(DLH) The Journal for Digital Legal History Open Access

Title:	Building and Deploying a Classification Schema using Open Standards and Technology
Author(s) and contributors:	C.A. Romein A.W. Wagner J.J. van Zundert
Issue:	"Gute Policey" and police ordinances: local regimes and digital methods
Year:	2023
DOI:	10.21825/dlh.85751
Keywords:	Legal History, Taxonomy, Tools, RHONDA, Tutorial, git, SKOS, GitHub pages, GitHub actions, Taxonomy, Tools, RHONDA, Tutorial, Git, SKOS, Reconciliation, GitHub pages, GitHub actions
Abstract:	This tutorial fully introduces Building and Deploying a Classification schema using Open Standards and Technology. Beyond accessing the command line, no prior knowledge is assumed as all the steps are described in detail. However, the tutorial will take you through some more complex technical steps. To profit best from the example workflows of the last chapter, it is good to know how to work with OpenRefine and/or have a TEI Publisher server set up to play around with. With this tutorial, you - the reader - should gain an understanding of: - what different classification schemes can contribute to your project, - what to pay attention to when building a classification scheme, - how to code the classification scheme (in SKOS), - how to publish the classification scheme (on GitHub pages), - what some possible scenarios of using classifications schemes in actual project workflows are.

Building and Deploying a Classification Scheme using Open Standards and Free Platforms

$C.A.Romein^{1}$; $A.Wagner^{2}$; $J.J.vanZundert^{1}$

[1] Huygens Instituut voor Nederlandse Geschiedenis en Cultuur, Amsterdam.
[2] Max Planck Institute for Legal History and Legal Theory, Frankfurt am Main.

All three authors contributed equally to this tutorial and are to be considered first authors, the order of the names is merely alphabetical and has no meaning other than this.

Keywords: *controlled vocabulary, classification, taxonomy, legal history, police ordinances, subject matter of legal regulations, tools, RHONDA, tutorial, linked open data, SKOS, reconciliation, data cleaning, annotation, persistent identifiers, git, GitHub pages, GitHub actions, w3id.org, skohub-vocabs, skohub-reconcile*

Contents

Introduction and Scope

This tutorial offers a full introduction to building and deploying a classification scheme using open standards and freely available internet infrastructure. “Full introduction” here means that it will cover all the steps from beginning to end, from conceptualising the classification scheme to publishing and intergating it in scholarly workflows. With this tutorial, you – the reader – should gain an understanding of:

- what different classification schemes can contribute to your project,
- what to pay attention to when building a classification scheme,

- how to encode the classification scheme (in the SKOS data model and in its turtle serialization),
- how to publish the classification scheme (on GitHub Pages taking advantage of GitHub Actions), and
- what some possible scenarios of using classifications schemes in actual project workflows (data reconciliation on OpenRefine and text annotation in TEI Publisher) are.

In providing a discussion of all the steps, this tutorial necessarily touches on on some very technical, as well as on some theoretical aspects. We acknowledge that this risks giving an incoherent character of the piece as a whole, or boring the reader with some of the material, but we think it is worthwhile bringing together all the various aspects for once.

Also we will assume we are talking of taxonomy development and usage as a collective and collaborative enterprise. The whole point of a taxonomy is to facilitate collaboration and interoperability and usually there is a scholarly community the resource is meant to serve. So while there may have been more simple and streamlined solutions, the mechanisms we will discuss are designed to accommodate collaborative scenarios.

The tutorial is split into five chapters, plus further reading.

Pre-requisites and requirements

A very basic understanding of using the command line will be needed. Other than that, no prior knowledge is assumed as all the steps are explained in detail. In particular, as chapters IV and V massively rely on `git`, we have included a detailed introduction to this software and the corresponding workflows in chapter III. If you are already familiar with this tool (and terms like “push”, “staging area”, “fork”, “pull request” do not even raise an eyebrow), you are invited to skip chapter III. We assume you have installed the git software and a steady internet connection that can support downloading software and uploading data. We have linked to resources in case you would need to acquire these basics.

This tutorial was written for and tested against:

- git 2.35.1
- skohub/skohub-vocabs-docker:latest Docker image from November 2021 (Digest 344c314bab2e)

When discussing particular software or platforms beyond these basics, we try to mention alternatives that you could use to achieve the same things.

To properly work with the deployment workflow we outline in chapter III, you will need an account on GitHub (or a comparable platform).

The example workflows of the last chapter demonstrate how to integrate a scheme's deployment with OpenRefine or TEI Publisher workflows, so you will profit best from this chapter if you are somewhat familiar with the purposes and typical workflows of these tools. However, the chapter has a rather illustrative function and knowledge of the tools is not necessary to get an idea of what is being demonstrated. If you do want to perform all the instructions of the last chapter yourself, you will need to install a copy of OpenRefine and have a TEI Publisher server set up to play around with.

How difficult is this tutorial?

The steps in this tutorial are unambiguous and there are very few choices you need to make as you work through this tutorial. However, it will take you through a few somewhat complex technical steps. You will learn some new terms and gain familiarity with the GitHub web interface.

The tutorial should take a few hours to complete. You may find it useful to repeat sections or chapters in order to strengthen your understanding. Technical terms have been linked to informative websites or blogs.

We would like to encourage you to think about a dataset of yours and a set of terms you would want to use in order to classify some of it. It will be much more effective – and more fun – for you to do all the steps with data and terms that you can relate to yourself.

In the further reading and resources section you may find additional information to help you with potential issues. For more background to some of the steps, it may be helpful to consult the Programming Historian lessons on OpenRefine, on GitHub pages, on Linked Data, or on the pull requests workflow at GitHub.

Scholarly case study: Police Ordinances

The material of this tutorial centers on Policeyordnungen der Frühen Neuzeit, an online repository of late medieval and early modern normative texts ('police ordinances'). This repository consists of data based on 12 printed volumes of the 'Repertorium der Policeyordnungen der Frühen Neuzeit' which were edited by Karl Härter and Michael Stolleis at the Max-Planck-Institute for Legal History and Legal Theory in Frankfurt am Main (Germany). It contains data of over 200,000 medieval and early modern normative texts from 68 territories and imperial cities, as well as from other indexing projects, from which there was no complete data-overview possible (due to archival damages etc.). It should be noted that these territories and imperial cities were not limited to the Holy Roman Empire ('Germany'), but Denmark, Sweden, and some towns of the Swiss Confederation have also been included.

Each of the contributors to the book series had set out to describe ordinances with metadata including legislators, dates, places, jurisdictions, types of ordinances, but also subject matters that the or-

dinances dealt with. In 2021, the online version of the Policeyordnungen der Frühen Neuzeit was launched with data from the German-speaking areas. This will be expanded with the Danish and Swedish sources in the coming months.

Of particular relevance for this tutorial, the subject matters of ordinances were classified based on an extensive catalog of descriptors that had been developed during the initial project in the 1990s and that has since been re-used in, or has inspired analogous catalogs in several other, independent projects. Given the European scale of 'police' legislation and the considerable interest in comparing the data of various projects, but also given the presence of various languages and the hierarchical structure of the descriptors in the original catalog, Annemieke Romein and Andreas Wagner jointly started working on a formally encoded, controlled vocabulary/taxonomy of subject matters from 2019 onwards. (See for an example within the project *Entangled Histories* here). This taxonomy will serve as an example throughout the tutorial; it can be accessed at https://w3id.org/rhonda/polmat/scheme (and comments are very welcome).

Our experience with setting up a controlled vocabulary helps explain the scholarly motivations behind technical choices in this tutorial, but the classification scheme you will develop over the course of this tutorial can be considerably smaller and simpler. You will finish with a site where visitors can browse and search your classification scheme, and with a permanent URL mechanism that allows records from independent projects to refer to concepts of your classification scheme. You will also have seen how such a resource can be integrated into different workflows of your own or other, independent projects.

I. Classification Schemes and (as) Interoperable Data

Interoperability and controlled vocabularies

In information sciences, classification schemes are used to organise information: You have a list of unequivocal terms and a rule prescribing, at least in a certain application context, to refer to phenomena of your domain of interest exclusively using the terms from your list (i.e. you have a controlled vocabulary), and you apply terms of your vocabulary to multiple pieces of your data and thus facilitate data profiling, and information retrieval. The idea of retrieving bits of information from different places of the data set because they are associated to the same descriptor can be extended to querying even multiple datasets/databases, provided they are using the same classification scheme: Understanding the logic of the vocabulary, the user of a cross-domain search engine would, ideally, not need to know the underlying data and what justified the application of descriptors in each individual data set, in order to collect a set of relevant data and then find out details in a second, close reading step.

For instance, imagine searching library holdings using the Dewey Decimal Classification (DDC) or the Library of Congress Subject Headings (LoCSH): you do not even have to learn about cataloging and indexing habits of your local library; instead, if you are sufficiently versed in the respective classification scheme, you can use the library's discovery system right away and find the literature that is of interest to you. What specific relation to the searched-for subject heading some particular book may have can only be understood by actually reading the book, but finding it was an important first step. Interoperability means that multiple systems, or parts of these systems, can exchange information and immediately use the exchanged information, and classification schemes and other types of controlled vocabulary are a central pillar of interoperability efforts (Zeng and Chan 2004). In terms of our example, imagine a single discovery service for multiple libraries like OCLC's worldcat, which presupposes that participating libraries are using the same classification scheme, and also that this information can be queried externally in an identical way.

As another example, consider annotation scenarios: While document retrieval relies mostly on classification of complete documents, you could also classify just portions of documents, like areas of images or spans of a text. Named Entity Recognition (NER), for instance, is a very common method of Natural Language Processing (NLP) that seeks to locate and classify mentions of persons, organisations, places, events, quantities etc. (without necessarily identifying *which* person is being mentioned). Suppose you want to compile training data for some task, and you manage to find a couple of datasets relevant for your domain, it is helpful if they in fact express the presence and location of the same entities in the same way, that is, if they use interoperable annotations. Similarly, large institutions concerned with collecting cultural data from various sources like Europeana often are striving to establish and promote with their data providers interoperability mechanisms at some level.

(Types of) controlled vocabularies

The most prominent structures of controlled vocabularies are, listed from concrete to abstract, or from flat to complex (see for more details e.g this blogpost):

- A ***glossary***, such as a keyword- or index-list is an unstructured (except for geographic locations or alphabetical order) list of words. In the analogue version of the 'Repertorium der Policeyordnungen', an equivalent can be found in the alphabetic index/indices of the volumes.
- A ***taxonomy*** refers explicitly to the classification of things and embeds some hierarchical relationships between its concepts, like "X is a subclass of Y". Ideally, it provides descriptors, definitions and examples for its concepts. (For completeness's sake, it should be mentioned that there are taxononies the main relations of which are not generic relations, i.e. class/subclass, but partitive relations, i.e. part/whole, or other types of relations or even mixtures of several types; and there are taxonomies where a term can have multiple parent/broader terms ("polyhierarchies"), e.g. "cats" having both "mammals" and "four-legged animals" as parent cate-

gories. But such increased flexibility comes at the price of reduced toolkits capable of processing such taxonomies (cf. this blog entry). In this tutorial, we concentrate on monohierarchical, generic taxonomies. For more details and recommendations, see SEMIC 2009.)

- A ***thesaurus*** is an extension to a taxonomy: It adds other relations to the concepts beyond super-/subclasses (e.g. opposites or other forms of 'related to' relations).
- An ***ontology*** is a 'formal, explicit specification of a shared conceptionalization' (Guarino et al. 2009) of a domain of the world. It describes what type of entities can exist in this domain and what relationships they may have. (On the distinction between ontologies and thesauri, cf. Kless et al. 2015.)

As tools of computational knowledge organization, all these vocabularies are encoded in some formal, machine-readable language. Moreover, all the entries in the above list could express or be used as classification schemes. but they differ in how much of your knowledge they allow to incorporate. In other words, you can use any of them to apply descriptors of your controlled vocabulary to the "things" that you want to classify, and if you then encounter an instance of such a classification, you already know that the classified entity also satisfies a more or less elaborate set of conditions: if you have been using a taxomony, you may may be able to infer that the entity belongs to a super-class and super-super-class, if you have been using a thesaurus, you may be able to infer that it is typically a part (or the opposite) of another entity, and if you have been using an ontology, then you may be able to infer that it must have certain properties and be related to other entities in specific ways.

Taxonomy classification schemes

Here and in the following, we will be using a monohierarchical taxonomy based on generic subclass-/superclass-relations. In the *Repertorium der Policeyordnungen* the subject matters of ordinances (German: Policeymaterien) show the hierarchical structure as indicated:

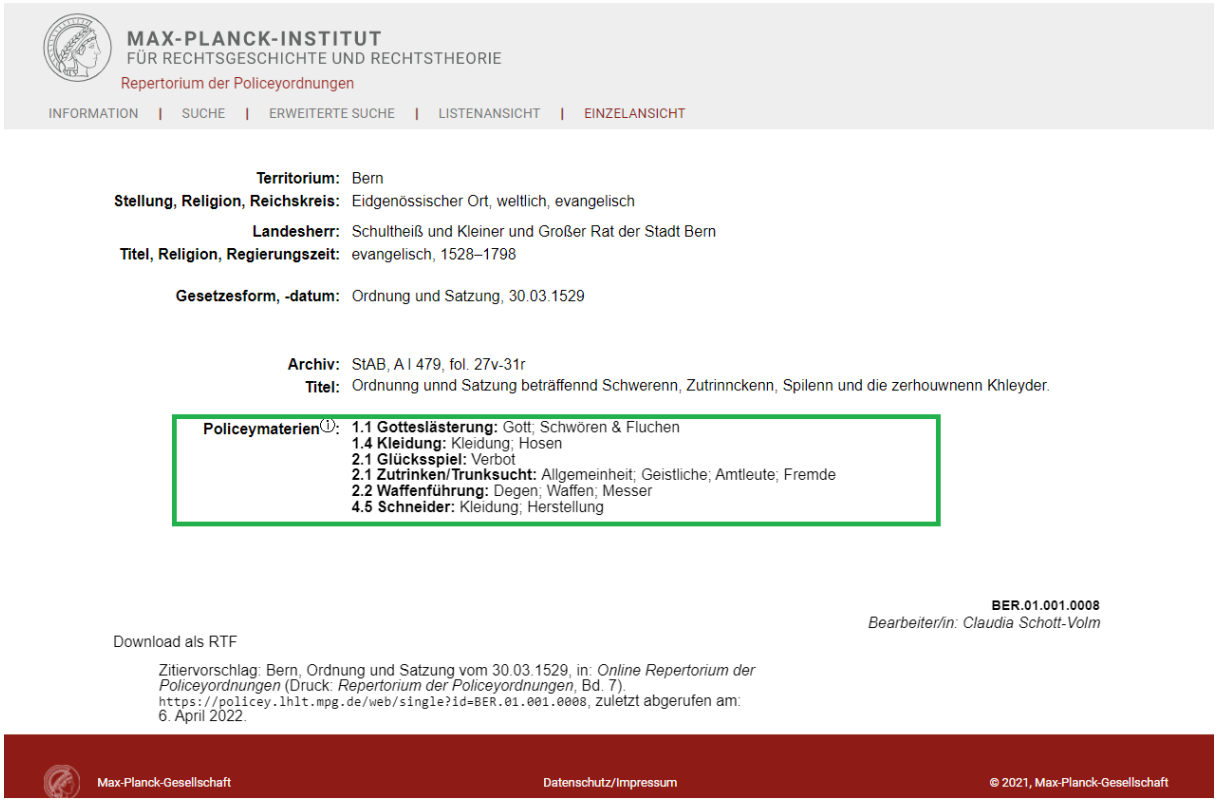


Image 1. Indication of the hierarchical order of the subject matters, as found on the website of the Repertorium der Policeyordnungen.
Source: https://policey.lhlt.mpg.de/web/

The numbers indicate the group and subgroup ("parent" classes), followed by a third specification and then, after the colon, individual tags. Thus, the classification of the matters as metadata to the original police ordinances is clearly visible to the user.

Authority data (providers)

Authority files can also function as way to standardise data and avoid double work when referring to the same 'thing'. The main distinction between controlled vocabularies and authority files is that the former provide information about types, i.e. general concepts ('universals'), whereas the latter provide information about *individuals* ('particulars') of a certain kind (and they usually, but not necessarily, express this information using a controlled vocabulary the definition of which is not their main business). For instance, contrast the DDC and LoCSH that have been mentioned with orcid, geonames, viaf or wikidata records. In fact, even book isbn numbers are an authority data mechanism. What makes an authority file is first of all a convention, in which a community of practice agrees to

use one single dataset of identifiers as point of reference for the task of expressing a specific information. Moreover, the dataset has to be accessible in ways that are convenient for the most frequent use cases - in practice, that means it has to be a webservice that allows for searching, getting contextual information and that maybe offers API access for software developers. In other words, authority data is a community convention and an information infrastructure. Which are what we are building around our classification scheme in this tutorial.

So why would you want to use a classification scheme?

In the case of our police ordinances, classes of subject matters are grouped into super- and subclasses, so we are talking of a taxonomy as an explicitly hierarchical classification scheme, a part of which could be visualised as follows:

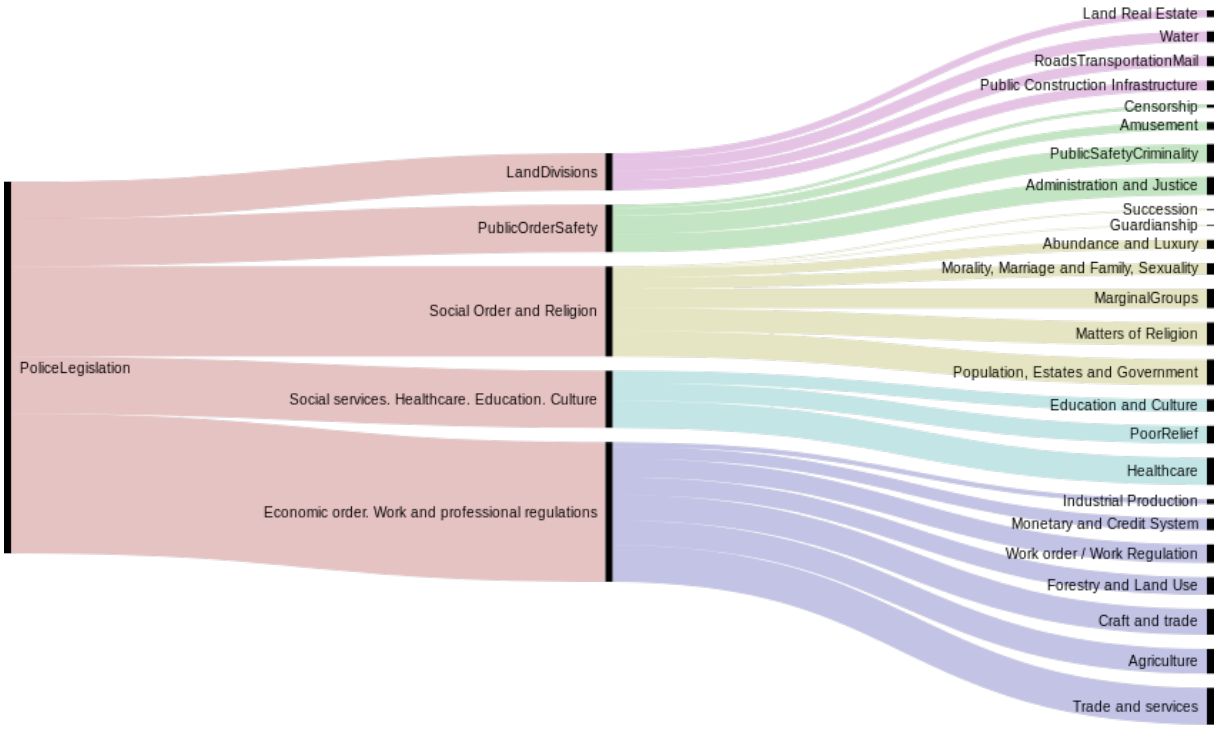


Image 2. Police ordinances, first levels of the hierarchy.
Source: C.A. Romein, S. Veldhoen, M. de Gruyter, 'The Datafication of Early Modern Ordinances', *DH Benelux Journal #2* (https://journal.dhbenelux.org/journal/issues/002/article-23-romein/Images/figure6.png)

This alluvial diagram is showing the five main topics of police ordinances in pink, subsequently split

out into 25 sublevels. Not shown in this visualisation, this is split even further into two more specific levels totalling up to 1,800 subcategories.

The high level of detail of our subject matters scheme does indeed serve a purpose: it allows researchers to aggregate or narrow down their search results flexibly and to a great extent: For example, searching for ordinances regulating land divisions, besides the entries that were tagged with "Land Divisions" directly, you could make the search engine also return *additional results* for all four of its subclasses (and their subclasses etc.). Or vice versa, searching for "Land Real Estate", one of "Land Divisions"'s subclasses, you could get results tagged with "Land Real Estate" or one of its subclasses, but you could also get *suggestions* for results in the other "Land Divisions" subclasses, or information like *how large a share* of "Land Divisions" these "Land Real Estate" results are. Finally, you could compare data from different contexts even if the specific tags, say "Jews" and "Indigenous Peoples", are different, based on a more general superclass like "Marginalized Groups"…

Also, by developing a classification scheme, a research project or researcher may hope to discipline themselves or other users in the use of synonyms and different grammatical forms of the terms. The idea is that it functions as a restricted vocabulary and you avoid one annotator marking something as "Craft and Trade" and the next one marking something comparable as "Crafts and Trades" (plural forms), lest you miss half of the entries when you search for either of the terms.

In a monohierarchical taxonomy such as this, there must not be an overlap between the classes. It does require a good understanding of the criteria of the classes and an explication of the relationship between the concepts. E.g. in our classification we have "Social Order and Religion" and a subclass are "Marginal Groups". These groups form a specific group within (or rather: outside) the society and as such the relation to 'Social Order' is specified. (Note, however, that a single ordinance or piece of data may be classified as related to several concepts. In this sense, there may be overlap, just not in the concepts and classes themselves.)

Surely, such classification schemes are always to a certain extent artificial, they force the material into preconceived "drawers" or "boxes", and they always have their blind spots and pitfalls. Thus, we want to highlight some principles and key takeaways to consider in building your own scheme.

II. Building Your Own Classification Scheme

Here are two pieces of advice that we recommend you keep in mind in all that follows after:

1. Keep your own research question in mind and make sure the scheme corresponds to it
2. Consider the relevant usage scenarios: is the scheme meant to support aggregating and quantitative analyses, or is it about search suggestions to lay people coming across your website.

Different scenarios call for different types of schemes, and maybe a taxonomy is not even the best choice

Obviously, there is a tension between, on the one hand, the scheme being a "living" thing that you keep developing and expanding, and, on the other hand, the need to have a settled vocabulary to refer to unambiguously. Thus we (as authors) applaud the possibility of adding information to classification schemes (progressive understanding one might call it), however, we also consider it good practice to leave the original structure intact and make changes only in new versions or releases.

Steps and considerations (a)

In general terms, the process of building a classification scheme can be summarized as *collecting terms*, *normalizing and consolidating terms* and *balancing the scheme*. (For a somewhat different presentation of the steps, recommendations that overlap a lot with the following, and a lot more literature, see also SEMIC 2009, in particular chapter 4.)

Where to start with finding concepts to build a classification scheme for historical material:

- Look within historical books and their indices
- Check historical classifications (e.g. archival schemes) to organise and classify information
- Search within research literature for 'standard' vocabulary within your field to name the concepts
- closely read a sample of your data to form an intuitive list of relevant concepts

Autiero et al. 2023 had recourse to a terminological repertory from 2007 (and to other sources like museum catalogues, repertories and databases), and the Policeyordnungen project collected terms from indices of 19th-century compilations of ordinances and from archival finding aids. Nijman and Pepping 2023 indicate that they plan to turn (among other sources) to a glossary from 2000. They also point out that it is important to be able to take a critical distance towards the language (they are planning a SKOS vocabulary for records of the Dutch East India Company and are confronted with biased and derogatory colonial language in the historical sources). Ernst et al. 2023 even describe and compare three different approaches for collecting terms ("deductive", "pragmatic-explorative", and "inductive-computational").

