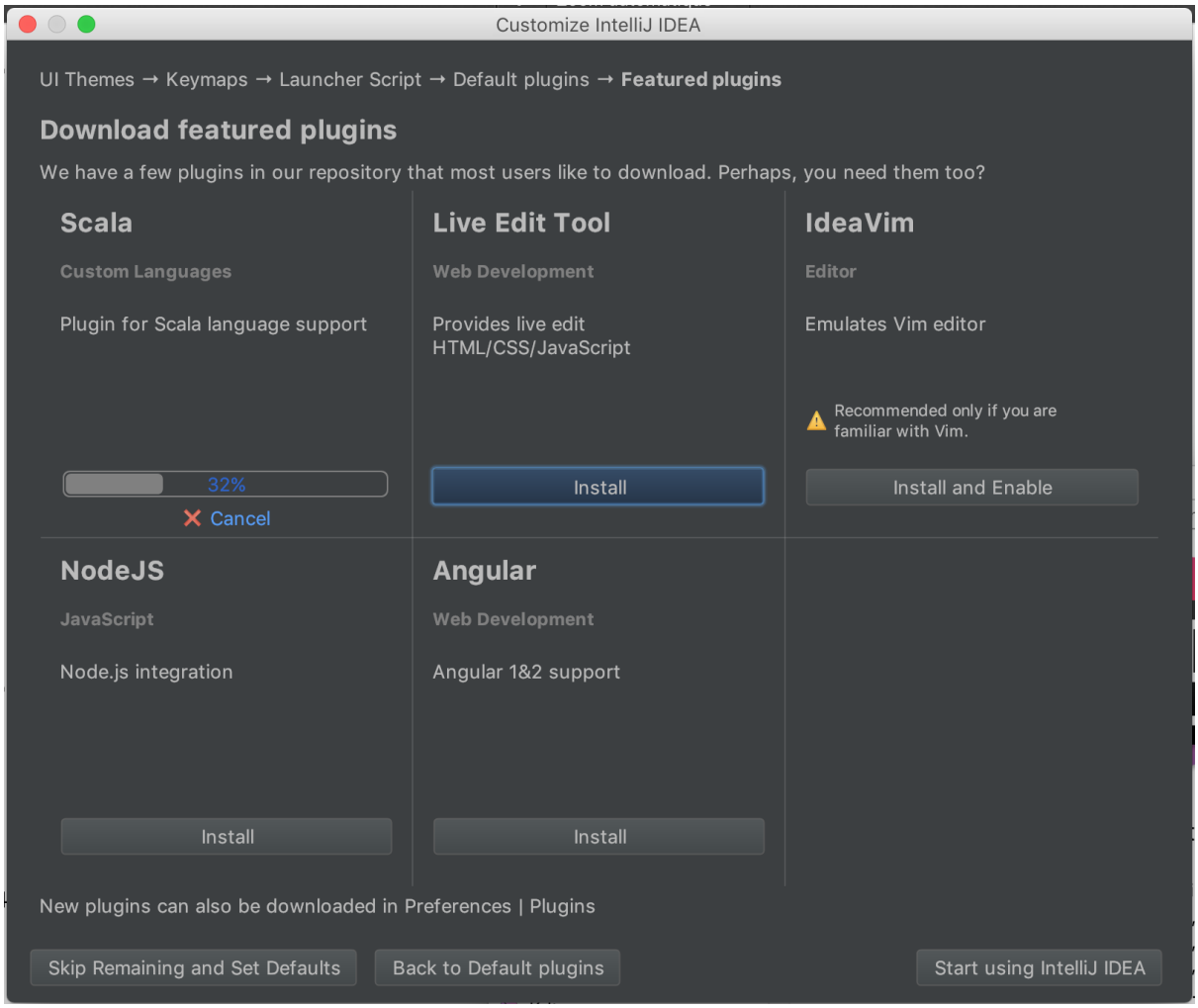


Fig. 4.2: screenshot 'Install Scala Plugin'

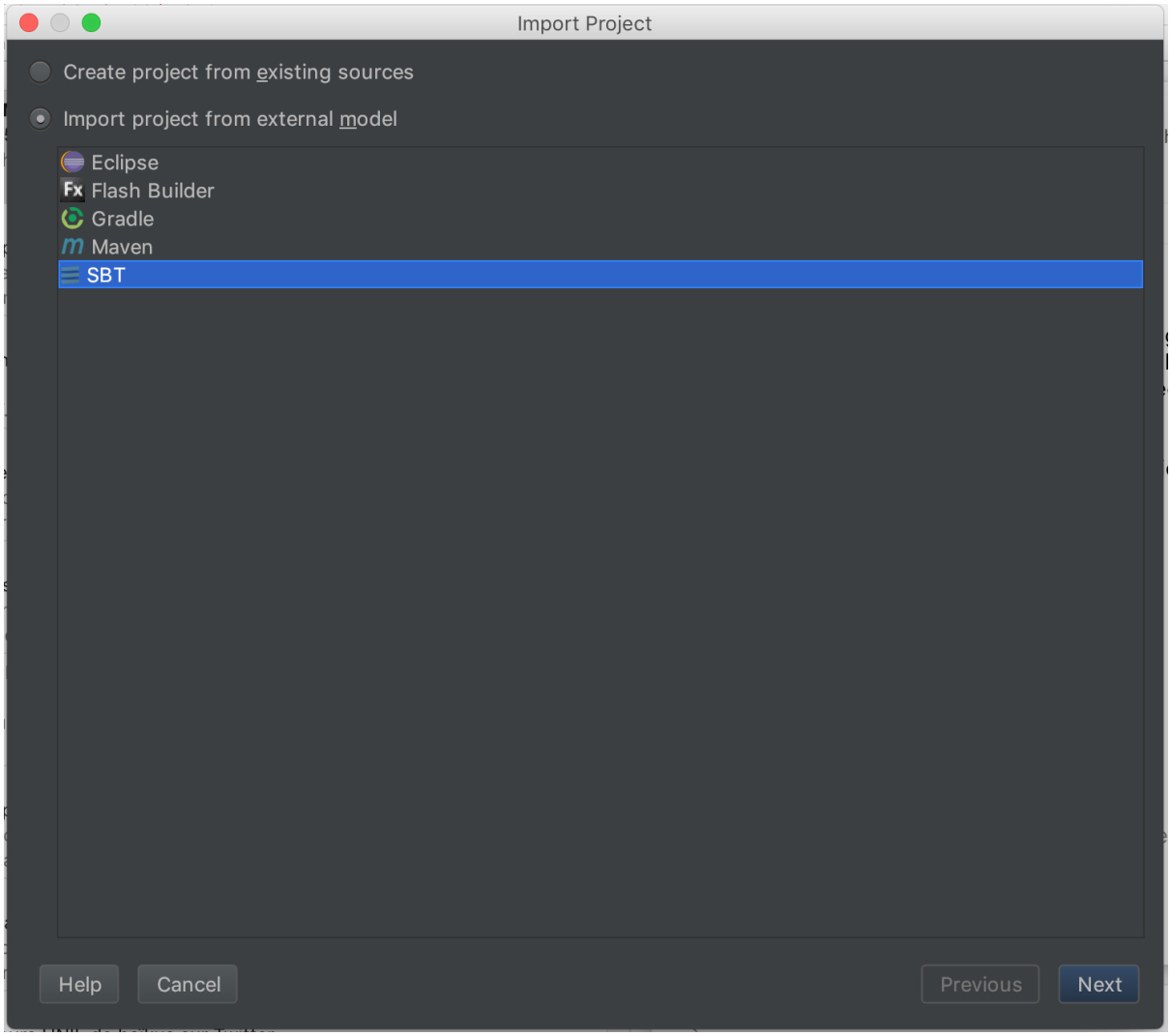


Fig. 4.3: screenshot 'import existing SBT project'

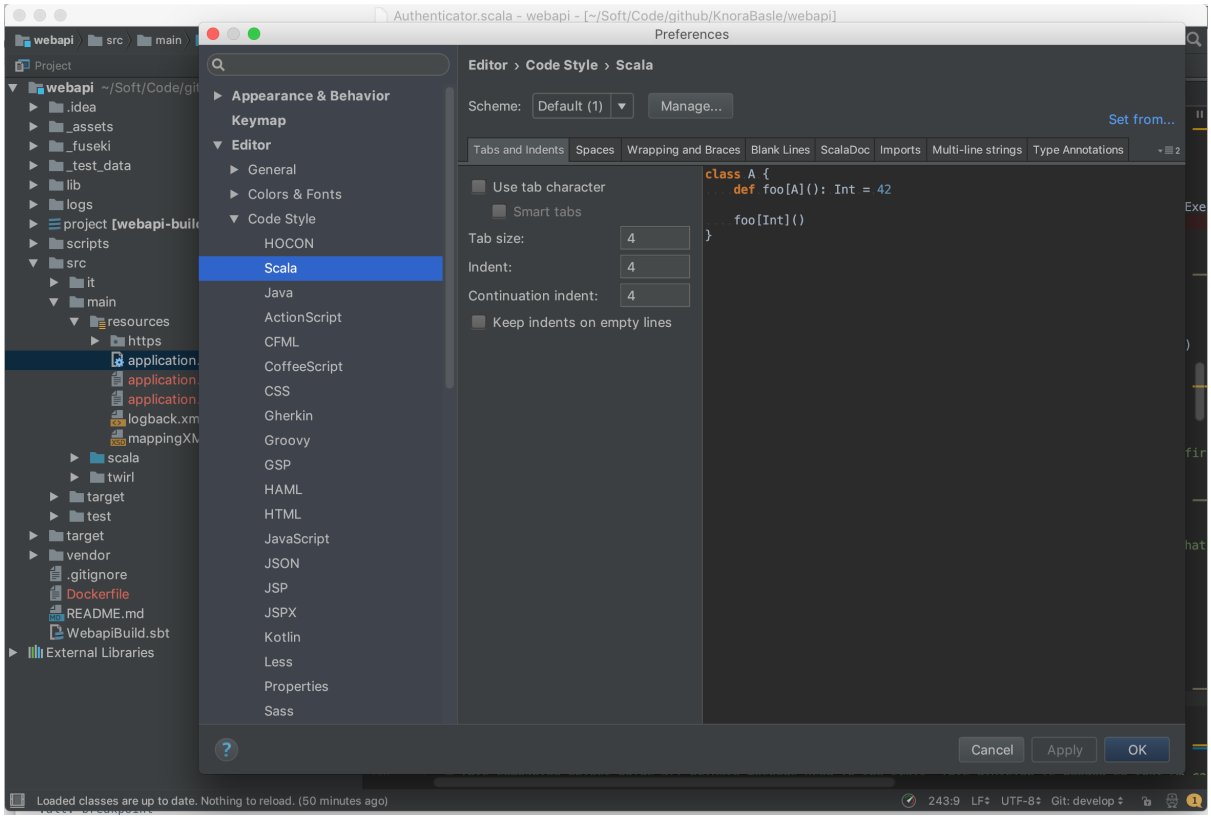


Fig. 4.4: screenshot 'setting tab size'

Twirl

By default, Intellij excludes some folders like the twirl template files. To include them, go to `Project Structure` and remove `target/scala-2.1*/twirl` from excluded folders. Then Intellij will correctly resolve the references to the template files.

Use IntelliJ IDEA's Debugger with the Knora API Server

- Create an application configuration:
- Click on the debugging symbol to start the application with a debugger attached
- Click on a line-number to add a breakpoint

Profile Knora Using VisualVM in IntelliJ

First, download and install VisualVM.

Then, in IntelliJ, under Preferences -> Plugins, search for the VisualVM Launcher, click on "Search in repositories", install the plugin, and restart IntelliJ. IntelliJ's toolbar should now contain a button with a green triangle on an orange circle, with the tooltip "Run with VisualVM":

You can use this button to run the class `org.knora.webapi.Main` and profile it in VisualVM. The first time you do this, IntelliJ will ask you for the path to the VisualVM executable. On macOS this is `/Applications/VisualVM.app/Contents/MacOS/visualvm`.

When VisualVM starts, it will open a window like this:

To use the profiler, click on the "Sampler" tab, then on the "CPU" button:

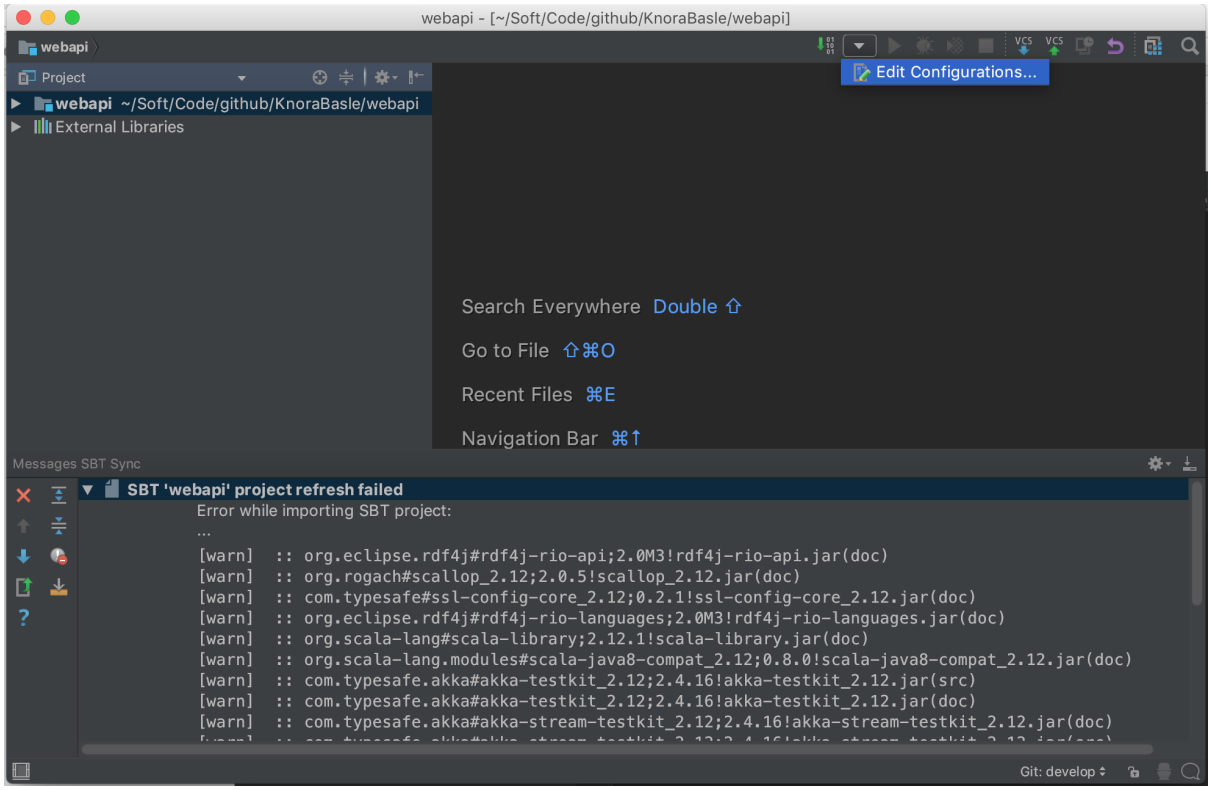


Fig. 4.5: screenshot 'edit application config'

