Knora Documentation

Release 0.1

Digital Humanities Lab, University of Basel

Jun 27, 2016

Part I

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 addresses 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 includes 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.
- *SALSAH*, 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.

Part II

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-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.

CHAPTER

ONE

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 document `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.

1.1 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 `The Knora Base Ontology`.

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`.

1.2 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 `The Knora Base Ontology`, 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 f
```

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`.

Part III

The Knora Ontologies

Please read `The Knora Base Ontology`.

Part IV

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.

CHAPTER

TWO

DEPLOYING THE KNORA API SERVER

2.1 Getting Started with the Knora API Server

2.1.1 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.

TODO: explain how to get started with these.

Load Test Data

In order to load the test data, go to `webapi/scripts` and run the script for the triplestore you have chosen. In case of Fuseki, run `fuseki-load-test-data.sh`, in case of GraphDB `graphdb-se-load-test-data.sh`

When working with GraphDB, you may encounter 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. Please delete all the data and start over:

- shutdown Tomcat: `$CATALINA_HOME/bin/shutdown.sh`
- go to http://localhost:8080/openrdf-sesame/system/overview.view and look for the field `Data directory` which indicates the location of the directory `.aduna`.
- Then remove this directory an restart tomcat. Now you should be able to load the test data correctly.

2.1.2 Creating a Test Installation

TODO: write subsections like this:

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

2.2 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 generall idea should be usable on all platforms with small changes.

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

Todo

Add link to where the official Knora API server distributions can be downloaded and to the description of how to create a distribution.

The jar distribution of the server can be either run manually or as a service for which we will use `supervisord` as described in the *Supervisord* section.

The supported triplestore can also be run manually (as described in the documentation of each distribution) or it can be run under an application server as described in the *Tomcat Application Server (Fuseki 2 and GraphDB)* section.

2.2.1 Supervisord

For running Knora-API we will use supervisord, which allows us to run our java application easily as a service.

- https://serversforhackers.com/monitoring-processes-with-supervisord

Configuration

- /etc/supervisord.d/knora-api.conf

```
[program:knora-api]
command=sh run.sh
directory=/var/www/vhosts/api.knora.org
autostart=true
autorestart=true
startretries=3
stderr_logfile=/var/log/knora-api/knora-api.err.log
stdout_logfile=/var/log/knora-api/knora-api.out.log
user=root
environment=
```

We need to create the directory for the log files!

Controlling Processes

```
$ supervisorctl reread $ supervisorctl update $ supervisorctl
```

2.2.2 Tomcat Application Server (Fuseki 2 and GraphDB)

The supported triplestores for the Knora API server are Fuseki 2 and GraphDB. Both come with a `.war` packaged distribution, which alows a deployment under an application server. We chose Tomcat, but there are other options available, e.g., Glassfish, Jetty, etc.

Installation

We use yum to install Tomcat:

:: $ yum install tomcat

Configuration

Fuseki 2 and GraphDB are deployed using tomcat.

The relevant directories are as follows:

- **Tomcat Webapps folder, where both the Fuseki 2 and GraphDB `.war` file is** dropped in: `/var/lib/tomcat/webapps`
- Fuseki configuration folder: `/etc/fuseki`
- **Data folder: `/usr/share/tomcat`**
  - for GraphDB: `/usr/share/tomcat/.aduna` and `/usr/share/tomcat/.graphdb-workbench`
  - for Fuseki: `usr/share/tomcat/.fuseki`

Administration

- `systemctl status tomcat`
- `systemctl start tomcat`
- `systemctl stop tomcat`

If someting is wrong, first check the log files:

- `/var/log/tomcat/`

CHAPTER

THREE

KNORA API SERVER DESIGN DOCUMENTATION

3.1 Knora API Server Design Overview

3.1.1 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 and the spray framework for web APIs. 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.

3.1.2 Design Diagram

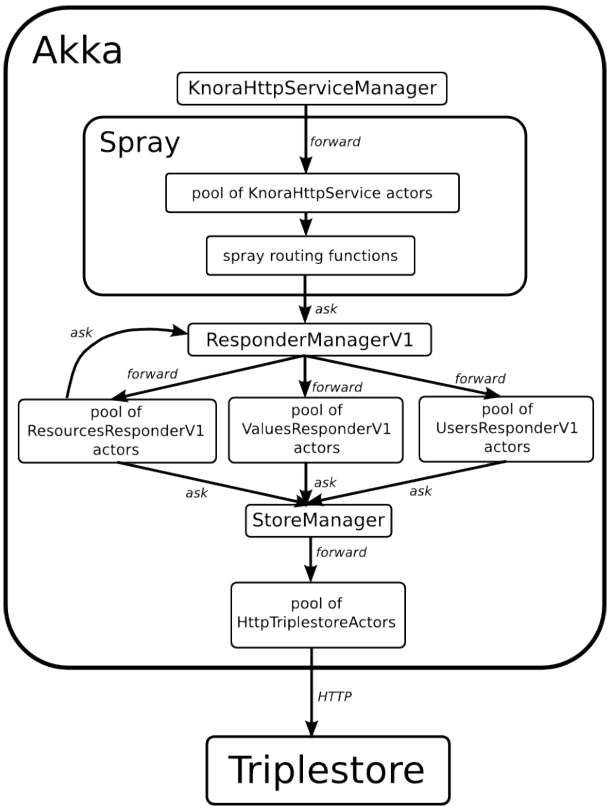

Akka
KnoraHttpServiceManager
Spray
forward
pool of KnoraHttpService actors
spray routing functions
ask
ResponderManagerV1
ask
forward
forward
forward
pool of ResourcesResponderV1 actors
pool of ValuesResponderV1 actors
pool of UsersResponderV1 actors
ask
ask
ask
StoreManager
forward
pool of HttpTriplestoreActors
HTTP
Triplestore

Fig. 3.1: A high-level diagram of the Knora API server.

3.1.3 Modules

HTTP Module

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

Responders Module

- org.knora.webapi.responders

Store Module

- org.knora.store

Shared Between Modules

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

3.1.4 Actor Supervision and Creation

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