When you start listing the terms, consider applying some normalisation and critical review of terminology:

- Lemmatisation (singular/plural, orthography, etc.)
- The composition or decomposition of words
- Often choosing modern terminology helps avoiding challenges with spelling-changes over time; but depending on your field you may need to defend this choice

- On the other hand, historic terminology often carries biases and misrepresentations that need to be addressed

Besides the more or less trivial orthographic and grammatical normalization, some more delicate questions are likely to surface: Nijman and Pepping 2023 point out that it is important to be able to take a critical distance towards the language. In their case, a SKOS vocabulary for records of the Dutch East India Company is being developed and they are confronted with derogatory and discriminatory colonial language in the historical sources. Ernst et al. 2023 are concerned with court records from the Nazi regime and discuss these problems in more depth.

Obviously, there will be a hermeneutic step of combining the categories in groups of various levels. In terms of the structure of your scheme, after building sub- and supergroups, you will need to balance your scheme and see that concepts at the same level of the hierarchy have a roughly analogous level of abstraction. E.g. Having 'economic affairs' on the same level as 'pavement' is not balanced as these terms are operating on different levels of abstraction. Adding 'infrastructure' as a super-class containing 'pavement' as a subcategory could solve the problem, as it balances them out, even though it may be introduced somewhat artificially. Be aware of super-concepts that could in effect be *collections* of multiple concepts rather than *concepts* in themselves: those you would need to break down into multiple categories or replace them with a concept that actually does reflect their common denominator to ensure clarity of your structure.

One other consideration that has proven to us to be valuable is keeping track of *what kind of objects* you are aiming to classify: Ideally all the terms in a scheme should classify the same "ontological" kind of thing: if you want to classify sectors of economic activity on the one hand, and physical objects on the other, perhaps you should simply create two separate schemes. This is not strictly necessary, however, as you might also think of a distinction between "processes" (with a sub-class "social practices", of which "economic activity" in turn could be a sub-class with different sectors) and "physical objects" as being a distinction of sub-classes of a common higher-level class of "things" as such. Then the scheme as a whole would again classify objects of the same kind, namely "things". But anyway, it is good to always be aware of what kind of object you are currently thinking of…

At this stage, the software that you are using to collect and arrange the classes of your classification scheme does not really matter. You can use a word processor (taking advantage of different levels of headings and perhaps automatic numbering to reflect different levels of the classes hierarchy), or you may use a mindmapping or other diagramming software, or you can even use a spreadsheet software with one concept per row and columns for concept identifier, preferred and alternative labels, optionally "parent" conept, definition etc. Maybe you want to be able to collapse/expand parts of the hierarchy, but many word processors and mindmapping tools are capable of doing so, and spreadsheet tools usually can filter rows as well.

Steps and considerations (b)

If your taxonomy aims at a certain scholarly community interested in the domain, and if you envision involving other members of the community or other stakeholders into the process of developing it, there are also issues of a different kind to consider:

- Are you going to collaborate, and with whom? Are these people knowledgeable enough to contribute and correct, or are they mere users? – How can you involve them in defining the classification scheme?
- How often and how far into the project can and will you revise your classification scheme?
- Can you document when and why choices were made, so that others can replicate your train of thought and can follow your classification scheme within a (slightly) different context?

Goulis 2021 and Edmond et al. 2023 discuss in detail the various facets and challenges of scenarios in which teams or communities aim to develop controlled vocabularies across disciplinary and institutional boundaries. While we, the authors, agree about the importance of *organizational* and *social* arrangements, we would like to add that methods and workflows also stand and fall with the *tooling* available and with the way the tools accommodate both users and their workflows. Thus, in the later chapters, we will discuss a software tool and a platform which greatly facilitate transparent collaborative editing of resources such as a taxonomy. (See Durost et al. 2021 and Almeida et al. 2021b who begin to cover this aspect.)

What is *SKOS* and how to encode your scheme?

In the previous chapter, you have encountered various examples of controlled vocabularies. Their type tells you something about what type of data they contain and whether its concrete/abstract or flat/complex. By now you should develop a first idea for a mental model or "conceptual scheme" of your classification scheme. You should prepare a diagram or document where all the concepts and their hierarchical relations are being collected. In order to express such structures in a way that can be parsed more easily by machines, we recommend and will use the *Simple Knowledge Organisation System (SKOS)*. It is a W3C standard recommendation, to express stuctures and contents of concept schemes.

As the W3C SKOS Primer explains, the purpose of this standard is just what this tutorial is about: deploying a classification scheme to facilitate its reuse.

> The aim of SKOS is not to replace original conceptual vocabularies in their initial context of use, but to allow them to be ported to a shared space, based on a simplified model, enabling wider re-use and better interoperability. *(Introduction)*

Beyond identifiers and labels, SKOS supports documentary information, hierarchical relations, a rather unspecific 'related-to'-relation and mapping relations (for connecting to other schemes), making it well suited to accommodate term lists and taxonomies. For thesauri and ontologies, more powerful standards are better suited, but since we are dealing with a taxonomy, SKOS is the perfect choice for us: The labels and documentary notes allow for synonyms and translations in various languages, examples (to provide a context), and definitions (explaining the terms). The hierarchical relations are expressed as concepts relating to *broader* (part of a higher level-category) and *narrower* (specification of the term) concepts.

Aiming to open up knowledge organisation schemes for wide re-use in the semantic web, SKOS itself is based on the Resource Description Framework (RDF) as the data model for any and all information on the semantic web. The RDF model expresses all information in the form of so-called "triples" of *subject*, *predicate* and *object*, with Unique/Uniform Resource Identifiers (URIs) being used as globally unambiguous identifiers for each of the three elements:

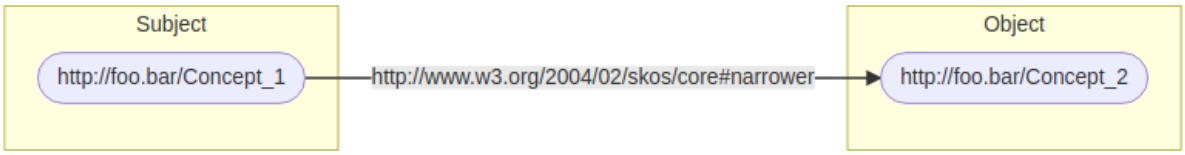


Image 3. Visualisation of a triple.

More on RDF and the Turtle (ttl)-syntax that we're using for serializing it can be found in the lesson of the Programming Historian dedicated to Linked Data.

Your concept is identified by an URI and you can link a number of labels in different languages (modern, dialect, slang or other language) to it, designating one label per language as the 'preferred' one (`prefLabel`).

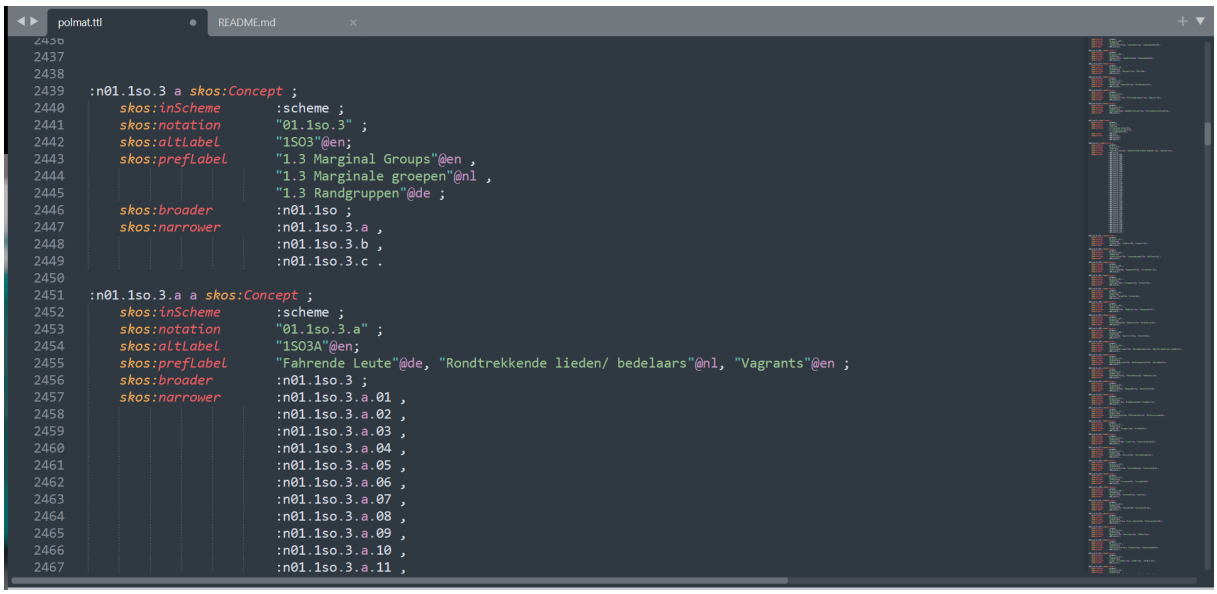


Image 4. How does a .ttl-file look like?

Source: https://w3id.org/rhonda/polmat/scheme.json

In detail:

```
@prefix : <https://w3id.org/rhonda/polmat/> .          // In the following, an identifier beginning with ':' should be expanded to 'https://w3id.org/rhonda/polmat/' etc.
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .  // In the following, an identifier beginning with 'skos:' should be expanded to 'http://www.w3.org/2004/02/skos/core#' etc.

:n01.1so.3 a skos:Concept ;                             // Here comes information about the concept "https://w3id.org/rhonda/polmat/n01.1so.3"
    skos:inScheme       :scheme ;                       // it is part of the concept scheme "https://w3id.org/rhonda/polmat/scheme"
    skos:prefLabel      "1.3 Marginal Groups"@en ,      // it has preferred labels in english, dutch and german languages...
                        "1.3 Marginale groepen"@nl ,
                        "1.3 Randgruppen"@de ;
    skos:broader        :n01.1so ;                      // this is its "parent" concept
    skos:narrower       :n01.1so.3.a ,                  // these are its three "child" concepts
                        :n01.1so.3.b ,
                        :n01.1so.3.c .
```

Image 5. Detail of a .ttl-file.

Source: https://w3id.org/rhonda/polmat/scheme.json

If you have a large classification scheme, manually creating such a SKOS file can be a tedious activity. Depending on how you have created your conceptual scheme in the preceding steps, there may be ways of transforming it with search and replace expressions (or a tool such as sparna's excel-to-skos converter). However, it is good to understand the elements, to manipulate or correct the information.

The most important ones are:

- `skos:broader` refers to the URI of the more abstract concept.
- `skos:narrower` is/are the more concrete concept(s).
- `skos:prefLabel` provides the words that you prefer to use to talk about the concept; the additional `@en` (English), `@nl` (Nederlands) or `@de` (Deutsch) are indications in which language the words are expressed.
- you might want to add `skos:scopeNote` with some explanation or `skos:definition` or `skos:example` in order to give a clearer picture of the concept; these fields can be given in several languages again.

For more information about other predicates (and entity types), see again the SKOS Primer. The sparna SKOS playground offers ways of testing and visualizing your SKOS file.

III. Introduction to Basic git Commands and GitHub Features

Having an overview of the files you are working on is crucial when working with multiple people on a project or when you need to return to an old project years later. Tracking changes made by contributors and preventing conflicting changes is especially useful for actively used documents with collaborative revisions and rewritings. Finally, when other projects want to reuse information it can be helpful to have some record of previous choices and developments of a project.

In this chapter, you will learn to use git and GitHub as tools for *version control*. If you are already familiar with these (and terms like "push", "staging area", "fork", "pull request" do not even raise an eyebrow), you are invited to skip this chapter and read on in chapter IV.

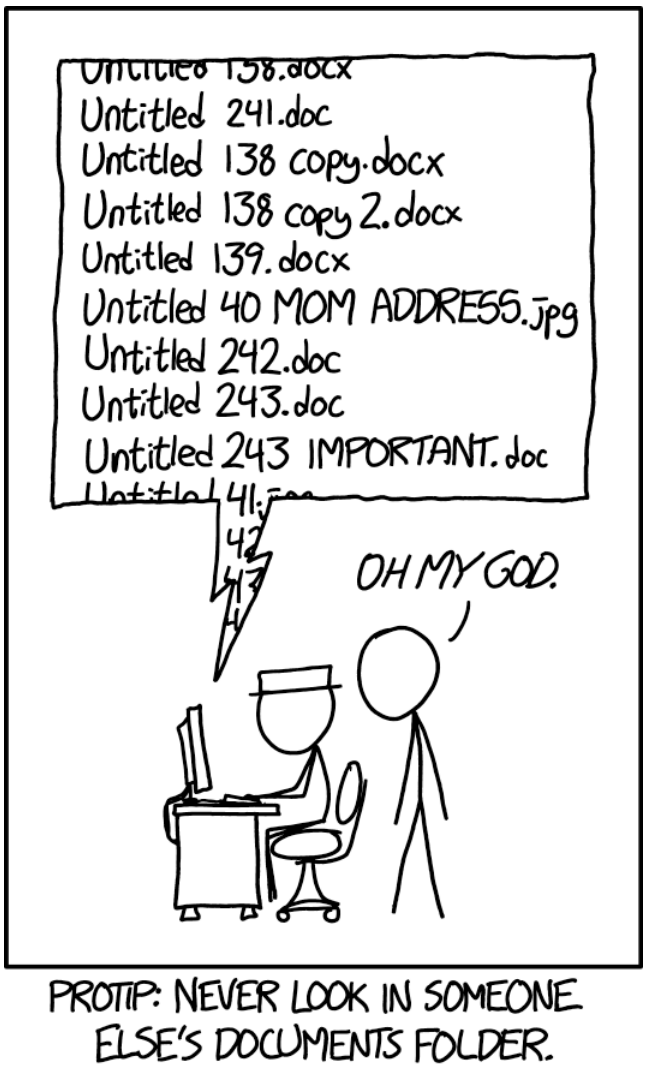


Image 6. Versionmanagement Old Style?
Source: https://imgs.xkcd.com/comics/documents_2x.png

Git is software originally created by 'Mr. Linux' Linus Torvalds in 2005. It was originally intended to managed the collaborative development of the operating system Linux. GitHub is a webservice that runs git and offers to host git repositories. It was acquired by Microsoft in 2018 and is by far the most popular free public repository for version-tracked files such as software code. Both git and GitHub are free to use.

To keep track of changes in files in a folder on your computer, you can use git. It acts as a *local repository* (archive). GitHub on the other hand, is a remote webservice that acts as a *remote repository*. You can think of the remote repository as a shared central archive, your local repository is your personal copy of it. This also means that you should consider the remote repository as having the "authori-

tative" version. This is because nobody knows anything about anything on your local computer, all your local stuff is private. Local changes are unknown and invisible to your collaborators until they are shared through the remote repository.

In this chapter you will first learn how to create the remote repository on GitHub, then you will learn how to clone (copy) this to your local machine. Then we will show you how to update the remote repository with changes you made locally, so you can share your edits with others.

Setting up a GitHub repository

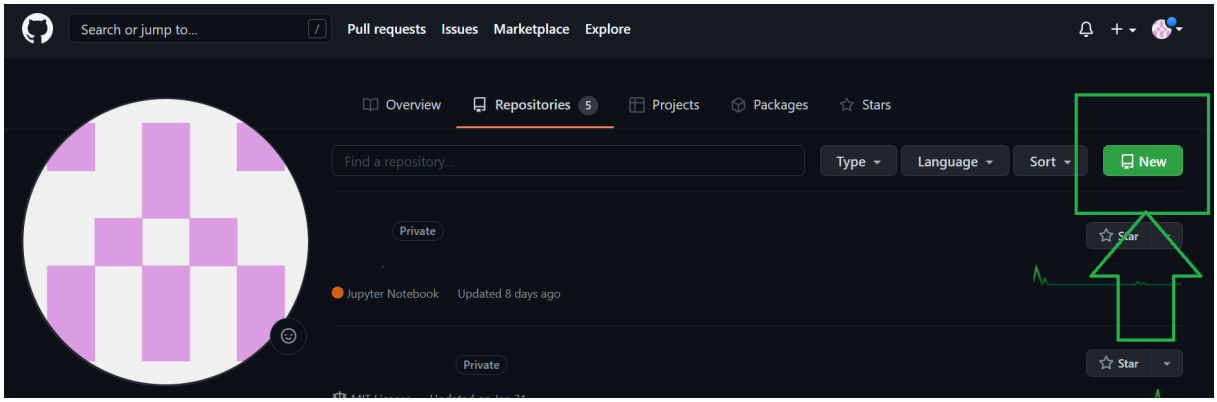


Image 7. Screenshot of the Repositories Tab in GitHub, with an arrow on the right to indicate the "New" Button where you can create a new repository.

To create a new GitHub Repository, you log in to your GitHub account and visit the tab "Repositories" (orange underlined). Here you can find a green button "New" (indicated with the arrow in the image) which you will need to click.

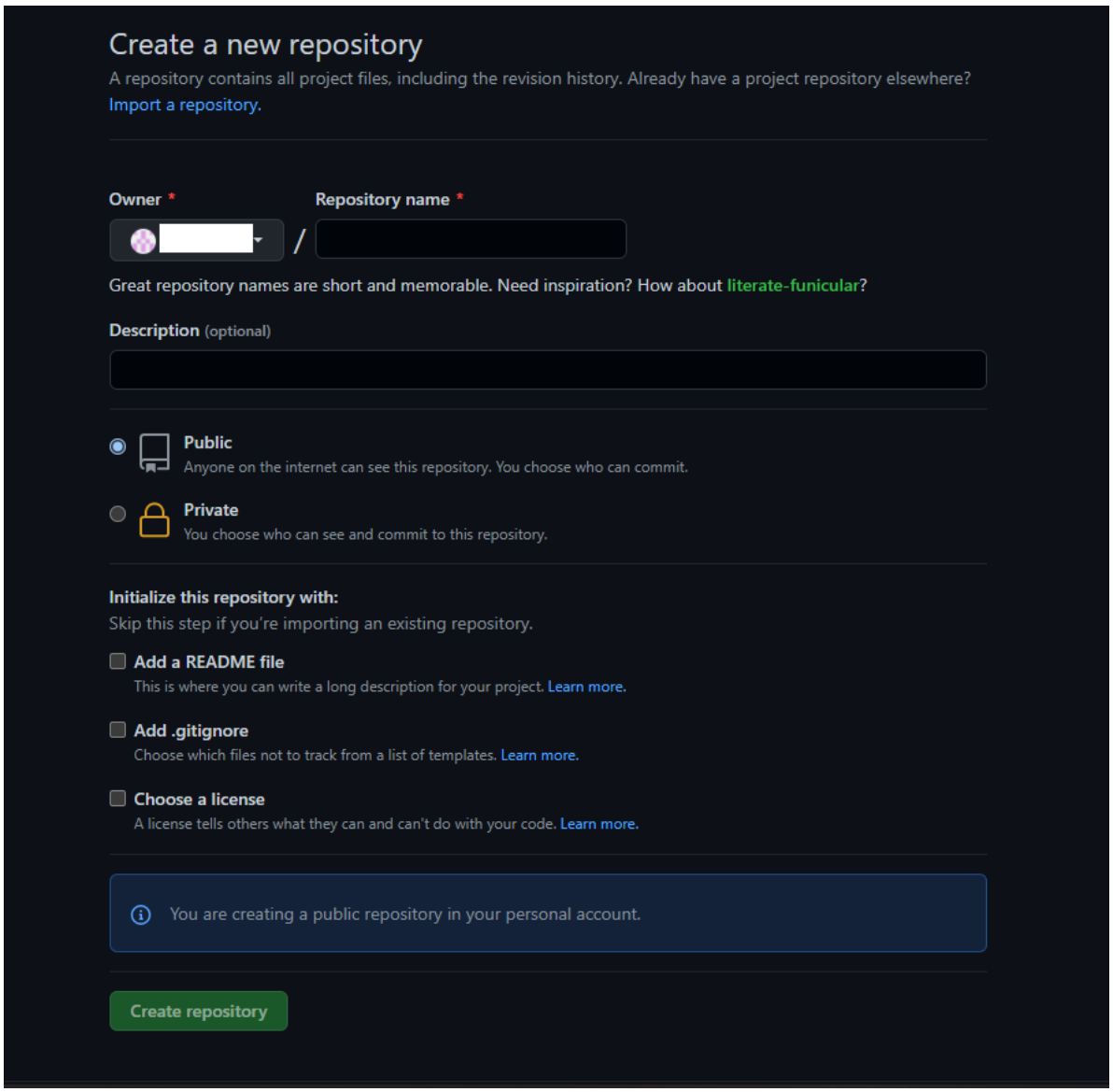


Image 8. Create a new repository.

When you create a new repository, you are asked to provide a name and a brief description. Additionally, you are choosing whether it is *public* or *private*. The contents of a public repositoy is visible to anyboy, the contents of a private repository is only visible to people whom you grant access. (How to grant access we will treat later.)

When setting up a new repository you are asked if you want to add some standard resources. One of these is the README.md file. This file is used for a brief description of the project. We sincerely recomend adding a README.md and use it to describe the aims, contents, and intended use of your project and files. The contents of the README file will be shown nicely readable on the front page of

your repository. It is an excellent way of introducing visitors to your project and goals. (Here and here are some hints about writing good READMEs.)

You can also choose to add a .gitignore file. This file is used to lists the files that you do not want to be integrated in the remote repository. For instance, suppose that you have a file in your local repository called `raw-notes.txt` and you do not want that to be integrated and shared through the repository, you would add a line saying simply `raw-notes.txt` to the .gitignore file. (Adding `*.txt` to the file would cause *all* text files to be ignored, adding `my_folder/*.jpg` will ignore all JPEG files in the folder named "my_folder" on your local repository.) Having such a possiblity to ignore files comes in very handy, so by default add it to your setup.

You will also be asked to chose a license fitting to your situation. The license determines how others can use and refer to your repository (and thus, later, your classification scheme).

After you have the choices that fit your needs, you click on "create repository" and you are redirected to your new repository.

The remote repository is now in existence and will have a little content in it. If you followed our defaults you would find the README.md, LICENSE, and .gitignore files listed.

Now we need to setup a local repository that will act as a "mirror" of the remote repository through git. To do this, find the HTTPS address for your repository. You will find this under the button "Code" (green arrow in the image):

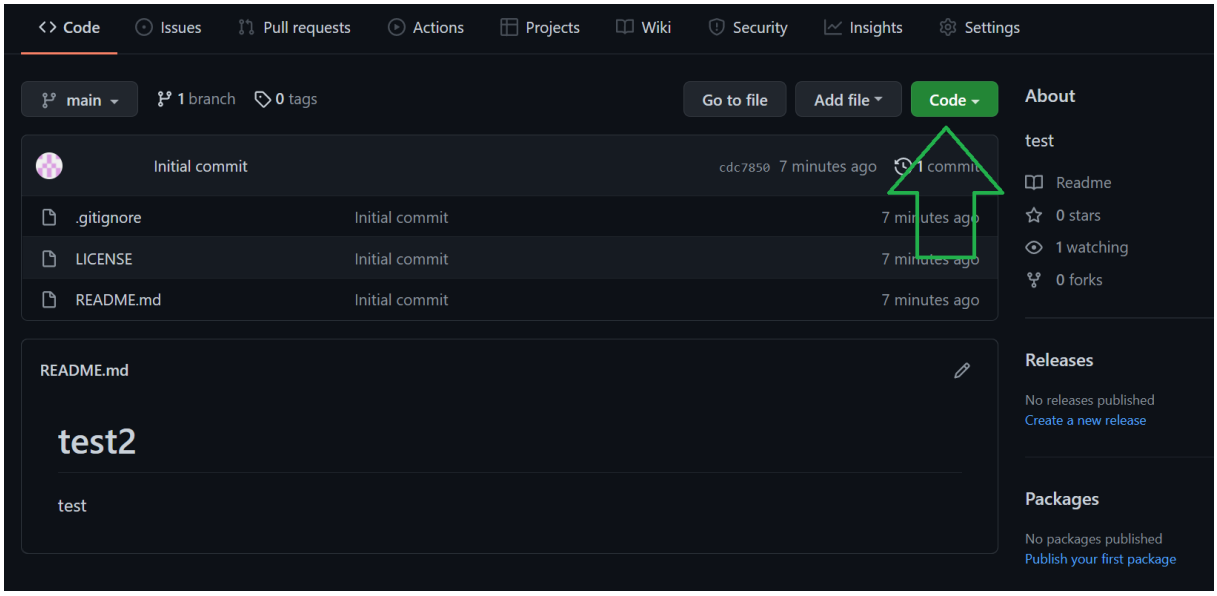


Image 9. Arrow indicating the button 'code' within the new repository.