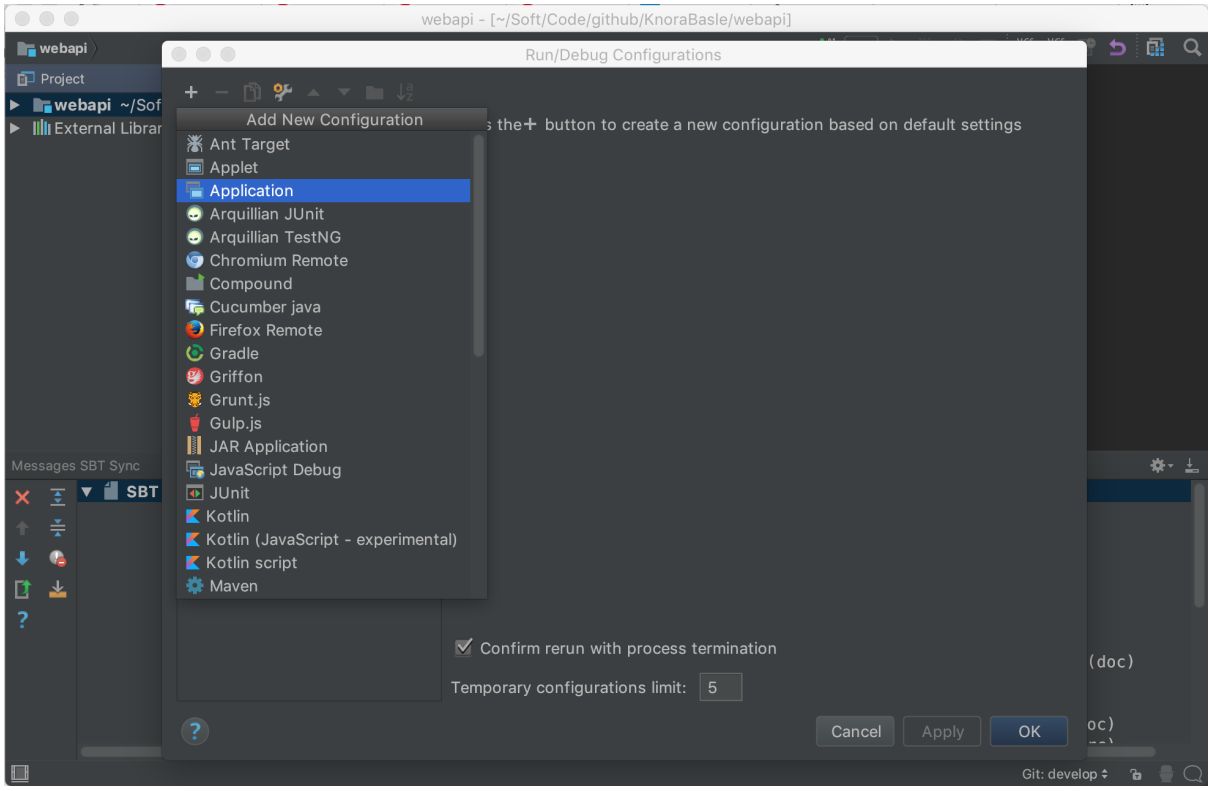


Fig. 4.6: screenshot 'create application configuration'

Fill in the configuration details:

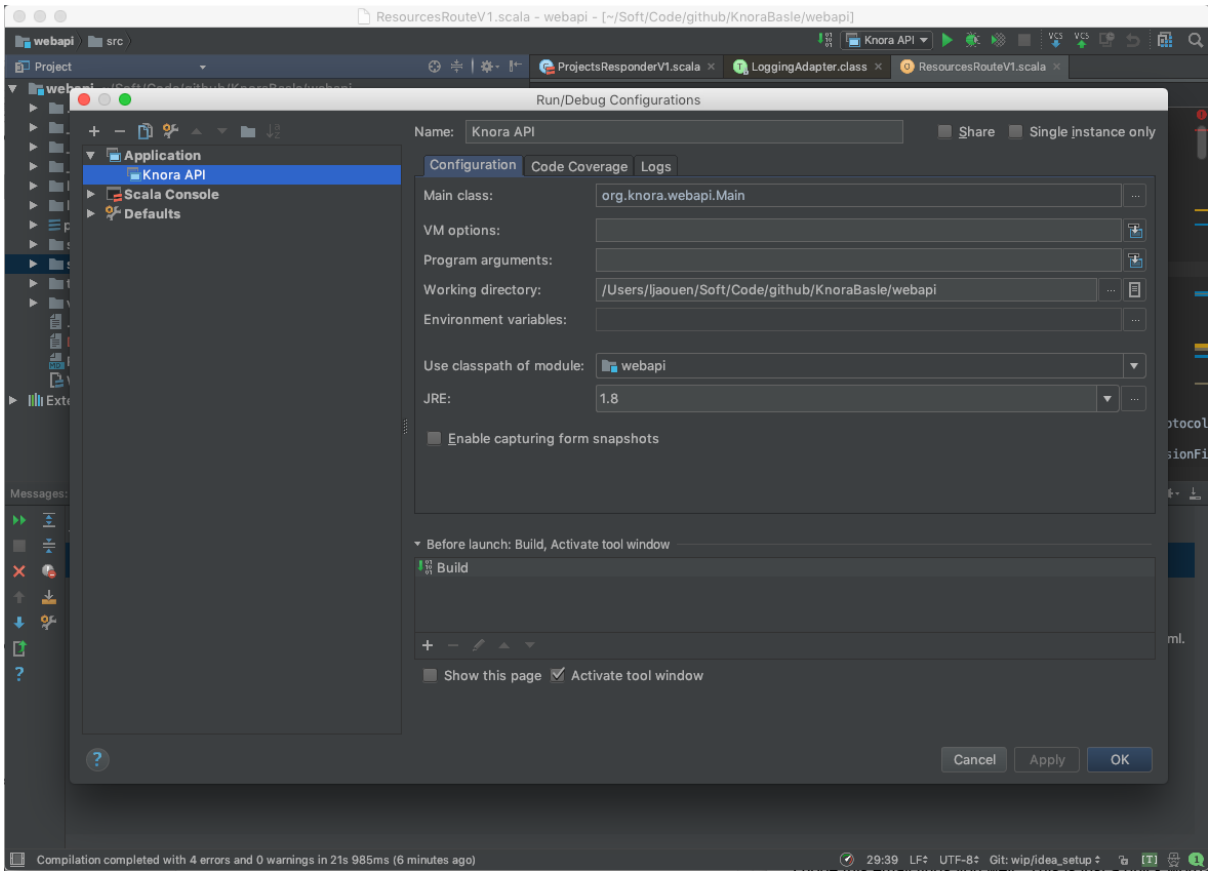


Fig. 4.7: screenshot ‘change application configuration’

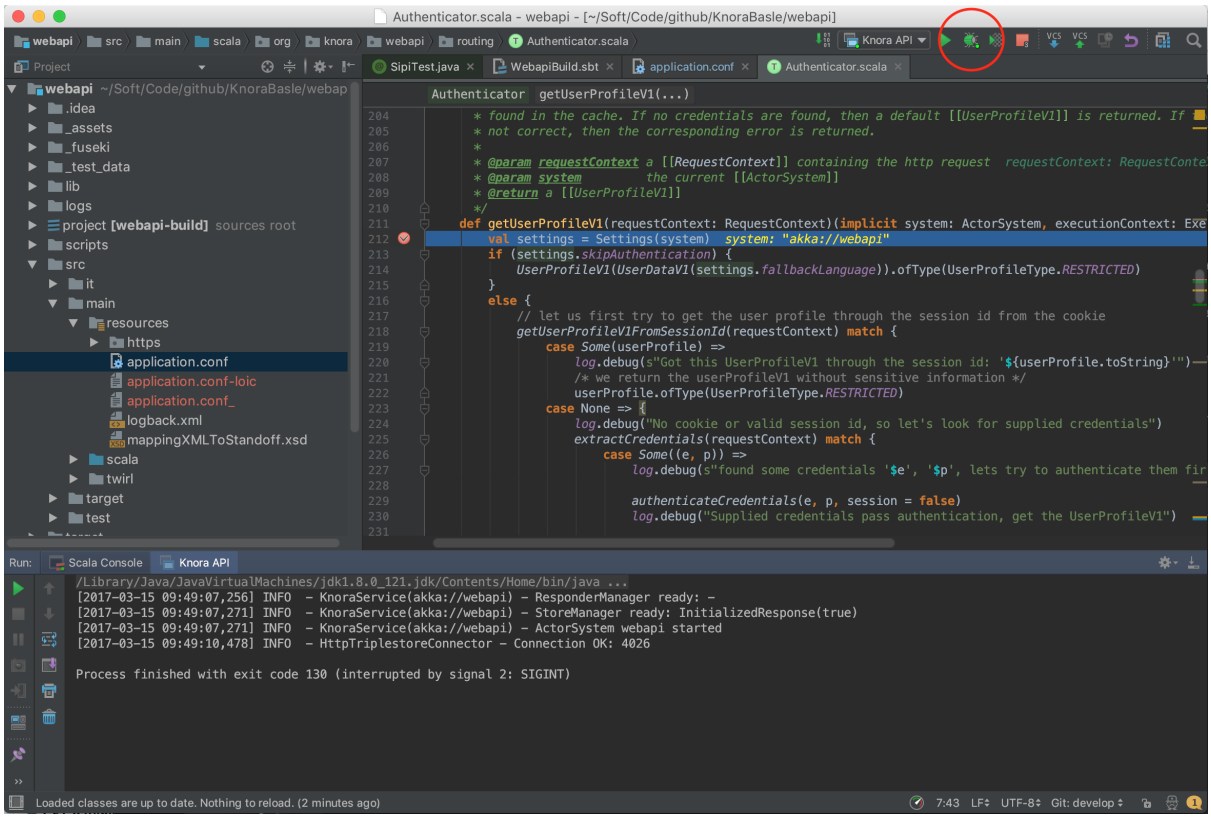


Fig. 4.8: screenshot 'debug'

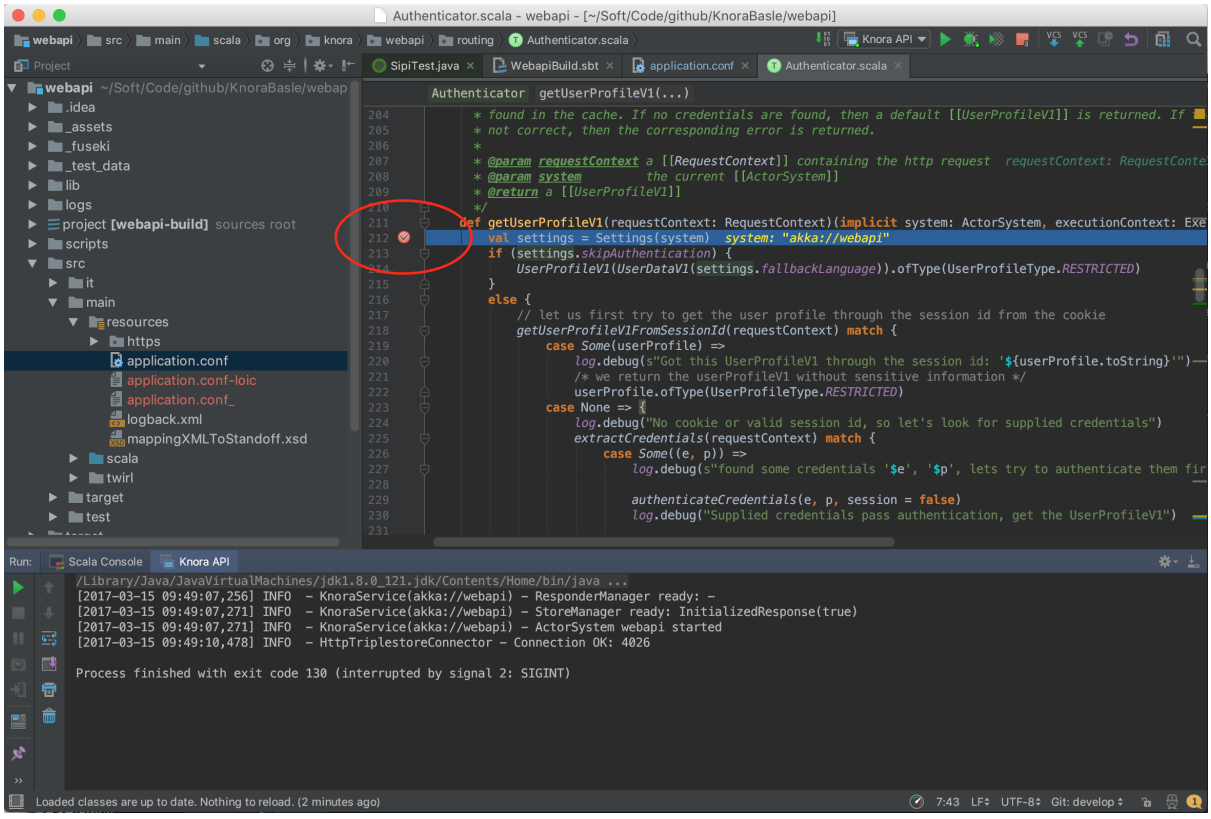


Fig. 4.9: screenshot 'set a breakpoint'

