Software Project Sbr-Regesten: Documentation

Susanne Fertmann
s9sufert@stud.uni-saarland.de

Tim Krones
tkrones@coli.uni-saarland.de

Conrad Steffens
conradsteffens@googlemail.com

under supervision of
Prof. Dr. Caroline Sporleder

February 8, 2013

Contents

1 Introduction

This documentation accompanies work that was carried out over the course of nine months by Susanne Fertmann, Tim Krones, and Conrad Steffens for the Software Project "Computerlinguistische Aufarbeitung kulturhistorischer Dokumente", under supervision of Prof. Dr. Caroline Sporleder. The project targets the book "Regesten zur Geschichte der Stadt Saarbrücken (bis 1545)" by Irmtraut Eder-Stein, which is a collection of historical deeds (*Urkunden*) pertaining to the history of the city of Saarbrücken.[1] Throughout this documentation, we will refer to this work simply as *Sbr-Regesten*. When referring to individual deeds, we will use the term *regest*.
The main goal of the project was to come up with a digital representation of the book in order to lay the foundation for making the information contained in the original text easily searchable, as well as for extracting different kinds of implicit knowledge from it.
To achieve this goal, the following (sub-)tasks were identified and completed within the scope of the project:

1. Design a custom *XML schema* for the Sbr-Regesten using the XML Schema Definition Language (XSD).
2. Extract all information contained in the original text and annotate it according to this schema.
3. Create a *database schema* based on the XML schema.
4. Store the information extracted from the Sbr-Regesten in a database created from this schema.
5. As a first step towards making the contents of the Sbr-Regesten available to a wider audience, set up a *web application* on top of the database that allows users with administrative rights to browse, add, modify, and delete database content.

The remainder of this document is structured as follows: Chapter 2 provides an overview of the different components that were implemented for the purpose of extracting, annotating, and storing information contained in the Sbr-Regesten, and how they are chained to form a full *extraction pipeline*. Chapters 3 and 4 provide a detailed account of the XML schema and extraction logic for the "Regesten" and "Index" chapters of the Sbr-Regesten. Chapter 5 explains the XML mark-up for the remaining parts of the book, and how they were extracted from the original text. The last chapter focuses on the web application, providing general information about the Django Web Application Framework, as well detailed guidance for installing, running, using, deploying and extending the application. It also includes information about the data model the web application is based on.
Chapter 4 was written by Susanne Fertmann, and Chapter 6 was written by Tim Krones. Chapter 3, as well as sections 5.3 and 5.7, will be supplied by Conrad Steffens at a later date. The remaining chapters represent a joint effort between Susanne Fertmann and Tim Krones.

[1] Hanns Klein/Irmtraud Eder-Stein, Regesten zur Geschichte der Stadt Saarbrücken (bis 1545), universaar, Saarbrücken 2012.

2 Extraction Process: Overview

The extraction process is based on an HTML version of the Sbr-Regesten. The latter was created using the `.doc` version of Sbr-regesten, obtained from the publishers. Microsoft Word from the 2010 edition of Microsoft Office was used to convert the document to `.html`. Compared to a range of other open-source tools available to us for conversion, its output provided the best structural information for our purpose. The HTML file is stored in `sbr-regesten/html/sbr-regesten.html`.[2]
In order to keep the architecture of the extraction process modular, individual chapters of the Sbr-Regesten were extracted using separate modules. Each of these modules works independently of the other modules, but to get a well-formed, schema-conforming XML document, they must be used in succession.
For ease of use, the extraction process was integrated into the Sbr-Regesten Web Application as a *management command*. This chapter focuses on where to find the extraction modules and how they are chained to form the overall extraction pipeline. For information on (re-)running the extraction process, please consult chapter 6.2.2.
The order of extraction for the individual parts of the Sbr-Regesten follows the order of chapters in the original text. The first module is responsible for inserting the start tag of the *root element* (`<sbr-regesten>`) into the output file, which is called `sbr-regesten.xml` and is located in the top-level directory of the project. The last module to process the HTML version of the Sbr-Regesten inserts the corresponding end tag (`</sbr-regesten>`). Book chapters and the modules responsible for extracting them are listed in the following table:

Chapter	Extraction Module
Frontmatter	`frontmatter_extractor.py`
Table of Contents	`toc_extractor.py`
Preface	`preface_extractor.py`
Bibliography	`bibliography_extractor.py`
List of Abbreviations	`abbrev_extractor.py`
List of Initials	`initials_extractor.py`
Regests	`regest_extractor.py`
List of Archives	`archives_extractor.py`
Index	`index_extractor.py`

Figure 1: Book chapters and corresponding extraction modules

The extraction modules are located in the `sbr-regesten/extraction` directory. As explained above, each one of them uses the HTML version of the Sbr-Regesten as input and implements a function called `extract_<part-of-book>`. This function takes no arguments and serves as an entry point for extracting the chapter it is responsible for. The `extract.py` module located in

`sbr-regesten/regesten_webapp/management/commands`

chains the calls to the entry points in the appropriate order:

```
class Command(NoArgsCommand):
    help = 'Starts and directs the extraction process for the Sbr-Regesten'
```

[2]Due to minor inconsistencies in the HTML, the file was manually adapted in some places, see Appendix A.

```
def handle_noargs(self, **options):
    frontmatter_extractor.extract_frontmatter()
    toc_extractor.extract_toc()
    preface_extractor.extract_preface()
    bibliography_extractor.extract_bibliography()
    abbrev_extractor.extract_abbrevs()
    initials_extractor.extract_initials()
    regest_extractor.extract_regests()
    archives_extractor.extract_archives()
    index_extractor.extract_index()
```

The following chapters describe in detail how individual parts of the Sbr-Regesten are extracted, and also provide information about relevant parts of the XML schema.

3 Extraction Process: Regesten

Author: Conrad Steffens

The "Regesten" part of the Sbr-Regesten contains the regest documents. The XML schema for regests is defined in

`sbr-regesten/regesten-schemas/sbr-regesten.xsd`

together with the schema for the other parts of the book. As described in Chapter 2, the module

`sbr-regesten/extraction/regest_extractor.py`

(which still needs to be implemented) takes care of extracting and annotating regests. It uses the HTML version of the Sbr-Regesten stored in `sbr-regesten.html` as input. The XML output it produces is added to `sbr-regesten.xml`, and a database entry is created for each individual regest. Additionally, a post-processing step for tagging proper names of e.g. persons and locations should be implemented: Once the index has been parsed, regest references in the index can be exploited in order to detect entities mentioned in a given regest, and annotate them using the `<name>` tag.

4 Extraction Process: Index

Author: Susanne Fertmann

Bonifatius IX., Papst (1389-1404) 1392-08-09
Boos von Waldeck, Familie von 1473-03-19
Karl 1495-03-31
Johann (†) 1435-11-22, 1501-08-24 (a)
Boppard, Stadt (Rhein-Hunsrück-Kreis, RLP)
Siehe Beyer von Boppard, Familie
Güter 1493-04-30
Born (Oberbronn, Dep. Bas-Rhin, F), Familie von
Hans, Junker 1436-05-02 (a), 1436-05-02 (b)
Leibeigene der Familie
- *Greden Hans von* Rheinheim 1436-05-02 (b), 1436-05-02 (b)
- *Niclaß Rode, Greden Hansen seligen sone von Rheinheym* 1436-05-02 (b)
Boß, Stephan 1531-03-16

Figure 2: Examples for index entries taken from the Sbr-Regesten

The index is a manually build list of entities that appear in the regests. It is the last part of the Sbr-Regesten and is 277 pages long. Figure 2 shows typical index entries taken from the Sbr-Regesten. The index entries differ from each other, but can roughly be divided into the following five groups: *locations, landmarks, persons, families* and *persongroups*. In total, there are 1035 index entries (see Figure 3 for the distribution of groups).