If you click on the button a panel will open where you will find the HTTPS weblink to your repository. Copy this link.

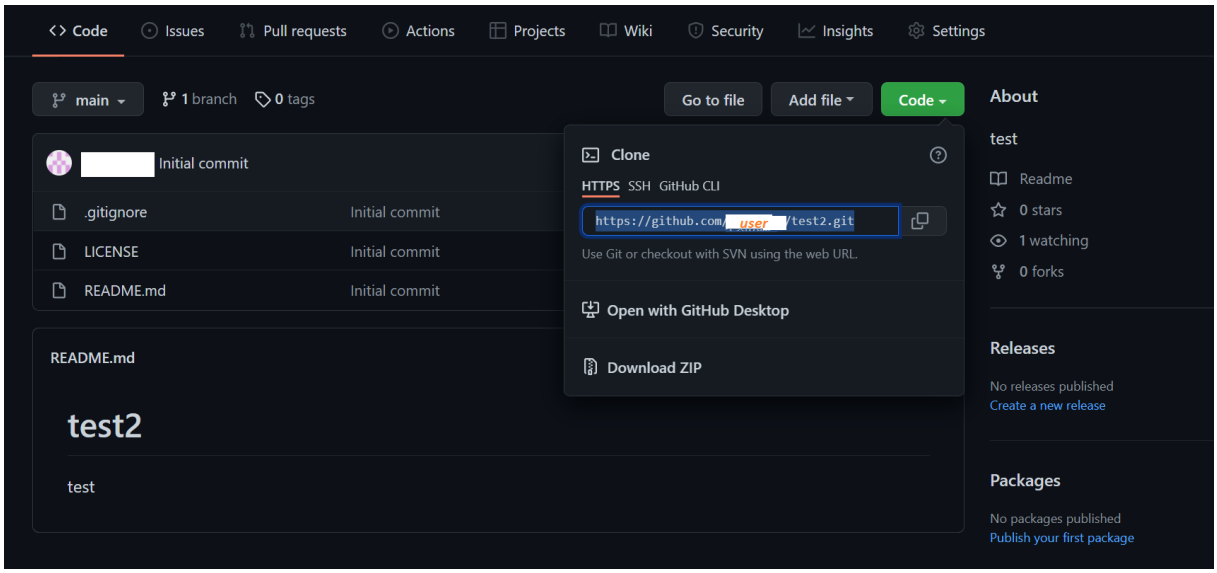


Image 10. Copy the HTTPS-link to your repository.

Setting up the local repository

Git offers various Graphical User Interfaces (GUIs). There are also code and text editors that integrate GUIs to use git for your resources (e.g. Visual Studio Code, Atom or Sublime Text). However, for the purpose of this tutorial you will work with git on the command line because you can use these commands exactly the same on all operating systems.

If you have not worked with the command line before, you may want to study a quick introduction to it. There are many tutorials for the command line around. You will find the Programming Historian's introduction to the command line here. Alternatively, look at one for the Mac here, and one for Windows here.

Access your command line and navigate to the folder where you want your local repository to be created as a sub-folder. On the command line now type `git clone`, type a single space, and add the HTTPS link from your GitHub repository. So, your command should look something like:

```
1 git clone https://github.com/yourname/yourproject
```

Now you press enter. Git will create a folder locally with all the content cloned from the remote repository. Following the example that folder would be named "yourproject", in your real context it will have exactly the same name as the remote repository's name.

Now change to this new folder as most of the git commands rely on the current folder being a git repository:

```
cd yourproject
```

Communicating between repositories

Obvisouly you want to communicate between your remote repository and the local repositories. The two most important commands that you need when you are updating either your local or remote repository are called `push` and `pull`. You **`push`** a new version of some resource **from** your **local** repository **to** a **remote** repository, using:

```
git push
```

And you **`pull`** a new version **from** a **remote** repository **to** your **local** repository, using:

```
git pull
```

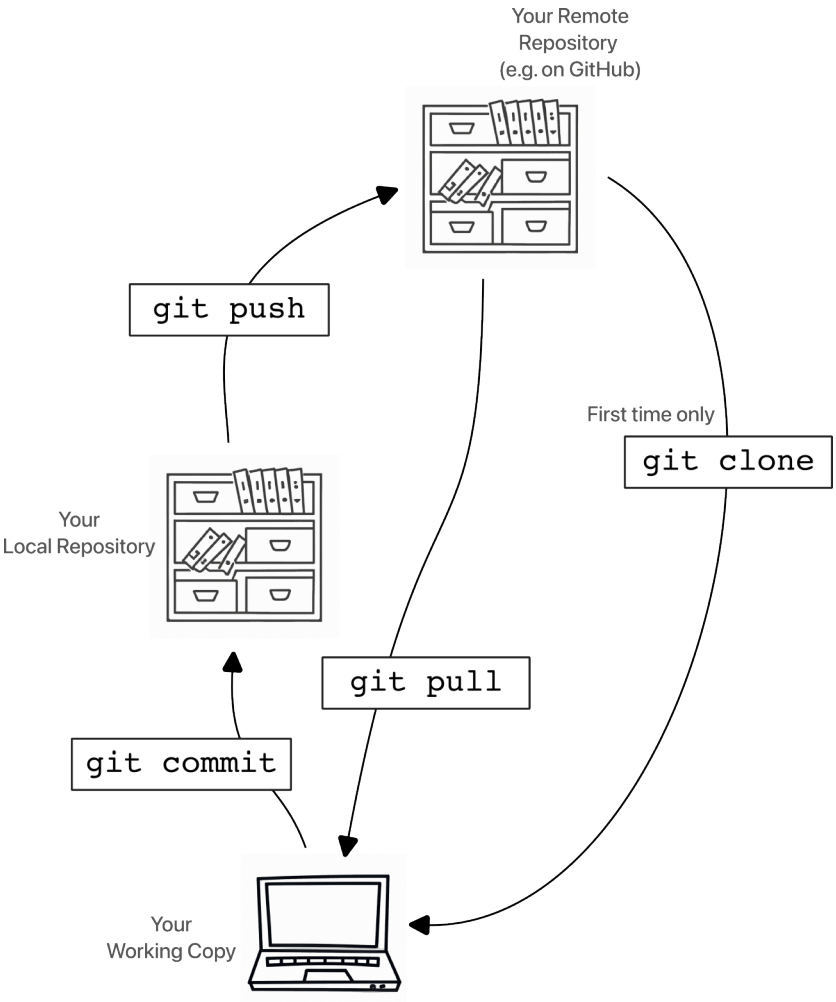


Image 11. Image indicating the GitHub remote repository and arrows pointing to and from local repository. The arrows are marked with the commands git push and git pull.
Source: J.J. van Zundert

However, before actually exchanging data and resources using `push` and `pull`, we need to understand if there are any changes either locally or remote. So, we need some ways to talk to both repositories through git about these potential changes. The most often used commands for this are:

- `fetch` downloads a "ledger" of latest changes in the remote repository.
- `status` indicates whether there exists any differences between local and remote repository.
- `add` adds a new file to the **index** of tracked files.
- `commit` pushes changes to your **local** repository.

Your first push to a repository

Let us see these commands in action. Of course, to do so, we do need an actual change first. So open up you favorite editor and make some little change to the README.md file that is tracked in your local directory since you used the `clone` command. When done, be sure to save your changes.[1]

You are now ready to commit your changes to the remote repository:

- Access your command line.
- Navigate to the project folder, i.e. the folder where your README.md file is stored.
- Type `git status` to check the state of your repository.

Git will now inform you that indeed something has changed. This will look something like the following.

```
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Image 12. Git status message.

Git is thus saying that the README.md file has changes but that these changes have not been committed to the repository. Technically, the process of getting stuff into the repository has several steps: you *edit* your files in what is called your "working copy"; then, of all the files you have edited, you *add* those that you are ready to commit to a "staging area"; and then you *commit* everything you have in this staging area to the local version-tracking "repository", adding an explanatory note. This way, while you may be editing in many files at the same time, you have the opportunity to compile just a selection of related changes for your commit and are not required to commit all or nothing. Long story short, next you use the `git add` command to add changes to the repository. You can use `git add [filename]` where you replace "[filename]" with a specific file. In our example you would use:

```
1 git add README.md
```

Instead of a specific file you can also add all changes found to the repository indiscriminately, like so:

```
1 git add .
```

[1] To work reliably with text and data resources use straight forward plain text editors such as Visual Studio Code, NotePad++, Sublime Text, Brackets etc. Do *not* use presentation oriented programs such as MSWord or OpenOffice or the like, that will contaminate your sources with unwanted additional presentation code or information.

People usually just use this last command, as it prevents a lot of repetitious typing. After you have done this, check the state of your repository again, using `git status`. The output will be slightly different:

```
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   README.md
```

Image 13. Git updated status message.

Git has now integrated all the changes we made in the local repository. We are ready to commit to our changes, which is the next step, using the command:

```
1  git commit -m [message]
```

In the place of `message` we add a reason for the changes or a short description of these. Do indicate what you changed, however comprehensively. If for some reason you ever need to revert changes this description will be very helpful in locating the version that you want to restore. In our example we could say:

```
1  git commit -m 'Small change for instruction purposes.'
```

After executing the command the change has been solidly locked into our local repository. However, now we still need to upload it to the central or remote repository. For this we utilize the `git push` command:

```
1  git push origin main
```

By default `origin` is the label that indicates the original remote repository we cloned from. It is possible to push to several different repositories, using different labels for various URLs where repositories are located. This advanced use of git is not covered here, however. Neither do we cover here the creation of different branches. Branches are a way of developing different variants of the same code. A variant (branch) is sublty different from a version. You might for instance have a Windows and a Mac branch of the same resource in one repository. Within each branch you may again have various versions. By default there is only one branch in a repository, which is called `main`. This is the second argument for our `git push` command. For our purposes it will be wholly sufficient to use only one origin and one branch, and therefore we will always use `git push origin main`.

Updating from a repository

It may be that you have been away for a while, and you have not tended to your collaborative project. However, it may be that in the meantime colleagues have added or changed resources. So, suppose you return after a few months, what do you do first? Indeed, you ***update*** your local repository with any changes committed (and pushed) by others to the remote repository. This will ensure that you are ***in sync*** and ready to work on the ***latest versions*** of all materials.

When you recommencing work on your local repository, it is thus good practice to start with `git fetch` and `git pull` to ensure you are working on the latest version. **Update often and certainly any time you start working again. Also, for instance, when you have just been away from your keyboard for a coffee break.** This is best way to prevent the headaches that come with a versioning hell. It may look a bit tedious at first, but you will come to love it after only a little while. Keep yourself happy and out of versioning mysery: use `git fetch` and `git pull` a lot.

To update your local repository, follow these steps:

- Check the status of your local repo: `git status`
- If your local repository does not report any changes, get the latest info from the remote repository: `git fetch`
- Check again the status of your local repo: `git status`
- If nothing has changed simply proceed to work.
- If there are changes reported, retrieve them by using `git pull`.

Also try not to leave changes in your local working copy or repository lying around for very long without committing/pushing them. In other words, commit and push often, certainly before taking a longer break from working on the project. This helps to avoid conflicting changes made by different collaborators.

Disciplining yourself: creating the routine

When you follow the routines described above you should soon be comfortable with retrieving changes from a remote repository and committing new materials to your local and remote repository. The more you `git fetch`, `git pull`, `git add .`, `git commit -m`, `git push origin main` the more it will become second nature and the more you will be able to avoid version conflicts.

Usually working with git and keeping versions in sync should be easy enough. But sometimes you may receive puzzling messages or errors. Suppose, for instance, that in the updating process just above for some reason the local repository would have reported some change. In that case you will not be able to update the local repository from the remote repository. Whatever the reason there was a change

(maybe you forgot to `git push` the last time you stopped working) and git will actively resist you doing any pull or push that would hurt the integrity of the remote authoritative version or that would overwrite any local changes.

In most cases it will be enough to carefully read what git reports, even if these messages may be a little hard to read. Sometimes git will say that it can "auto merge" local and remote changes. If this happens, just follow instructions and you will be fine.

If git really does not want to copy changes, you still have the option to overwrite your local repository with the remote data. Keep in mind however, that all possible changes on your local repository will be lost and the remote data will prevail. If you know for sure that you do not need the local changes, do:

```
1 git reset --hard origin/main
```

There is also a way to forcibly push your local state to the authoritative source on the remote repository. This is however a **nuclear** option, really. You will have to be utterly sure and convinced and able to prove that your local version is better than anything else there is in the remote repository. This is almost never the case. In any case, after you have written a 500 words essay on why you are doing this, you may overwrite the remote repository by using:

```
1 git push -f origin main
```

But seriously, consider all other options first.

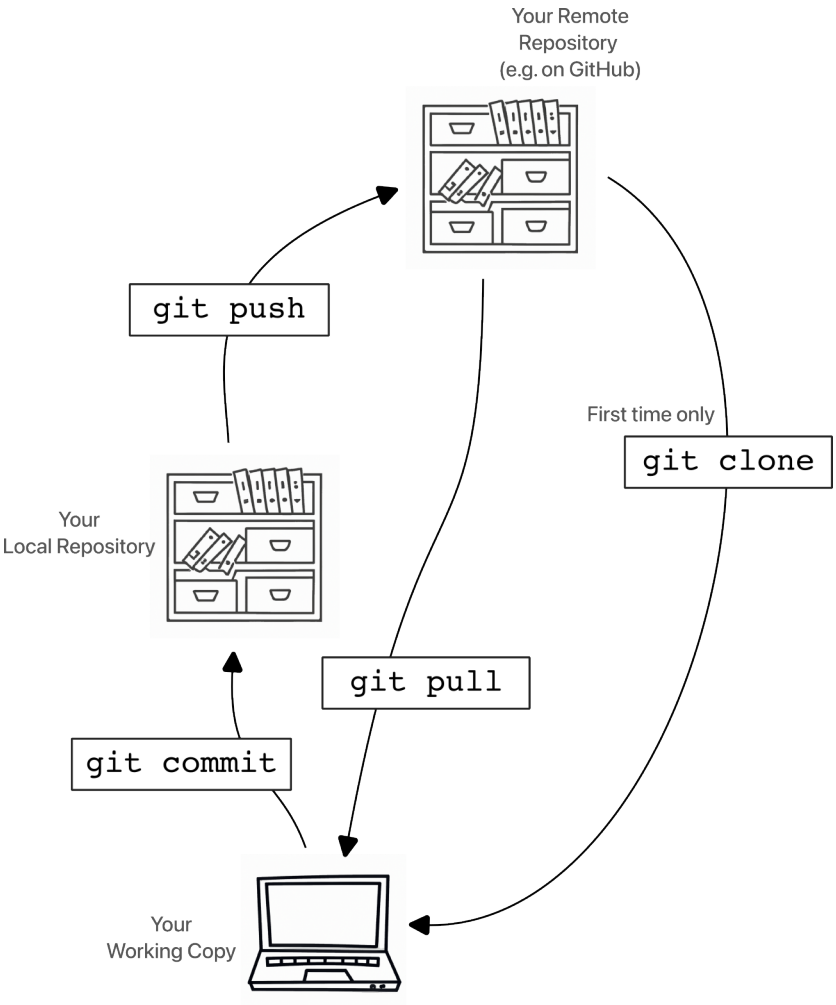


Image 14. Summary of the essential Git commands and what they do.
Source: J.J. van Zundert

Collaborating with others on your repository

Collaborating with others on the same files in your own repository is as easy as allowing sufficient access to the remote repository for other users. For this, your collaborators need an account on GitHub (if your repository is stored there), and you need to grant them access via the settings page on GitHub (see image 15). Git will take care that no one is accidentally removing or overriding changes by any collaborators unchecked. Your collaborators will be able to use the repository in exactly the same way as you are interacting with it (as depicted in image 16.).

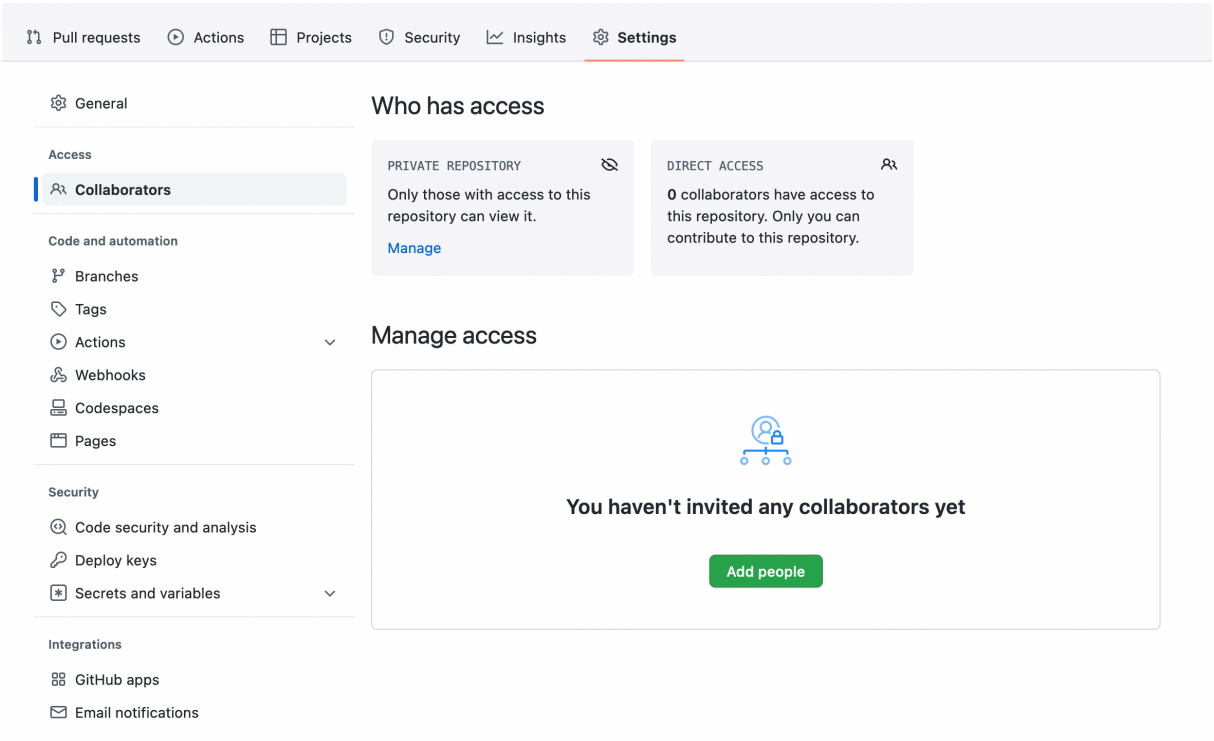


Image 15. GitHub: inviting collaborators.

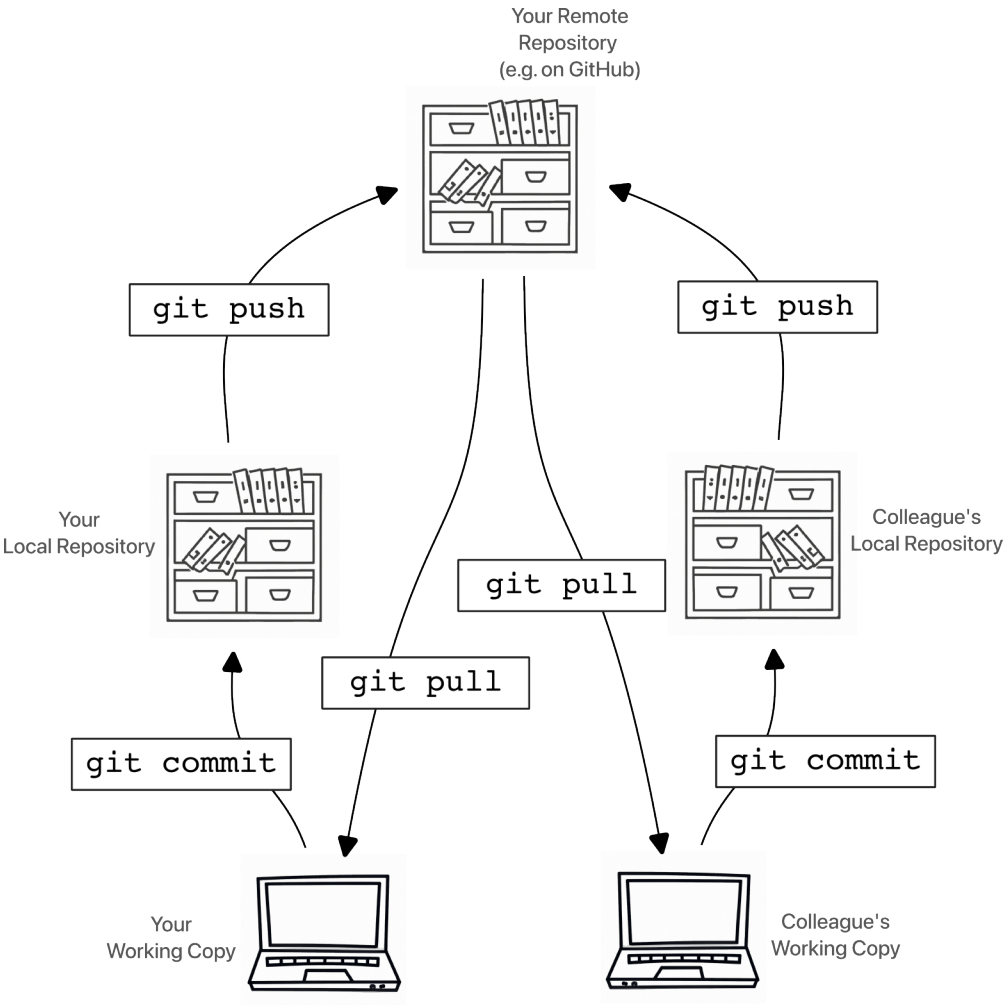


Image 16. Git: Multiple users working on the same repository.
Source: J.J. van Zundert

IV. Deploying your Classification Scheme

After having worked through the *what* and *why* of classification schemes, and after also having completed the *how* of building and collaboratively developing your own one, you should now have a classification scheme of your own, encoded in turtle language, in a git-controlled repository. The next step will be to deploy it, which means publishing your classification scheme online so that you, your collaborators, and perhaps others that might be interested can consult and use it as a controlled vocabulary in a linked data setting. As we said above, authority data is a controlled set of records, covered by a consensus of a community of researchers, that is available in an infrastructure offering certain services and it is this latter part that we will be addressing now.

Let us talk about the "services" that we aim for. We want to:

- Offer permanent URIs for your scheme and for each of its concepts, so other projects can refer to them unequivocally,
- Make these URIs resolvable, i.e. behave like URLs that provide both human readers and machines with more information about the resources (in the correct file format), i.e.
- Browse and search the terms and the hierarchy of your scheme in a web browser,
- Integrate the scheme with annotation and data cleaning tools.

Interestingly, for some of these services, you can rely on the internet infrastructure that you are already using anyway to take care of collaborative development: The GitHub platform offers, for free, possibilties to create a user-friendly web page for your project and stable access to your data. And it also supports automatic actions to be executed whenever your data or code gets updated in the remote (i.e. GitHub-hosted) repository.

In this chapter, we will see how we can use the skohub-vocabs software to *create* the necessary additional resources based on our .ttl file. We can use "GitHub pages" to *publish* these resources. And we can use "GitHub actions" to do this *automatically* with every update of our vocabulary. Finally, we will use the w3id.org web service to mint *permanent URLs* for all our resources.

After these publication steps, we will take it one notch further in the final chapter and discuss how to integrate the scheme with example tools of a data cleaning and an annotation workflow.

Building your files automatically with GitHub Actions and the skohub-vocabs Docker image

Before we discuss how to run tasks automatically on the GitHub platform, we should see what the task that we want to be executed *actually is*. Up to now, we only have a single turtle file containing your whole SKOS scheme, but the services outlined above expect other file formats and other granularities. We would need data to offer when the user (or an automatic client) requests information about a single concept, too. And we would need all this data in a format nicely readable for perhaps not so technically savvy human users who want to learn about our vocabulary using a web browser (i.e. not in the turtle format described above), and in at least one of the RDF serializations (like turtle, but maybe other formats are even more popular among automatic clients, respectively among the developers of such clients).

Enter skohub-vocabs, a so-called static site generator. If you run this software, it takes all turtle files in its `./data` folder and creates a `./public` folder with html files and subfolder structure. If you copy this folder to a webserver, you have a full-fledged web presence for your classification scheme: In addition to creating individual webpages for each concept (in the case of our policey ordinances subject matters, this means more than 1,800 individual webpages) and an index file to get information about

the scheme as a whole, it also creates such files in multiple formats that automated clients might request (JSON, JSON-LD). On the webpages, you can switch languages if you have added multilingual information to your turtle file; you also have a search form, and a tree view of your categories:

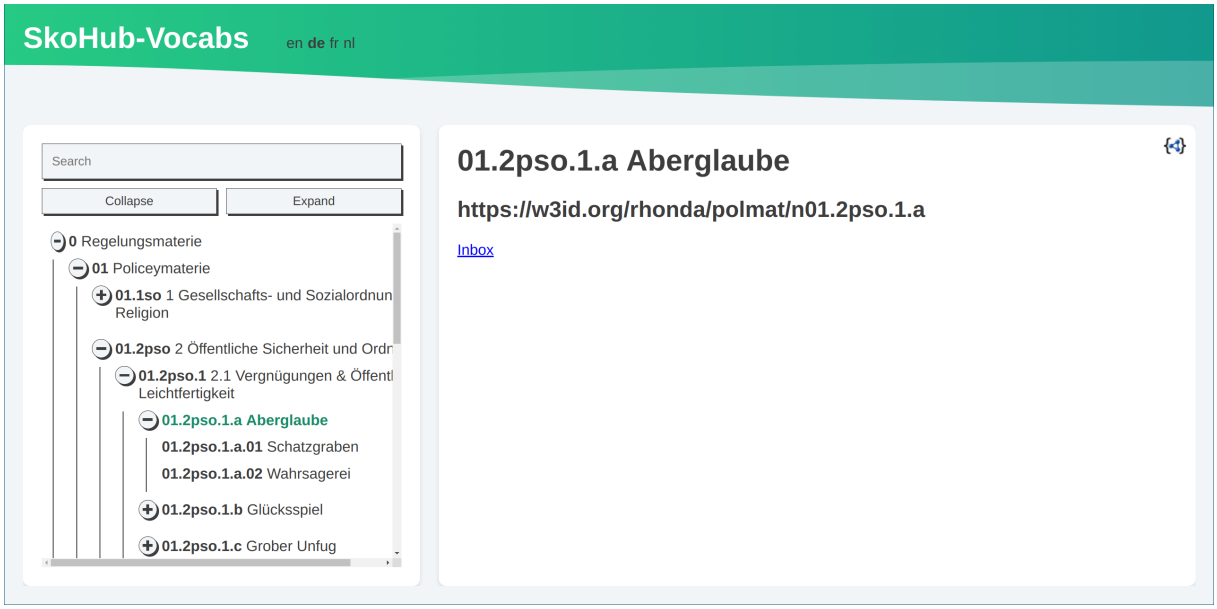


Image 17. Screenshot of concept on a skohub-built site.
Source: https://w3id.org/rhonda/polmat/n01.2pso.1.a.de.html

The developers of skohub-vocabs also suggest an interesting way of using the software: Instead of downloading, installing, configuring and executing the skohub-vocabs software, an alternative, quite clever way of running the software on your .ttl file is by making use of a *docker image*, a snapshot of a virtual environment in which the software and all its dependencies are already installed and preconfigured and which is capable of executing a command such as building all those files we need. Thankfully, the skohub people maintain such a docker image of skohub-vocabs available on docker hub, an online collection of such images.

Even more clever is running the docker image not on our local computer, but on the GitHub platform: With GitHub actions, you can specify processes that are triggered by events in your repository. Originally, this was developed to automatically run validation tests on software code upon changes, and - if the tests are successful -, deploy the new code to where it would be executed. However, the actions are immensely flexible and powerful. Thus, whenever code is pushed to our repository, we can have the platform launch a virtualisation environment (downloaded from a public collection of docker images), feed it with the files in our repository, run the processing, and push the resulting files to some specific branch of our repository. That saves us the hassle of bringing the image and our .ttl file to-

gether, and of collecting and uploading the resulting files after processing - the GitHub platform does all this, we just have to configure our "action":

Assuming you are working in a local copy of the main branch of your project, this is how you do it:

- Make sure your .ttl file is in a subfolder of the main folder called `data`
- Create a subfolder of the main folder called `.github` (note the leading dot), and in it a subfolder called `workflows`, and in this folder, create a file called `main.yml` with this content:

`.github/workflows/main.yml`:

```yaml
name: Build /public and deploy to gh-pages with docker container

on:
  push:
    branches:
      - master
      - main
  workflow_dispatch:
    inputs:
      logLevel:
        description: 'Log level'
        required: true
        default: 'warning'
      tags:
        description: 'Test scenario tags'

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v2
        with:
          persist-credentials: false

      - name: remove public and data-dir if already exists
        run: rm -rf public data

      - run: mkdir public

      - run: mkdir data

      - run: git clone https://github.com/skohub-io/skohub-docker-
          vocabs.git data/

      - name: make .env.production file

        run: echo "BASEURL=/skohub-docker-vocabs" > .env.production

      - name: build public dir with docker image
        run: docker run -v $(pwd)/public:/app/public -v $(pwd)/data:/
            app/data -v $(pwd)/.env.production:/app/.env.production
            skohub/skohub-vocabs-docker:latest

      - name: Deploy
        uses: peaceiris/actions-gh-pages@v3
        with:
          github_token: ${{ secrets.GITHUB_TOKEN }}
          publish_dir: ./public
```

(here is the documentation for the syntax of GitHub's workflow files.)

- change line 33 to have ***your*** full repository URL instead of `https://github.com/skohub-io/skohub-docker-vocabs.git`
- change line 37 to have ***your*** repository name instead of `skohub-docker-vocabs`
- commit your changes and push them to the platform

This will automatically run the whole build process on every push to the main branch and put the resulting files (inside the docker container, they are in the folder **`public`**) in a branch of your repository called "gh-pages". Depending on the size of your classification scheme, this may well take some minutes. If you go to the "Actions tab" in your repository's web site, you can watch the progress and check if everything was successful.

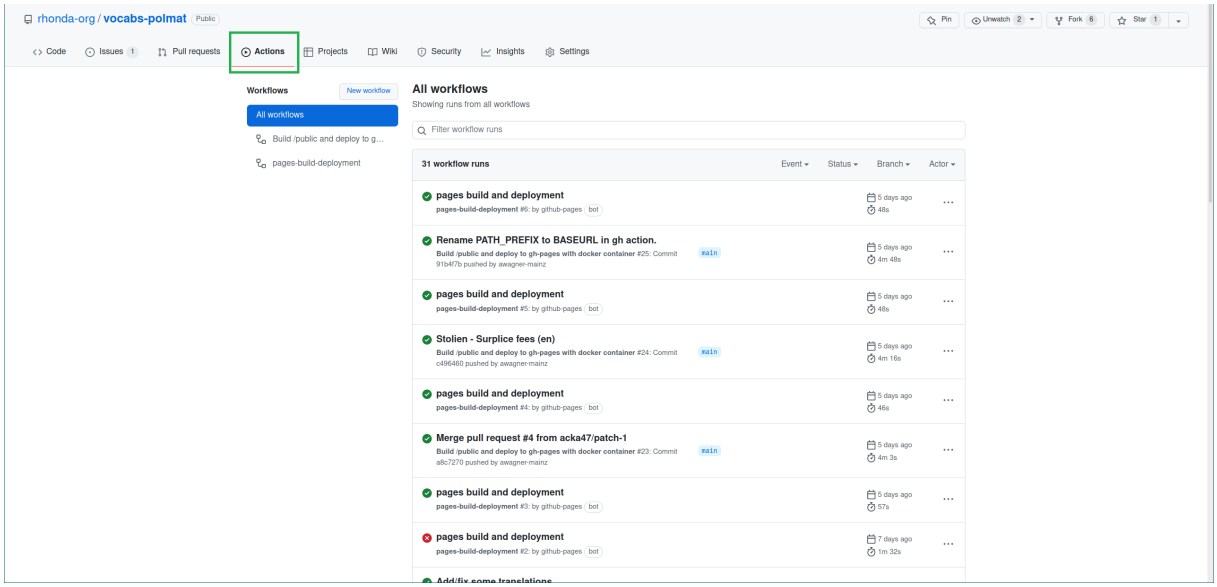


Image 18. Screenshot of GitHub's "Action" tab.

What is still missing now (and what eventually may have caused your first build to run into an error) is that we have not yet defined what this "gh-pages" branch is and how we can use it to host our html files as a beautiful website (instead of a repository of code files).

Publishing your classification scheme with GitHub Pages

"GitHub pages" is a feature of the GitHub platform where you use it as a web server rather than a git repository. Instead of building a UI with notifications, tabs to access and manage issues, actions and repository settings around the listing of files (and, in viewing a single file, displaying the actual

file content, with line numbers, information about change history etc.), the platform just delivers the raw file as-is. Obviously this only makes sense if the file has a format that a browser can understand, but if so, you may even skip an expensive web space provider and just use GitHub. If you have come across webpages the URL of which begins like `https://foo-bar.github.io/`, that's how they are hosted.

Above, we had our GitHub action populate the root folder of the `gh-pages` branch of our repository and so we just need to tell GitHub to use this folder as base folder for hosting our website.

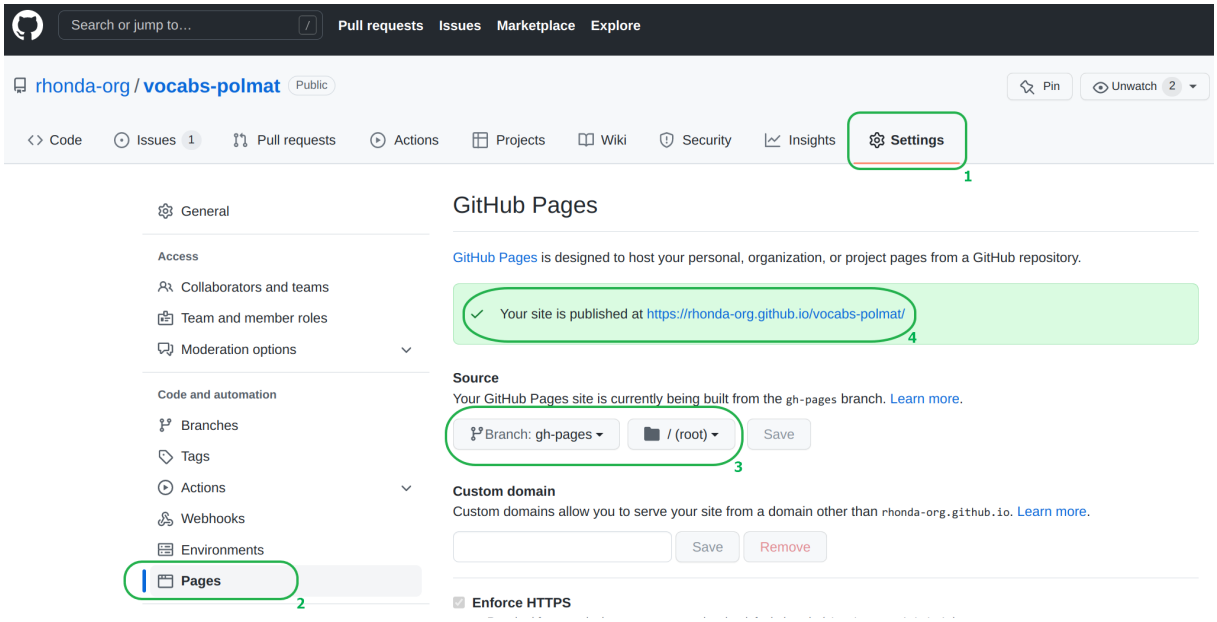


Image 19. GitHub-pages settings.

We do this by navigating to the repository "Settings" tab (box number 1 in the image), then selecting the "Pages" item in the list on the left (box 2) and selecting the branch "gh-pages" and the "/ (root)" folder (box 3). (If there is no branch "gh-pages" available you can create it if you click on the dropdown button "main" on your main repository page, entering "gh-pages" in the "Find or create a branch…" slot and then clicking on "Create branch: gh-pages from main") Click on the "Save" button and wait some minutes.

That's it. You should be able to access your classification scheme at `https://your-github-username.github.io/your-repository-name/` (or what is reported in box 4). If not, then maybe the build has not been triggered and you could make some small change to your turtle file and push it to the platform again (to trigger a re-build of the webpages). Or maybe it just takes a bit longer to build the pages, then just wait a bit longer.

Note that while the preceding sections have discussed the whole workflow based on the offerings of GitHub, there are alternative platforms you might prefer. GitHub platform provides all the features described above for free (for up to 2,000 minutes of processing time per month spent on executing your actions), but it's a private company owned by Microsoft. Alternative platforms, like GitLab, BitBucket or others, may offer a different set of advantages and drawbacks, but in terms of git compatibility, "actions", and "pages", they usually have quite comparable features.

Creating Persistent Identifiers with w3id.org

The following is also described in other web tutorials by the skohub team and an earlier one by Daniel Garijo.

In academic discourse, addressing resources – literature, but also datasets and vocabularies – by persistent identifiers is considered good practice. (Think of articles cited by DOI.) Such identifiers facilitate referring unambiguously to the same resource even when the infrastructure serving it changes over time. Since we are dealing with resources on the (Semantic) Web, it is quite natural to use URIs/URLs as such identifiers, hence Persistent Uniform Resource Locators (PURLs) are adequate persistent identifiers for our resources.[2] Consider that, while your vocabulary and your concepts may have parts of their URI that depend on the underlying infrastructure (like the `github.io/` in the hostname), you would like its users to refer to it via URIs that avoid such components. Then you set up an infrastructure that offers such "neutral" URIs and forwards requests to your actual (github, in this case) URL, allowing you to reconfigure the redirection when necessary. In this way, if at some point in the future you migrate from GitHub pages (with that `https://*.github.io` URI) to another platform, say Gitlab (giving you a `https://*.gitlab.io` URI), you just have to change the redirections, but all the urls "out there in the wild" remain valid because the platform is "masked". Of course, it requires effort of setting up and eventually updating the redirection, but we suggest it is worth the effort.

Once more, what is needed is an infrastructure and social promises of maintenance (by the infrastructure provider and by the redirection maintainer). Two popular platforms where you can maintain PURLs for your own resources are w3id and purl.org. The latter is managed by the Internet Archive and has a web user interface for maintaining redirections, but it requires an account with the Internet Archive. The former is more community-driven insofar as it is managed by the W3C Permanent Identifier Community Group and it offers more immediate access to the redirection server setup. So for this tutorial, we will set up redirections with w3id.

[2] Sometimes the concept of permalinks is mentioned in this context as well, and the difference between PURLs and Permalinks is not a categorical one, but rather "about domain name and time scale: PURL uses an independent dedicated domain name, and is intended to last for decades; permalinks usually do not change the URL's domain, and are intended for use on timescales of years."(Wikipedia). With this tutorial, you are going for a PURL.

As has been mentioned, w3id.org is just a redirection platform the configuration (or "routing") of which is configured in a collection of snippets of configuration files for an Apache webserver. The collection is maintained as a github project and maintained in subfolders, so that for a redirection of `https//w3id.org/myname/` to `https://my.site.com/`, you would have a folder `myname` in the w3id github project. In this folder there would be a `.htaccess` file, containing, most importantly, one or more `RewriteRule` instructions defining the redirects. For our vocabulary of police ordinances subject matters, the full file reads like this:

```
1 Options +FollowSymLinks
2 RewriteEngine on
3 RewriteRule ^polmat/(.*) https://rhonda-org.github.io/vocabs-polmat/
     w3id.org/rhonda/polmat/$1 [R=302,L]
```

The last line is the crucial one, defining what to redirect *from* with a regular expression (`^polmat/(.*)` meaning a path component `polmat`, followed by "anything") and where to redirect *to* as a full web URL (in our case, that's the rhonda-org github pages site, with a variable `$1` at the end, where the "anything" extracted from the original request is inserted). Note that the whole request matching expression is *appended* to the main w3id domain *plus* a path component reflecting the w3id subfolder: Our example file resides in a folder called `rhonda`, so the rewrite rule would match everything beginning with "https://w3id.org/rhonda/polmat/". The expression in square brackets at the end of the line defines the status of the redirection: the HTTP status code `302` ("FOUND" instructs clients to re-post their request at the new URL, but to keep the original url as the canonical address for the resource in their bookmarks and databases. The `L` tells the webserver that this is the last rule and if it matches, no further rules should be processed. (This way, you can have several rules and only the first one that matches will be executed.)

So, it should be relatively easy to adapt this file to your own needs/domain: You have to figure out how your "project" should be called in the w3id context (that will be the first path component of your persistent URLs), create a corresponding folder, copy the contents above into a file called `.htaccess` in this folder (note the leading dot in the filename), and specify redirection source and destination for *your* vocabulary.

What remains to be done now is integrating this with the configuration of the main w3id site. For this, we rely on another feature of the GitHub platform…

Another git/GitHub feature: forking and merging repositories As has been mentioned, git is a decentralized version control system, meaning every copy has the same "authority" and when one copy is lost, changes can be synchronized with any of the other copies. This facilitates collaboration, but it still does not settle the question who has write access to your project. In the spirit of Free and Open Source Software, you would want everybody to be able to make a copy of your code and adapt it

to their needs, but in order to feed back useful those changes into your own codebase, you most likely do not want to give everybody write access to your repository. For such a scenario, *forking* is a strategy that is frequently used, but it presupposes a platform on which different projects can exchange code updates:

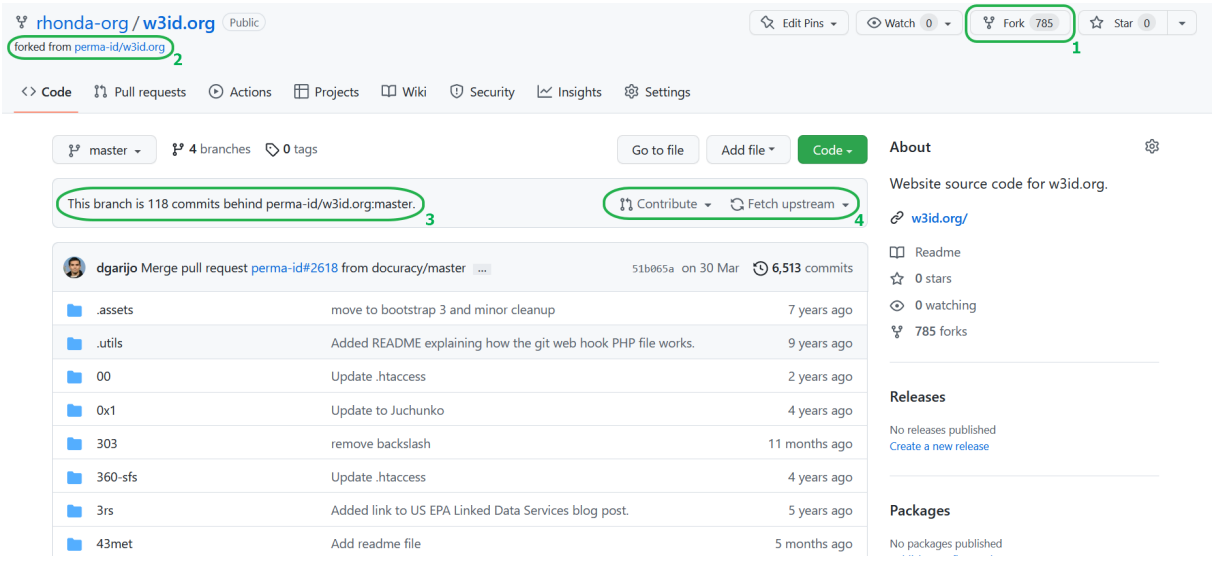


Image 20. GitHub Forks.

If you visit any project on GitHub, you will see a button “Fork” in the upper right corner (box 1 in the screenshot) and if you click on the button (on the text “Fork”), you will create a copy of the repository in your own account. (The number on the “Fork” button tells you how many users have already pulled a copy of the repository into their own namespace. If you click on the number instead of on the text “Fork”, you will be taken to a list of these copies.)

If you create a fork, it resides in your account or namespace and you are its owner, so when you clone, branch, pull, push your code from this (i.e. your) repository, you will have full access rights and can develop as if you had been the original creator. However, the platform remembers the relation between your repository and the original one (see boxes 2 and 3). (Often, when discussing such a scenario, your repository on the github platform is called the “origin” remote repository, and the repository you had forked from is called the “upstream” remote repository.) What we will want to do is collect information about how your fork has diverged from the upstream repository (this should be only your project folder with the custom `.htaccess` file) and submit this to the upstream maintainer so, that they can “pull” the changes to their original codebase and deploy them to the redirection service running at w3id.org. For this, the GitHub platform offers you to open a “Pull Request” (for what it’s worth, in GitLab this is called “merging” and a “Merge Request”).

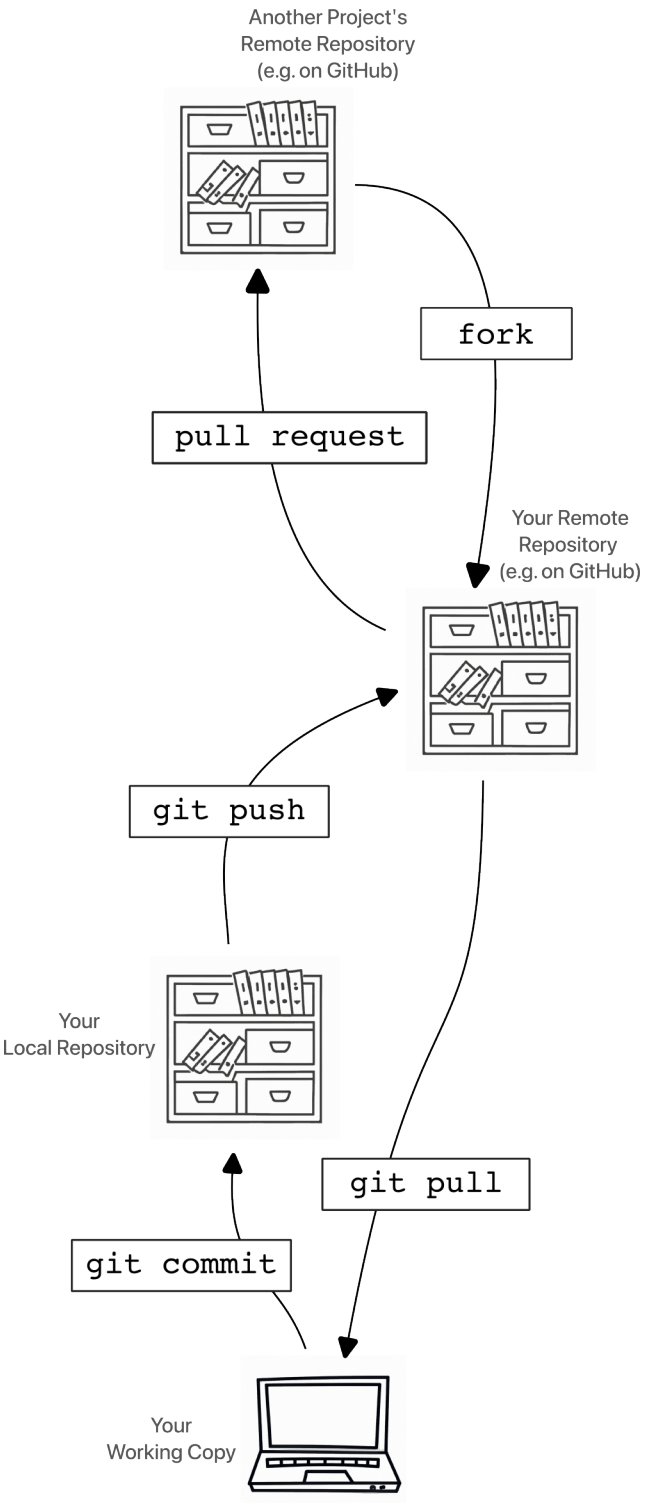


Image 21. A GitHub fork in relation to your repository.
Source: J.J. van Zundert

The process you need to perform now is this:

- Point your web browser to the original ("upstream") w3id repository at https://github.com/perma-id/w3id.org
- "Fork" it, i.e. push the fork button and you end up with a copy of the repository in your own GitHub account
- Download or "clone" your fork locally
- Add a new folder with your project name, containing your custom `.htaccess` file (and per good netiquette also add a REAMDE file).
- Commit and push your changes to the repository on GitHub
- (If it has been some time since you had forked, you may want to make sure the rest of the repository is up to date by clicking on "Fetch upstream" - see box 4 in the screenshot above)
- Submit your changes to upstream: Click on "Contribute" and then on "Open Pull Request"

After some time, one of the upstream w3id maintainers will have a look at your request and either approve (and pull the changes and subsequently close) the request or get back to you with questions to clarify or rectify something. Whatever it is, you will get a notification via the email address you had registered your GitHub account with.

If the Pull request has been approved and merged successfully, you can test if the redirection works: point your browser to `https://w3id.org/yourproject/yourvocabulary/scheme.html` and see if you end up on your vocabulary's web presence at github.io. (The last bit of the URL depends on the identifier you are using for your `conceptScheme` in your SKOS file.)

It is important to have your concepts have these persistent URLS as their identifiers also in how they are described in the SKOS file. Here, we are declaring a very short prefix `:` to correspond to the base w3id address and then this prefix leads all references to concepts, conceptScheme etc.:

```
@prefix : <https://w3id.org/rhonda/polmat/> .