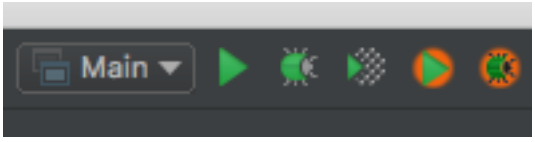


Fig. 4.10: screenshot 'Run with VisualVM button'

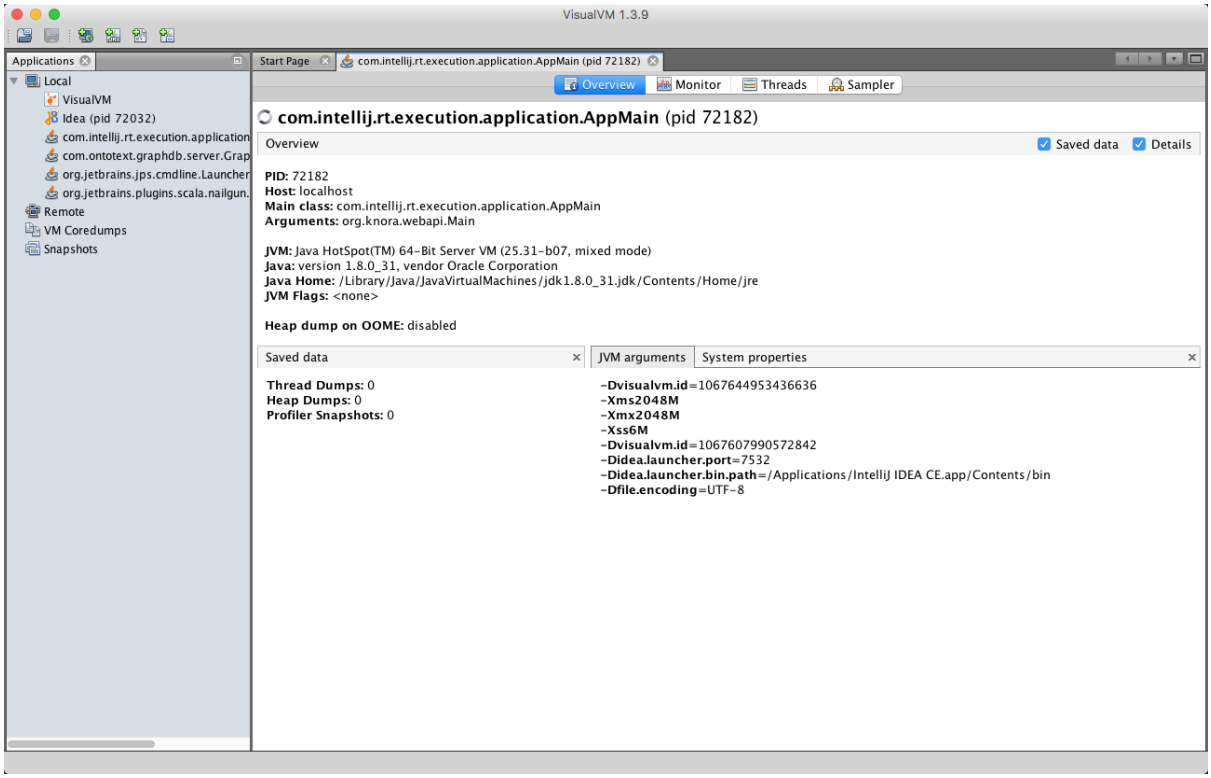


Fig. 4.11: screenshot 'VisualVM overview'

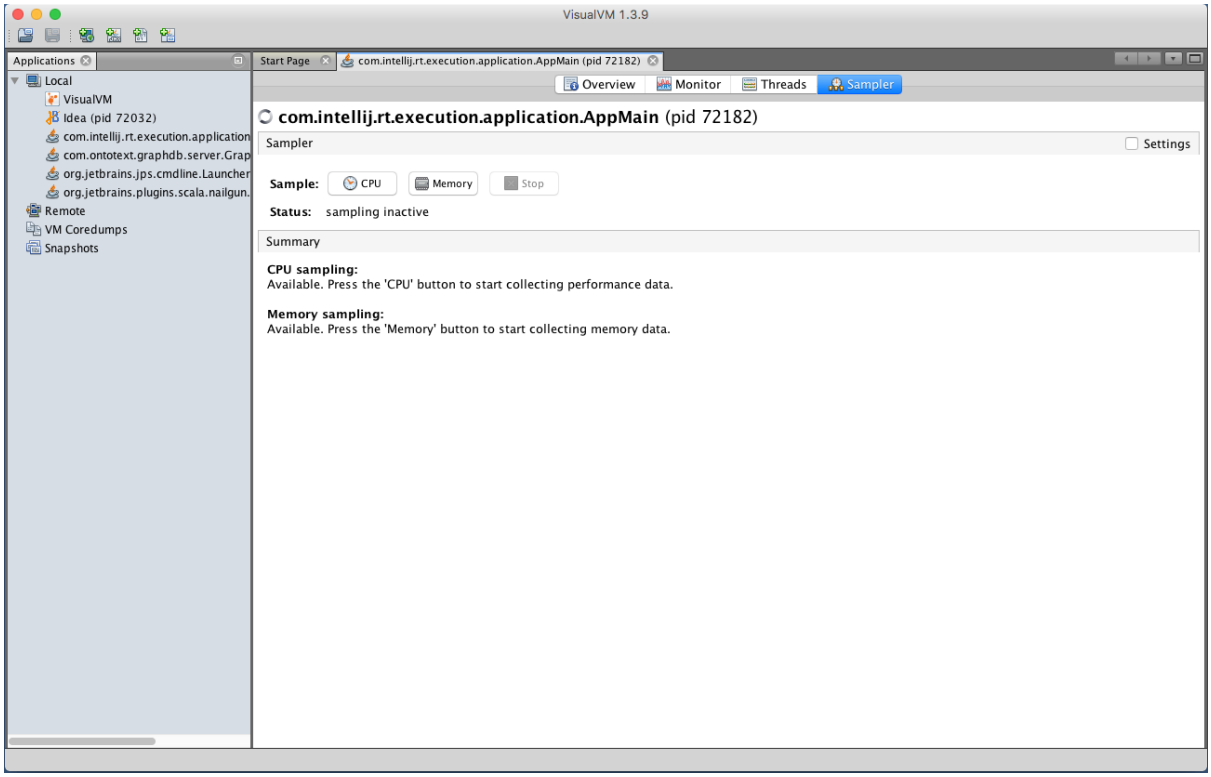


Fig. 4.12: screenshot 'VisualVM sampler'

Now run some Knora API operations that you're interested in profiling, preferably several times to allow the sampler to collect enough data. Then click on the "Snapshot" button:

In the snapshot, you'll see a list of threads that were profiled:

You can then browse the call tree for each thread, looking for Knora method calls, to see the total time spent in each method:

Documentation Guidelines

The Knora documentation uses reStructuredText as its markup language and is built using Sphinx.

For more details, see The Sphinx Documentation and Quick reStructuredText.

Sections

Section headings are very flexible in reST. We use the following convention in the Knora documentation based on the Python Documentation Conventions:

- # (over and under) for parts
- * (over and under) for chapters
- = for sections

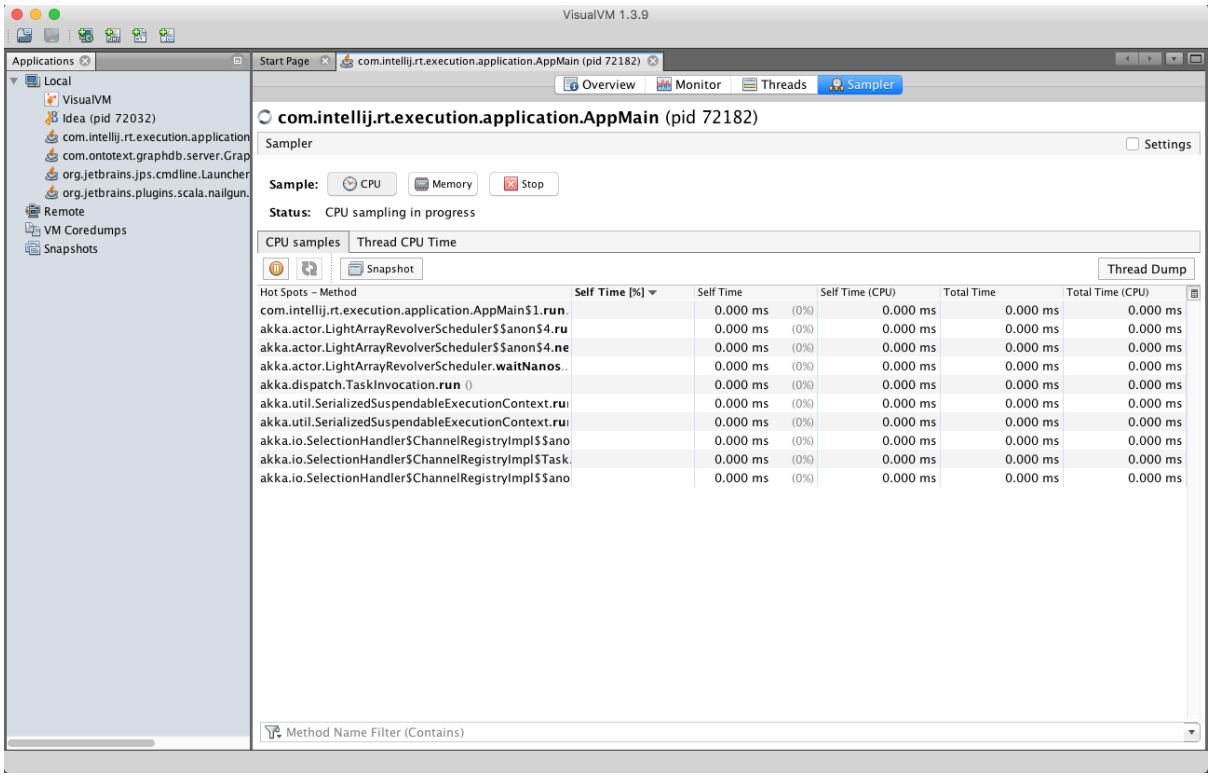


Fig. 4.13: screenshot 'VisualVM snapshot button'

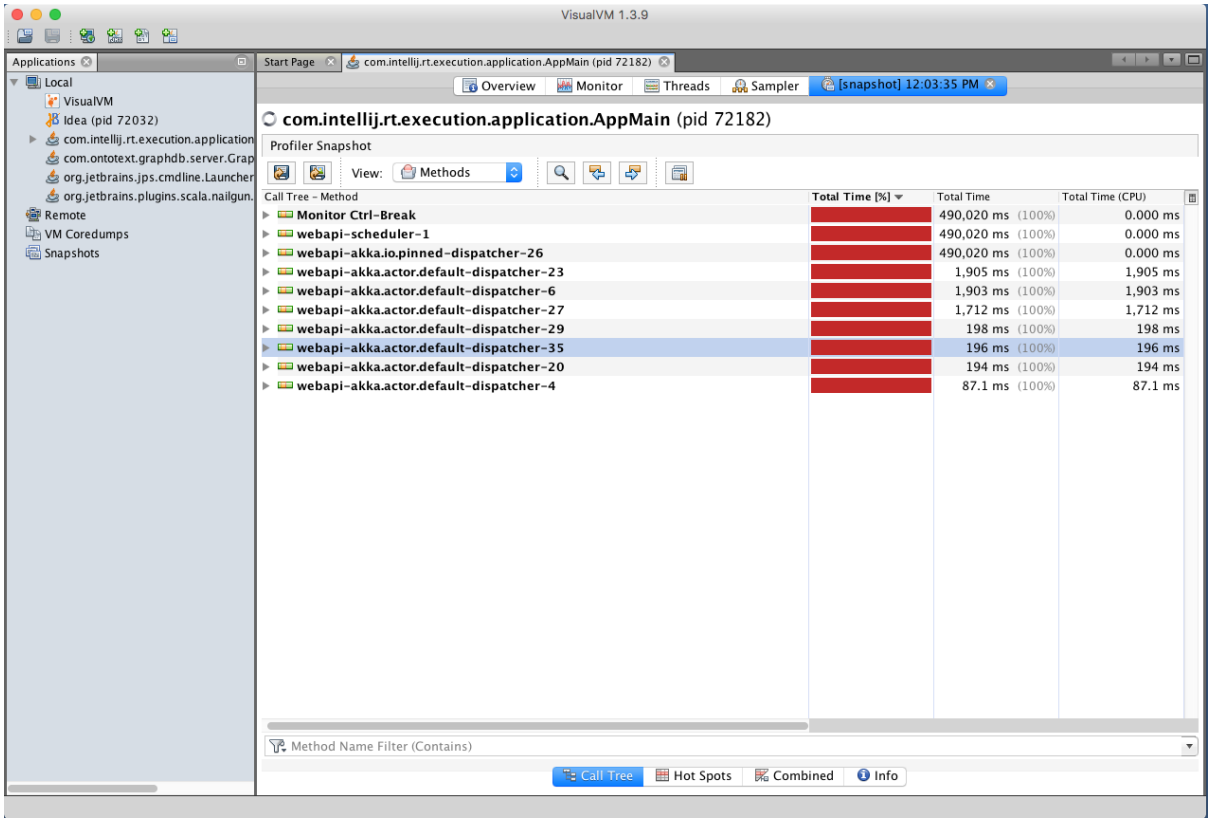


Fig. 4.14: screenshot 'VisualVM snapshot'