```
val httpServiceManager = system.actorOf(Props(new KnoraHttpServiceManager with LiveActorMaker), na
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/httpServiceManager/httpServiceRouter {
            router = round-robin-pool
            nr-of-instances = 100
        }

        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
        }
    }
}
```

3.1.5 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).

3.1.6 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`).

3.1.7 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.

3.1.8 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
}
```

3.1.9 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 and report it to the actor's supervisor, 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 actor's supervisor will receive it).

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.

We still need to add error-handling strategies in supervisor actors.

See also *Futures with Akka*.

3.1.10 API Routing

The API routes in the `routing` package are defined using the DSL provided by the spray-routing 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 constructs a `Try` in which the following operations are performed:

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 `Try` to `org.knora.webapi.routing.RouteUtils.runJsonRoute()`, which takes care of sending the message to `ResponderManagerV1` and returning a response to the client, as well as handling errors.

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

3.1.11 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.

3.2 Futures with Akka

3.2.1 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`.

3.2.2 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.

3.2.3 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.

3.2.4 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
}
```

3.2.5 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`).

3.2.6 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.

3.3 HTTP Module

TODO

3.4 Responders Module

3.4.1 Version 1.0 Responders

TDOO

ResponderManagerV1

CkanResponderV1

HierarchicalListsResponderV1

OntologyResponderV1

ProjectsResponderV1

RepresentationsResponderV1

ResourcesResponderV1

SearchResponderV1

UsersResponderV1

ValuesResponderV1

Shared

- ResponderV1
- ValueUtilV1

3.5 Store Module

3.5.1 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.

3.5.2 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.).

3.5.3 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

3.5.4 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 stor
        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-demo-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())
}
```

3.6 Shared Packages

TODO

3.7 How to Add an API Route

3.7.1 Write SPARQL templates

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

3.7.2 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.

3.7.3 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.

3.7.4 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 rapierPath(_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 requestMessageTry = Try {
                    val userProfile = getUserProfileV1(requestContext)
                    makeRequestMessage(iri, userProfile)
                }

                RouteUtils.runJsonRoute(
                    requestMessageTry,
                    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)
    }
}
```

It's important that inside the `get` (or `post` or whatever), everything before the call to `RouteUtils.runJsonRoute` is wrapped in a `Try` constructor. This allows `RouteUtils.runJsonRoute` to handle input validation errors.

Finally, add your `rapierPath()` function to the `routes` member variable in `KnoraHttpService`.

3.8 Triplestore Updates

3.8.1 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.
- Delete a resource.

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 deleted, 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.

'Deleting' a value means creating a new version, marked with `knora-base:isDeleted` and pointing to the previous version. A triple then points from the resource to this new, deleted version of the value. 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.

Unlike values, resources (which belong to 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. A resource can be deleted by marking it with `knora-base:isDeleted` and a timestamp. Once this is done, the resource 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.

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.

3.8.2 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.

Consistency Checks

Knora enforces consistency constraints in three ways: by doing pre-update checks, by doing checks in the `WHERE` clauses of SPARQL updates, and by using GraphDB's built-in consistency checker. 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.

GraphDB's consistency checker can be turned on to ensure that each update transaction respects the consistency constraints, as described in the section Consistency checks of the GraphDB documentation. This makes it possible to catch consistency constraint violations caused by bugs in Knora, and it also checks data that is uploaded directly into the triplestore without going through the Knora API. However, this feature is only partly enabled, because of problems described in issue 33.

GraphDB's consistency checker requires the repository to be created with reasoning enabled. GraphDB's reasoning rules are defined in rule files with the `.pie` filename extension, as described in Reasoning in the GraphDB documentation. To use consistency checking, it is necessary to modify one of GraphDB's standard `.pie` files by adding consistency rules. 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.

A GraphDB consistency rule is composed of two parts: a pattern that will match if corresponding triples are found in the data, and an optional pattern that will match if corresponding triples are not found in the data. If both parts match, this means that there is a consistency violation, and the transaction will be rolled back. A rule is written in this form:

```
Consistency: <rule name>
    <pattern for triples found in the data>
    -------------------------------
    <pattern for triples not found in the data>
```

The triple patterns can contain variable names for subjects, predicates, and objects, as well as actual property names.

The consistency rules are currently being revised, so here are just two examples.

knora-base:subjectClassConstraint

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

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

knora-base:objectClassConstraint

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

If resource `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.

3.8.3 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:hasDeletePermisson knora-admin:Owner ;
              knora-base:hasModifyPermission knora-admin:ProjectMember ;
              knora-base:hasViewPermission knora-admin:KnownUser ,
                                           knora-admin:UnknownUser ;
              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:hasDeletePermisson knora-admin:Owner ;
              knora-base:hasModifyPermission knora-admin:ProjectMember ;
              knora-base:hasViewPermission knora-admin:KnownUser ,
                                           knora-admin:UnknownUser ;
              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.

3.9 Authentication in the Knora API Server

3.9.1 Scope

Authentication is the process of making sure that if someone is accessing something that 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.

3.9.2 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.

3.9.3 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.

Username 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.

Workflow

1. The login form on the client can use */v1/authentication* to check if the username/password combination provide by the user is correct. The username and password can be provided as URL parameters

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

3.10 Plans for Knora API v2

3.10.1 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.

3.10.2 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.

3.10.3 Redundancy

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

3.10.4 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.

3.10.5 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.

3.10.6 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.

3.10.7 Annotating values

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

3.10.8 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.

3.10.9 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.

CHAPTER FOUR

DEVELOPING THE KNORA API SERVER

4.1 Build Process

TODO: complete this file.

- SBT
- Using GraphDb for development and how to initializing the 'knora-test-unit' repository
- Using Fuseki for development
- Using embedded JenaTDB
- Using docker for all of the above

4.1.1 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
```