Type of index entry	Number of entries
Location	550
Landmark	22
Person	103
Persongroup	46
Family	313
Other	1
sum	1035

Figure 3: Distribution of index entries

4.1 Index Entries

An index entry can intuitively be divided into two parts, which we name *header* and *body*. The header gives more details about the entity, e.g. where a location is located, what profession a person has or in which regest to find the entity. The body gives a list of concepts/entities which are in some (sometimes specified) relation to the index entity. For example *Boos von Waldeck* (see Figure 2) would have the header *Boos von Waldeck, Familie von 1473-03-19*. The remaining part of the index entry would be the body.
Irrespective of the group they belong to, index entries have some elements in common. Each index entry starts with its name, which is written in bold. The name might be followed by alternative or additional names. These are usually written in italics (which indicates that they are original text) and in parenthesis right after the name. The header of index entries usually ends with a set

of numbers in the form of dates, which refer to the particular regests, where this entity is named. Right after these regest-references, there might appear references to other index entities. They start with the word *siehe* (*see*).
The body of an index entry is a list of concepts, which are related to the entity. For example *Karl* stands in some relation to the family *Boos von Waldeck*, namely he is a member of it. There exist different layers of such related concepts. If *Karl* had a son for example, he could appear indented below *Karl*. But apparently, a concept on the third level must not obligatorily be in relation with the one the first level. The house where Karl lived could appear below *Karl* as well, although it is not directly related to the family. Often, the authors manually introduced a level between the index entity and the actual body-entities/concepts in order to group the items. For example *Einwohner* or *Güter*, below which there can be found a list of inhabitants/goods which in the regests appeared in relation with the index entity. Each concept might be followed by a set of numbers, which refers the reader to the regest in which the concept was named.
Apart from these commonalities, the headers and bodies are distinct for the different groups. The following sections describe the distinctive parts each of the five groups.

Group	**Possible types**
location types	Dorf, Stadt, Stadtteil, Burgsiedlung, Burg, ehem. Burg, Hofgut, Hof, Ort, Örtlichkeit, Gemeinde, Kloster, Abtei, Schloss, Herrschaft, Herzogtum, Hzgt., Grafschaft, Gft., Fürstentum, Kgr., Deutschordenskommende, Bistum, Vogtei, Regierungssitz, Hochstift, Pfarrei, Erzstift, Erzbistum, Dekanat, Domstift, Reichsland, Deutschordensballei, Wasserburg, Mühle, Zisterzienserabtei, Region, Deutschordenshaus
landmark types	Berg, Gau, Talschaft, Bach, Tal, Landschaft, Au, Waldung, Wald, Gemeindewald
persongroups	Notare, Grafen, Markgrafen, Herzöge, Bischöfe, Edelknechte, Herrn von, Herren, Fürsten, Personen, Könige, Ritter von, Einwohner, Päpste, Wildgrafen, Dominikaner

Figure 4: Types for locations, landmarks and persongroups found in the index

Locations The locations found in the index are geographical or political. They might also be buildings. Among them are towns, castles, parishes, kingdoms or mills. A full list of the types of locations found in the index can be found in Figure 4. In the index entry, the type of location is usually specified right after the name or eventual additional names. It is often followed by district (Gemeinde, Stadt, Stadtverband, Kreis), region (Province, Bundesland or Departement) and/or country. Most of the locations are in Germany. The country is only specified, when the location is not in Germany. Sometimes additional descriptions of where to find the location can be found right after the type name (e.g. *Dorf im Köllertal*, *Hof bei St. Ingbert*). Locations also specify if a location is no longer inhabited. Entries of such abandoned villages (*Wüstungen*) contain references where to find them in Staerk's catalogue[3]).

[3]Dieter Staerk, Die Wüstungen des Saarlandes, Minerva-Verlag, Saarbrücken, 1974.