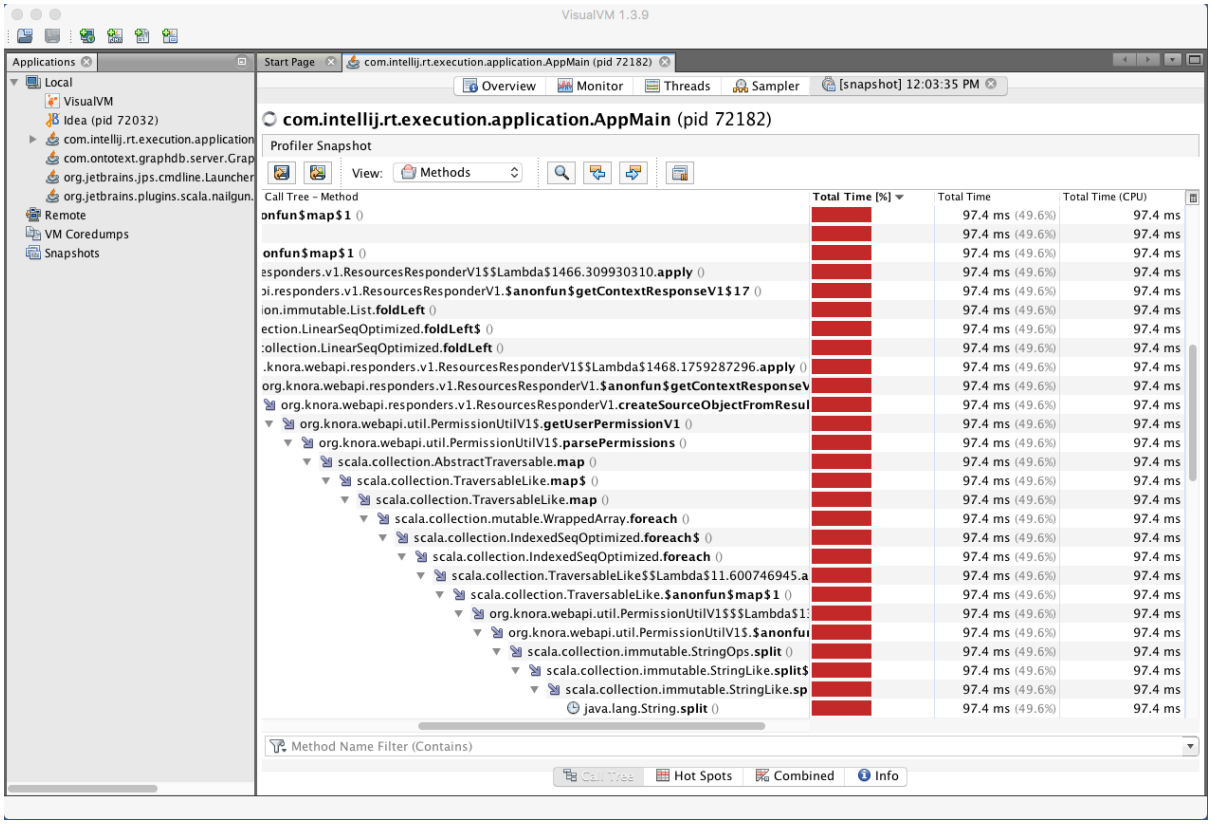


Fig. 4.15: screenshot 'VisualVM call tree'

- – for subsections
- ^ for subsubsections
- ~ for subsubsubsections

Cross-referencing

Sections that may be cross-referenced across the documentation should be marked with a reference. To mark a section use `.. _ref-name:` before the section heading. The section can then be linked with `:ref:`ref-name``. These are unique references across the entire documentation.

For example:

```
.. _knora_part::

##########
Knora Part
##########

.. _knora-chapter:

*************
Knora Chapter
*************

This is the chapter documentation.

.. _knora-section:
```

```
Knora Section
=============

Knora Subsection
----------------

Here is a reference to "knora section": :ref:`knora-section` which will have the
name "Knora Section".
```

Build the documentation

First install Sphinx. See below.

For the html version of the docs:

```
sbt sphinx:generateHtml

open <project-dir>/akka-docs/target/sphinx/html/index.html
```

For the pdf version of the docs:

```
sbt sphinx:generatePdf

open <project-dir>/akka-docs/target/sphinx/latex/AkkaJava.pdf
or
open <project-dir>/akka-docs/target/sphinx/latex/AkkaScala.pdf
```

Installing Sphinx on OS X

Install Homebrew.

Install Python with Homebrew:

```
brew install python
```

Homebrew will automatically add Python executable to your $PATH and pip is a part of the default Python installation with Homebrew.

More information in case of trouble: Homebrew and Python.

Install sphinx:

```
pip install sphinx
```

Install the BasicTeX package.

Add texlive bin to $PATH:

```
export TEXLIVE_PATH=/usr/local/texlive/2015basic/bin/universal-darwin
export PATH=$TEXLIVE_PATH:$PATH
```

Add missing tex packages:

```
sudo tlmgr update --self
sudo tlmgr install titlesec
sudo tlmgr install framed
sudo tlmgr install threeparttable
sudo tlmgr install wrapfig
sudo tlmgr install helvetic
sudo tlmgr install courier
sudo tlmgr install multirow
```

If you get the error `unknown locale: UTF-8` when generating the documentation, the solution is to define the following environment variables:

```
export LANG=en_GB.UTF-8
export LC_ALL=en_GB.UTF-8
```

Test Tags

Tags can be used to mark tests, which can then be used to only run tests with a certain tag, or exclude them.

There is now the **org.knora.webapi.testing.tags.SipiTest** tag (in the *test* folder), which marks tests that require the Sipi image server. These tests can be excluded from running with the following command in sbt:

```
test-only * -- -l org.knora.webapi.testing.tags.SipiTest
```

Testing with Fuseki 2

Inside the Knora API server git repository, there is a folder called `_fuseki` containing a script named `fuseki-server`. All needed configuration files are in place. To start Fuseki 2, just run this script.

How to Write Your Test

1. Inside a test, at the beginning, add the following (change the paths to the test data as needed):

```
val rdfDataObjects = List (
       RdfDataObject(path = "_test_data/ontologies/knora-base.ttl", name = "http://www.knora.org/
       RdfDataObject(path = "_test_data/ontologies/knora-dc.ttl", name = "http://www.knora.org/ont
       RdfDataObject(path = "_test_data/ontologies/salsah-gui.ttl", name = "http://www.knora.org/
       RdfDataObject(path = "_test_data/ontologies/incunabula-onto.ttl", name = "http://www.knora
       RdfDataObject(path = "_test_data/responders.v1.ValuesResponderV1Spec/incunabula-data.ttl",
)

"Reload data " in {
    storeManager ! ResetTripleStoreContent(rdfDataObjects)
    expectMsg(15.seconds, ResetTripleStoreContentACK())
}
```

2. In the config section add `fuseki` as the `dbtype`:

```
app {
     triplestore {
         //dbtype = "embedded-jena-tdb"
         dbtype = "fuseki"
     ...
}
```

Important

The reloading of the test data should be always done at the beginning of the test, because when using Fuseki in combination with `reload-on-start`, the data is not loaded in time (when the actor starts), so that the tests already run without all the data inside the triple store.

Docker Cheat Sheet

A complete cheat sheet can be found here

Lifecycle

- [*docker create*](https://docs.docker.com/reference/commandline/create) creates a container but does not start it.
- [*docker run*](https://docs.docker.com/reference/commandline/run) creates and starts a container in one operation.
- [*docker rename*](https://docs.docker.com/engine/reference/commandline/rename/) allows the container to be renamed.
- [*docker rm*](https://docs.docker.com/reference/commandline/rm) deletes a container.
- [*docker update*](https://docs.docker.com/engine/reference/commandline/update/) updates a container's resource limits.

If you want a transient container, `docker run --rm` will remove the container after it stops.

If you want to map a directory on the host to a docker container, `docker run -v $HOSTDIR:$DOCKERDIR`.

Starting and Stopping

- [*docker start*](https://docs.docker.com/reference/commandline/start) starts a container so it is running.
- [*docker stop*](https://docs.docker.com/reference/commandline/stop) stops a running container.
- [*docker restart*](https://docs.docker.com/reference/commandline/restart) stops and starts a container.
- [*docker pause*](https://docs.docker.com/engine/reference/commandline/pause/) pauses a running container, "freezing" it in place.
- [*docker attach*](https://docs.docker.com/reference/commandline/attach) will connect to a running container.

Info

- [*docker ps*](https://docs.docker.com/reference/commandline/ps) shows running containers.
- [*docker logs*](https://docs.docker.com/reference/commandline/logs) gets logs from container. (You can use a custom log driver, but logs is only available for *json-file* and *journald* in 1.10)
- [*docker inspect*](https://docs.docker.com/reference/commandline/inspect) looks at all the info on a container (including IP address).
- [*docker events*](https://docs.docker.com/reference/commandline/events) gets events from container.
- [*docker port*](https://docs.docker.com/reference/commandline/port) shows public facing port of container.
- [*docker top*](https://docs.docker.com/reference/commandline/top) shows running processes in container.
- [*docker stats*](https://docs.docker.com/reference/commandline/stats) shows containers' resource usage statistics.
- [*docker diff*](https://docs.docker.com/reference/commandline/diff) shows changed files in the container's FS.

`docker ps -a` shows running and stopped containers.

`docker stats --all` shows a running list of containers.

Executing Commands

- [*docker exec*](https://docs.docker.com/reference/commandline/exec) to execute a command in container.

To enter a running container, attach a new shell process to a running container called foo, use: `docker exec -it foo /bin/bash`.

Images

Images are just [templates for docker containers](https://docs.docker.com/engine/understanding-docker/#how-does-a-docker-image-work).

- [*docker images*](https://docs.docker.com/reference/commandline/images) shows all images.
- [*docker build*](https://docs.docker.com/reference/commandline/build) creates image from Dockerfile.

Using API V1

Introduction: Using API V1

RESTful API

Knora API V1 is a RESTful API that allows for reading and adding of resources from and to Knora and changing their values using HTTP requests. The actual data is submitted as JSON (request and response format). The diverse HTTP methods are applied according to the widespread practice of RESTful APIs: GET for reading, POST for adding, PUT for changing resources and values, and DELETE to delete resources or values (see RESTful_API).

Knora IRIs

Every resource that is created or hosted by Knora is identified by a unique id, a so called Internationalized Resource Identifier (IRI). The IRI is required for every API operation to identify the resource in question. A Knora IRI has itself the format of a URL. For some API operations, the IRI has to be URL-encoded (HTTP GET requests).

Different API Operations

In the following sections, the diverse API operations are described including their request and response formats:

- *Reading and Searching Resources*: Get a specific resource or resource class by its IRI or search for resources
- *Adding Resources*: Create a new resource
- *Reading Values*: Get a specific value or its version history
- *Adding a Value*: Add values to a resource
- *Changing a Value*: Change the values of a resource
- *Deleting Resources and Values*: Delete resources and values

V1 Path Segment

Every request to API V1 includes `v1` as a path segment, e.g. `http://host/v1/resources/http%3A%2F%2Fdata.knora`
Accordingly, requests to another version of the API will require another path segment.

Knora API Response Format

In case an API request could be handled successfully, Knora responds with a 200 HTTP status code. The actual answer from Knora (the representation of the requested resource or information about the executed API operation) is sent in the HTTP body, encoded as JSON (using UTF-8). In this JSON, an API specific status code is sent (member `status`).

The JSON formats are formally defined as TypeScript interfaces (located in `salsah/src/typescript_interfaces`). Build the HTML documentation of these interfaces by executing `make jsonformat` (see `docs/Readme.md` for further instructions).

Placeholder `host` in sample URLs

Please note that all the sample URLs used in this documentation contain `host` as a placeholder. The placeholder `host` has to be replaced by the actual hostname (and port) of the server the Knora instance is running on.

Authentication

For all API operations that target at changing resources or values, the client has to provide credentials (username and password) so that the API server can authenticate the user making the request. When using the SALSAH web interface, after logging in a session is established (cookie based). When using the API with another client application, credentials can be sent as a part of the HTTP header or as parts of the URL (see *Authentication in the Knora API Server*).

Also when reading resources authentication my be needed as resources and their values may have restricted view permissions.

Reading and Searching Resources

In order to get an existing resource, the HTTP method `GET` has to be used. The request has to be sent to the Knora server using the `resources` path segment (depending on the type of request, this segment has to be exchanged, see below). Reading resources may require authentication since some resources may have restricted viewing permissions.

Get the Representation of a Resource by its IRI

Simple Request of a Resource (full Resource Request)

A resource can be obtained by making a GET request to the API providing its IRI. Because a Knora IRI has the format of a URL, its IRI has to be URL encoded.

In order to get the resource with the IRI `http://data.knora.org/c5058f3a` (an incunabula book contained in the test data), make a HTTP GET request to the resources route (path segment `resources` in the API call) and append the URL encoded IRI:

```
HTTP GET to http://host/v1/resources/http%3A%2F%2Fdata.knora.org%2Fc5058f3a
```

More formalized, the URL looks like this:

```
HTTP GET to http://host/v1/resources/resourceIRI
```

As an answer, the client receives a JSON that represents the requested resource. It has the following members:

- `status`: The Knora status code, `0` if everything went well
- `userdata`: Data about the user that made the request
- `resinfo`: Data describing the requested resource and its class
- `resdata`: Short information about the resource and its class (including information about the given user's permissions on the resource)
- `incoming`: Resources pointing to the requested resource
- `props`: Properties of the requested resource.

For a complete and more formalized description of a full resource request, look at the TypeScript interface `resourceFullResponse` in the module `resourceResponseFormats`.

Provide Request Parameters

To make a request more specific, the following parameters can be appended to the URL (`http://www.knora.org/resc`

- **`reqtype=info|context|rights`: Specifies the type of request.**
  - Setting the parameter's to value `info` returns short information about the requested resource (contains only `resinfo` and no properties, see TypeScript interface `resourceInfoResponse` in module `resourceResponseFormats`).
  - Settings the parameter's value to `context` returns context information (`resource_context`) about the requested resource: Either the dependent parts of a compound resource (e.g. pages of a book) or the parent resource of a dependent resource (e.g. the book a pages belongs to). By default, a context query does not return information about the requested resource itself, but only about its context (see TypeScript interface `resourceContextResponse` in module `resourceResponseFormats`). See below how to get additional information about the resource.
  - The parameter `rights` returns only the given user's permissions on the requested resource (see TypeScript interface `resourceRightsResponse` in module `resourceResponseFormats`).
- `resinfo=true`: Can be used in combination with `reqtype=context`: If set, `resinfo` is added to the response representing information about the requested resource (complementary to its context), see TypeScript interface `resourceContextResponse` in module `resourceResponseFormats`.

Obtain an HTML Representation of a Resource

In order to get an HTML representation of a resource (not a JSON), the path segment `resources.html` can be used:

```
HTTP GET to http://host/v1/resources.html/resourceIRI?reqtype=properties
```

The request returns the properties of the requested resource as an HTML document.

Get only the Properties belonging to a Resource

In order to get only the properties of a resource without any other information, the path segment `properties` can be used:

```
HTTP GET to http://host/v1/properties/resourceIRI
```

The JSON contains just the member `properties` representing the requested resource's properties (see TypeScript interface `resourcePropertiesResponse` in module `resourceResponseFormats`).