  ...

:scheme a skos:ConceptScheme ;
    dct:title "RHONDA Categories of Matters"@en ,
    ...

:n0 a skos:Concept ;
    skos:inScheme       :scheme ;
    skos:topConceptOf   :scheme ;
    skos:narrower       :n01 ,
                        :n02 ;
    ...

:n01 a skos:Concept ;
    skos:inScheme       :scheme ;
    skos:broader        :n0 ;
    skos:narrower       :n01.1so ,
    ...

:n01.1so a skos:Concept ;
    skos:inScheme       :scheme ;
    skos:broader        :n01 ;
    skos:narrower       :n01.1so.1 ,
                        :n01.1so.2 ,
                        :n01.1so.3 ,
                        :n01.1so.4 ,
                        :n01.1so.5 ,
                        :n01.1so.6 ,
                        :n01.1so.7 .

etc.
```

(In the example code above, all the lines with labels, definitions and so on have been eliminated, so you can see just how the `:` prefix works.)

DONE! Congratulations, you have created a classification scheme and published it as a website for everybody to consult.

V. Integrating your Classification Scheme in Workflows

Now that you have your classification scheme built and deployed, we want to illustrate how such resources can be used with two example workflows where the classification scheme plays a central role. At present, almost all of the SKOS development and management tools require you to run your

own server in order to host your vocabulary, or to be allowed to upload your vocabulary onto such a server that is being run and maintened by someone else. To some extent, we have already worked our way around this by hosting our vocabulary on GitHub pages. However, we would like to integrate our vocabulary more tightly into our workflows than to just point users to a website where the concepts are explained. After all, it's a digital resource and so far, it would be analogous to a dictionary on our bookshelf. Instead, we would to have it more "at our fingertips" in the tools we are using in our regular workflows. To achieve this, we would need our online vocabulary resource to respond to some dedicated data exchange protocols, which, unfortunately, the hosting at GitHub pages does not do.

This final chapter will thus focus on ways of integrating your vocabulary in workflows by providing it via an application programming interface (API) that your other tooling can interact with. One API standard that is not necessarily connected to SKOS, but is particularly fitting for several integration scenarios is the Reconciliation Service API.

As this is meant rather to illustrate how such a concept scheme (service) can be used in the context of concrete scholarly workflows than to explain the context and setup of the workflows in detail, we will briefly describe the steps necessary to integrate the vocabulary and show you how an integrated workflow would look like, but we will more or less presuppose that the working environment has otherwise already been set up. For more detailed information about the methods used in our example workflows, we will be referring you to other tutorials.

We will present two example workflows, both of which rely on another free platform, supporting the data exchange protocol mentioned above: The skohub developers (staff at North Rhine-Westphalian Library Service Centre) have a companion to skohub-vocabs called skohub-reconcile, which can serve a Reconciliation API for a SKOS scheme submitted. And they even have a server where anybody can submit their vocabulary (in turtle SKOS format) to have it served via Reconciliation API. Be aware that this is currently in alpha status and it is quite likely that the service will eventually migrate to another address/domain. But we expect few problems in the actual operation, and with the w3id.org redirection we have already seen how we can respond to the service moving to another URL. The ability to respond to GitHub webhooks is on the roadmap for the service, too, so you may at one point have a fully automated, always up-to-date reconciliation service for your vocabulary. Hopefully other SKOS platforms (and working enviroment tools) will start supporting the reconciliation API so that choices will become more numerous and diverse. To the best of our knowledge, though, the skohub initiative is the only one where hosting is included and you don't have to run your own server. (You might try to get in touch with the SSH Vocabulary Commons project and ask them about hosting your vocabulary, though, too).

Preparation: Offer your vocabulary via Reconciliation API with skohub-reconcile

Reconciliation means to align your dataset with an external (authority) resource, in order to either

consistently use the normalized terms of your vocabulary everywhere in your dataset, or directly use the unique identifiers provided by the auhtority database in place of the terms exclusively. Historically, The data cleaning tool OpenRefine (or Freebase Gridworks that would later become OR) has pioneered the development of a communication protocol for this purpose: You'd need a standardized way of asking the authority database "what entries do you have that match the string 'abcde' in some way?" and a standardized way for the database to reply "I have four entries, which I am listing here along with their ID and their type (person/book/concept/place)…". Soon, many databases supported the way OpenRefine was asking such things and the form in which it would expect answers. Since 2019, this communication protocol has been maintained and developed by a W3C community group as an independent API specification. A long list of databases that support queries via reconciliation API can be found in the Reconciliation API testbench (at the time of writing, this list has almost 50 entries), and clients capable of dispatching such queries as well as libraries facilitating the development of such clients are listed in the Reconciliation census.

All the authority files we mentioned above offer a reconciliation API and thus can be used to match your data with. However, we are interested in providing a reconciliation API endpoint for or *our own* vocabulary, thus enabling users from various projects to use their clients to access our controlled vocabulary. For this, we go to https://reconcile-publish.skohub.io/index.html, enter an account name (this is to help distinguish between several users submitting concept schemes that happen to have the same identifier), a main language and select our turtle file for uploading:

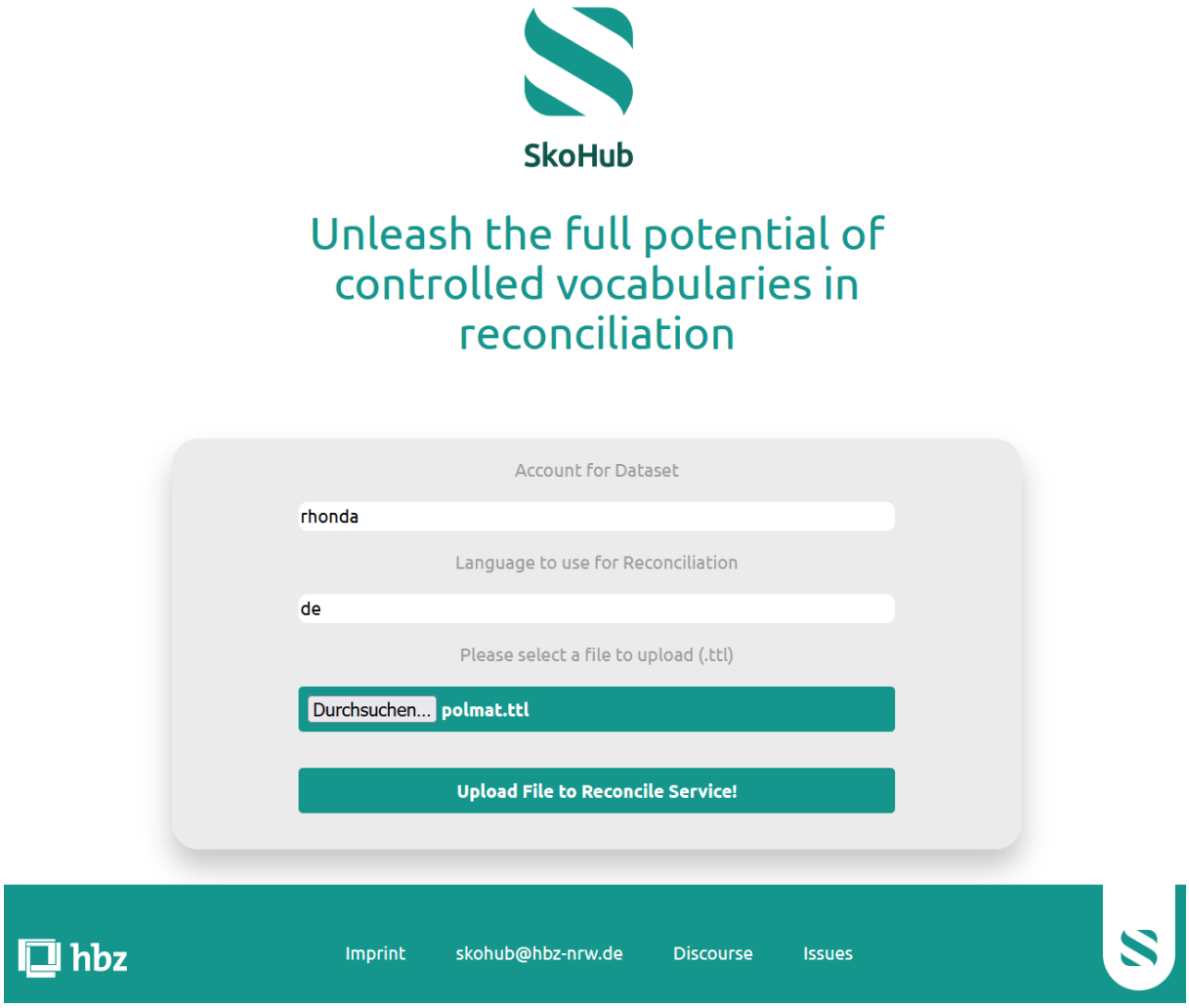


Image 22. Skohub-reconcile: Prepare to upload.
Source: https://reconcile-publish.skohub.io/

After we push the “Upload File to Reconcile Service!” button and wait a few minutes, we get a confirmation screen with the URL for the reconciliation service based on our vocabulary:

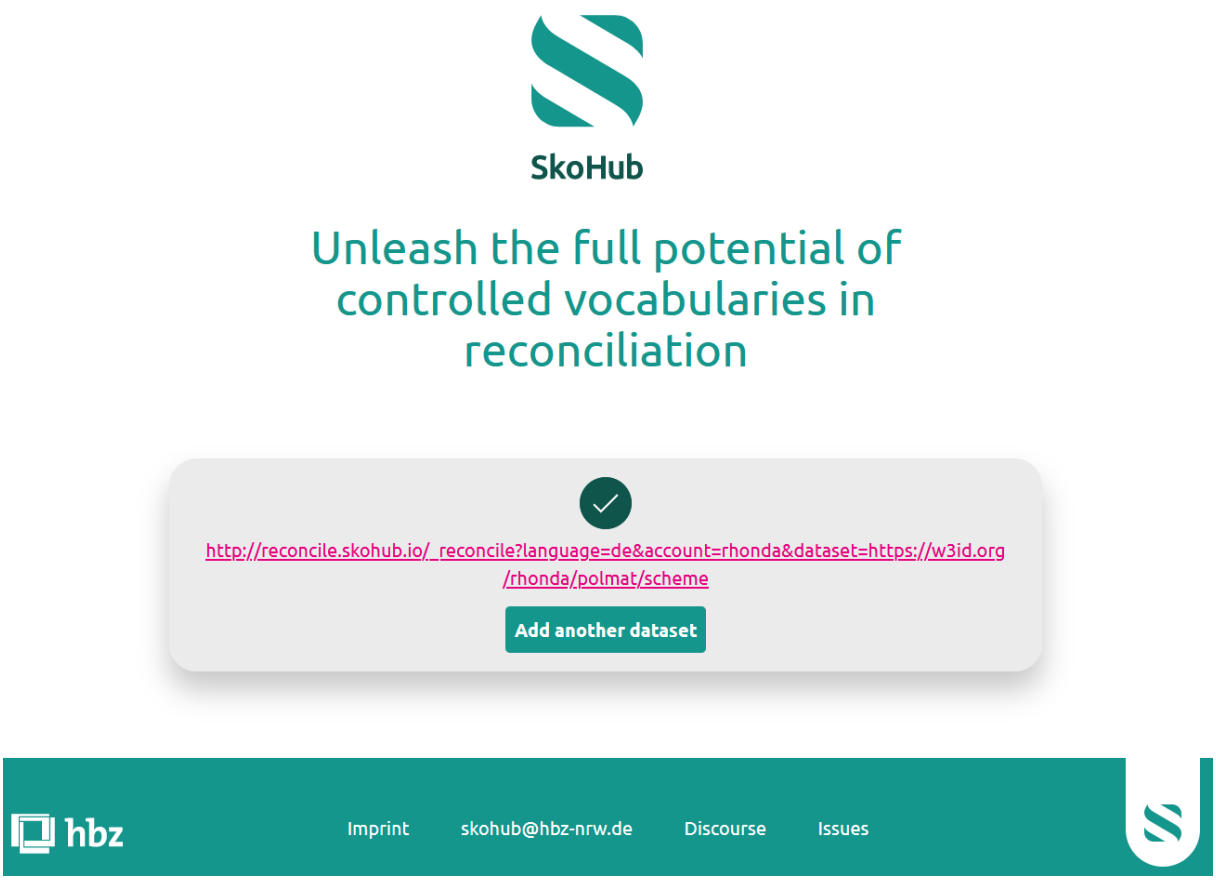


Image 23. Skohub-Reconcile: Upload done.
Source: https://reconcile-publish.skohub.io/

In the URL, you can see that account name and language reappear and the dataset parameter is the URI for the `conceptScheme` (resulting from the expanded prefix and the identifier) in our SKOS file. Yours might obviously have different values there. In the following two example workflows, we will thus be using this reconciliation service for the matters of police ordinances vocabulary, available at http://reconcile.skohub.io/reconcile?language=de&account=rhonda&dataset=https://w3id.org/rhonda/polmat/scheme.

Workflow 1: Reconciling a dataset with OpenRefine

As Seth van Hooland, Ruben Verborgh and Max De Wilde have shown in their Programming Historian tutorial on Cleaning Data with OpenRefine, the OpenRefine tool is easy to use in order to clean data of various types of datasets. You can do this with regular expressions, the General Refine Expression Language (GREL) or Jython but you can use the built-in features of the GUI as well. However, van

Hooland et al. mention OpenRefine's reconciliation function only in passing. In this section, you will learn how to reconcile your data with your controlled vocabulary, i.e. to make sure all the relevant values in your dataset conform to and make use of the controlled vocabulary and its concept PIDs.

If you have not already done so,

- download the most recent openrefine zip from https://github.com/OpenRefine/OpenRefine/releases
- extract it, and then in the resulting folder,
- run the openrefine executable (`refine.bat` if you are on MS Windows, `refine` otherwise)
- open your browser and navigate to OpenRefine's user interface at http://127.0.0.1:3333/