Behren/Behren-lès-Forbach, Dorf (Dep. Moselle, F)
Einwohner
- Lambrecht, Bruder von Wilhelm 1301-07-08
- Wilhelm, Bruder von Lambrecht 1301-07-08
Zinsgut 1301-07-08
Bellheim, Dorf (Kr. Germersheim, RLP) 1469-06-12
Benningen, Dorf (Wüstung bei Fechingen, Kr. Saarbrücken; Staerk, Wüstungen Nr. 30)
Güter 1369-07-24

Figure 5: Examples for location index entries taken from the Sbr-Regesten

Landmarks Landmarks (*Flurnamen*) are geographical units as forests, mountains, valleys. For the whole list of landmarks found in the index, see Figure 4. Landmarks do not provide any additional unique information.

Rossel (*Russel*), Fluss
Fischerei, 1444-12-03 (a)
Rosselgau, Gau 1046-05-25

Figure 6: Examples for landmark index entries taken from the Sbr-Regesten

Persons Person-headers may contain forenames, surnames, maiden names any generational names (as *Senior, the first, II.*). Additionally they can specify the role or occupation of the person, biographical information such as date of birth or date of the beginning the role and relations to other persons (e.g. *married with, son of*).

Johanna, Tochter von Gf. Simon III. von Saarbrücken 1235-04
Johanna von Loen zu Heinsberg, 1456 oo Graf Johann III. von Nassau-Saarbrücken 1469-09-03
Johannes XXII., Papst (1316-1334) 1325-03-30
Johannes de Salburgo, Dominikaner 1478-02-20

Figure 7: Examples for person index entries taken from the Sbr-Regesten

Families Family-headers have no special elements. But their body can be defined more precisely. Family-bodies do not only represent a list of related entities/concepts, but they are a list of the members of the family. After each of them, the regest where it appeared might be specified. Below each family member (at the next level) there might appear concepts related to that specific member. This might again be a family member but not necessarily.

Persongroups Persongroups are index entries that contain a group of persons. For example, there is a list for popes, for inhabitants of Saarbrücken, bishops etc. The complete list is shown in Figure 4. The group name is specified in the header. Similar to the families, the header does not contain any specific elements, whereas the body is a list of concepts which can be specified more concretely. In the case of persongroups, each concept on the first level of the body is a person and member of the group.

Böckling von Böcklingsau, elsäss. Adelsfamilien
Ludwig *Bocklin von Bockelnauwe* 1518-05-26
Bodman (*Bodenheim*), Familie von
Barbel, oo Friedrich [der Junge] von Fleckenstein, Herr zu Dagstuhl
1489-03-30 (a), 1489-03-30 (b)

Figure 8: Examples for family index entries taken from the Sbr-Regesten

Savoyen, Herzöge
Amadeus 1419-06-01
Karl III. 1546-03-20
Sayn, Grafen von
Gottfried 1270-08-03
Heinrich, Propst des Klosters St. Remigiusberg 1402

Figure 9: Examples for persongroup index entries taken from the Sbr-Regesten

Others Only one index entry is very unusual and does not fit in any of the proposed groups. It is the entry *Saarbrücken, Gliederung*, which shows an overview over the *Saarbrücken* entry (see Figure 10). The *Saarbrücken* entry is the largest index entry with almost 100 pages. It is therefore split up into seven entries. *Saarbrücken, Gliederung* gives an overview over these entries.