Get Information about a Resource Class

Get a Resource Class by its IRI

In order to get information about a resource class, the path segment `resourcetypes` can be used. Append the IRI of the resource class to the URL (e.g. `http://www.knora.org/ontology/incunabula#book`).

```
HTTP GET to http://host/v1/resourcetypes/resourceClassIRI
```

In the JSON, the information about the resource class and all the property types that it may have are returned. Please note that none of these are actual instances of a property, but only types (see TypeScript interface `resourceTypeResponse` in module `resourceResponseFormats`).

Get all the Property Types of a Resource Class or a Vocabulary

To get a list of all the available property types, the path segment `propertylists` can be used. It can be restricted to a certain vocbulary using the parameter `vocabulary` or to a certain resource class using the parameter `restype`.

```
# returns all the property types for incunabula:page
HTTP GET to http://host/v1/propertylists?restype=resourceClassIRI

# returns all the property types for the incunabula vocabulary
HTTP GET to http://host/v1/propertylists?vocabulary=vocabularyIRI
```

Both of these queries return a list of property types. The default value for the parameter `vocabulary` is `0` and means that the resource classes from all the available vocabularies are returned. See TypeScript interface `propertyTypesInResourceClassResponse` in module `resourceResponseFormats`.

Get the Resource Classes of a Vocabulary

Resource classes and property types are organized in (project specific) name spaces, so called vocabularies. In order to get all the resource classes defined for a specific vocabulary (e.g. `incunabula`), the parameter `vocabulary` has to be used and assigned the vocabulary's IRI:

```
HTTP GET to http://host/v1/resourcetypes?vocabulary=vocabularyIRI
```

This returns all the resource classes defined for the specified vocabulary and their property types. The default value for the parameter `vocabulary` is `0` and means that the resource classes from all the available vocabularies are returned. See TypeScript interface `resourceTypesInVocabularyResponse` in module `resourceResponseFormats`.

Get all the Vocabularies

To get a list of all available vocabularies, the path segment `vocabularies` can be used:

```
HTTP GET to http://host/v1/vocabularies
```

The response will list all the available vocabularies. See TypeScript interface `vocabularyResponse` in module `resourceResponseFormats`.

Search for Resources

Search for Resources by their Label

This is a simplified way for searching for resources just by their label. It is a simple string-based method:

```
HTTP GET to http://host/v1/resources?searchstr=searchValue
```

Additionally, the following parameters can be appended to the URL (search value is `Zeitglöcklein`):

- `restype_id=resourceClassIRI`: This restricts the search to resources of the specified class (subclasses of that class will also match). `-1` is the default value and means no restriction to a specific class. If a resource class IRI is specified, it has to be URL encoded (e.g. `http://www.knora.org/v1/resources?searchstr=Zeitgl%C3%B6cklein&restype_id=http%3`
- `numprops=Integer`: Specifies the number of properties returned for each resource that was found (sorted by GUI order), e.g. `http://www.knora.org/v1/resources?searchstr=Zeitgl%C3%B6cklein&numprops=4`.
- `limit=Integer`: Limits the amount of results returned (e.g. `http://www.knora.org/v1/resources?searchstr=Zeitgl%C3%B6cklein&limit=1`).

The response lists the resources that matched the search criteria (see TypeScript interface `resourceLabelSearchResponse` in module `resourceResponseFormats`).

Fulltext Search

Knora offers a fulltext search that searches through all textual representations of values. You can separate search terms by a white space and they will be combined using the Boolean `AND` operator. Please note that the search terms have to be URL encoded.

```
HTTP GET to http://host/v1/search/searchValue?searchtype=fulltext[&filter_by_restype=resourceClass
[&filter_by_project=projectIRI][&show_nrows=Integer]{[&start_at=Integer]
```

The parameter `searchtype` is required and has to be set to `fulltext`. Additionally, these parameters can be set:

- `filter_by_restype=resourceClassIRI`: restricts the search to resources of the specified resource class (subclasses of that class will also match).
- `filter_by_project=projectIRI`: restricts the search to resources of the specified project.
- `show_nrows=Integer`: Indicates how many reults should be presented on one page. If omitted, the default value `25` is used.
- `start_at=Integer`: Used to enable paging and go through all the results request by request.

The response presents the retrieved resources (according to `show_nrows` and `start_at`) and information about paging. If not all resources could be presented on one page (`nhits` is greater than `shown_nrows`), the next page can be requested (by increasing `start_at` by the number of `show_nrows`). You can simply go through the elements of `paging` to request the single pages one by one. See TypeScript interface `searchResponse` in module `searchResponseFormats`.

Extended Search for Resources

```
HTTP GET to http://host/v1/search/?searchtype=extended
[&filter_by_restype=resourceClassIRI][&filter_by_project=projectIRI][&filter_by_owner=userIRI]
(&property_id=propertyTypeIRI&compop=comparisonOperator&searchval=searchValue)+
[&show_nrows=Integer][&start_at=Integer]
```

The parameter `searchtype` is required and has to be set to `extended`. An extended search requires at least one set of pa

- `property_id=propertyTypeIRI`: the property the resource has to have (subproperties of that property will also match).
- `compop=comparisonOperator`: the comparison operator to be used to match between the resource's property value and the search term.
- `searchval=searchTerm`: the search value to look for.

You can also provide several of these sets to make your query more specific.

The following table indicates the possible combinations of value types and comparison operators:

Value Type	Comparison Operator
Date Value	EQ, !EQ, GT, GT_EQ, LT, LT_EQ, EXISTS
Integer Value	EQ, !EQ, GT, GT_EQ, LT, LT_EQ, EXISTS
Float Value	EQ, !EQ, GT, GT_EQ, LT, LT_EQ, EXISTS
Text Value	MATCH_BOOLEAN, MATCH, EQ, !EQ, LIKE, !LIKE, EXISTS
Geometry Value	EXISTS
Resource Pointer	EQ, EXISTS
Color Value	EQ, EXISTS
List Value	EQ, EXISTS

Explanation of the comparison operators:

- `EQ`: checks if a resource's value *equals* the search value. In case of a text value type, it checks for identity of the strings compared. In case of a date value type, it checks if the dates are equal or if the specified date encompasses it (internally, dates are always treated as periods).
- `!EQ`: checks if a resource's value *does not equal* the search value. In case of a text value type, it checks if the compared strings are different. In case of a date value type, it checks if the dates are not equal or if the specified date does not encompass it (internally, dates are always treated as periods).
- `GT`: checks if a resource's value is *greater than* the search value. In case of a date value type, it checks if the resource's period begins after the indicated date.
- `GT_EQ`: checks if a resource's value *equals or is greater than* the search value. In case of a date value type, it checks if the resource's period equals the end of the indicated period or begins after the indicated period.
- `LT`: checks if a resource's value is *lower than* the search value. In case of a date value type, it checks if the resource's period begins before the indicated date.
- `LT_EQ`: checks if a resource's value *equals or is lower than* the search value. In case of a date value type, it checks if the resource's period equals the begin of the indicated period or begins before the indicated period.

- `EXISTS`: checks if an instance of the indicated property type *exists* for a resource. **Please always provide an empty search value when using EXISTS: "searchval="**. Otherwise, the query syntax rules would be violated.
- `MATCH`: checks if a resource's text value *matches* the search value. The behaviour depends on the used triplestore's full text index.
- `LIKE`: checks if the search value is contained in a resource's text value.
- `!LIKE`: checks if the search value is not contained in a resource's text value.
- `MATCH_BOOLEAN`: checks if a resource's text value *matches* the provided list of positive (exist) and negative (do not exist) terms. The list takes this form: `([+-]term\s)+`.

Additionally, these parameters can be set:

- `filter_by_restype=resourceClassIRI`: restricts the search to resources of the specified resource class (subclasses of that class will also match).
- `filter_by_project=projectIRI`: restricts the search to resources of the specified project.
- `filter_by_owner`: restricts the search to resources owned by the specified user.
- `show_nrows=Integer`: Indicates how many reults should be presented on one page. If omitted, the default value `25` is used.
- `start_at=Integer`: Used to enable paging and go through all the results request by request.

Some sample searches:

- `http://localhost:3333/v1/search/?searchtype=extended&filter_by_restype=http%3A%2` searches for books that have a title that does not equal "Zeitglöcklein des Lebens und Leidens Christi".
- `http://www.knora.org/v1/search/?searchtype=extended&filter_by_restype=http%3A%2F` searches for resources of type `incunabula:book` whose titles match "Zeitglöcklein" and were published in the year 1490 (according to the Julian calendar).

The response presents the retrieved resources (according to `show_nrows` and `start_at`) and information about paging. If not all resources could be presented on one page (`nhits` is greater than `shown_nrows`), the next page can be requested (by increasing `start_at` by the number of `show_nrows`). You can simply go through the elements of `paging` to request the single pages one by one. See the TypeScript interface `searchResponse` in module `searchResponseFormats`.

Get a Graph of Resources

The path segment `graphdata` returns a graph of resources that are reachable via links to or from an initial resource.

```
HTTP GET to http://host/v1/search/graphdata/resourceIRI?depth=Integer
```

The parameter `depth` specifies the maximum depth of the graph, and defaults to 4. If `depth` is 1, the operation will return only the initial resource and any resources that are directly linked to or from it.

The graph includes any link that is a subproperty of `knora-base:hasLinkTo`, except for links that are subproperties of `knora-base:isPartOf`. Specifically, if resource `R1` has a link that is a subproperty of `knora-base:isPartOf` pointing to resource `R2`, no link from `R1` to `R2` is included in the graph.

The response represents the graph as a list of nodes (resources) and a list of edges (links). For details, see the TypeScript interface `graphDataResponse` in module `graphDataResponseFormats`.

Get Hierarchical Lists

The knora-base ontology allows for the definition of hierarchical lists. These can be queried by providing the IRI of the root node. Selections are hierarchical list that are just one level deep. Internally, they are represented as hierarchical lists.

You can get a hierarchical by using the path segment `hlists` and appending the hierarchical list's IRI (URL encoded):

```
HTTP GET to http://host/v1/hlists/rootNodeIRI
```

The response shows all of the list nodes that are element of the requested hierarchical list as a tree structure. See TypeScript interface `hierarchicalListResponse` in module `hierarchicalListResponseFormats`.

For each node, the full path leading to it from the top level can be requested by making a query providing the node's IRI and setting the param `reqtype=node`:

```
HTTP GET to http://host/v1/hlists/nodeIri?reqtype=node
```

The response presents the full path to the current node. See TypeScript interface `nodePathResponse` in module `hierarchicalListResponseFormats`.

XML to Standoff Mapping

The Knora Standard Mapping

A mapping allows for the conversion of XML to standoff representation in RDF and back. In order to create a TextValue with markup, the text has to be provided in XML format, along with the IRI of the mapping that will be used to convert the markup to standoff. However, a mapping is only needed if a TextValue with markup should be created. If a text has no markup, it is submitted as a mere sequence of characters.

The two cases are described in the TypeScript interfaces `simpletext` and `richtext` in module `basicMessageComponents`.

Knora offers a standard mapping with the IRI `http://data.knora.org/projects/standoff/mappings/StandardM`
The standard mapping covers the HTML elements and attributes supported by the GUI's text editor CKEditor [1] (please note that the HTML has to be encoded in strict XML syntax). The standard mapping contains the following elements and attributes that are mapped to standoff classes and properties defined in the ontology:

- `<text>` -> `standoff:StandoffRootTag`
- `<p>` -> `standoff:StandoffParagraphTag`
- `<em>` -> `standoff:StandoffItalicTag`
- `<strong>` -> `standoff:StandoffBoldTag`
- `<u>` -> `standoff:StandoffUnderlineTag`
- `<sub>` -> `standoff:StandoffSubscriptTag`
- `<sup>` -> `standoff:StandoffSuperscriptTag`

[1] CKeditor offers the possibility to define filter rules (CKEditor). They should reflect the elements supported by the mapping (see `jquery.htmleditor.js`).

- <strike> -> standoff:StandoffStrikeTag
- <a href="URL"> -> knora-base:StandoffUriTag
- <a class="salsah-link" href="Knora IRI"> -> knora-base:StandoffLinkTag
-
 -> standoff:StandoffBrTag

The HTML produced by CKEditor is wrapped in an XML doctype and a pair of root tags <text>...</text> and then sent to Knora. The XML sent to the GUI by Knora is unwrapped accordingly (see jquery.htmleditor.js). Although the GUI supports HTML5, it is treated as if it was XHTML in strict XML notation.

Creating a custom Mapping

The Knora standard mapping only supports a few HTML tags. In order to submit more complex XML markup to Knora, a custom mapping has to be created first. Basically, a mapping expresses the relations between XML elements and attributes and their corresponding standoff classes and properties. The relations expressed in a mapping are one-to-one relations, so the XML can be recreated from the data in RDF. However, since HTML offers a very limited set of elements, Knora mappings support the combination of element names and classes. In this way, the same element can be used several times in combination with another classname (please note that <a> without a class is a mere hyperlink whereas <a class="salsah-link"> is an internal link/standoff link).

Basic Structure of a Mapping

The mapping is written in XML itself (for a formal description, see webapi/src/resources/mappingXMLToStandoff.xsd). It has the following structure (the indentation corresponds to the nesting in XML):

- **<mapping>: the root element**
  - **<mappingElement>: an element of the mapping (at least one)**
    - **<tag>: information about the XML element that is mapped to a standoff class**
      - <name>: name of the XML element
      - <class>: value of the class attribute of the XML element, if any. If the element has no class attribute, the keyword noClass has to be used.
      - <namespace>: the namespace the XML element belongs to, if any. If the element does not belong to a namespace, the keyword noNamespace has to be used.
      - <separatesWords>: a Boolean value indicating whether this tag separates words in the text. Once an XML document is converted to RDF-standoff the markup is stripped from the text, possibly leading to continuous text that has been separated by tags before. For structural tags like paragraphs etc., <separatesWords> can be set to true in which case a special separator is inserted in the the text in the RDF representation. In this way, words stay separated and are represented in the fulltext index as such.
    - **<standoffClass>: information about the standoff class the XML element is mapped to**
      - <classIri>: Iri of the standoff class the XML element is mapped to
      - **<attributes>: XML attributes to be mapped to standoff properties (other than id or class)**

        <attribute>: an XML attribute to be mapped to a standoff property, may be repeated

        <attributeName>: the name of the XML attribute

<namespace>: the namespace the attribute belogs to, if any. If the attribute does not belong to a namespace, the keyword noNamespace has to be used.