Before starting with reconciliation, you may want to have a quick check of the data and, depending on whether the reconciliation data source supports fuzzy searches or not, make sure that the spelling of your words is uniform. For such initial data profiling and cleaning, we refer you to the Cleaning Data with OpenRefine tutorial.

But now, let's start reconciling your data. This you can do per column again, by going to the dropdown menu of the respective column (we will use column "Scope_clean 2" in the screenshots) and then go to the "Reconcile"-menu. Here you can chose to 'Start Reconciling'.

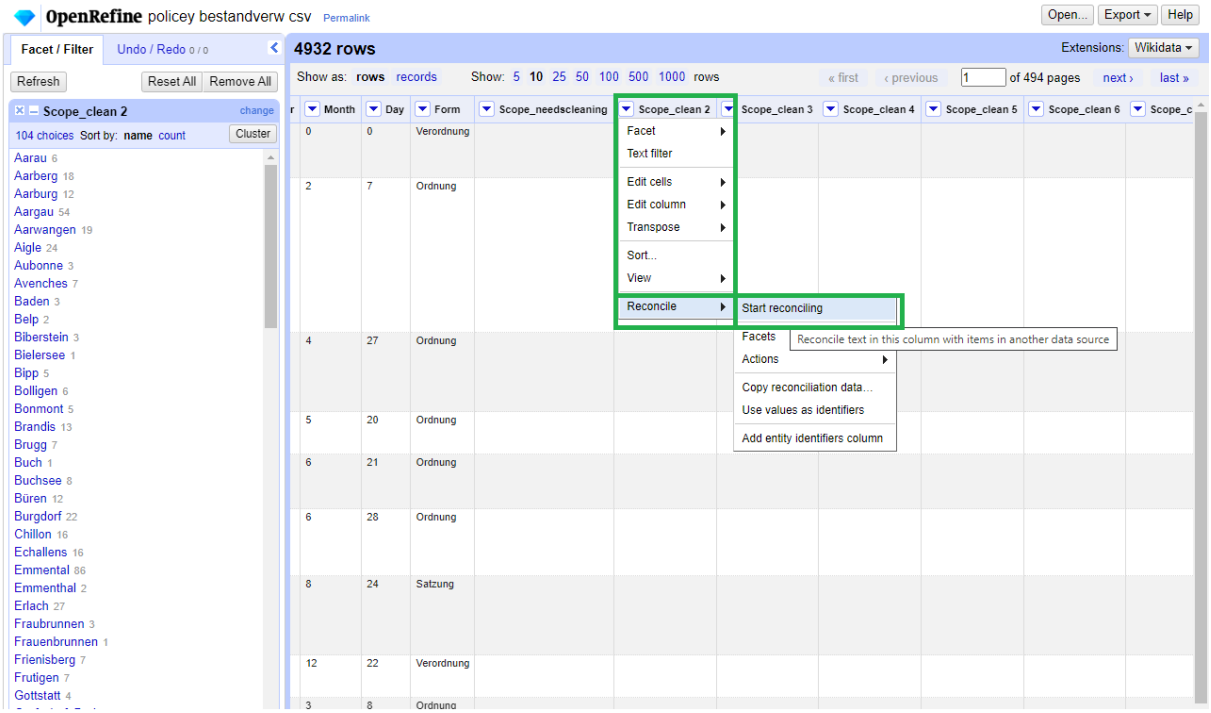


Image 24. Start Reconciling.

In the following menu, you have two options. Either you choose one of the available services (left field) or you click on the “add standard service”. OpenRefine has already a few built-in options, but you can add other trusted (by you) services that follow the Reconciliation API protocol.

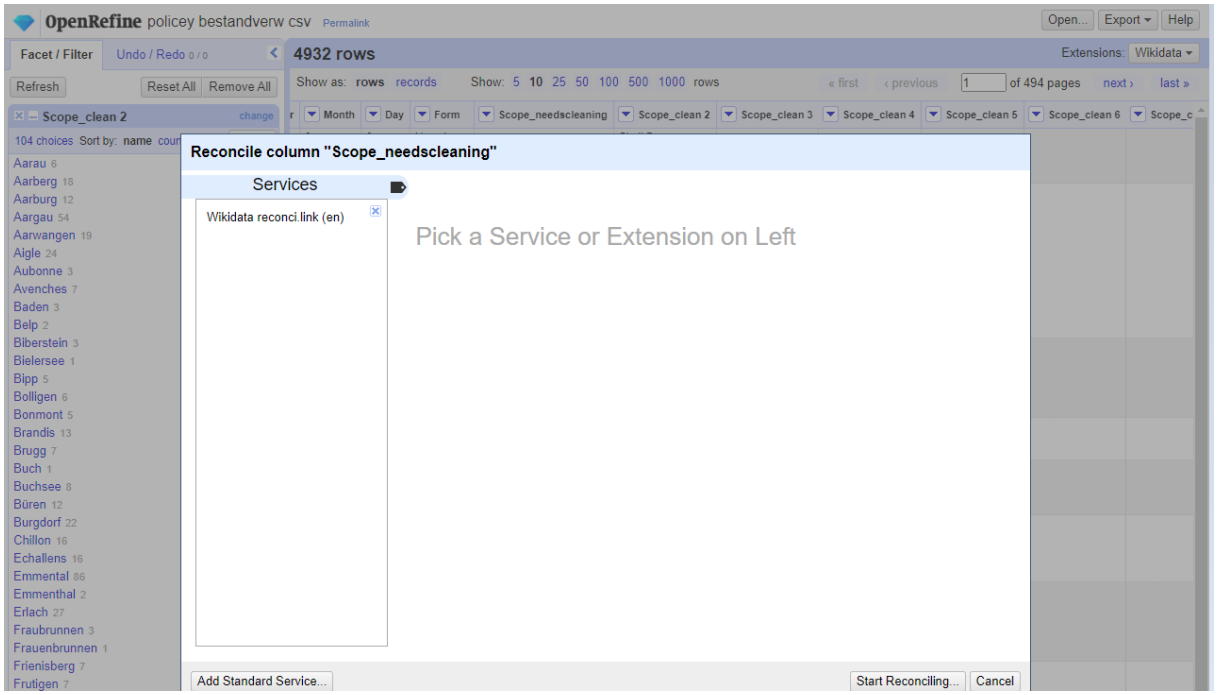


Image 25. Add a reconciliation service.

First, we will show you how you can reconcile placenames. When you have clicked on the ‘wikidata reconci.link(en)’ you are immediately offered some suggestions, based upon your column’s content. You can restrict the types of entities that should be taken into account, specify additional columns holding information that might be helpful in deciding uncertain cases (e.g. the country column when you are reconciling places, or the birthdate column when you are reconciling persons), and tell OpenRefine to skip creating a menu for you to pick one of multiple candidates when the match is already quite unambiguous. In this screenshot, we are telling wikidata to return entries that are Swiss municipalities, to consider data in the “Scope_clean 3” column for finding good matches, and to indeed skip creating a menu when matches are unambiguous (“Auto-match candidates with high confidence”):

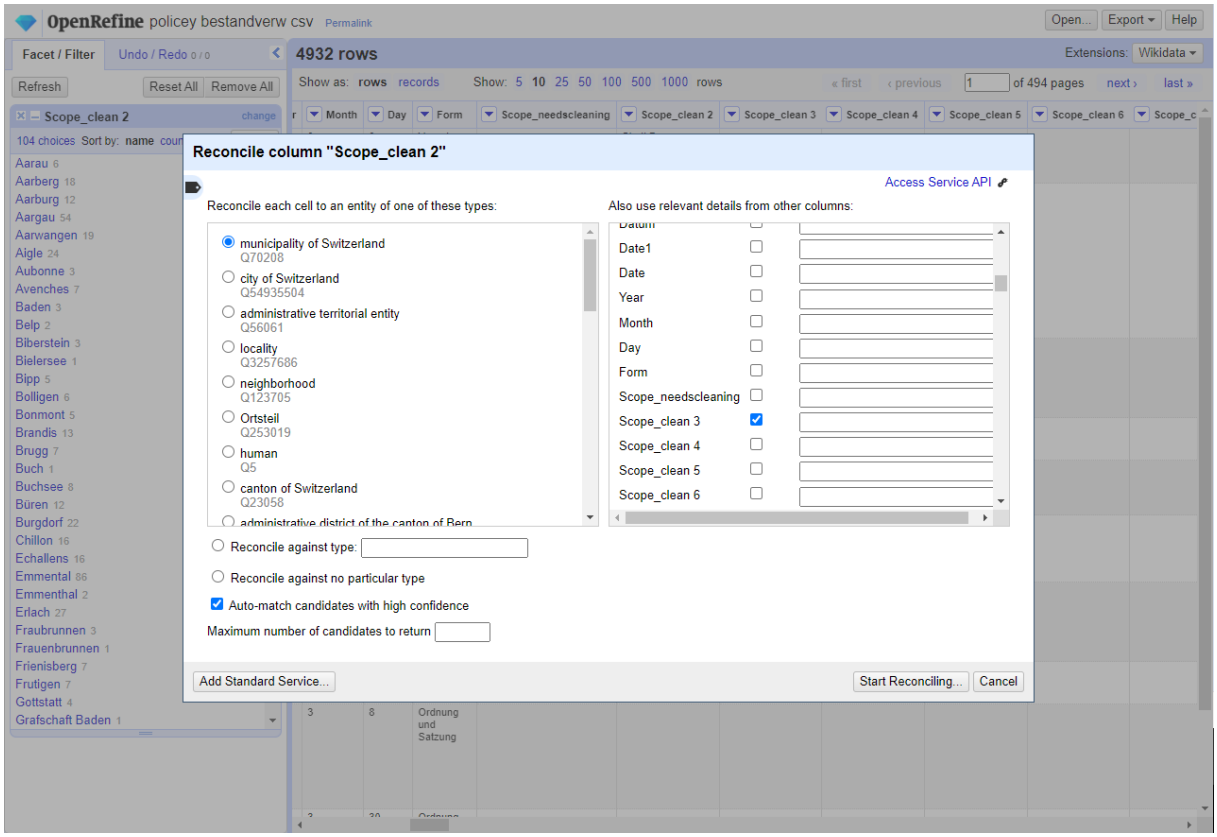


Image 26. Auto-match candidates against geographical locations (example municiplity of Switzerland).

After selecting all these options and clicking on *Start Reconciliation* OpenRefine is matching the listed names. The result is shown in the next image:

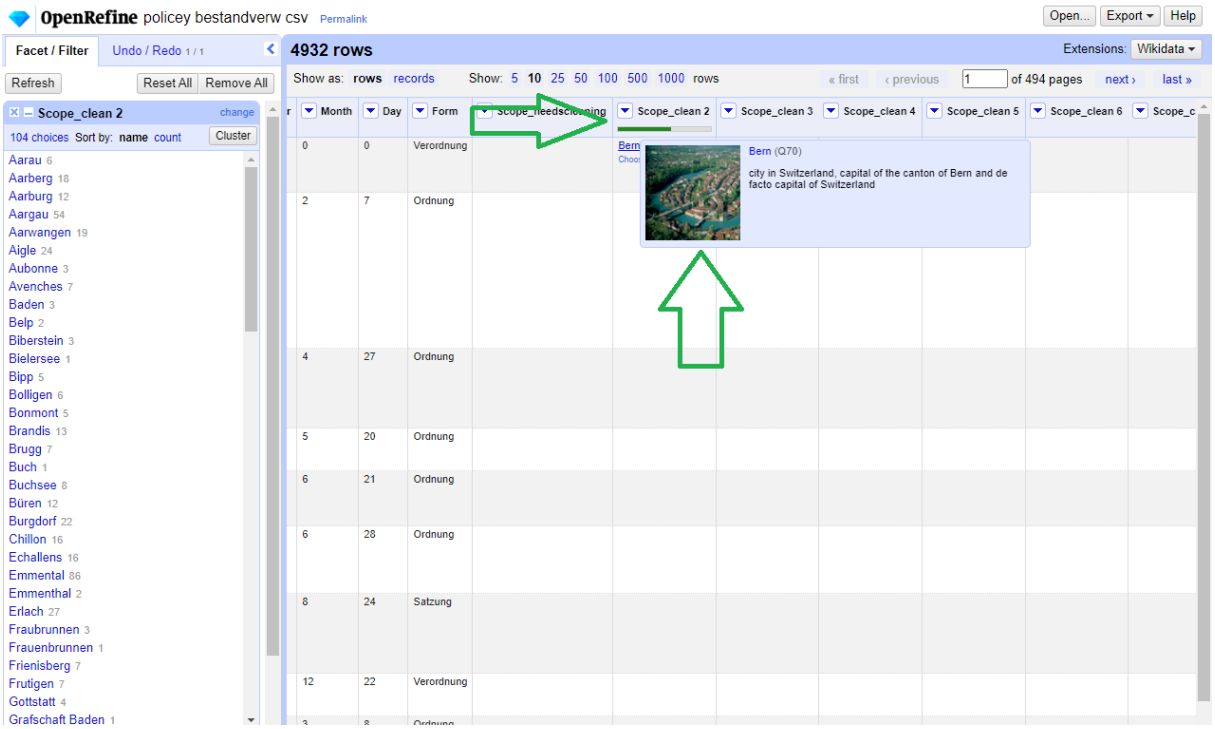


Image 27. Reconciliation results - with geographical references.

The upper, right-facing green arrow points to information about how much of the column has been matched against the chosen autority files. In this case, the bar indicates that about 58% could be matched against wikidata records and about 42% still needs reconciliation. When you hover your mouse cursor over the hyperlink of one of the places that *has* been matched, you will see a preview of the wikidata record this has been matched to, as well as the ID-number of the autority file ("Q70" for Berne, in this case).

In cases where no sufficiently unambigous match was available, or where there was a high uncertainty, OpenRefine provides suggestions, such as the candidates for "Aargau" here. With these suggstions, you can define whether the value in the current row corresponds to one of the entries (the checkbox with the single checkmark icon), or whether all the entries in all the rows that have the same value ("Aargau") should be linked to the corresponding suggestion (the checkbox with the double check-marks icon):

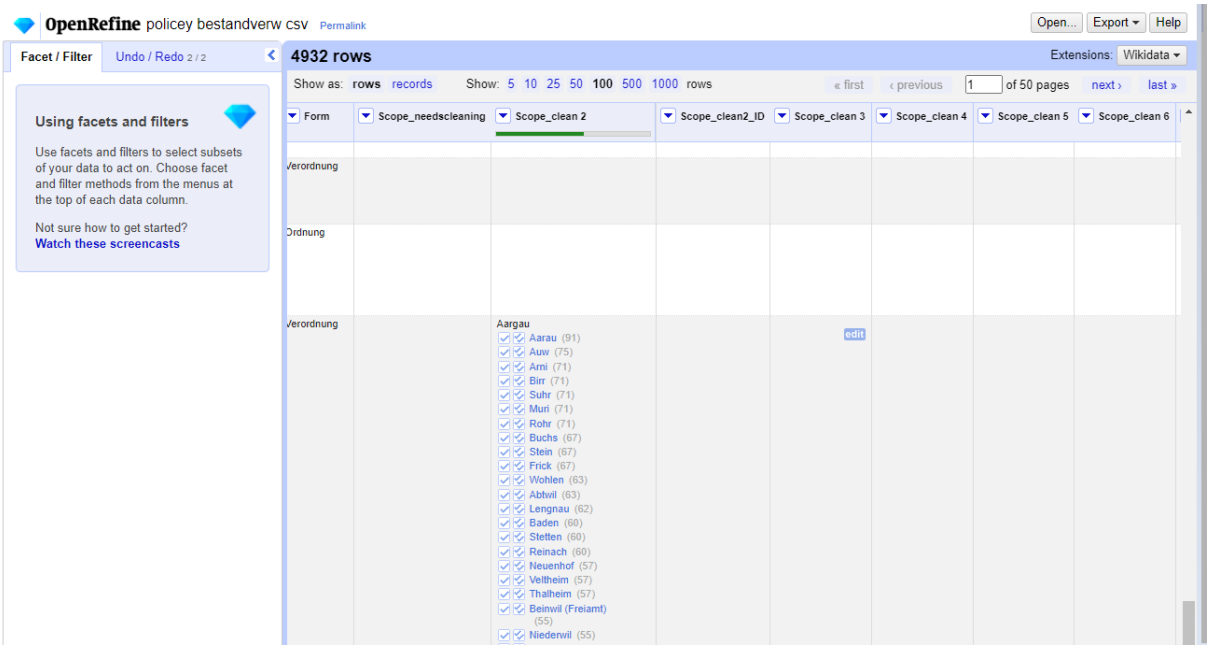


Image 28. Suggested reconciliation candidates.

We can see that places that are not Swiss municipalities have not been matched and also do not appear as candidates. Had we selected “administrative territorial entity” in the reconciliation options above, “Aargau” might have been matched directly. If the table included cities from other countries, “municipality of Switzerland” would have been too restrictive, too; and if it included non-administrative geographical regions like the Black Forest, even “administrative territorial entity” may not be tolerant enough and we should have chosen “Location” or even “Reconcile against no particular type” (N.B. Aargau is a Canton these days, but was a region within Berne back in the period this data comes from).

For the clarity of your database, you may want to consider adding an additional column with the *entity identifiers* (like “Q70” in our example). By going to *Reconcile* in the column’s drop-down menu, you can click on *Add entity identification column*:

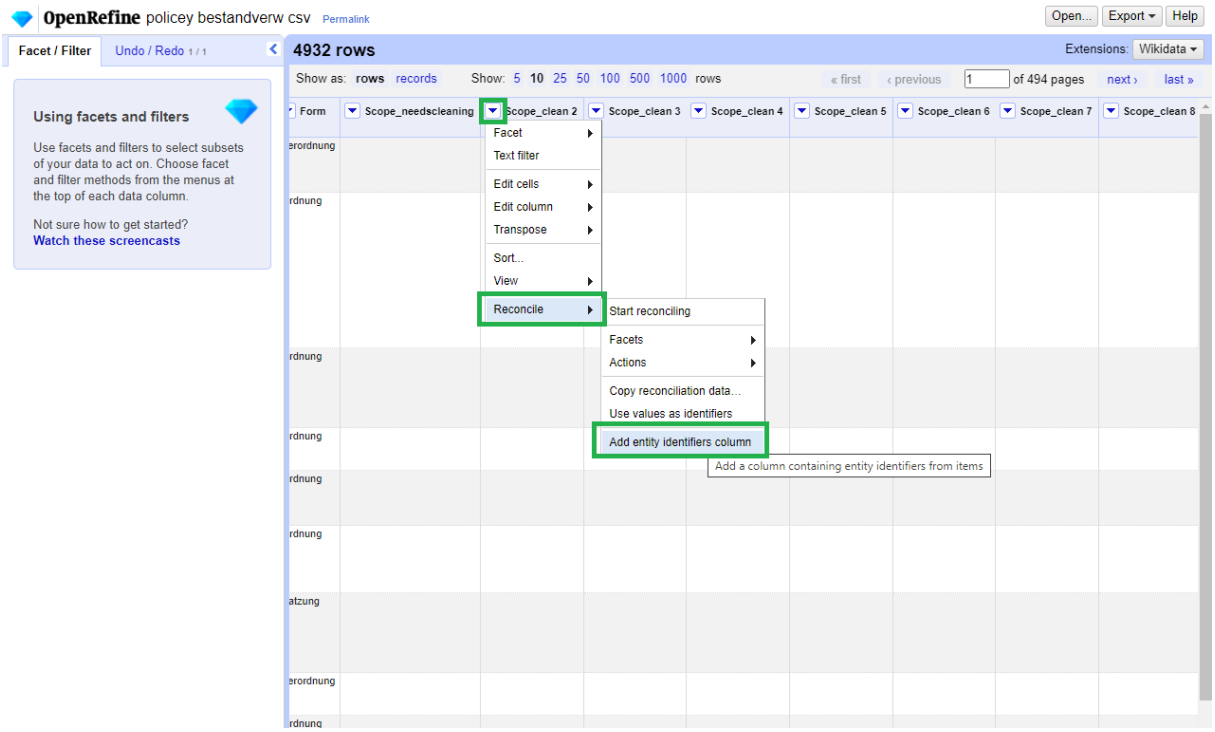


Image 29. Add entity identification column.

You are asked to provide a name for this new column:

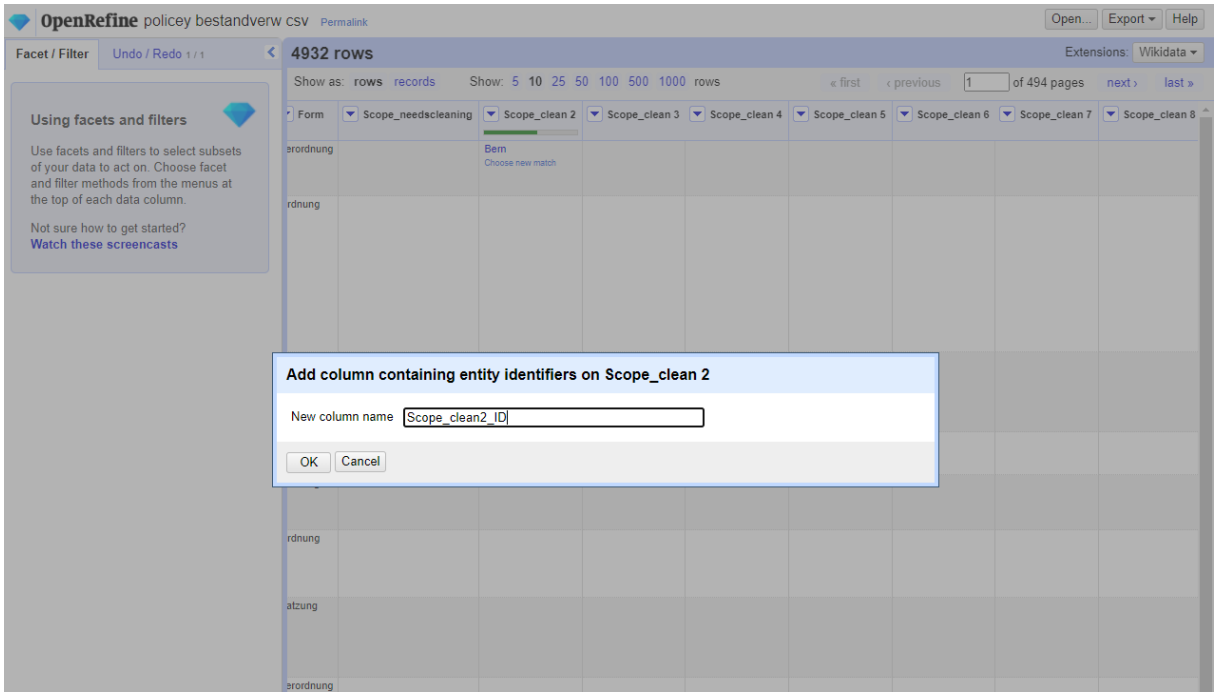


Image 30. Add column containing entity identifiers - New column name.

And consequently the *entity identifications* are shown in the column next to the original. Here you see that Zolfingen has the ID: Q63986 in Wikidata Swiss Municipalities:

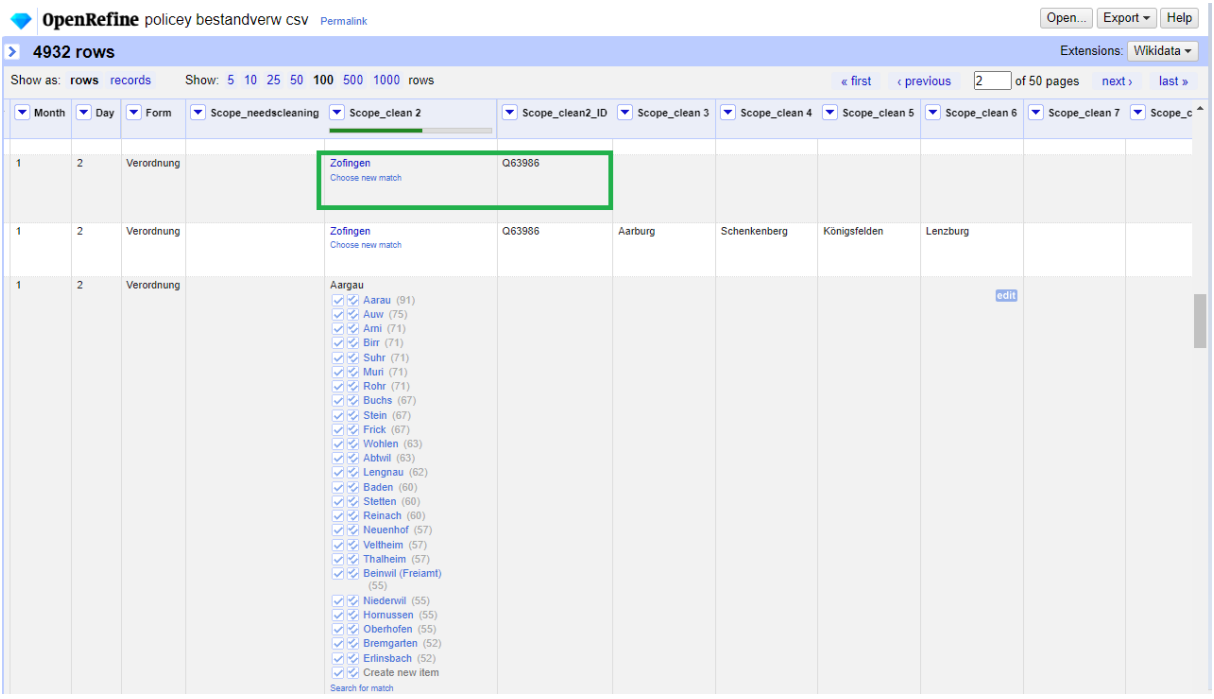


Image 31. Entity identifiers.

You can find more helpful information about data reconciliation with OpenRefine, including more advanced aspects like adding data from authority databases to your tables, or using reconciliation facets, in these resources: Chapter “5 Hands-on: Reconciliation” in John Little’s “Cleaning Data with OpenRefine” Workshop, the Getty Vocabularies OpenRefine Tutorial, and of course the OpenRefine Manual.

Now, while geographical locations can be considered a restricted vocabulary, they are not the classificaton scheme we have built and published earlier. But the matters-columns (the topics) of our early modern ordinances can be matched to our own created classification scheme. Here the same principles are at work, though you cannot chose a standard available service but have to add our custom reconciliation endpoint as a new reconciliation service to OpenRefine. Once we have created the new reconciliation service in OpenRefine, all the other steps described above then apply to our vocabulary in very much the same way. So, we only have to make our vocabulary reconciliation service known to OpenRefine…

We do this by going to the dropdown menu of the column we want to reconcile against our vocabulary (in this case “Matter_1”), select “Reconcile” and then “Start reconciling”.

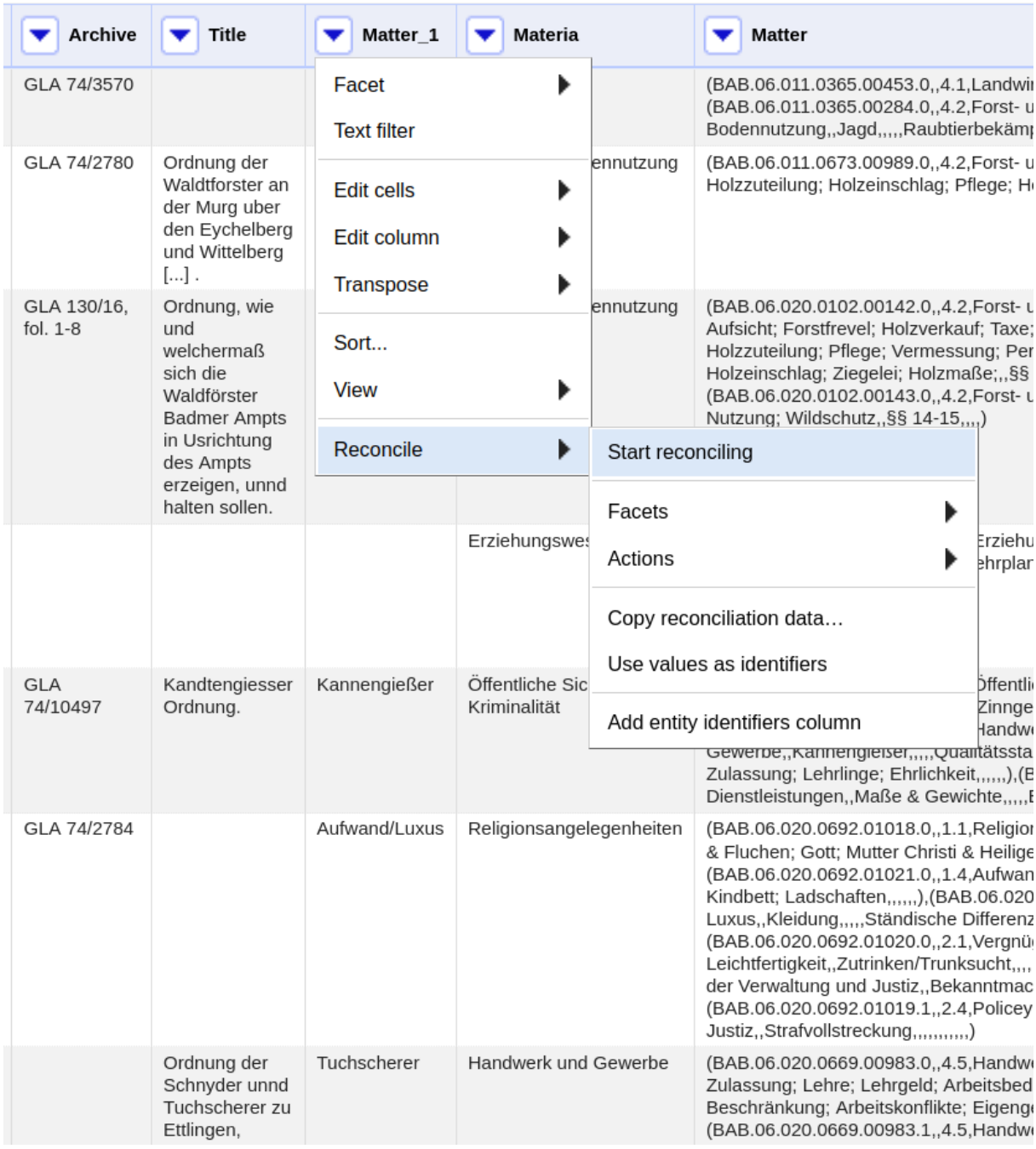


Image 32. Reconciliation against taxonomy.

Again, we are presented the the list of available services:

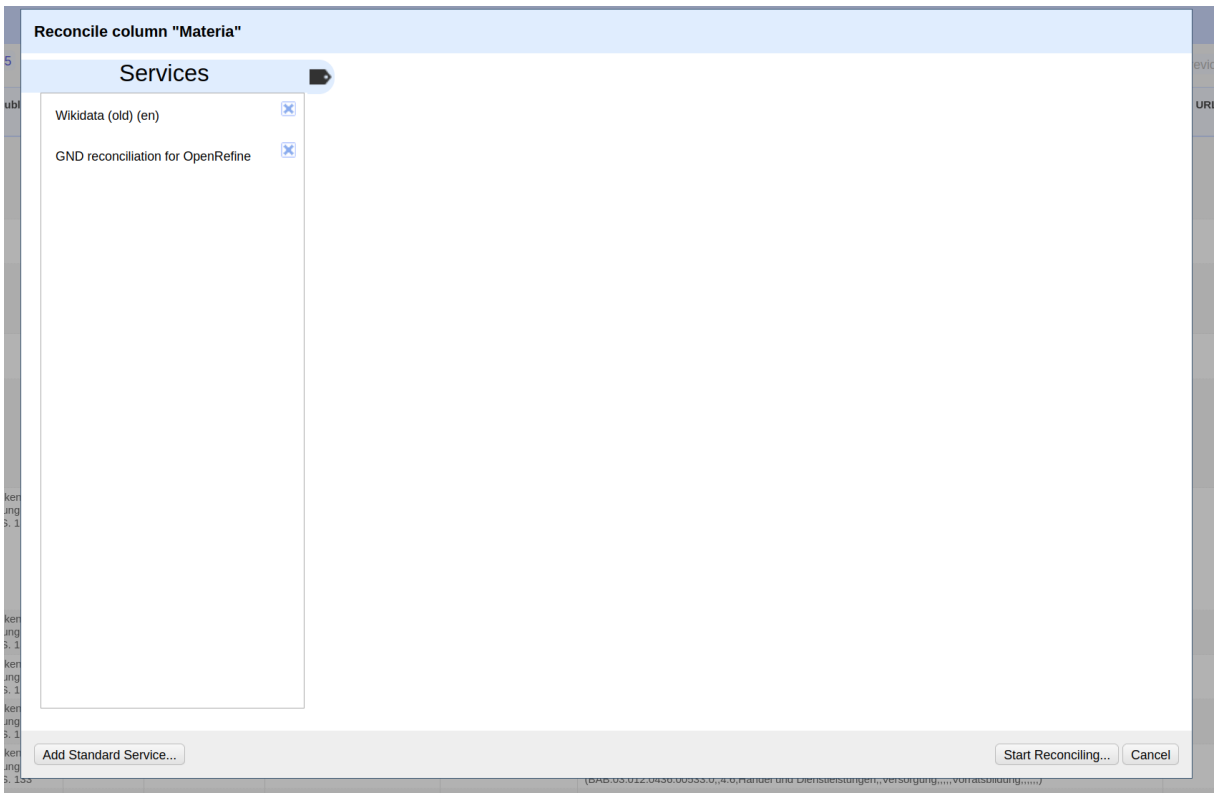


Image 33. Reconciliation services.

Clicking on “Add Standard Service…” in the lower right corner takes us to a dialogue where we can enter the service’s URL. In our case, that would be `https://reconcile.skohub.io/reconcile?account=rhonda&dataset=https://w3id.org/rhonda/polmat/scheme`, or, since our “Matter_1” and all the other entries are in German, even better enter `https://reconcile.skohub.io/reconcile?account=rhonda&dataset=https://w3id.org/rhonda/polmat/scheme&language=de` with a German language tag attached to the URL (`&language=de`).

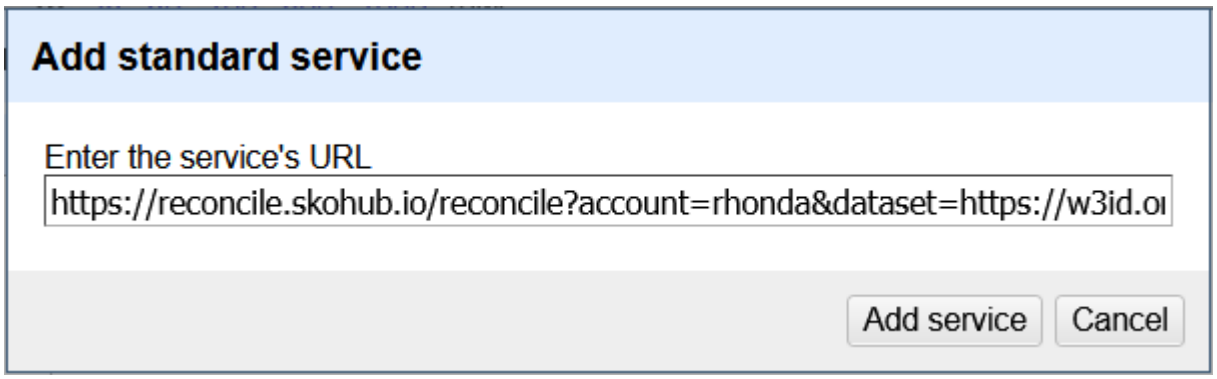


Image 34. Entering own service's URL.

Clicking on "Add service" will have OpenRefine query the service and see what type of entities it can offer for the values in our "Matter_1" column. In this case, the service can offer only "Concept" (and "ConceptScheme") type entries anyway, so this is what we are being offered in the reconciliation options:

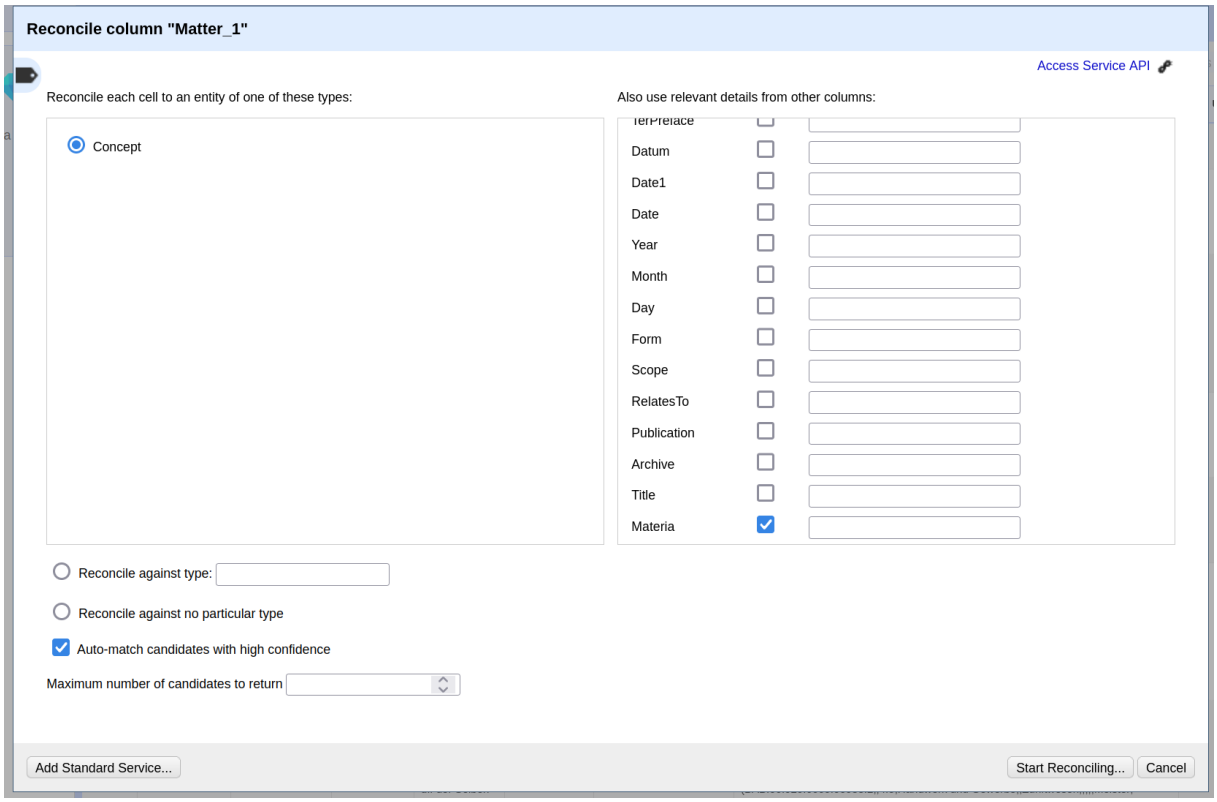
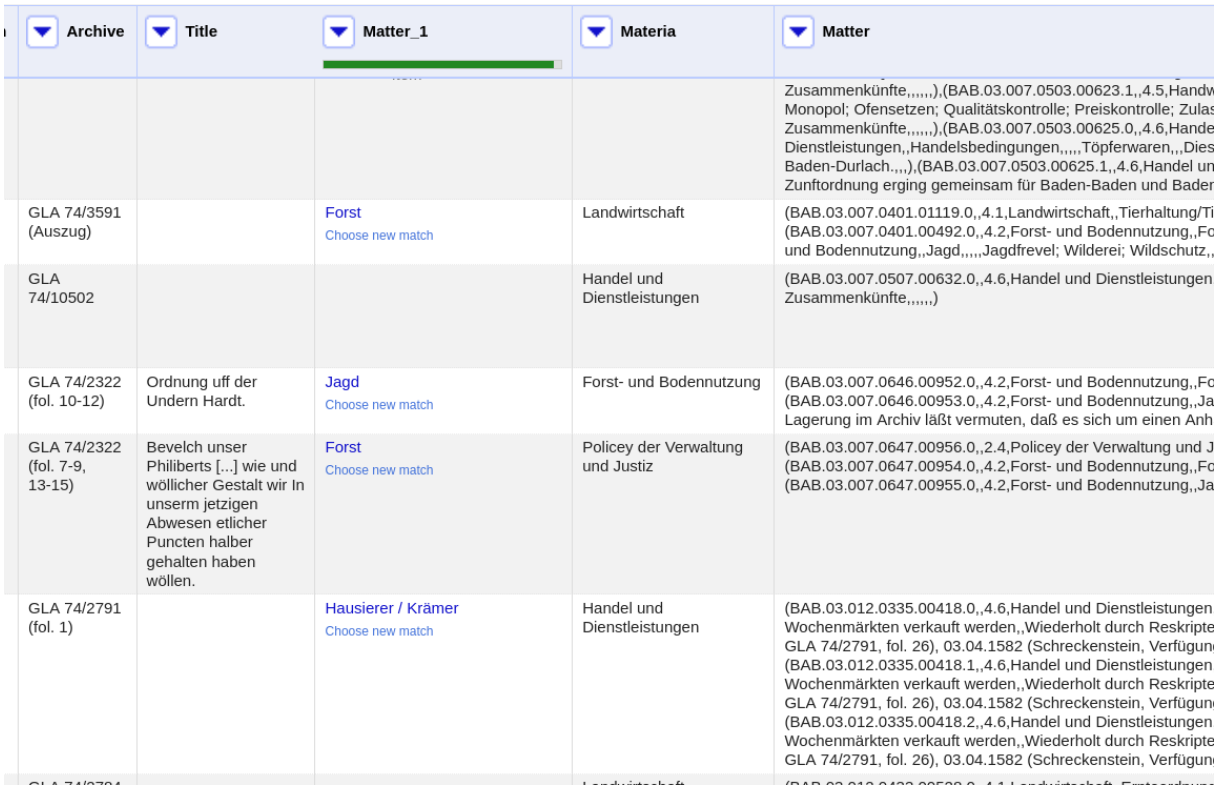


Image 35. Reconcile Concepts.

(In this example, we can specify that another column called “Materia” holds information that may be helpful for deciding ambiguous cases.)

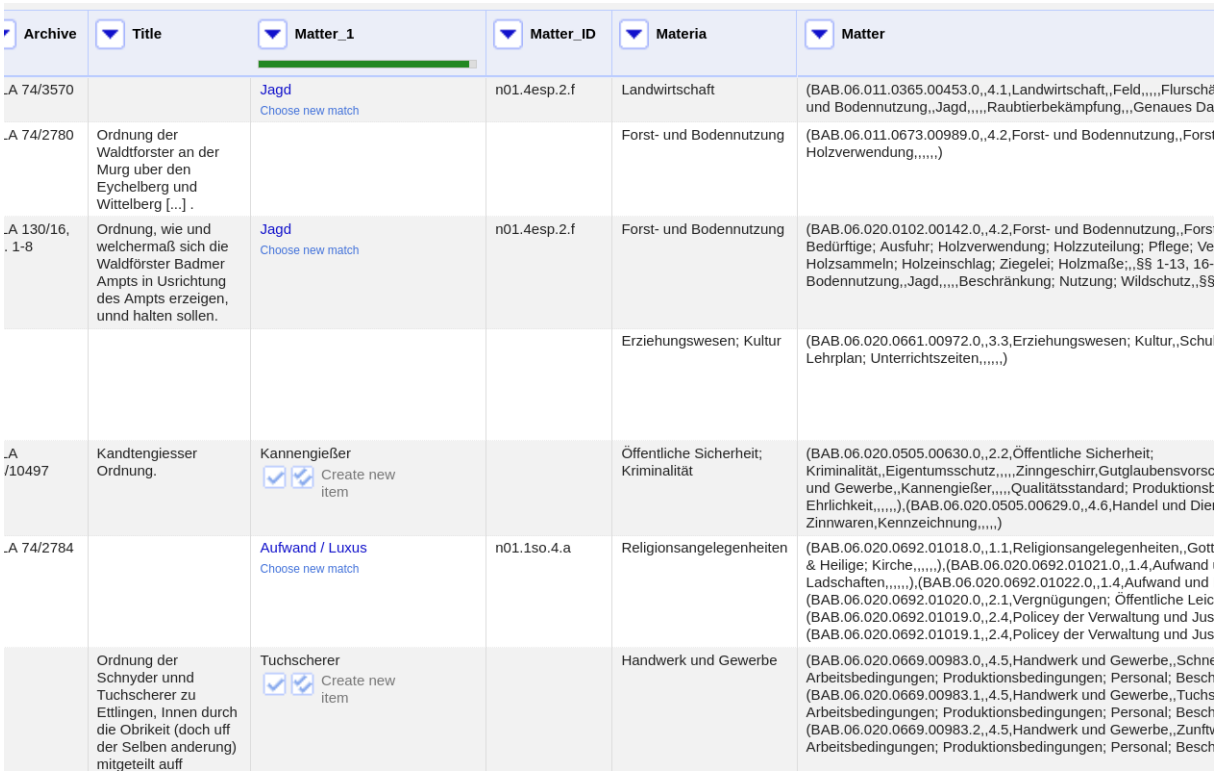
After clicking on “Start Reconciling…” and waiting a bit for the lookups to settle, we end up with most of the entries being successfully mapped and replaced with the preferred label stored in the vocabulary:



Archive	Title	Matter_1	Materia	Matter
				Zusammenkünfte,,,,,,),(BAB.03.007.0503.00623.1,,4.5,Handw Monopol; Ofensetzen; Qualitätskontrolle; Preiskontrolle; Zulas Zusammenkünfte,,,,,,),(BAB.03.007.0503.00625.0,,4.6,Hande Dienstleistungen,,Handelsbedingungen,,,,,Töpferwaren,,,Dies Baden-Durlach.,,,),(BAB.03.007.0503.00625.1,,4.6,Handel un Zunftordnung erging gemeinsam für Baden-Baden und Bader
GLA 74/3591 (Auszug)		Forst Choose new match	Landwirtschaft	(BAB.03.007.0401.01119.0,,4.1,Landwirtschaft,,Tierhaltung/Ti (BAB.03.007.0401.00492.0,,4.2,Forst- und Bodennutzung,,Fo und Bodennutzung,,Jagd,,,,,Jagdfrevel; Wilderei; Wildschutz,,
GLA 74/10502			Handel und Dienstleistungen	(BAB.03.007.0507.00632.0,,4.6,Handel und Dienstleistungen Zusammenkünfte,,,,,,)