Saarbrücken, Gliederung
1. Burg/Schloss
2. Burgsiedlung/Dorf/Stadt
3. Stadt, Einwohner
4. Deutschordenkommende
5. Ritter von
6. Grafschaft
7. Grafen

Figure 10: The index entry *Saarbrücken, Gliederung*

4.2 XML Schema for the Index

One of the main contributions of the Sbr-Regesten project was to create an XML schema for the Sbr-Regesten. The schema can be found in the file

`sbr-regesten/regesten-schemas/sbr-regesten.xsd`

It was as far as it was applicable designed to comply with the TEI Guidelines[4]. This section describes the schema for the index.
The index consists of an `index-info`, which contains general information about the index, followed by an unrestricted number of `items`. Each `item` represents an index entry and is either a location, a person, a landmark, a family or a persongroup. The schema covers each of the groups respectively. Each index item requires the following attributes:

- `id`: a unique id (e.g. item_381)

[4]TEI P5: Guidelines for Electronic Text Encoding and Interchange by the TEI Consortium, Version 2.0.2, 2012

- `type`: location, landmark, person, family or persongroup
- `value`: the name of the item (printed in bold in the printed Sbr-Regesten)

An index `item` consists of an `item-header` and does usually also have an `item-body`. Both are abstract elements, which are substituted by the respective header and body types. This decision was made in order to have only one type of items with consecutive IDs, although the headers and bodies highly vary depending on their type.
Irrespective its type, a header usually ends with the element `mentioned-in`, which might only be followed by `index-refs`. `mentioned-in` is of type `mentionings` and contains a sequence of references to the regests (`reg-ref`). The attribute `regest` specifies the id from the regest it refers to. `index-refs` contains a sequence of references to other index entries (`index-ref`). The attribute `itemid` contains the id from the index item it refers to.
The next sections will describe in more detail the elements that items have depending on their type. They will also provide a XML example for each header and body type.

4.2.1 Headers

Location-Header Figure 12 and 13 show examples for location headers. Location headers contain the element `placeName`, which corresponds mostly to TEI's `placeName`. Figure 11 shows an example for a comparison of the two. The Sbr-Regesten schema adopts TEI's `settlement`, `district`, `region` and `country`. It introduces two new attributes for the `settlement` element, aside from `type`. It gets the additional attributes `abandoned-village` and `av-ref` to encode information about abandoned villages (*Wüstungen*). `abandoned-village` is an obligatory attribute, which denotes whether the settlement is an abandoned village. If that is the case, the optional attribute `av-ref` encodes a reference to the catalogue of Staerk[5]. Region types may be `Bundesland` (if located in Germany), `Departement` (France) or `Province` (Belgium). `country` has no attributes. The TEI guidelines define a `district` as "any kind of subdivision of a settlement, such as a parish, ward, or other administrative or geographic unit."[6] In contrast, in our index-locations a `district` is any administrative unit below a `region` (*Kreis, Gemeinde, Stadtverband*).

```
<placeName>
    <settlement type="city">Rochester</settlement>,
    <region type="state">New York</region>
</placeName>

<placeName>
    <settlement type="Stadt" abandoned-village="false">Saarbrücken</settlement>,
    <region type="Bundesland">SL</region>
</placeName>
```

Figure 11: `placeName` in TEI vs. `placeName` in the Sbr-Regesten schema

Additional to `placeName`, the index schema for locations introduces two elements: `addNames` and `reference-point`. `addNames` contains a list of additional names of the location, each of them tagged

[5]Dieter Staerk, Die Wüstungen des Saarlandes, Minerva-Verlag, Saarbrücken, 1974.
[6]TEI P5: Guidelines for Electronic Text Encoding and Interchange by the TEI Consortium, Version 2.0.2, 2012

as addName. reference-point includes further information that is not covered by district, region or country. It specifies closer where to find the location, e.g. *Im Köllertal, bei St. Ingbert* (*in the Köllertal, near St. Ingbert*). All elements in placeName are optional (even settlement, as placeName also occurs in family headers, where locations are given, without their names being explicitly specified.)

```
<location-header>
   <placeName>
      <settlement type="Dorf" abandoned-village="false">Frauenberg</settlement>
      <addNames> (
         <addName>Frauwenberg</addName>)
      </addNames>, Dorf (
      <region type="Departement">Dep. Moselle</region>,
      <country>F</country>)
   </placeName>
   <mentioned-in>
      <reg-ref regest="regest_235">1452-12-26</reg-ref>,
      <reg-ref regest="regest_984">1459-02-19</reg-ref>
   </mentioned-in> siehe
   <index-refs>
      <index-ref itemid="item_534">Lenterdingen</index-ref>,
      <index-ref itemid="item_956">Volkersweiler</index-ref>
   </index-refs>
</location-header>
```