<propertyIri>: the Iri of the standoff property the XML attribute is mapped to.

- **<datatype>: the data type of the standoff class, if any.**

  <type>: the Iri of the data type standoff class

  <attributeName>: the name of the attribute holding the typed value in the expected Knora standard format

XML structure of a mapping:

```
<mapping>
    <mappingElement>
        <tag>
            <name>XML element name</name>
            <class>XML class name or "noClass"</class>
            <namespace>XML namespace or "noNamespace"</namespace>
            <separatesWords>true or false</separatesWords>
        </tag>
        <standoffClass>
            <classIri>standoff class Iri</classIri>
            <attributes>
                <attribute>
                    <attributeName>XML attribute name</attributeName>
                    <namespace>XML namespace or "noNamespace"</namespace>
                    <propertyIri>standoff property Iri</propertyIri>
                </attribute>
            <datatype>
                <type>standoff data type class</type>
                <attributeName>XML attribute with the typed value</attributeName>
            </datatype>
        </standoffClass>
    </mappingElement>
    <mappingElement>
       ...
    </mappingElement>
</mapping>
```

Please note that the absence of an XML namespace and/or a class have to be explicitly stated using the keywords noNamespace and noClass [2].

id and class Attributes

The id and class attributes are supported by default and do not have to be included in the mapping like other attributes. The id attribute identifies an element and must be unique in the document. id is an optional attribute. The class attribute allows for the reuse of an element in the mapping, i.e. the same element can be combined with different class names and mapped to different standoff classes (mapping element <class> in <tag>).

Respecting Cardinalities

A mapping from XML elements and attributes to standoff classes and standoff properties must respect the cardinalities defined in the ontology for those very standoff classes. If an XML element is mapped to a certain standoff class and this class requires a standoff property, an attribute must be defined for the XML element mapping to that very standoff property. Equally, all mappings for attributes of an XML element must have corresponding cardinalities for standoff properties defined for the standoff class the XML element maps to.

[2] This is because we use XML Schema validation to ensure the one-to-one relations between XML elements and standoff classes. XML Schema validations unique checks do not support optional values.

However, since an XML attribute may occur once at maximum, it makes sense to make the corresponding standoff property required (`owl:cardinality` of one) in the ontology or optional (`owl:maxCardinality` of one), but not allowing it more than once.

Standoff Data Types

Knora allows the use of all its value types as standoff data types (defined in `knora-base.ttl`):

- `knora-base::StandoffLinkTag`: Represents a reference to a Knora resource (the IRI of the target resource must be submitted in the data type attribute).
- `knora-base:StandoffInternalReferenceTag`: Represents an internal reference inside a document (the id of the target element inside the same document must be indicated in the data type attribute), see *Internal References in an XML Document*.
- `knora-base::StandoffUriTag`: Represents a reference to a URI (the URI of the target resource must be submitted in the data type attribute).
- `knora-base::StandoffDateTag`: Represents a date (a Knora date string must be submitted in the data type attribute, e.g. `GREGORIAN:2017-01-27`).
- `knora-base::StandoffColorTag`: Represents a color (a hexadecimal RGB color string must be submitted in the data type attribute, e.g. `#0000FF`).
- `knora-base::StandoffIntegerTag`: Represents an integer (the integer must be submitted in the data type attribute).
- `knora-base::StandoffDecimalTag`: Represents a number with fractions (the decimal number must be submitted in the data type attribute, e.g. `1.1`).
- `knora-base::StandoffIntervalTag`: Represents an interval (two decimal numbers separated with a comma must be submitted in the data type attribute, e.g. `1.1,2.2`).
- `knora-base::StandoffBooleanTag`: Represents a Boolean value (`true` or `false` must be submitted in the data type attribute).

The basic idea is that parts of a text can be marked up in a way that allows using Knora's built-in data types. In order to do so, the typed values have to be provided in a standardized way in an attribute that has to be defined in the mapping.

Data type standoff classes are standoff classes with predefined properties (e.g., a `knora-base:StandoffLinkTag` has a `knora-base:standoffTagHasLink` and a `knora-base:StandoffIntegerTag` has a `knora-base:valueHasInteger`). Please note the data type standoff classes can not be combined, i.e. a standoff class can only be the subclass of **one** data type standoff class. However, standoff data type classes can be subclassed and extended further by assigning properties to them (see below).

The following simple mapping illustrates this principle:

```
<?xml version="1.0" encoding="UTF-8"?>
<mapping>
     <mappingElement>
        <tag>
            <name>text</name>
            <class>noClass</class>
            <namespace>noNamespace</namespace>
            <separatesWords>false</separatesWords>
        </tag>
        <standoffClass>
            <classIri>http://www.knora.org/ontology/standoff#StandoffRootTag</classIri>
        </standoffClass>
    </mappingElement>

    <mappingElement>
```

```
        <tag>
            <name>mydate</name>
            <class>noClass</class>
            <namespace>noNamespace</namespace>
            <separatesWords>false</separatesWords>
        </tag>
        <standoffClass>
            <classIri>http://www.knora.org/ontology/anything#StandoffEventTag</classIri>
            <attributes>
                <attribute>
                    <attributeName>description</attributeName>
                    <namespace>noNamespace</namespace>
                    <propertyIri>http://www.knora.org/ontology/anything#standoffEventTagHasDescrip
                </attribute>
            </attributes>
            <datatype>
                <type>http://www.knora.org/ontology/knora-base#StandoffDateTag</type>
                <attributeName>knoraDate</attributeName>
            </datatype>
        </standoffClass>
    </mappingElement>
<mapping>
```

`<datatype>` **must** hold the Iri of a standoff data type class (see list above). The `<classIri>` must be a subclass of this type or this type itself (the latter is probably not recommendable since semantics are missing: what is the meaning of the date?). In the example above, the standoff class is `anything:StandoffEventTag` which has the following definition in the ontology `anything-onto.ttl`:

```
anything:StandoffEventTag rdf:type owl:Class ;

    rdfs:subClassOf knora-base:StandoffDateTag,
                    [
                       rdf:type owl:Restriction ;
                       owl:onProperty :standoffEventTagHasDescription ;
                       owl:cardinality "1"^^xsd:nonNegativeInteger
                    ] ;

    rdfs:label "Represents an event in a TextValue"@en ;

    rdfs:comment """Represents an event in a TextValue"""@en .
```

`anything:StandoffEventTag` is a subclass of `knora-base:StandoffDateTag` and therefore has the data type date. It also requires the standoff property `anything:standoffEventTagHasDescription` which is defined as an attribute in the mapping.

Once the mapping has been created, an XML like the following could be sent to Knora and converted to standoff:

```
<?xml version="1.0" encoding="UTF-8"?>
<text>
    We had a party on <mydate description="new year" knoraDate="GREGORIAN:2016-12-31">New Year's
</text>
```

The attribute holds the date in the format of a Knora date string (the format is also documented in the typescript type alias `dateString` in module `basicMessageComponents`. There you will also find documentation about the other types like color etc.). Knora date strings have this format: `GREGORIAN|JULIAN):YYYY[-MM[-DD]][:YYYY[-MM[-DD]]]`. This allows for different formats as well as for imprecision and periods. Intervals are submitted as one attribute in the following format: `interval-attribute="1.0,2.0"` (two decimal numbers separated with a comma).

You will find a sample mapping with all the data types and a sample XML file in the the test data: `webapi/_test_data/test_route/texts/mappingForHTML.xml` and `webapi/_test_data/test_route/texts/HTML.xml`.

Internal References in an XML Document Internal references inside an XML document can be represented using the data type standoff class `knora-base:StandoffInternalReferenceTag` or a subclass of it. This class has a standoff property that points to a standoff node representing the target XML element when converted to RDF.

The following example shows the definition of a mapping element for an internal reference (for reasons of simplicity, only the mapping element for the element is question is depicted):

```
<mappingElement>
    <tag>
        <name>ref</name>
        <class>noClass</class>
        <namespace>noNamespace</namespace>
        <separatesWords>false</separatesWords>
    </tag>
    <standoffClass>
        <classIri>http://www.knora.org/ontology/knora-base#StandoffInternalReferenceTag</classIri
        <datatype>
            <type>http://www.knora.org/ontology/knora-base#StandoffInternalReferenceTag</type>
            <attributeName>internalRef</attributeName>
        </datatype>
    </standoffClass>
</mappingElement>
```

Now, an internal reference to an element in the same document can be made that will be converted to a pointer in RDF:

```
<?xml version="1.0" encoding="UTF-8"?>
<text>
    This is an <sample id="1">element</sample> and here is a reference to <ref internalRef="#1">it
</text>
```

An internal reference in XML has to start with a `#` followed by the value of the `id` attribute of the element referred to.

Predefined Standoff Classes and Properties

The standoff ontology `standoff-onto.ttl` offers a set of predefined standoff classes that can be used in a custom mapping like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<mapping>
    <mappingElement>
        <tag>
            <name>myDoc</name>
            <class>noClass</class>
            <namespace>noNamespace</namespace>
            <separatesWords>false</separatesWords>
        </tag>
        <standoffClass>
            <classIri>http://www.knora.org/ontology/standoff#StandoffRootTag</classIri>
            <attributes>
                <attribute>
                    <attributeName>documentType</attributeName>
                    <namespace>noNamespace</namespace>
                    <propertyIri>http://www.knora.org/ontology/standoff#standoffRootTagHasDocument
                </attribute>
            </attributes>
        </standoffClass>
    </mappingElement>
```

```
    <mappingElement>
        <tag>
            <name>p</name>
            <class>noClass</class>
            <namespace>noNamespace</namespace>
            <separatesWords>true</separatesWords>
        </tag>
        <standoffClass>
            <classIri>http://www.knora.org/ontology/standoff#StandoffParagraphTag</classIri>
        </standoffClass>
    </mappingElement>

    <mappingElement>
        <tag>
            <name>i</name>
            <class>noClass</class>
            <namespace>noNamespace</namespace>
            <separatesWords>false</separatesWords>
        </tag>
        <standoffClass>
            <classIri>http://www.knora.org/ontology/standoff#StandoffItalicTag</classIri>
        </standoffClass>
    </mappingElement>
<mapping>
```

Predefined standoff classes may be used by various projects, each providing a custom mapping to be able to recreate the original XML from RDF. Predefined standoff classes may also be inherited and extended in project specific ontologies.

The mapping above allows for an XML like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<myDoc documentType="letter">
    <p>
        This my text that is <i>very</i> interesting.
    </p>
    <p>
        And here it goes on.
    </p>
</myDoc>
```

Respecting Property Types

When mapping XML attributes to standoff properties, attention has to be paid to the properties' object constraints.

In the ontology, standoff property literals may have one of the following `knora-base:objectDatatypeConstraint`:

- `xsd:string`
- `xsd:integer`
- `xsd:boolean`
- `xsd:decimal`
- `xsd:anyURI`

In XML, all attribute values are submitted as strings. However, these string representations need to be convertible to the types defined in the ontology. If they are not, the request will be rejected. It is recommended to enforce types on attributes by applying XML Schema validations (restrictions).

Links (object property) to a `knora-base:Resource` can be represented using the data type standoff class `knora-base::StandoffLinkTag`, internal links using the data type standoff class `knora-base:StandoffInternalReferenceTag`.

Validating a Mapping and sending it to Knora

A mapping can be validated before sending it to Knora with the following XML Schema file: `webapi/src/resources/mappingXMLToStandoff.xsd`. Any mapping that does not conform to this XML Schema file will be rejected by Knora.

The mapping has to be sent as a multipart request to the standoff route using the path segment `mapping`:

```
HTTP POST http://host/v1/mapping
```

The multipart request consists of two named parts:

- "json" ->:

```
{
  "project_id": "projectIRI",
  "label": "my mapping",
  "mappingName": "MappingNameSegment"
}
```

- "xml" ->:

```
<?xml version="1.0" encoding="UTF-8"?>
<mapping>
    ...
</mapping>
```

A successful response returns the Iri of the mapping. However, the Iri of a mapping is predictable: it consists of the project Iri followed by `/mappings/` and the `mappingName` submitted in the JSON (if the name already exists, the request will be rejected). Once created, a mapping can be used to create TextValues in Knora. The formats are documented in the typescript interfaces `addMappingRequest` and `addMappingResponse` in module `mappingFormats`

Adding Resources

In order to create a resource, the HTTP method `POST` has to be used. The request has to be sent to the Knora server using the `resources` path segment:

```
HTTP POST to http://host/v1/resources
```

Unlike in the case of GET requests, the request body consists of JSON describing the resource to be created.

Creating resources requires authentication since only known users may add resources.

Adding Resources without a digital Representation

The format of the JSON used to create a resource without a digital representation is described in the TypeScript interface `createResourceWithoutRepresentationRequest` in module `createResourceFormats`. It requires the IRI of the resource class the new resource belongs to, a label describing the new resource, the IRI of the project the new resource belongs to, and the properties to be assigned to the new resource.

The request header's content type has to be set to `application/json`.

Adding Resources with a digital Representation

Certain resource classes allow for digital representations (e.g. an image). There are two ways to attach a file to a resource: Either by submitting directly the binaries of the file in a HTTP Multipart request or by indicating the location of the file. The two cases are referred to as Non GUI-case and GUI-case (see *Interaction Between Sipi and Knora*).

Including the binaries (Non GUI-case)

In order to include the binaries, a HTTP Multipart request has to be sent. One part contains the JSON (same format as described for *Adding Resources without a digital Representation*) and has to be named `json`. The other part contains the file's name, its binaries, and its mime type and has to be named `file`. The following example illustrates how to make this type of request using Python3:

```
#!/usr/bin/env python3

import requests, json

# a Python dictionary that will be turned into a JSON object
resourceParams = {
   'restype_id': 'http://www.knora.org/ontology/test#testType',
   'properties': {
       'http://www.knora.org/ontology/test#testtext': [
           {'richtext_value': {'utf8str': "test", 'textattr': json.dumps({}), 'resource_reference
       ],
       'http://www.knora.org/ontology/test#testnumber': [
           {'int_value': 1}
       ]
   },
   'label': "test resource",
   'project_id': 'http://data.knora.org/projects/testproject'
}

# the name of the file to be submitted
filename = "myimage.jpg"

# a tuple containing the file's name, its binaries and its mimetype
file = {'file': (filename, open(filename, 'rb'), "image/jpeg")} # use name "file"

# do a POST request providing both the JSON and the binaries
r = requests.post("http://host/v1/resources",
                  data={'json': json.dumps(resourceParams)}, # use name "json"
                  files=file,
                  auth=('user', 'password'))
```

Please note that the file has to be read in binary mode (by default it would be read in text mode).