GLA 74/2322 (fol. 10-12)	Ordnung uff der Undern Hardt.	Jagd Choose new match	Forst- und Bodennutzung	(BAB.03.007.0646.00952.0,,4.2,Forst- und Bodennutzung,,Fo (BAB.03.007.0646.00953.0,,4.2,Forst- und Bodennutzung,,Ja Lagerung im Archiv läßt vermuten, daß es sich um einen Anh
GLA 74/2322 (fol. 7-9, 13-15)	Bevelch unser Philiberts [...] wie und wöllicher Gestalt wir In unserm jetzigen Abwesen etlicher Puncten halber gehalten haben wöllen.	Forst Choose new match	Policey der Verwaltung und Justiz	(BAB.03.007.0647.00956.0,,2.4,Policey der Verwaltung und J (BAB.03.007.0647.00954.0,,4.2,Forst- und Bodennutzung,,Fo (BAB.03.007.0647.00955.0,,4.2,Forst- und Bodennutzung,,Ja
GLA 74/2791 (fol. 1)		Hausierer / Krämer Choose new match	Handel und Dienstleistungen	(BAB.03.012.0335.00418.0,,4.6,Handel und Dienstleistungen Wochenmärkten verkauft werden,,Wiederholt durch Reskripte GLA 74/2791, fol. 26), 03.04.1582 (Schreckenstein, Verfügun (BAB.03.012.0335.00418.1,,4.6,Handel und Dienstleistungen Wochenmärkten verkauft werden,,Wiederholt durch Reskripte GLA 74/2791, fol. 26), 03.04.1582 (Schreckenstein, Verfügun (BAB.03.012.0335.00418.2,,4.6,Handel und Dienstleistungen Wochenmärkten verkauft werden,,Wiederholt durch Reskripte GLA 74/2791, fol. 26), 03.04.1582 (Schreckenstein, Verfügun

Image 36. Reconciled Matters example.

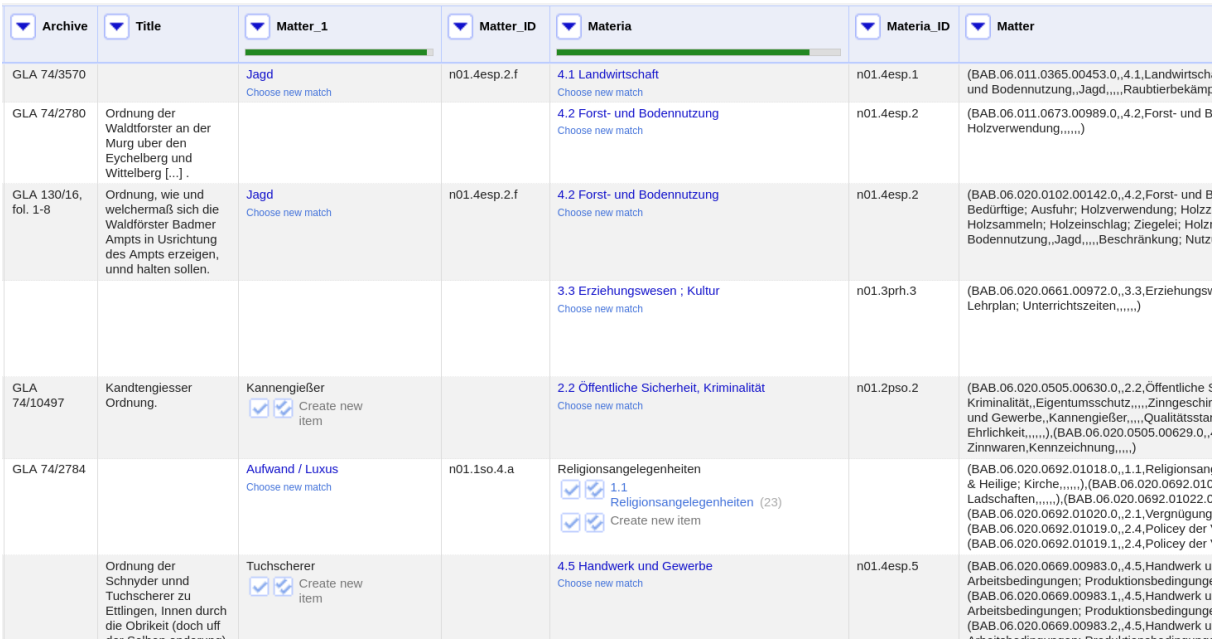
Going to the “Matter_1” dropdown menu and the “Reconciliation” item again, we can also tell OpenRefine to “Add entity identifiers column”. Then it will ask us for a name for this column, and we end up with an additional column holding the concept identifiers that our authority database holds.



Archive	Title	Matter_1	Matter_ID	Materia	Matter
.A 74/3570		Jagd Choose new match	n01.4esp.2.f	Landwirtschaft	(BAB.06.011.0365.00453.0,,4.1,Landwirtschaft,,Feld,,,,,Flurschä und Bodennutzung,,Jagd,,,,,Raubtierbekämpfung,,,Genaues Da
.A 74/2780	Ordnung der Waldtforster an der Murg uber den Eychelberg und Wittelberg [...] .			Forst- und Bodennutzung	(BAB.06.011.0673.00989.0,,4.2,Forst- und Bodennutzung,,Forst Holzverwendung,,,,,,)
.A 130/16, . 1-8	Ordnung, wie und welchermaß sich die Waldförster Badmer Ampts in Usrichtung des Ampts erzeigen, unnd halten sollen.	Jagd Choose new match	n01.4esp.2.f	Forst- und Bodennutzung	(BAB.06.020.0102.00142.0,,4.2,Forst- und Bodennutzung,,Forst Bedürftige; Ausfuhr; Holzverwendung; Holzzuteilung; Pflege; Ve Holzsammeln; Holzeinschlag; Ziegelei; Holzmaße;,,§§ 1-13, 16- Bodennutzung,,Jagd,,,,,Beschränkung; Nutzung; Wildschutz,,§§
				Erziehungswesen; Kultur	(BAB.06.020.0661.00972.0,,3.3,Erziehungswesen; Kultur,,Schu Lehrplan; Unterrichtszeiten,,,,,,)
.A /10497	Kandtengiesser Ordnung.	Kannengießer Create new item		Öffentliche Sicherheit; Kriminalität	(BAB.06.020.0505.00630.0,,2.2,Öffentliche Sicherheit; Kriminalität,,Eigentumsschutz,,,,,Zinngeschirr,Gutglaubensvorsc und Gewerbe,,Kannengießer,,,,,Qualitätsstandard; Produktionsb Ehrlichkeit,,,,,),(BAB.06.020.0505.00629.0,,4.6,Handel und Die Zinnwaren,Kennzeichnung,,,,,)
.A 74/2784		Aufwand / Luxus Choose new match	n01.1so.4.a	Religionsangelegenheiten	(BAB.06.020.0692.01018.0,,1.1,Religionsangelegenheiten,,Gott & Heilige; Kirche,,,,,),(BAB.06.020.0692.01021.0,,1.4,Aufwand Ladschaften,,,,,),(BAB.06.020.0692.01022.0,,1.4,Aufwand und (BAB.06.020.0692.01020.0,,2.1,Vergnügungen; Öffentliche Leic (BAB.06.020.0692.01019.0,,2.4,Policey der Verwaltung und Jus (BAB.06.020.0692.01019.1,,2.4,Policey der Verwaltung und Jus
	Ordnung der Schnyder unnd Tuchscherer zu Ettlingen, Innen durch die Obrikeit (doch uff der Selben anderung) mitgeteilt auff	Tuchscherer Create new item		Handwerk und Gewerbe	(BAB.06.020.0669.00983.0,,4.5,Handwerk und Gewerbe,,Schne Arbeitsbedingungen; Produktionsbedingungen; Personal; Besch (BAB.06.020.0669.00983.1,,4.5,Handwerk und Gewerbe,,Tuchs Arbeitsbedingungen; Produktionsbedingungen; Personal; Besch (BAB.06.020.0669.00983.2,,4.5,Handwerk und Gewerbe,,Zunftw Arbeitsbedingungen; Produktionsbedingungen; Personal; Besch

Image 37. Additional column with identifiers.

Doing the same for the “Materia” column (which contains more general terms), we get our final result, with PIDs available for the matters in our table. If later, we build a proper database out of our table and want to offer multilingual interfaces for it, we could use these IDs to present the terms in the respective current language of the user interface (provided our vocabulary has all the necessary translations).



Archive	Title	Matter_1	Matter_ID	Materia	Materia_ID	Matter
GLA 74/3570		Jagd Choose new match	n01.4esp.2.f	4.1 Landwirtschaft Choose new match	n01.4esp.1	(BAB.06.011.0365.00453.0,,4.1,Landwirtsch und Bodennutzung,,Jagd,,,,,Raubtierbekämp
GLA 74/2780	Ordnung der Waldtforster an der Murg uber den Eychelberg und Wittelberg [...] .			4.2 Forst- und Bodennutzung Choose new match	n01.4esp.2	(BAB.06.011.0673.00989.0,,4.2,Forst- und B Holzverwendung,,,,,,)
GLA 130/16, fol. 1-8	Ordnung, wie und welchermaß sich die Waldförster Badmer Ampts in Usrichtung des Ampts erzeigen, unnd halten sollen.	Jagd Choose new match	n01.4esp.2.f	4.2 Forst- und Bodennutzung Choose new match	n01.4esp.2	(BAB.06.020.0102.00142.0,,4.2,Forst- und B Bedürftige; Ausfuhr; Holzverwendung; Holzz Holzsammeln; Holzeinschlag; Ziegelei; Holzı Bodennutzung,,Jagd,,,,,Beschränkung; Nutz
				3.3 Erziehungswesen ; Kultur Choose new match	n01.3prh.3	(BAB.06.020.0661.00972.0,,3.3,Erziehungsv Lehrplan; Unterrichtszeiten,,,,,,)
GLA 74/10497	Kandtengiesser Ordnung.	Kannengießer Create new item		2.2 Öffentliche Sicherheit, Kriminalität Choose new match	n01.2pso.2	(BAB.06.020.0505.00630.0,,2.2,Öffentliche S Kriminalität,,Eigentumsschutz,,,,,Zinngeschir und Gewerbe,,Kannengießer,,,,,Qualitätsstar Ehrlichkeit,,,,,),(BAB.06.020.0505.00629.0,, Zinnwaren,Kennzeichnung,,,,,)
GLA 74/2784		Aufwand / Luxus Choose new match	n01.1so.4.a	Religionsangelegenheiten 1.1 Religionsangelegenheiten (23) Create new item		(BAB.06.020.0692.01018.0,,1.1,Religionsan & Heilige; Kirche,,,,,,),(BAB.06.020.0692.010 Ladschaften,,,,,,),(BAB.06.020.0692.01022.0 (BAB.06.020.0692.01020.0,,2.1,Vergnügung (BAB.06.020.0692.01019.0,,2.4,Policey der ' (BAB.06.020.0692.01019.1,,2.4,Policey der '
	Ordnung der Schnyder unnd Tuchscherer zu Ettlingen, Innen durch die Obrikeit (doch uff	Tuchscherer Create new item		4.5 Handwerk und Gewerbe Choose new match	n01.4esp.5	(BAB.06.020.0669.00983.0,,4.5,Handwerk u Arbeitsbedingungen; Produktionsbedingunge (BAB.06.020.0669.00983.1,,4.5,Handwerk u Arbeitsbedingungen; Produktionsbedingunge (BAB.06.020.0669.00983.2,,4.5,Handwerk u

Image 38. Reconciling additional columns.

Workflow 2: Annotating full text with TEI Publisher

A basic method in humanities or historical projects is annotating some source text with tags and vocabulary relevant to your research question (cf. Nantke/Schlupkothen 2020). For this, many annotation platforms are available, but we will focus on TEI Publisher as an example because of two points: it allows you to annotate TEI-XML encoded full-text documents without taking you away from this format; and it can get its annotation data from an authority database that offers a Reconciliation API. In other words, you can annotate with all the authority databases we have mentioned, plus your own classification scheme.

There is a public playground available where you can try out the annotation tools, too. However, we need to change some configuration and code files that cannot be manipulated in this playground, so we have to assume you have set up a TEI Publisher server for your project.

The TEI Publisher documentation describes the configuration of annotation using authority lookups quite well, but we briefly resume it here, adding details about our vocabulary service where necessary. We need to:

a. edit the `templates/pages/annotate.html` file so that the annotation page has buttons for accessing our vocabulary service,
b. edit the `modules/annotation-config.xqm` file to explain how our new type of annotation should appear in the TEI XML source files, and

c. edit the ODD file in order to define how the annotations should be rendered in web views (normally, we would do this in the `resources/odd/annotations.odd` file, but if want the rendering instructions to apply to all the views, and not just to the annotation view, then we do this in the `resources/odd/teipublisher.odd` file).

In TEI Publisher's own source code editor, eXide, open the file `templates/pages/annotate.html` and find the passage where it says:

```
<main>
    <aside class="annotation-editor">
        <div class="toolbar">
```

with lots of `<paper-icon-button ... />` entries in it. These are the various buttons for the toolbar, and you should add an entry like the following to where you see fit:

```
<paper-icon-button class="annotation-action authority" data-i18n="[
   title]AZ_Polmat" title="Policey matter" data-type="polmat" icon="
   icons:android" data-shortcut="mac+shift+r,ctrl+shift+p" disabled="
   disabled"/>
```

The **`class`**`="annotation-action authority"` specifies that this component represents an annotation action using an authority database lookup. The `type="polmat"` bit is important too, as it links the component to the configuration instructions that we will discuss in the next step. You can choose a suitable icon from https://kevingleason.me/Polymer-Todo/bower_components/iron-icons/demo/index.html, we chose the android droid for simplicity's sake.

Next, find the passage with

```
<paper-dialog id="authority-dialog">
    <paper-dialog-scrollable>
        <pb-authority-lookup subscribe="transcription" emit="
           transcription">
```

This is where the configuration of authority lookups goes. Add an entry like the following:

```
<pb-authority connector="ReconciliationService" debug="debug" prefix="
   polmat" name="polmat" endpoint="https://reconcile.skohub.io/
   reconcile?account=rhonda&amp;dataset=https://w3id.org/rhonda/polmat/
   scheme&amp;language=de"/>
```

The `connector="ReconciliationService"` bit specifies that the authority database uses the Reconciliation API, the `endpoint` gives the service URL, the `prefix` controls how the ID of the annotated concept will be coded in some TEI attribute, and the `name` bit is what the button component above was referring to. Save the file in eXide.

Next, open the file `modules/annotation-config.xqm` in eXide and find the passage

```
declare function anno:annotations($type as xs:string, $properties as
    map(*), $content as function(*)) {
    switch ($type)
```

Add a new entry like the following:

```
case "polmat" return
    <term xmlns="http://www.tei-c.org/ns/1.0" type="polmat" ref="{
        $properties?ref}">{$content()}</term>
```

This defines how lookup results from the annotation mechanism will be encoded in the TEI files: We see that whatever is selected in the annotation view (the `$content`) is wrapped in a `<term>` element, the `@ref` attribute of which will contain what the lookup has returned as its `ref`. Again, save the file, and if you will, close the eXide editor.

Finally, we need to define how our new `<term type="polmat"/>` elements should be rendered in the display: from one of the pages in your TEI Publisher, click on the "Admin" menu at the top and select the "Edit ODD: teipublisher.odd" entry. This will take you to TEI Publisher's ODD editor GUI. If then there is no "term" entry in the "Element Specs" list on the left, enter `term` below the "Add Element" prompt and click on the plus icon. If there is one, click on the "term" option in the list on the left.

Click on the plus icon in the upper right corner to create a new rendering instruction for "term" elements, select "model" and fill in the following details:

- Output: `web` (as we will create some interactive stuff that will not work in pdf or other formats)
- Predicate: `@type="polmat"` (so that this will apply only to the new term elements identifiable by this type, and not to others)
- Behaviour: `alternate` (this provides a tooltip on hover, according to the documentation)
- CSS class: `annotation annotation-term annotation-polmat` (allows CSS styling to apply to our polmat term elements)
- Click on the plus icon to create the following parameters:
  - "default": `.` (content to display by default, i.e. what text had been selected in annotation)
  - "persistent": `false()` (activates the hover popup. If `true()`, a persistent popup is shown when the element is clicked on)
  - "alternate": `concat("Concept: ", ./@ref)` (this is what is printed inside the popup. In this case, it's the concept ID with the prefix configured above and an explanatory "Concept:" label attached in front.)
- In renditions, just add `text-decoration: underline solid violet 3px;` or whatever other formatting you want to apply to the annotation.

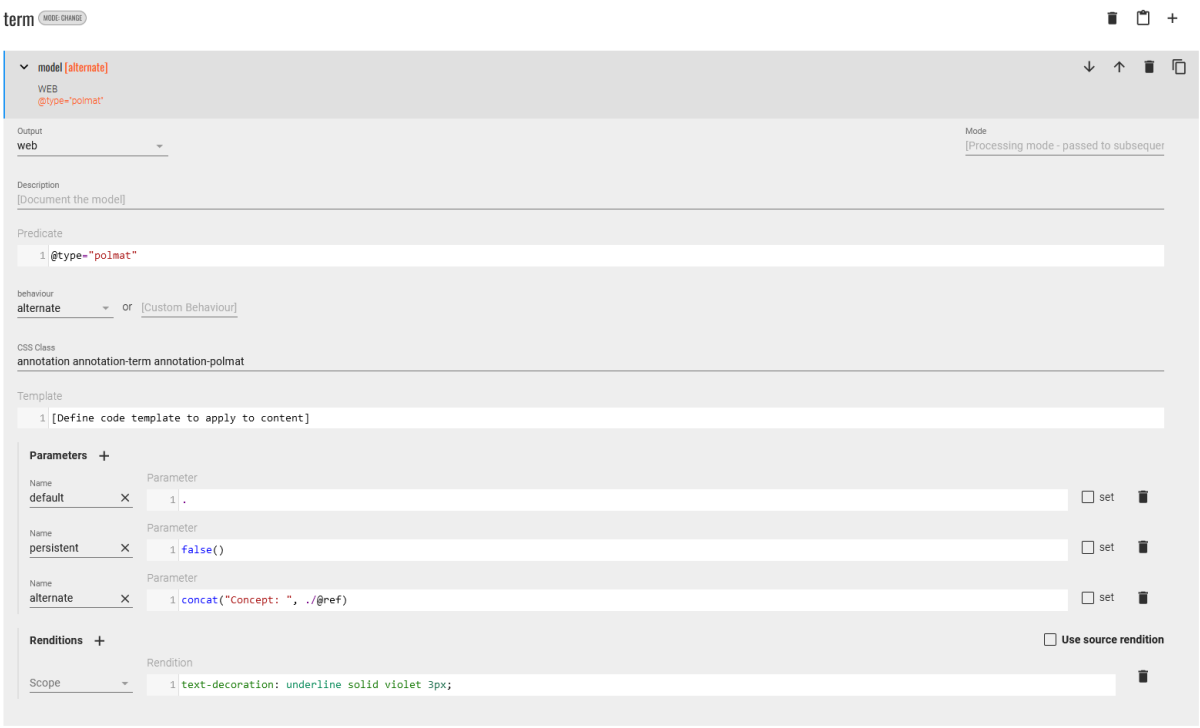


Image 39. ODD instructions for rendering polmat-type terms.

Click on the Save icon near the “teipublisher.odd” title on the upper left and wait until TEI publisher has created the actual rendering functions from your ODD file. Then you can close the ODD editor, too.

Now, when you open a document in TEI Publisher and select the “Annotation Editing” view in the “hamburger”/settings menu on the upper right, you will be taken to the annotation view with your new annotation button waiting in the toolbar. Selecting a term like “Erkrankte” in the main text area, the buttons become available for clicking, but clicking on the android icon takes us to an empty list, indicating that the vocabulary has no entry for “Erkrankte”. However, if we change the text in the popup’s search field to “krank” and click on the search icon, the list is updated with two candidates “Kranke / Krankheiten” and “Krankenwärter”. Clicking on the candidates’ entries themselves takes us to the respective term’s “homepage”, but clicking on the plus icon near a candidate will insert the annotation.

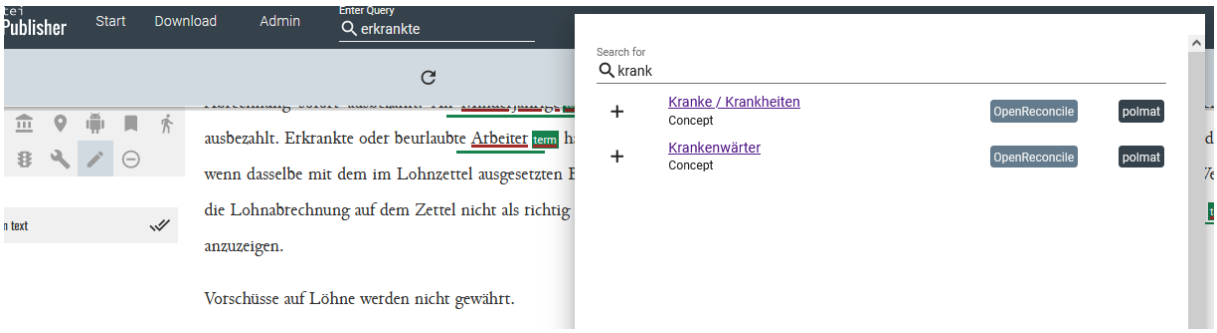


Image 40. Reconciliation results for annotation.

If we click on one of the annotations in the text, a popup shows a preview of the concept (image 41, box 1) so that we can get more information and eventually verify we have the correct entry. Also, on the left we are offered a list of all occurrences of the character sequence we have just selected and annotated, so if we want to tag all of them, we can do so easily (image 41, box 2). Also you can inspect the html preview and the TEI preview on the right (image 41, box 3):

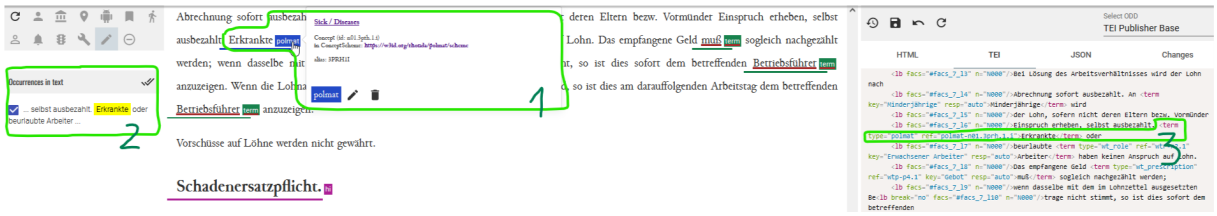


Image 41. View of annotations in web view and TEI.

If you are happy with the annotation, click the save button in the menu on the right and you are done. You can use the categories of your classification scheme in a flexible environment and in an efficient workflow for annotating TEI XML full text documents.

Conclusion

The two example workflows should have given you an idea of how taxonomies can be used in a variety of workflows. This chapter should also have demonstrated in which sense integration of a (normative) conceptual resource such as a taxonomy into a project means more than an intellectual resolution and good ideas about who does what: it also means finding good tools and tuning them to a workflow – in the case of a collaborative project, configuring networked resources and services so that everyone's tool can benefit from and contribute to the intellectual efforts. Again it shows that an authority file

– this is what our taxonomy functions as – means a combination of intellectual resource, social convention and networked infrastructure and foregoing any of these three aspects risks diminishing the success of the endeavour.

VI. Further Reading and Resources

Literature

- Almeida, Bruno/Costa, Rute/Medeiros, Filipa (eds., 2021a): *Controlled vocabularies and knowledge organisation for the digital humanities: proceedings*. Lisbon: NOVA FCSH/CLUNL. DOI:10.34619/pgtp-upne.
- Almeida, Bruno/Freire, Nuno/Salgueiro, Ângela/Monteiro, Daniel (2021b): "The ROSSIO vocabularies: development and publication as linked open data", in: Almeida, Bruno/Costa, Rute/Medeiros, Filipa (eds.): *Controlled vocabularies and knowledge organisation for the digital humanities: proceedings*. Lisbon: NOVA FCSH/CLUNL, 2021. DOI:10.34619/pgtp-upne.
- Autiero, Serena/Elwert, Frederik/Moscatelli, Cristiano/Pons, Jessie (2023): "The Seven Steps: Building the DiGA Thesaurus", in: *Journal of Open Humanities Data* 9. DOI:10.5334/johd.111.
- Durost, Sébastien/Reich, Guillaume/Girard, Jean Pierre (2021): "Vocabulaires de recherche, vocabulaire contrôlé et modèle de données: une chaîne opératoire pour le partage de données archéologiques", in: Almeida, Bruno/Costa, Rute/Medeiros, Filipa (eds.): *Controlled vocabularies and knowledge organisation for the digital humanities: proceedings*. Lisbon: NOVA FCSH/-CLUNL, 2021. DOI:10.34619/pgtp-upne.
- Edmond, Jennifer/Benito Santos, Alejandro/Doran, Michelle/Kozak, Michał/Mazurek, Cezary/Wandl-Vogt, Eveline/Rocha Sepulveda, Aleyda (2023): "Making the Whole Greater than the Sum of its Parts: Taxonomy Development as a Site of Negotiation and Compromise in an Interdisciplinary Software Development Project", in: Gerstorfer, Dominik/Gius, Evelyn/-Jacke, Janina (eds.) *Digital Humanities Quarterly* 17(3) Special Issue on "Categories in Digital Humanities". http://www.digitalhumanities.org/dhq/vol/17/3/000701/000701.html.
- Ernst, Marlene/Gassner, Sebastian/Gerstmeier, Markus/Rehbein, Malte (2023): "Categorising Legal Records – Deductive, Pragmatic, and Computational Strategies", in: Gerstorfer, Dominik/Gius, Evelyn/Jacke, Janina (eds.) *Digital Humanities Quarterly* 17(3) Special Issue on "Categories in Digital Humanities". http://www.digitalhumanities.org/dhq/vol/17/3/000708/000708.html.
- Gerstorfer, Dominik/Gius, Evelyn/Jacke, Janina (2023): "Working on and with Categories for Text Analysis. Challenges and Findings from and for Digital Humanities Practices" (Editorial), in:

Gerstorfer, Dominik/Gius, Evelyn/Jacke, Janina (eds.) *Digital Humanities Quarterly* 17(3) Special Issue on "Categories in Digital Humanities". http://www.digitalhumanities.org/dhq/vol/17/3/000704/000704.html

- Goulis, Helen (2021): "The BBT meta-thesaurus model: building interoperable thesauri for humanities researchers", in: Almeida, Bruno/Costa, Rute/Medeiros, Filipa (eds.): *Controlled vocabularies and knowledge organisation for the digital humanities: proceedings*. Lisbon: NOVA FCSH/CLUNL, 2021. DOI:10.34619/pgtp-upne.
- Guarino, Nicola/Oberle, Daniel/Staab, Steffen (2009): "What is an ontology?", in: Steffen Staab, Rudi Studer (eds.): *Handbook on Ontologies*. Berlin: Springer, 2009: 1-17. DOI:10.1007/978-3-540-92673-3_0 (also: https://iaoa.org/isc2012/docs/Guarino2009_What_is_an_Ontology.pdf).
- Kless, Daniel/Milton, Simon/Kazmierczak, Edmund/Lindenthal, Jutta (2015): "Thesaurus and ontology structure: Formal and pragmatic differences and similarities", in: *Journal of the Association for Information Science and Technology* 66: 1348-1366. DOI:10.1002/asi.23268.
- Nantke, Julia/Schlupkothen, Frederik (eds., 2020): *Annotations in Scholarly Editions and Research. Functions, Differentiation, Systematization*. Berlin: De Gruyter. DOI:10.1515/9783110689112.
- Nijman, Brecht/Pepping, Kay (2023): "Building a VOCabulary: the uses and challenges of thesauri for working with early modern recognized entities". Abstract for a paper presented at DHBenelux 2023. DOI:10.5281/zenodo.7973694. (See other resources from the project at https://globalise.huygens.knaw.nl/output/.)
- SEMIC (2009): "Guidelines and good practices for taxonomies", Report of the Semantic Interoperability Centre Europe, version 1.3. https://publica.fraunhofer.de/handle/publica/294684.
- Zeng, Marcia Lei/Chan, Lois Mai (2004): "Trends and issues in establishing interoperability among knowledge organization systems", in: *Journal of the American Society for Information Science and Technology (JASIST)* 55(5): 377-395. DOI:10.1002/asi.10387.
- Zeng, Marcia Lei (2019): "Interoperability", in: *Knowledge Organization* 46(2): 122-146. Also available in Hjørland, Birger and Gnoli, Claudio (eds.): *ISKO Encyclopedia of Knowledge Organization*, https://www.isko.org/cyclo/interoperability

Software and Platforms

Git

- git source-control management
- GitHub Platform
- GitLab Platform
- BitBucket Platform
- git cheat sheet, including user information set up

Code Editors

- BBEdit
- Notepad++
- SublimeText
- TextMate [MAC only]
- Visual Studio Code
- Free Online Turtle Editor (1): http://onto.fel.cvut.cz/turtle-editor/turtle-editor.html
- Free Online Turtle Editor (2): https://felixlohmeier.github.io/turtle-web-editor/

SKOS

- Host a SKOS resource:
  - SkoHub.io
    - blog
    - github repositories
    - skohub-vocabs repo
    - skohub-docker-vocabs repo
    - skohub-reconcile repo
    - polmat Vocabulary (browse)
    - polmat Vocabulary (source)
    - polmat GitHub Workflow file
  - SSH Vocabulary Commons
- Manage a SKOS resource:
  - Skosmos
  - VocBench
  - iQvoc
  - OpenTheso
  - UniLex
- Test and visualise SKOS thesaurus: Sparna Playground
- clean SKOS thesaurus: Skosify
- Directories of thesauri, classification schemes etc. (search for related ones, consider submitting yours):
  - Basic Register of Thesauri, Ontologies & Classifications (BARTOC)
  - Linked Open Vocabularies

  - Social Sciences & Humanities Open Cloud (SSHOC) Vocabulary Commons
  - Vocabularies hosted at the Austrian Center for Digital Humanities
- BARTOC's list of software for controlled vocabularies
- Search for prefixes: prefix.cc
- Using SKOS:Concept Wiki article: http://ontologydesignpatterns.org/wiki/Community:Using_SKOS_Concept
- The Accidental Taxonomist (Blog)

Persistent Identifiers

- w3id.org
  - w3id github repository
  - help for `git squash`
- purl.org
  - archive.org

Data Cleaning / OpenRefine

- OpenRefine

Annotation / TEI Publisher

- https://newscatcherapi.com/blog/top-6-text-annotation-tools
- TEI Publisher

Acknowledgements

This tutorial is based on a workshop provided to participants of the *2021 Linked Pasts Conference #7* (Ghent). It has been revised for the *Journal for Digital Legal History* - Special Issue *"Gute Policey" and police ordinances: local regimes and digital methods (2023)'.*

Funding: CAR was funded by a postdoctoral fellowship from the Dutch Research Council/Nederlandse Organisatie voor Wetenschappelijk Onderzoek [VI.Veni.191H.035].

Author contributions: *Conceptualisation*: C.A.R., A.W.; *Formal analysis:* C.A.R., A.W., J.J.v.Z.; *Resources:* C.A.R., A.W., J.J.v.Z.; *Methodology:* C.A.R., A.W., J.J.v.Z.; *Writing – original draft:* C.A.R., A.W., J.J.v.Z; *Writing – review and editing:* C.A.R., A.W., J.J.v.Z..

Competing interests: the authors declare no competing interests.

About the Authors

- Annemieke Romein is a Post-Doctoral Researcher at the Huygens Institute *for history and culture of the Netherlands* in Amsterdam (the Netherlands).
- Andreas Wagner is Digital Humanities Officer at the Max Planck Institute for Legal History and Legal Theory in Frankfurt am Main (Germany) and a researcher for the project "The School of Salamanca. A digital collection of sources and a dictionary of its juridical-political language" of the Academy of Sciences and Literature | Mainz (Germany).
- Joris van Zundert is a Senior Researcher and Developer at the Huygens Institute *for history and culture of the Netherlands* in Amsterdam (the Netherlands).