To shut down the Knora API server:

```
> re-stop
```

4.1.2 Run the automated tests

Make sure you've started Fuseki as shown above. Then at the SBT prompt:

```
> fuseki:test
```

4.1.3 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  sudo: required
2  git:
3      depth: 5
4  language: scala
5  scala:
6  - 2.11.7
7  jdk:
8  - oraclejdk8
9  cache:
10   directories:
11     - $HOME/.ivy2
12 script:
13 - cd triplestores/fuseki/ && ./fuseki-server &
14 - cd webapi/ && sbt test
15 notifications:
16   slack:
17     secure: AJZARDC7P6bwjFwk6gpe+p2ozLj+bH3h83PapfCTL0xi7frHd4y6/jXOs9ac+m7ia5FlnzgBxrf0lmaE+Ikql
```

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)

4.1.4 SBT Build Configuration

```
import sbt._
import sbt.Keys._
import spray.revolver.RevolverPlugin._
import com.typesafe.sbt.SbtNativePackager.autoImport._


// Bring the sbt-aspectj settings into this build
//aspectjSettings

lazy val webapi = (project in file(".")).
        configs(
            FusekiTest,
            FusekiTomcatTest,
            GraphDBTest,
            GraphDBFreeTest,
            SesameTest,
            EmbeddedJenaTDBTest
        ).
        settings(webApiCommonSettings:  _*).
        settings(inConfig(FusekiTest)(
            Defaults.testTasks ++ Seq(
                fork := true,
                javaOptions ++= javaFusekiTestOptions,
                testOptions += Tests.Argument("-oDF")
            )
        ): _*).
        settings(inConfig(FusekiTomcatTest)(
            Defaults.testTasks ++ Seq(
                fork := true,
                javaOptions ++= javaFusekiTomcatTestOptions,
                testOptions += Tests.Argument("-oDF")
            )
        ): _*).
        settings(inConfig(GraphDBTest)(
            Defaults.testTasks ++ Seq(
                fork := true,
                javaOptions ++= javaGraphDBTestOptions,
                testOptions += Tests.Argument("-oDF")
            )
        ): _*).
        settings(inConfig(GraphDBFreeTest)(
            Defaults.testTasks ++ Seq(
                fork := true,
                javaOptions ++= javaGraphDBFreeTestOptions,
                testOptions += Tests.Argument("-oDF")
            )
        ): _*).
        settings(inConfig(SesameTest)(
            Defaults.testTasks ++ Seq(
                fork := true,
                javaOptions ++= javaSesameTestOptions,
                testOptions += Tests.Argument("-oDF")
            )
        ): _*).
        settings(inConfig(EmbeddedJenaTDBTest)(
            Defaults.testTasks ++ Seq(
                fork := true,
                javaOptions ++= javaEmbeddedJenaTDBTestOptions,
                testOptions += Tests.Argument("-oDF")
            )
        ): _*).
        settings(
            //javaOptions in FusekiTest ++= javaFusekiTestOptions,
            //javaOptions in FusekiTomcatTest ++= javaFusekiTomcatTestOptions,
            //javaOptions in GraphDBTest ++= javaGraphDBTestOptions,
            //javaOptions in EmbeddedJenaTDBTest ++= javaEmbeddedJenaTDBTestOptions,
            //fork in FusekiTest := true,
            //parallelExecution in FusekiTest := false,
            //testOptions in FusekiTest += Tests.Argument("-oDF")
        ).
        settings(
            libraryDependencies ++= webApiLibs,
            scalacOptions ++= Seq("-feature", "-deprecation", "-Yresolve-term-conflict:package"),
            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,
            testOptions in Test += Tests.Argument("-oDF") // show full stack traces and test case
        ).
        settings(Revolver.settings: _*).
        enablePlugins(SbtTwirl) // Enable the SbtTwirl plugin

lazy val webApiCommonSettings = Seq(
    organization := "org.knora",
    name := "webapi",
    version := "0.1.0",
    ivyScala := ivyScala.value map { _.copy(overrideScalaVersion = true) },
    scalaVersion := "2.11.7"
)

lazy val webApiLibs = Seq(
    // akka
    "com.typesafe.akka" % "akka-actor_2.11" % "2.4.0",
    "com.typesafe.akka" %% "akka-agent" % "2.4.0",
    // "com.typesafe.akka" % "akka-stream-experimental_2.11" % "1.0-M3",
    // "com.typesafe.akka" % "akka-http-experimental_2.11" % "1.0-M3",
    // "com.typesafe.akka" % "akka-http-core-experimental_2.11" % "1.0-M3",
    // spray
    "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",
    // 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
    "org.slf4j" % "slf4j-api" % "1.7.12",
    "org.slf4j" % "jcl-over-slf4j" % "1.7.12",
    "ch.qos.logback" % "logback-core" % "1.1.3",
    "ch.qos.logback" % "logback-classic" % "1.1.3",
    "com.typesafe.akka" % "akka-slf4j_2.11" % "2.4.0",
    "com.typesafe.scala-logging" %% "scala-logging" % "3.1.0",
    // input validation
    "commons-validator" % "commons-validator" % "1.4.1",
    // pretty printing
    "com.googlecode.kiama" % "kiama_2.11" % "1.8.0",
    // authentication
    "com.github.t3hnar" %% "scala-bcrypt" % "2.4",
    // caching
    "net.sf.ehcache" % "ehcache" % "2.10.0",
    // monitoring
    //"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 versi
    "org.jodd" % "jodd" % "3.2.6",
    "joda-time" % "joda-time" % "2.9.1",
    "org.joda" % "joda-convert" % "1.8",
    // testing
    "com.typesafe.akka" %% "akka-testkit" % "2.4.0" % "test, fuseki, fuseki-tomcat, graphdb, tdb",
    "org.scalatest" %% "scalatest" % "2.2.6" % "test, fuseki, fuseki-tomcat, graphdb, tdb",
    "io.spray" %% "spray-testkit" % "1.3.3" % "test, fuseki, fuseki-tomcat, graphdb, tdb"
)