Indicating the location of a file (GUI-case)

This request works similarly to *Adding Resources without a digital Representation*. The JSON format is described in the TypeScript interface `createResourceWithRepresentationRequest` in module `createResourceFormats`. The request header's content type has to set to `application/json`.

In addition to *Adding Resources without a digital Representation*, the (temporary) name of the file, its original name, and mime type have to be provided (see *GUI-Case*).

Response to a Resource Creation

When a resource has been successfully created, Knora sends back a JSON containing the new resource's IRI (`res_id`) and its properties. The resource IRI identifies the resource and can be used to perform future Knora API V1 operations.

The JSON format of the response is described in the TypeScript interface `createResourceResponse` in module `createResourceFormats`.

Changing a resource's label

A resource's label can be changed by making a PUT request to the path segments `resources/label`. The resource's Iri has to be provided in the URL (as its last segment). The new label has to submitted as JSON in the HTTP request's body.

```
HTTP PUT to http://host/v1/resources/label/resourceIRI
```

The JSON format of the request is described in the TypeScript interface `changeResourceLabelRequest` in module `createResourceFormats`. The response is described in the TypeScript interface `changeResourceLabelResponse` in module `createResourceFormats`.

Adding Multiple Resources in a Single Request

Multiple resources can be created in a single request. This is especially useful if the resources have links to one another. The entire request will be checked for consistency as a whole.

The resources to be created must be described in an XML file. The XML file containing the resource descriptions can be imported directly to Knora by a POST request. The request has to be sent to the Knora server using the `resources/xml` path segment.

```
HTTP POST to http://host/v1/resources/xml
```

XML File Format

The ontologies containing the resource classes must be given as XML namespaces. For example, if resource classes from the `beol` and `biblio` ontologies are used in the XML file, these ontologies can be specified as follows:

```
<xml xmlns:beol="http://www.knora.org/ontology/beol"
   xmlns:biblio="http://www.knora.org/ontology/biblio">
```

Each XML element representing a resource or property must have the name of a resource class or property defined in one of the specified ontologies. The cardinalities defined in the ontologies must also be respected. For example, if the resource class `person` in the `beol` ontology has the properties `hasGivenName` and `hasFamilyName`, a `person` resource could be created as follows:

```
<beol:person id="abel">
      <beol:hasGivenName>Niels Henrik</beol:hasGivenName>
      <beol:hasFamilyName>Abel</beol:hasFamilyName>
</beol:person>
```

Every resource must have an `id` attribute containing a unique identifier, which will be stored as its `rdfs:label`.

The property values of resources should be in the format specified for that property in the ontology. For example, if a property is defined in the ontology as having a value of type `knora-base:DateValue`, a Knora date string must be submitted as its value in the XML, e.g.:

```
<biblio:publicationHasDate>GREGORIAN:1974</biblio:publicationHasDate>
```

An element representing a link to another resource must have a child element specifying the type of the target resource, and a `ref` attribute referring to the `id` attribute of the XML element representing the target resource. For example:

```
<biblio:publicationHasAuthor>
   <beol:person ref="abel"/>
</biblio:publicationHasAuthor>
```

Reading Values

In order to get an existing value, the HTTP method GET has to be used. The request has to be sent to the Knora server using the `values` path segment. Reading values may require authentication since some resources may have restricted viewing permissions.

Reading a Value

The representation of a value can be obtained by making a GET request providing the value's IRI:

```
HTTP GET to http://host/v1/values/valueIRI
```

In the response, the value's type and value are returned (see TypeScript interface `valueResponse` in module `valueResponseFormats`).

Getting a Value's Version History

In order to get the history of a value (its current and previous versions), the IRI of the resource it belongs to, the IRI of the property type that connects the resource to the value, and its **current** value IRI have to be submitted. Each of these elements is appended to the URL and separated by a slash. Please note that all of these have to be URL encoded.

Additionally to `values`, the path segment `history` has to be used:

```
HTTP GET to http://host/v1/values/history/resourceIRI/propertyTypeIRI/valueIRI
```

In the response, the value's versions returned (see TypeScript interface `valueVersionsResponse` in module `valueResponseFormats`).

Getting a Linking Value

In order to get information about a link between two resources, the path segment `links` has to be used. The IRI of the source object, the IRI of the property type linking the the two objects, and the IRI of the target object have to be provided in the URL separated by slashes. Each of these has to be URL encoded.

```
HTTP GET to http://host/links/sourceObjectIRI/linkingPropertyIRI/targetObjectIRI
```

In the response, information about the link is returned such as a reference count indicating how many links of the specified direction (source to target) and type (property) between the two objects exist (see TypeScript interface `linkResponse` in module `valueResponseFormats`).

Adding a Value

In order to add values to an existing resource, the HTTP method `POST` has to be used. The request has to be sent to the Knora server using the `values` path segment. Creating values requires authentication since only known users may add values.

Adding a Property Value

In order to add a value to a resource, its property type, value, and project has to be indicated in the JSON. Also the IRI of the resource the new value belongs has to be provided in the JSON.

```
HTTP POST to http://host/v1/values
```

Depending on the type of the new value, one of the following formats (all TypeScript interfaces defined in module `addValu`

- `addRichtextValueRequest`
- `addLinkValueRequest`
- `addIntegerValueRequest`
- `addDecimalValueRequest`
- `addBooleanValueRequest`
- `addUriValueRequest`
- `addDateValueRequest` (see `dateString` in `basicMessageComponents` for the date format)
- `addColorValueRequest`
- `addGeometryValueRequest`
- `addHierarchicalListValueRequest`
- `addintervalValueRequest`
- `addGeonameValueRequest`

Response on Value Creation

When a value has been successfully created, Knora sends back a JSON with the new value's IRI. The value IRI identifies the value and can be used to perform future Knora API V1 operations.

The JSON format of the response is described in the TypeScript interface `addValueResponse` in module `addValueFormats`.

Changing a Value

In order to add values to an existing resource, the HTTP method `PUT` has to be used. Changing values requires authentication since only known users may change values.

Modifying a Property Value

The request has to be sent to the Knora server using the `values` path segment followed by the value's IRI:

```
HTTP PUT to http://host/values/valueIRI
```

Please note that the value IRI has to be URL encoded.

In order to change an existing value (creating a new version of it), the value's current IRI and its new value have to be submitted as JSON in the HTTP body.

Depending on the type of the new value, one of the following formats (all TypeScript interfaces defined in module `changeV`

- `changeRichtextValueRequest`
- `changeLinkValueRequest`
- `changeIntegerValueRequest`
- `changeDecimalValueRequest`
- `changeBooleanValueRequest`
- `changeUriValueRequest`
- `changeDateValueRequest`
- `changeColorValueRequest`
- `changeGeometryValueRequest`
- `changeHierarchicalListValueRequest`
- `changeIntervalValueRequest`
- `changeGeonameValueRequest`

Modifying a File Value

In order to exchange a file value (digital representation of a resource), the path segment `filevalue` has to be used. The IRI of the resource whose file value is to be exchanged has to be appended:

```
HTTP PUT to http://host/filevalue/resourceIRI
```

Please note that the resource IRI has to be URL encoded.

There are two ways to change a file of a resource: Either by submitting directly the binaries of the file in a HTTP Multipart request or by indicating the location of the file. The two cases are referred to as Non GUI-case and GUI-case (see *Interaction Between Sipi and Knora*).

Including the binaries (Non GUI-case)

Here, a HTTP MULTIPART request has to be made simply providing the binaries (without JSON):

```
#!/usr/bin/env python3

import requests, json, urllib

# the name of the file to be submitted
filename = 'myimage.tif'

# a tuple containing the file's name, its binaries and its mimetype
files = {'file': (filename, open(filename, 'rb'), "image/tiff")}

resIri = urllib.parse.quote_plus('http://data.knora.org/xy')

r = requests.put("http://host/filevalue/" + resIri,
                 files=files)
```

Please note that the file has to be read in binary mode (by default it would be read in text mode).

Indicating the location of a file (GUI-case)

Here, simply the location of the new file has to be submitted as JSON. The JSON format is described in the TypeScript interface `changeFileValueRequest` in module `changeValueFormats`. The request header's content type has to set to `application/json`.

Response on Value Change

When a value has been successfully changed, Knora sends back a JSON with the new value's IRI. The value IRI identifies the value and can be used to perform future Knora API V1 operations.

The JSON format of the response is described in the TypeScript interface `changeValueResponse` in module `changeValueFormats`.

Deleting Resources and Values

Knora does not actually delete resources or values; it just marks them as deleted. To mark a resource or value as deleted, you must use the HTTP method `DELETE` has to be used. This requires authentication.

Mark a Resource as Deleted

The delete request has to be sent to the Knora server using the `resources` path segment.

```
HTTP DELETE to http://host/resources/resourceIRI?deleteComment=String
```

The resource IRI must be URL-encoded. The `deleteComment` is an optional comment explaining why the resource is being marked as deleted.

Mark a Value as Deleted

The delete request has to be sent to the Knora server using the `values` path segment, providing the valueIRI:

```
HTTP DELETE to http://host/values/valueIRI?deleteComment=String
```

The value IRI must be URL-encoded. The `deleteComment` is an optional comment explaining why the value is being marked as deleted.

Once a value has been marked as deleted, no new versions of it can be made.

CHAPTER 5

SALSAH

SALSAH - System for Annotation and Linkage of Sources in Arts and Humanities

Developing SALSAH

Build Process

TODO: complete this file.

- SBT

Building and Running

Start the provided Fuseki triplestore:

```
$ cd KNORA_PROJECT_DIRECTORY/triplestores/fuseki
$ ./fuseki-server
```

Then in another terminal, load some test data into the triplestore:

```
$ cd KNORA_PROJECT_DIRECTORY/webapi/scripts
$ ./fuseki-load-test-data.sh
```

Then go back to the webapi root directory and use SBT to start the API server:

```
$ cd KNORA_PROJECT_DIRECTORY/webapi
$ sbt
> compile
> re-start allowResetTriplestoreContentOperationOverHTTP
```

Then in another terminal, go to the SIPI project root directory and start the server:

```
$ ./local/bin/sipi --config=config/sipi.knora-config.lua (for production)
$ ./local/bin/sipi --config=config/sipi.knora-test-config.lua (for running tests)
```

Then in another terminal, go to the SALSAH root directory and start the server:

```
$ cd KNORA_PROJECT_DIRECTORY/salsah
$ sbt
> compile
> re-start
```

To shut down the SALSAH server:

```
> re-stop
```

Run the automated tests

In order to run the tests, the Selenium driver for Chrome has to be installed.

It is architecture-dependant, please go to `salsah/lib/chromedriver` directory and unzip the distribution that matches your architecture, or download it from here and install it in this directory.

Then, launch the services as described above; the triple store with the test data, the api server with the `allowResetTriplestoreContentOperationOverHTTP` option, sipi with the test configuration and salsah where you can run the tests in the same SBT session:

```
$ cd KNORA_PROJECT_DIRECTORY/salsah
$ sbt
> compile
> re-start
> test
```

Note: please be patient as salsah can take up to one mimute (end of a time-out) before reporting some errors.

SBT Build Configuration

```
import sbt._
import sbt.Keys.{licenses, mainClass, mappings, _}
import spray.revolver.RevolverPlugin._
import com.typesafe.sbt.SbtNativePackager.autoImport._
import com.typesafe.sbt.packager.MappingsHelper.{contentOf, directory}

lazy val salsah = (project in file(".")).
        settings(salsahCommonSettings:  _*).
        settings(
            libraryDependencies ++= salsahLibs,
            logLevel := Level.Info,
            fork in run := true,
            javaOptions in run ++= javaRunOptions,
            mainClass in (Compile, run) := Some("org.knora.salsah.Main"),
            fork in Test := true,
            javaOptions in Test ++= javaTestOptions,
            parallelExecution in Test := false,
            /* show full stack traces and test case durations */
            testOptions in Test += Tests.Argument("-oDF")
        ).
        settings( // enable deployment staging with `sbt stage`
            mappings in Universal ++= {
                // copy the public folder
                directory("src/public") ++
                // copy configuration files to config directory
                contentOf("src/main/resources").toMap.mapValues("config/" + _)
            },
            // add 'config' directory first in the classpath of the start script,
            scriptClasspath := Seq("../config/") ++ scriptClasspath.value,
            // add license
            licenses := Seq(("GNU AGPL", url("https://www.gnu.org/licenses/agpl-3.0"))),
            // need this here, but why?
            mainClass in Compile := Some("org.knora.salsah.Main")).
        settings(Revolver.settings: _*).
        enablePlugins(JavaAppPackaging) // Enable the sbt-native-packager plugin
```

```
lazy val salsahCommonSettings = Seq(
    organization := "org.knora",
    name := "salsah",
    version := "0.1.0",
    scalaVersion := "2.11.7"
)

lazy val javaRunOptions = Seq(
    // "-showversion",
    "-Xms2048m",
    "-Xmx4096m"
    // "-verbose:gc",
    //"-XX:+UseG1GC",
    //"-XX:MaxGCPauseMillis=500"
)

lazy val javaTestOptions = Seq(
    // "-showversion",
    "-Xms2048m",
    "-Xmx4096m"
    // "-verbose:gc",
    //"-XX:+UseG1GC",
    //"-XX:MaxGCPauseMillis=500",
    //"-XX:MaxMetaspaceSize=4096m"
)

lazy val salsahLibs = Seq(
    // akka
    "com.typesafe.akka" % "akka-http-core-experimental_2.11" % "2.0-M2",
    "com.typesafe.akka" % "akka-http-experimental_2.11" % "2.0-M2",
    "com.typesafe.akka" % "akka-http-spray-json-experimental_2.11" % "2.0-M2",
    "com.typesafe.akka" % "akka-http-xml-experimental_2.11" % "2.0-M2",
    // testing
    "com.typesafe.akka" %% "akka-http-testkit-experimental" % "2.0-M2" % "test",
    "org.scalatest" %% "scalatest" % "2.2.5" % "test",
    "org.seleniumhq.selenium" % "selenium-java" % "2.35.0" % "test",
    "io.spray" %% "spray-http" % "1.3.3",
    "io.spray" %% "spray-httpx" % "1.3.3",
    "io.spray" %% "spray-util" % "1.3.3",
    "io.spray" %% "spray-io" % "1.3.3",
    "io.spray" %% "spray-can" % "1.3.3",
    "io.spray" %% "spray-caching" % "1.3.3",
    "io.spray" %% "spray-routing" % "1.3.3",
    "io.spray" %% "spray-json" % "1.3.2",
    "io.spray" %% "spray-client" % "1.3.2"
)
```

SALSAH Design Documentation

SALSAH Design Overview

Introduction

Some text about salsah version 2

CHAPTER 6

Sipi

Sipi is a high-performance media server written in C++, for serving and converting binary media files such as images and video. Sipi can efficiently convert between many different formats on demand, preserving embedded metadata, and implements the International Image Interoperability Framework (IIIF). Knora is designed to use Sipi for converting and serving media files.