Figure 12: Sample location-header XML

```
<location-header>
   <placeName>
      <settlement type="Dorf" abandoned-village="true" av-ref="Staerk, Wüstungen Nr. 19">
         Arschofen</settlement>
      <ddNames> (
         <addName>Arßhoffen</addName>)
      </addNames>, Dorf
      <reference-point>im Köllertal</reference-point> (Wüstung,
      <district>Gde. Gersweiler, Stadtverband Sb.</district>,
      <region type="Bundesland">SL</region>; Staerk, Wüstungen Nr. 19)
   </placeName>
</location-header>
```

Figure 13: Sample location-header XML of an abandoned village *Wüstung*

Landmark-Header Figure 14 shows an example for a landmark-header. Landmark-headers consist of the elements geogName, mentioned-in and index-refs. The element geogName captures the same concept as TEI's geogName. But it is differently realized. Namely, it contains exactly the two elements name and addNames (additional names).

```
<landmark-header>
   <geogName type="Fluss">
      <name>Rossel</name>
         <addNames>(
         <addName>Russel</addName>)
      /addNames>
   </geogName>, Fluss
</landmark-header>
```

Figure 14: Sample for `landmark-header`

Person-Header Figure 15 shows an example for a `person-header`. Person headers provide the element `person`. Person consists of `persName` and `description`. `persName` is very close to TEI's `persName`. It contains (optional) `forename`, `surname`, `genName` (generational name, such as *Senior, the first, II.*, see TEI Guidelines) and `roleName` (name of the role or official position of the person, such as *Fürst, Herzog, Papst*, see TEI Guidelines.). The only addition to TEI in `persName` is `maidenname` (maiden name). Also here we do not have a single additional name (`addName`), but the element `addNames`, which contains a list of `addName` elements. It is also important to know that a `person` has not only an item id (as the other headers do), but also an id that refers to the person itself. This is due to the fact, that the `person` element is also used in the `listing-body`. A person therefore has two attributes: `id` and `itemid`. The other element of `person`, `description`, provides additional information about the person, such as occupation, relation to other persons (*married to, son of,* etc.). It is of type `content`, which means, that it can contain `quote` elements, if there is original text in the `description`.

```
<person-header>
   <person id="person_412">
      <persName>
         <forename>Johann</forename>
         <genName>I.</genName>,
         <roleName>Graf</roleName>
      </persName>
     <description> von Saarbrücken-Commercy (1307-1341),
                   oo Mathilde von Apremont    </description>
   </person>
   <mentioned-in>
      <reg-ref regest="regest_543">1310</reg-ref>,
      <reg-ref regest="regest_1353">1310-10-21</reg-ref>,
   </mentioned-in>
</person-header>
```

Figure 15: Sample for `person-header`

Family-Header Figure 16 displays an example for a `family-header`. Family headers consist of the elements `family-name`, `location`, `mentioned-in` and `index-refs`. `family-name` consists of

name and addNames (additional names). location is of type placeName, which is the same type that location-headers have. See Section 4.2.1 *location-header* for more details.

```
<family-header>
  <family-name>
     <name>Brandscheid</name>
     <addNames>
        <addName>Brandscheidt<addName>
     </addNames>
  </family-name> (
  <location>
     <district>Kr. Bitburg-Prüm</district>,
     <region type="Bundesland">RLP</region>
  </location>), Familie von
</family-header>
```

Figure 16: Sample for family-header

Persongroup-Header Figure 17 shows an example for a persongroup-header. Persongroup headers consist of the elements group-name, mentioned-in and index-refs. group-name contains the name of the persongroup.

```
<persongroup-header>
   <group-name>Notare </group-name>
</persongroup-header>
```

Figure 17: Sample for persongroup-header

4.2.2 Bodies

There are two different types of bodies: concept-body and listing-body.