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

// skip test before creating fat-jar
test in assembly := {}
```

```
// set fat-jar main class
mainClass in assembly := Some("org.knora.webapi.Main")

// change merge strategy for fat-jar
assemblyMergeStrategy in assembly := {
    case PathList("org", "apache", "commons", "logging", xs @ _*)    => MergeStrategy.first
    case PathList("META-INF", xs @ _*) =>
    xs.map(_.toLowerCase) match {
        case ("manifest.mf" :: Nil) | ("index.list" :: Nil) | ("dependencies" :: Nil) =>
        MergeStrategy.discard
        case ps @ (x :: xs) if ps.last.endsWith(".sf") || ps.last.endsWith(".dsa") ||  ps.last.end
        MergeStrategy.discard
        case "plexus" :: xs =>
        MergeStrategy.discard
        case "services" :: xs =>
        MergeStrategy.filterDistinctLines
        case ("spring.schemas" :: Nil) | ("spring.handlers" :: Nil) =>
        MergeStrategy.filterDistinctLines
        case ps@(x :: xs) if ps.last.endsWith("aop.xml") => MergeStrategy.first
        case _ => MergeStrategy.deduplicate
    }
    case x =>
    val oldStrategy = (assemblyMergeStrategy in assembly).value
    oldStrategy(x)
}

// Custom run task
//lazy val generateFakeTriplestore = taskKey[Unit]("Generate fake triplestore from a list of reque

//fullRunTask(generateFakeTriplestore, Test, "org.knora.webapi.GenFakeTripleStore")
```

4.1.5 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
```

4.2 Setup IntelliJ for development of Knora

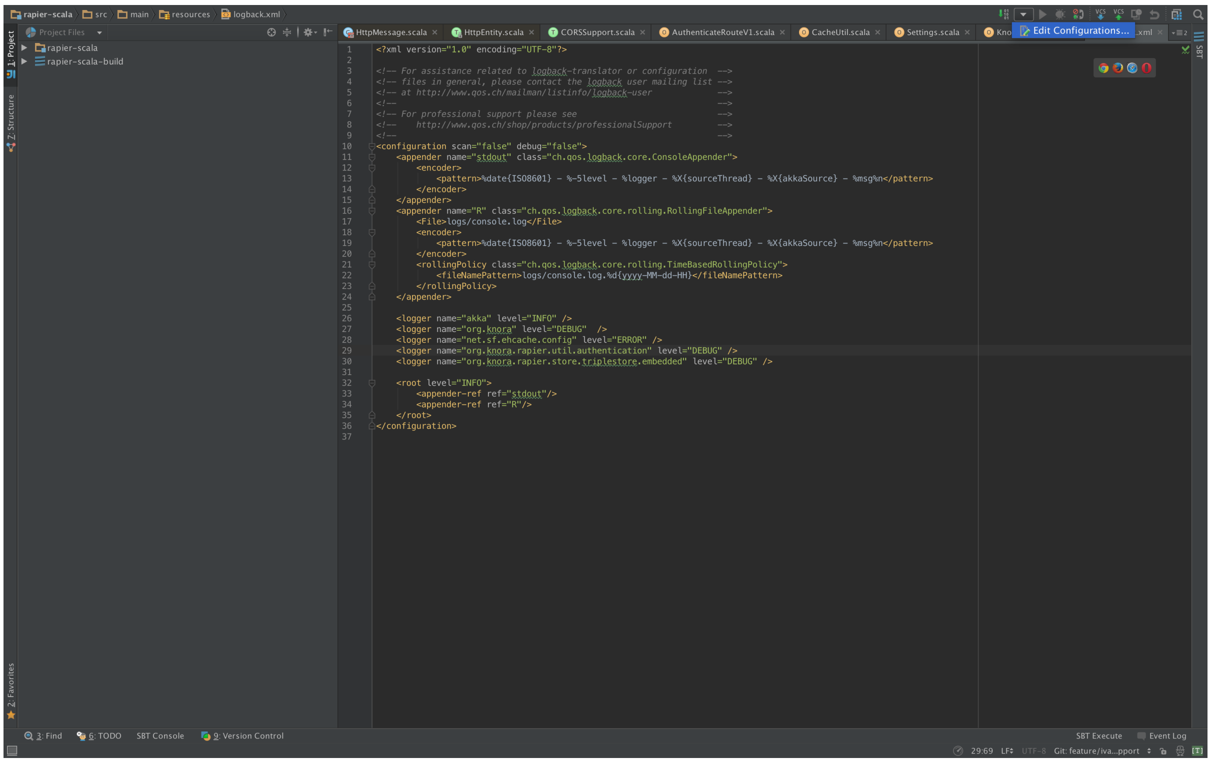
- Download and install IntelliJ
- To open the gitrep `rapier-scala` with IntelliJ's full scala support, do the following: `Import Project` -> Choose the option `module SBT`
- Then install the Scala plugin for IntelliJ
- make sure that the tab size is set correctly to **4 spaces** (so you can use automatic code reformatting): Preferences -> Code Style and also Preferences -> Code Style -> Scala.