Setup Sipi for Knora

Setup and Execution

In order to serve files to the client application like the Salsah GUI, Sipi must be set up and running. Sipi can be downloaded from its own github-repository: https://github.com/dhlab-basel/Sipi. Please follow the instructions given in the README to compile it on your system.

Once it is compiled, you can run Sipi with the following option: `./local/bin/sipi --config=config/sipi.knora-config.lua` (or `./local/bin/sipi --config=sipi.knora-test-config.lua` for using sipi for testing). Please see `sipi.knora-config.lua` for the settings like URL, port number etc. These settings need to be set accordingly in Knora's `application.conf`. If you use the default settings both in Sipi and Knora, there is no need to change these settings.

Whenever a file is requested from Sipi (e.g. a browser trying to dereference an image link served by Knora), a preflight function is called. This function is defined in `sipi.init-knora.lua` present in the Sipi root directory. It takes three parameters: `prefix`, `identifier` (the name of the requested file), and `cookie`. File links created by Knora use the prefix `knora`, e.g. `http://localhost:1024/knora/incunabula_0000000002.jp2/full/2613,3505/0/default.jpg`.

Given these information, Sipi asks Knora about the current's users permissions on the given file. The cookie contains the current user's Knora session id, so Knora can match Sipi's request with a given user profile and determine the permissions this user has on the file. If the Knora response grants sufficient permissions, the file is served in the requested quality. If the suer has preview rights, Sipi serves a reduced quality or integrates a watermark. If the user has no permissions, Sipi refuses to serve the file. However, all of this behaviour is defined in the preflight function in Sipi and not controlled by Knora. Knora only provides the permission code.

See *Sharing the Session ID with Sipi* for more information about sharing the session id.

Using Sipi in Test Mode

If you just want to test Sipi with Knora without serving the actual files (e.g. when executing browser tests), you can simply start Sipi like this: `./local/bin/sipi --config=config/sipi.knora-test-config.lua`. Then always the same test file will be served which is included in Sipi. In test mode, Sipi will not aks Knora about the user's permission on the requested file.

Interaction Between Sipi and Knora

General Remarks

Knora and Sipi (Simple Image Presentation Interface) are two **complementary** software projects. Whereas Knora deals with data that is written to and read from a triplestore (metadata and annotations), Sipi takes care of storing, converting and serving image files as well as other types of files such as audio, video, or documents (binary files it just stores and serves).

Knora and Sipi stick to a clear division of responsibility regarding files: Knora knows about the names of files that are attached to resources as well as some metadata and is capable of creating the URLs for the client to request them from Sipi, but the whole handling of files (storing, naming, organization of the internal directory structure, format conversions, and serving) is taken care of by Sipi.

Adding Files to Knora: Using the GUI or directly the API

To create a resource with a digital representation attached to, either the browser-based GUI (SALSAH) can be used or this can be done by *directly* [1] addressing the API. The same applies for changing an existing digital representation for a resource. Subsequently, the first case will be called the *GUI-case* and the second the *non GUI-case*.

GUI-Case

In this case, the user may choose a file to upload using his web-browser. The file is directly sent to Sipi (route: `create_thumbnail`) to calculate a thumbnail hosted by Sipi which then gets displayed to the user in the browser. Sipi copies the original file into a temporary directory and keeps it there (for later processing in another request). In its answer (JSON), Sipi returns:

- `preview_path`: the path to the thumbnail (accessible to a web-browser)
- `filename`: the name of the temporarily stored original file (managed by Sipi)
- `original_mimetype`: mime type of the original file
- `original_filename`: the original name of the file submitted by the client

Once the user finally wants to attach the file to a resource, the request is sent to Knora's API providing all the required parameters to create the resource along with additional information about the file to be attached. **However, the file itself is not submitted to the Knora Api, but its filename returned by Sipi.**

Create a new Resource with a Digital Representation

The POST request is handled in `ResourcesRouteV1.scala` and parsed to a `CreateResourceApiRequestV1`. Information about the file is sent separately from the other resource parameters (properties) under the name `file`:

- `originalFilename`: original name of the file (returned by Sipi when creating the thumbnail)
- `originalMimeType`: original mime type of the file (returned by Sipi when creating the thumbnail)
- `filename`: name of the temporarily stored original file (returned by Sipi when creating the thumbnail)

In the route, a `SipiResponderConversionFileRequestV1` is created representing the information about the file to be attached to the new resource. Along with the other parameters, it is sent to the resources responder.

See *Further Handling of the GUI and the non GUI-case in the Resources Responder* for details of how the resources responder then handles the request.

[1] Of course, also the GUI uses the API. But the user does not need to know about it.

Change the Digital Representation of a Resource

The request is taken care of in `ValuesRouteV1.scala`. The PUT request is handled in path `v1/filevalue/{resIri}` which receives the resource Iri as a part of the URL: *The submitted file will update the existing file values of the given resource.*

The file parameters are submitted as json and are parsed into a `ChangeFileValueApiRequestV1`. To represent the conversion request for the Sipi responder, a `SipiResponderConversionFileRequestV1` is created. A `ChangeFileValueRequestV1` containing the resource Iri and the message for Sipi is then created and sent to the values responder.

See *Further Handling of the GUI and the non GUI-case by the Values Responder* for details of how the values responder then handles the request.

Non GUI-Case

In this case, the API receives an HTTP multipart request containing the binary data.

Create a new Resource with a Digital Representation

The request is handled in `ResourcesRouteV1.scala`. The multipart POST request consists of two named body parts: `json` containing the resource parameters (properties) and `file` containing the binary data as well as the file name and its mime type. Using Python's request module, a request could look like this:

```
import requests, json

params = {...} // resource parameters
files = {'file': (filename, open(path + filename, 'rb'), mimetype)} // filename, binary data,

r = requests.post(knora_url + '/resources',
                  data={'json': json.dumps(params)},
                  files=files,
                  headers=None)
```

The binary data is saved to a temporary location by Knora. The route then creates a `SipiResponderConversionPathRequestV1` representing the information about the file (i.e. the temporary path to the file) to be attached to the new resource. Along with the other parameters, it is sent to the resources responder.

See *Further Handling of the GUI and the non GUI-case in the Resources Responder* for details of how the resources responder then handles the request.

Change the Digital Representation of a Resource

The request is taken care of in `ValuesRouteV1.scala`. The multipart PUT request is handled in path `v1/filevalue/{resIri}` which receives the resource Iri as a part of the URL: *The submitted file will update the existing file values of the given resource.*

For the request, no json parameters are required. So its body just consists of the binary data (cf. *Python code example*). The values route stores the submitted binaries as a temporary file and creates a `SipiResponderConversionPathRequestV1`. A `ChangeFileValueRequestV1` containing the resource Iri and the message for Sipi is then created and sent to the values responder.

See *Further Handling of the GUI and the non GUI-case by the Values Responder* for details of how the values responder then handles the request.

Further Handling of the GUI and the non GUI-case in the Resources Responder

Once a `SipiResponderConversionFileRequestV1` (GUI-case) or a `SipiResponderConversionPathRequestV1` (non GUI-case) has been created and passed to the resources responder, the GUI and the non GUI-case can be handled in a very similar way. This is why they are both implementations of the trait `SipiResponderConversionRequestV1`.

The resource responder calls the ontology responder to check if all required properties were submitted for the given resource type. Also it is checked if the given resource type may have a digital representation. The resources responder then sends a message to Sipi responder that does a request to the Sipi server. Depending on the type of the message (`SipiResponderConversionFileRequestV1` or `SipiResponderConversionPathRequestV1`), a different Sipi route is called. In the first case (GUI-case), the file is already managed by Sipi and only the filename has to be indicated. In the latter case, Sipi is told about the location where Knora has saved the binary data to.

To make this handling easy for Knora, both messages have their own implementation for creating the parameters for Sipi (declared in the trait as `toFormData`). If Knora deals with a `SipiResponderConversionPathRequestV1`, it has to delete the temporary file after it has been processed by SIPI. Here, we assume that we deal with an image.

For both cases, Sipi returns the same answer containing the following information:

- `file_type`: the type of the file that has been handled by Sipi (image | video | audio | text | binary)
- `mimetype_full` and `mimetype_thumb`: mime types of the full image representation and the thumbnail
- `original_mimetype`: the mime type of the original file
- `original_filename`: the name of the original file
- `nx_full`, `ny_full`, `nx_thumb`, and `ny_thumb`: the x and y dimensions of both the full image and the thumbnail
- `filename_full` and `filename_full`: the names of the full image and the thumbnail (needed to request the images from Sipi)

The `file_type` is important because representations for resources are restricted to media types: image, audio, video or a generic binary file. If a resource type requires an image representations (subclass of `StillImageRepresentation`), the `file_type` has to be an image. Otherwise, the ontology's restrictions would be violated. Because of this requirement, there is a construct `fileType2FileValueProperty` mapping file types to file value properties. Also all the possible file types are defined in enumeration.

Depending on the given file type, Sipi responder can create the apt message (here: `StillImageFileValueV1`) to save the data to the triplestore.

Further Handling of the GUI and the non GUI-case by the Values Responder

In the values responder, `ChangeFileValueRequestV1` is passed to the method `changeFileValueV1`. Unlike ordinary value change requests, the Iris of the value objects to be updated are not known yet. Because of this, all the existing file values of the given resource Iri have to be queried first. Also their quality levels are queried because in case of a `StillImageFileValue`, we have to deal with a file value for the thumbnail and another one for the full quality representation. When these two file values are being updated, the quality levels have to be considered for the sake of consistency (otherwise a full quality value's `knora-base:previous-value` may point to a thumbnail file value).

With the file values being returned, we actually know about the current Iris of the value objects. Now the Sipi responder is called to handle the file conversion request (cf. *Further Handling of the GUI and the non GUI-case in the Resources Responder*). After that, it is checked that the `file_type` returned by Sipi responder corresponds to the property type of the existing file values. For example, if the `file_type` is an image, the property pointing to the current file values must be a `hasStillImageFileValue`. Otherwise, the user submitted a non image file that has to be rejected.

Depending on the `file_type`, messages of type `ChangeValueRequestV1` can be created. For each existing file value, such a message is instantiated containing the current value Iri and the new value to be created (returned by the sipi responder). These messages are passed to `changeValueV1` because with the described handling done in `changeFileValueV1`, the file values can be changed like any other value type.

In case of success, a `ChangeFileValueResponseV1` is sent back to the client, containing a list of the single `ChangeValueResponseV1`.

Retrieving Files from Sipi

URL creation

Binary representions of Knora locations are served by Sipi. For each file value, Knora creates several locations representing different quality levels:

```
"resinfo": {
   "locations": [
      {
         "duration": 0,
         "nx": 95,
         "path": "http://sipiserver:port/knora/incunabula_0000000002.jpg/full/full/0/default.jpg",
         "ny": 128,
         "fps": 0,
         "format_name": "JPEG",
         "origname": "ad+s167_druck1=0001.tif",
         "protocol": "file"
      },
      {
         "duration": 0,
          "nx": 82,
          "path": "http://sipiserver:port/knora/incunabula_0000000002.jp2/full/82,110/0/default.jp
          "ny": 110,
          "fps": 0,
          "format_name": "JPEG2000",
          "origname": "ad+s167_druck1=0001.tif",
          "protocol": "file"
      },
      {
          "duration": 0,
          "nx": 163,
          "path": "http://sipiserver:port/knora/incunabula_0000000002.jp2/full/163,219/0/default.
          "ny": 219,
          "fps": 0,
          "format_name": "JPEG2000",
          "origname": "ad+s167_druck1=0001.tif",
          "protocol": "file"
      }
      ...
   ],
"restype_label": "Seite",
"resclass_has_location": true,
```

Each of these paths has to be handled by the browser by making a call to Sipi, obtaining the binary representation in the desired quality. To deal with different image quality levels, Sipi implements the IIIF standard. The different quality level paths are created by Knora in `ValueUtilV1`.

Whenever Sipi serves a binary representation of a Knora file value (indicated by using the prefix `knora` in the path), it has to make a request to Knora's Sipi responder to get the user's permissions on the requested file. Sipi's request to Knora contains a cookie with the Knora session id the user has obtained when logging in to Knora: As a response to a successful login, Knora returns the user's session id and this id is automatically sent to Sipi by the browser, setting a second cookie for the communication with Sipi. The reason the Knora session id is set in two

cookies, is the fact that cookies can not be shared among different domains. Since Knora and Sipi are likely to be running under different domains, this solution offers the necessary flexibility.

Sharing the Session ID with Sipi

Whenever a file is requested, Sipi asks Knora about the currents user's permissions on the given file. This is achieved by sharing the Knora session id with Sipi. When the user logs in to Knora using his browser, a request is sent to Sipi submitting the session id the user got back from Knora, setting a second session cookie. Now the user has two session cookies containing the same session id: one for the communication with Knora and one for the communication with Sipi. However, Sipi does not handle sessions. It just sends the given Knora session id to Knora.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

[Schmidt2016] Schmidt, Desmond. 2016. “Using Standoff Properties for Marking-up Historical Documents in the Humanities.” *It – Information Technology* 58: 1. http://ecdosis.net/papers/schmidt.d.2016.pdf.