4.2.1 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.

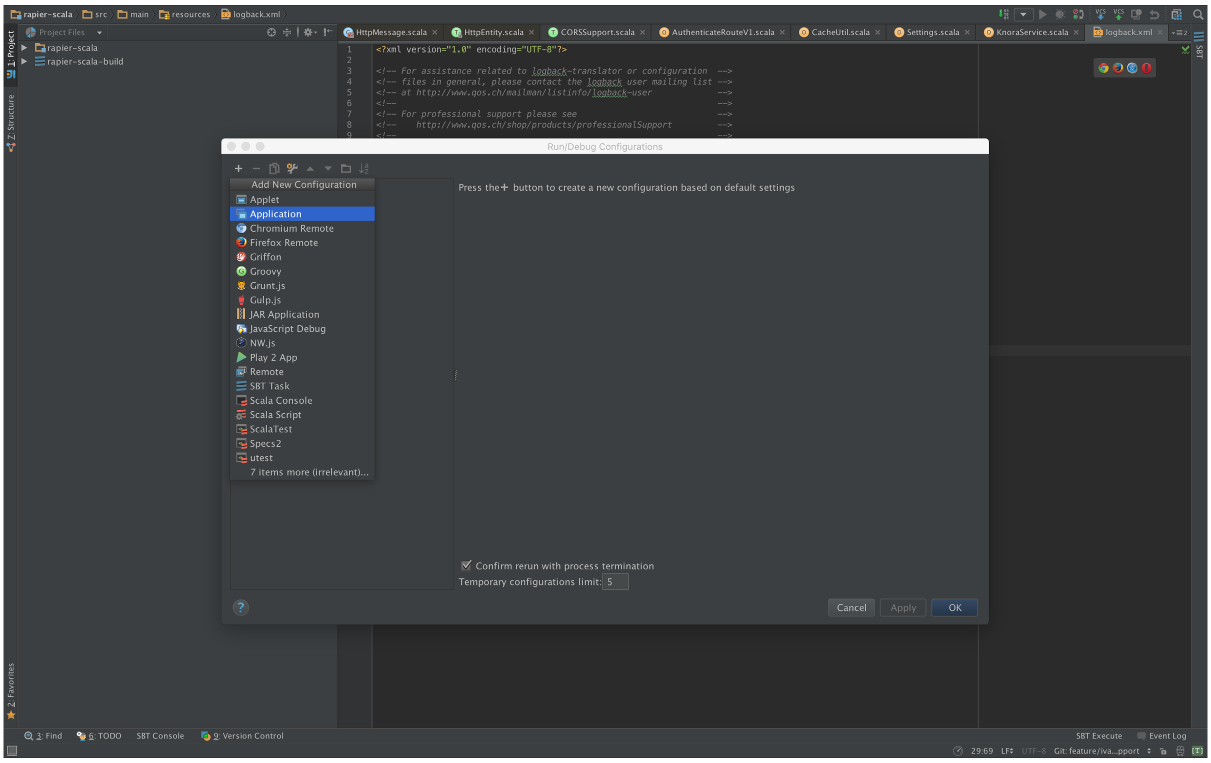
4.2.2 How Use IntelliJ IDEA's Debugger with the Knora API Server

- 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



```
<?xml version="1.0" encoding="UTF-8"?>

<!-- For assistance related to logback-translator or configuration  -->
<!-- files in general, please contact the logback user mailing list -->
<!-- at http://www.qos.ch/mailman/listinfo/logback-user             -->
<!--                                                                -->
<!-- For professional support please see                           -->
<!--    http://www.qos.ch/shop/products/professionalSupport         -->
<!--                                                                -->
<configuration scan="false" debug="false">
    <appender name="stdout" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%date{ISO8601} - %-5level - %logger - %X{sourceThread} - %X{akkaSource} - %msg%n</pattern>
        </encoder>
    </appender>
    <appender name="R" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <File>logs/console.log</File>
        <encoder>
            <pattern>%date{ISO8601} - %-5level - %logger - %X{sourceThread} - %X{akkaSource} - %msg%n</pattern>
        </encoder>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>logs/console.log.%d{yyyy-MM-dd-HH}</fileNamePattern>
        </rollingPolicy>
    </appender>

    <logger name="akka" level="INFO" />
    <logger name="org.knora" level="DEBUG"  />
    <logger name="net.sf.ehcache.config" level="ERROR" />
    <logger name="org.knora.rapier.util.authentication" level="DEBUG" />
    <logger name="org.knora.rapier.store.triplestore.embedded" level="DEBUG" />

    <root level="INFO">
        <appender-ref ref="stdout"/>
        <appender-ref ref="R"/>
    </root>
</configuration>
```

Fig. 4.1: Screenshot_2015-07-22_16.36.14

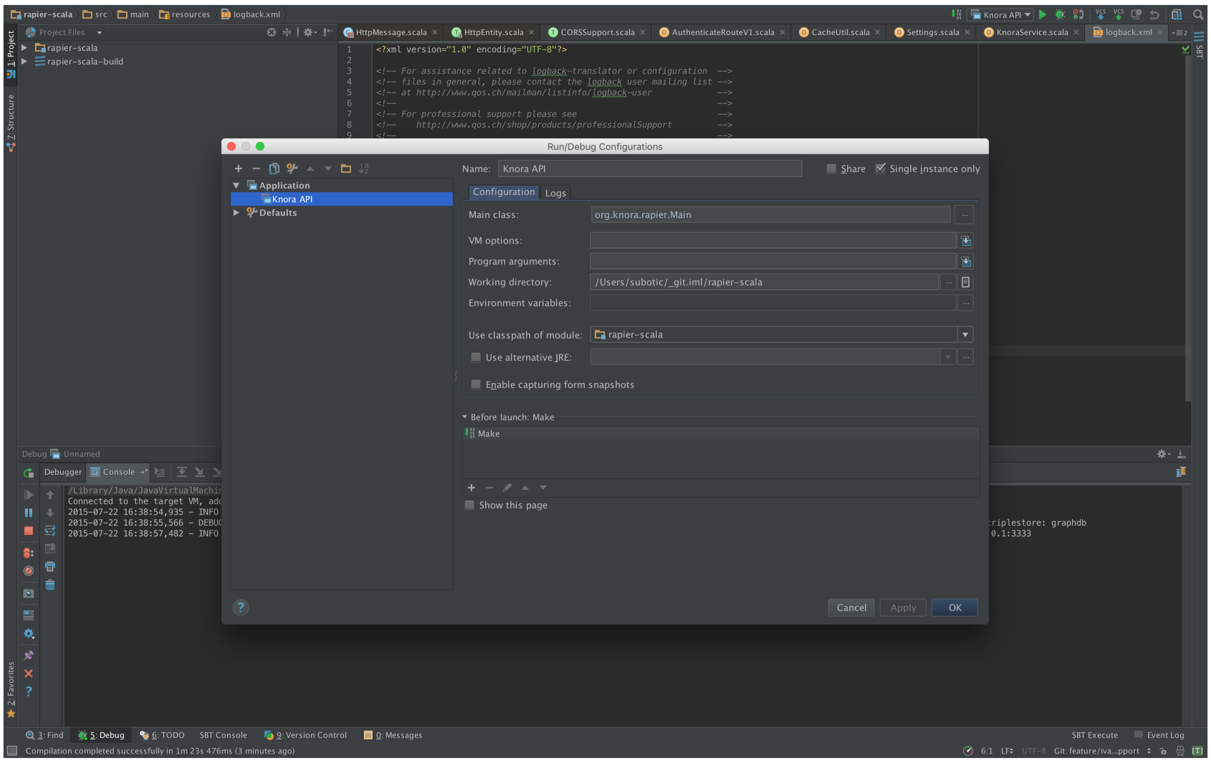


Press the + button to create a new configuration based on default settings

☑ Confirm rerun with process termination

Temporary configurations limit: 5

Fig. 4.2: Screenshot_2015-07-22_16.36.28



```
<?xml version="1.0" encoding="UTF-8"?>

<!-- For assistance related to logback-translator or configuration  -->
<!-- files in general, please contact the logback user mailing list -->
<!-- at http://www.qos.ch/mailman/listinfo/logback-user             -->
<!--                                                                -->
<!-- For professional support please see                            -->
<!--    http://www.qos.ch/shop/products/professionalSupport         -->
<!--                                                                -->
```

Name: Knora API   ☐ Share   ☑ Single instance only

Main class: org.knora.rapier.Main

VM options:

Program arguments:

Working directory: /Users/subotic/_git.iml/rapier-scala

Environment variables:

Use classpath of module: rapier-scala

☐ Use alternative JRE:

☐ Enable capturing form snapshots

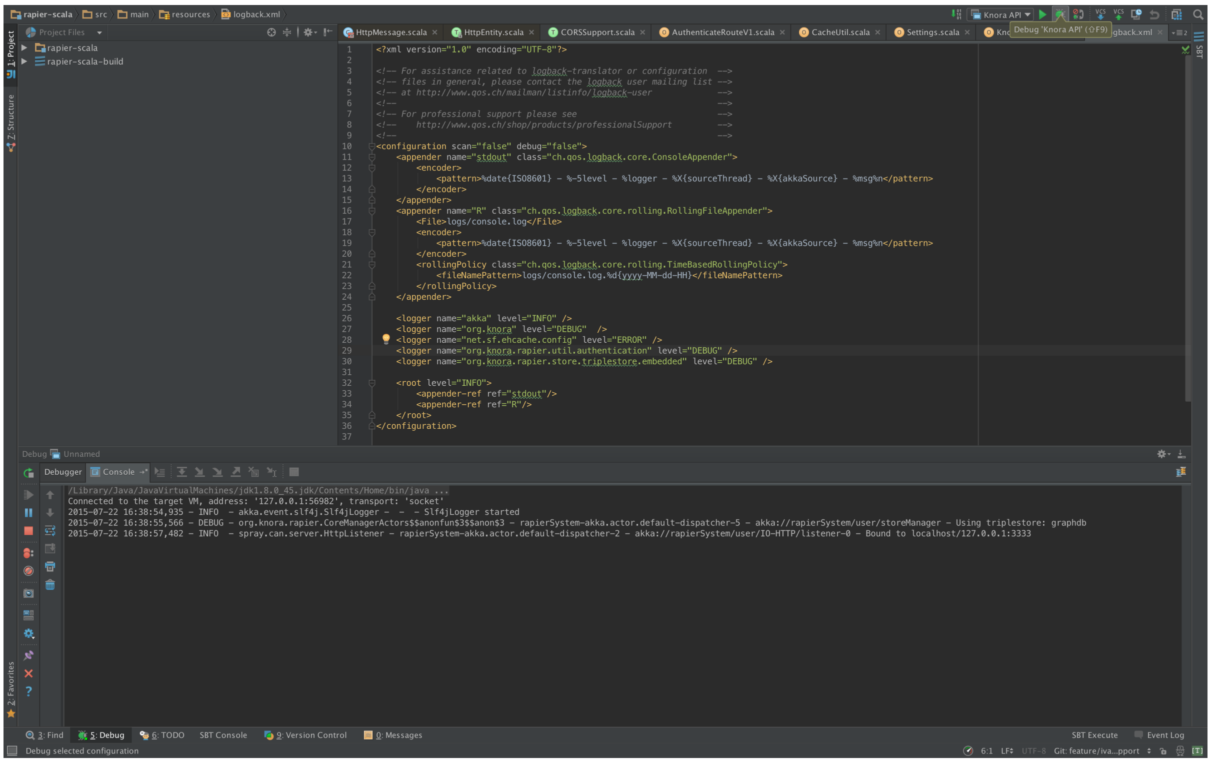
Before launch: Make

Make

☐ Show this page

Compilation completed successfully in 1m 23s 476ms (3 minutes ago)

Fig. 4.3: Screenshot_2015-07-22_16.42.35

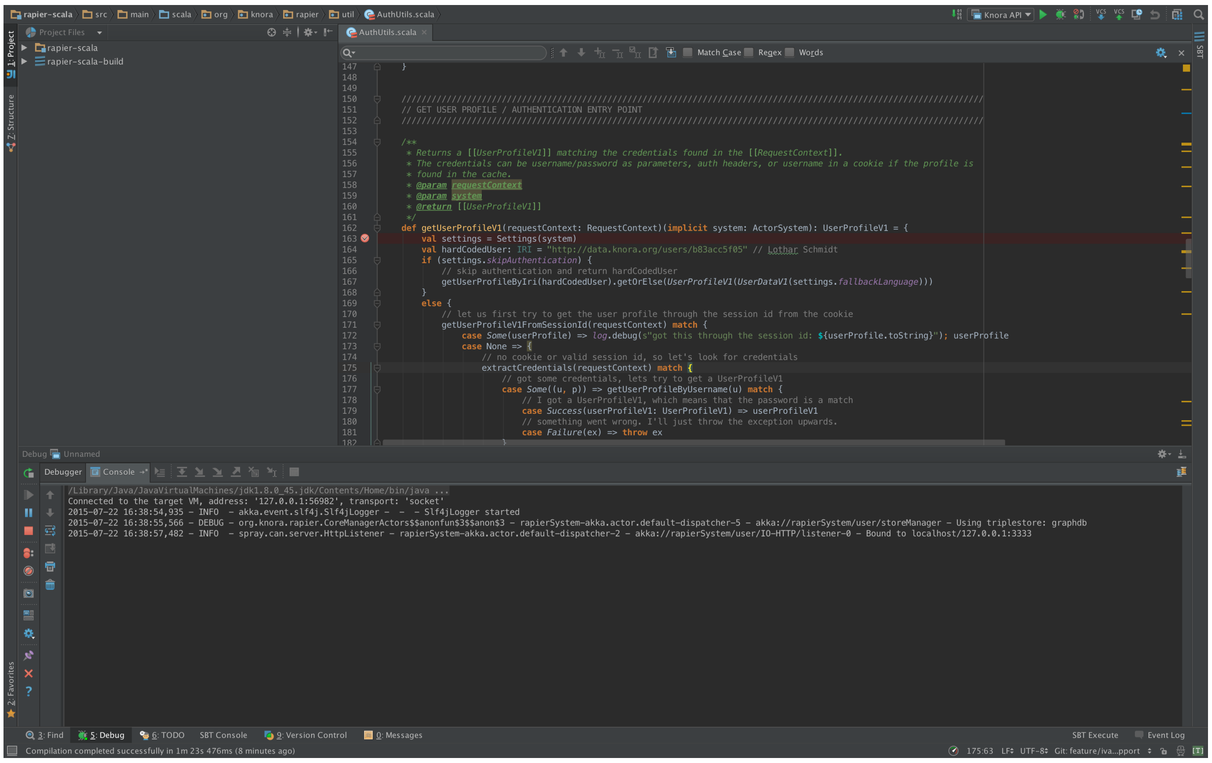


```
<?xml version="1.0" encoding="UTF-8"?>

<!-- For assistance related to logback-translator or configuration  -->
<!-- files in general, please contact the logback user mailing list -->
<!-- at http://www.qos.ch/mailman/listinfo/logback-user             -->
<!--                                                                -->
<!-- For professional support please see                            -->
<!--    http://www.qos.ch/shop/products/professionalSupport         -->
<!--                                                                -->
<configuration scan="false" debug="false">
    <appender name="stdout" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%date{ISO8601} - %-5level - %logger - %X{sourceThread} - %X{akkaSource} - %msg%n</pattern>
        </encoder>
    </appender>
    <appender name="R" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <File>logs/console.log</File>
        <encoder>
            <pattern>%date{ISO8601} - %-5level - %logger - %X{sourceThread} - %X{akkaSource} - %msg%n</pattern>
        </encoder>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>logs/console.log.%d{yyyy-MM-dd-HH}</fileNamePattern>
        </rollingPolicy>
    </appender>

    <logger name="akka" level="INFO" />
    <logger name="org.knora" level="DEBUG"  />
    <logger name="net.sf.ehcache.config" level="ERROR" />
    <logger name="org.knora.rapier.util.authentication" level="DEBUG" />
    <logger name="org.knora.rapier.store.triplestore.embedded" level="DEBUG" />

    <root level="INFO">
        <appender-ref ref="stdout"/>
        <appender-ref ref="R"/>
    </root>
</configuration>
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home/bin/java ...
Connected to the target VM, address: '127.0.0.1:56982', transport: 'socket'
2015-07-22 16:38:54,935 - INFO  - akka.event.slf4j.Slf4jLogger -  -  - Slf4jLogger started
2015-07-22 16:38:55,566 - DEBUG - org.knora.rapier.CoreManagerActors$$anonfun$3$$anon$3 - rapierSystem-akka.actor.default-dispatcher-5 - akka://rapierSystem/user/storeManager - Using triplestore: graphdb
2015-07-22 16:38:57,482 - INFO  - spray.can.server.HttpListener - rapierSystem-akka.actor.default-dispatcher-2 - akka://rapierSystem/user/IO-HTTP/listener-0 - Bound to localhost/127.0.0.1:3333
```

Debug selected configuration

Fig. 4.4: Screenshot_2015-07-22_16.42.46



```
    }


    ////////////////////////////////////////////////////////////////////////////////////////////////////////////
    // GET USER PROFILE / AUTHENTICATION ENTRY POINT
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////

    /**
     * Returns a [[UserProfileV1]] matching the credentials found in the [[RequestContext]].
     * The credentials can be username/password as parameters, auth headers, or username in a cookie if the profile is
     * found in the cache.
     * @param requestContext
     * @param system
     * @return [[UserProfileV1]]
     */
    def getUserProfileV1(requestContext: RequestContext)(implicit system: ActorSystem): UserProfileV1 = {
        val settings = Settings(system)
        val hardCodedUser: IRI = "http://data.knora.org/users/b83acc5f05" // Lothar Schmidt
        if (settings.skipAuthentication) {
            // skip authentication and return hardCodedUser
            getUserProfileByIri(hardCodedUser).getOrElse(UserProfileV1(UserDataV1(settings.fallbackLanguage)))
        }
        else {
            // let us first try to get the user profile through the session id from the cookie
            getUserProfileV1FromSessionId(requestContext) match {
                case Some(userProfile) => log.debug(s"got this through the session id: ${userProfile.toString}"); userProfile
                case None => {
                    // no cookie or valid session id, so let's look for credentials
                    extractCredentials(requestContext) match {
                        // got some credentials, lets try to get a UserProfileV1
                        case Some((u, p)) => getUserProfileByUsername(u) match {
                            // I got a UserProfileV1, which means that the password is a match
                            case Success(userProfileV1: UserProfileV1) => userProfileV1
                            // something went wrong. I'll just throw the exception upwards.
                            case Failure(ex) => throw ex
                        }
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home/bin/java ...
Connected to the target VM, address: '127.0.0.1:56982', transport: 'socket'
2015-07-22 16:38:54,935 - INFO  - akka.event.slf4j.Slf4jLogger -  -  - Slf4jLogger started
2015-07-22 16:38:55,566 - DEBUG - org.knora.rapier.CoreManagerActors$$anonfun$3$$anon$3 - rapierSystem-akka.actor.default-dispatcher-5 - akka://rapierSystem/user/storeManager - Using triplestore: graphdb
2015-07-22 16:38:57,482 - INFO  - spray.can.server.HttpListener - rapierSystem-akka.actor.default-dispatcher-2 - akka://rapierSystem/user/IO-HTTP/listener-0 - Bound to localhost/127.0.0.1:3333
```

Compilation completed successfully in 1m 23s 476ms (8 minutes ago)

Fig. 4.5: Screenshot_2015-07-22_16.47.04

4.3 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.

4.3.1 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
- − for subsections
- ^ for subsubsections
- ~ for subsubsubsections

4.3.2 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".
```

4.3.3 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
```

4.3.4 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
```

4.4 Test Tags

Todo

add example of how to tag a test.

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
```

4.5 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.

4.5.1 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/o
       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/o
       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"
     ...
}
```

4.5.2 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.

Part V

SALSAH

SALSAH - System for Annotation and Linkage of Sources in Arts and Humanities

CHAPTER

FIVE

DEVELOPING SALSAH

5.1 Build Process

TODO: complete this file.

- SBT

5.1.1 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 SALSAH root directory and start the server:

```
$ cd KNORA_PROJECT_DIRECTORY/salsah
$ sbt
> compile
> re-start
```

To shut down the SALSAH server:

```
> re-stop
```

5.1.2 Run the automated tests

Make sure you've started Fuseki and the API server as shown above. In order to run the tests, the Selenium driver for Chrome has to be installed. Please download it from here and save it as `salsah/lib/chromedriver`. Also, please make sure to start the API server with the `allowResetTriplestoreContentOperationOverHTTP` flag. For more information about this flag, see *Webapi Server Startup-Flags*

Then at the SBT prompt:

```
> test
```

5.1.3 SBT Build Configuration

```
import sbt._
import sbt.Keys._
import spray.revolver.RevolverPlugin._
import com.typesafe.sbt.SbtNativePackager.autoImport._

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(Revolver.settings: _*)

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"
)
```

Part VI

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.

CHAPTER SIX

SETUP SIPI FOR KNORA

6.1 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: `build/sipi -config sipi.knora-config.lua`. 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.

6.2 Test Sipi

If you just want to test Sipi with Knora without serving the actual files, you can simply start Sipi like this: `build/sipi -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.

CHAPTER

SEVEN

INTERACTION BETWEEN SIPI AND KNORA

7.1 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.

7.2 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*.

7.2.1 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`:

[1] Of course, also the GUI uses the API. But the user does not need to know about it.

- `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.

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.

7.2.2 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.

7.2.3 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.

7.2.4 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`.

7.3 Retrieving Files from Sipi

7.3.1 URL creation

Binary represention 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.j
          "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.

7.3.2 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.

Part VII

Indices and tables

CHAPTER EIGHT

OTHER INDICES

- search