Concept-body A concept-body contains the element related-concepts. related-concepts, in turn, consists of an unrestricted number of concepts and persons[7]. A concept provides a name, a description and mentioned-in. name contains the name of the concept and description any additional information provided. Additionally, a concept can recursively contain an element related-concepts. For the description of a person, see Section 4.2.1 *Person-Header*.

[7]Note, that currently only concepts are extracted.

```
<concept-body>
   <related-concepts>
      <concept>
         <name> Einwohner </name>
         <related-concepts> -
            <concept>
               <name> Hans gen. Phennewert </name>
               <mentioned-in>
                  <reg-ref regest="regest_64">1455-11-24</reg-ref>
               </mentioned-in>
            </concept> -
            <concept>
               <name> Philipps Pfenwert von Hermanßhusen</name>
               <description>, Bürger zu Saarbrücken, oo Ursulla </description>
               <mentioned-in>
                  <reg-ref regest="regest_756">1510-11-09</reg-ref>,
                  <reg-ref regest="regest_834">1521-12-24</reg-ref>
               </mentioned-in>
            </concept>
         </related-concepts>
      </concept>
      <concept>
         <name> Güter </name>
         <mentioned-in>
            <reg-ref regest="regest_43">1296-12-29</reg-ref>
         </mentioned-in>
      </concept>
   </related-concepts>
</concept-body>
```

Figure 18: Sample for `concept-body`

Listing-body Figure 19 illustrates an example for a `listing-body`. A listing-body consists of the element `members`. `members` consists of a sequence of `persons`, which have the type as the persons appearing in a `person-header` (see Section sec:person-header *Person-Header*). Each `person` can have a `related-concepts` element, which recursively can contain related concepts again, as in a concept-body. That means, that in a `persongroup` or `family`, the first layer displays members of that family or group. The concepts on deeper levels might also be family or group members but not obligatory. Therefore they are not modeled as members on their own, but as related-concepts of that member.

```
<listing-body>
  <members>
     <person id="person_89">
        <persName>
           <forename> Dietrich </forename>gen.
           <addNames>
              <addName> Gebürghin </addName>
           </addNames>
        </persName>
        <mentioned-in>
           <reg-ref regest="regest_823">1460-12-01</reg-ref>
        </mentioned-in>
        <related-concepts> -
           <concept>
              <name> Knecht Peter </name>
              <mentioned-in>
                 <reg-ref regest="regest_823">1460-12-01</reg-ref>
              </mentioned-in>
           </concept>
        </related-concepts>
     </person>
     <person id="person_90">
        <persName>
           <forename> Jakob von Brandscheid</forename>
        </persName>
        <description>, Amtmann zu Saargemünd </description>
        <mentioned-in>
           <reg-ref regest="regest_934">1519-03-23</reg-ref>
        </mentioned-in>
     </person>
  </members>
</listing-body>
```

Figure 19: Sample for `listing-body`

4.3 Index Extraction

In order to convert the Sbr-Regesten index into an XML representation, relevant pieces of information have to be extracted from the HTML file and tagged according to the schema. Additionally, the index entries are written into the database `sbr-regesten.db`. This chapter will describe in more detail how the XML for the index is extracted. Generally, the index suffers many inconsistencies due to its manual construction. That complicates the index extraction.
Figure 20 shows the basic architecture of the module (`index_extractor.py`), which is responsible for the index extraction. Note that it does not show any classes, but is aimed to illustrate the logical structure/work-flow of `index_extractor.py` to demonstrate its functionality. In short, the `index_extractor` module finds the index in the HTML and extracts single index items. These are post-processed before yielding the final XML for the index entries and writing them form the XML

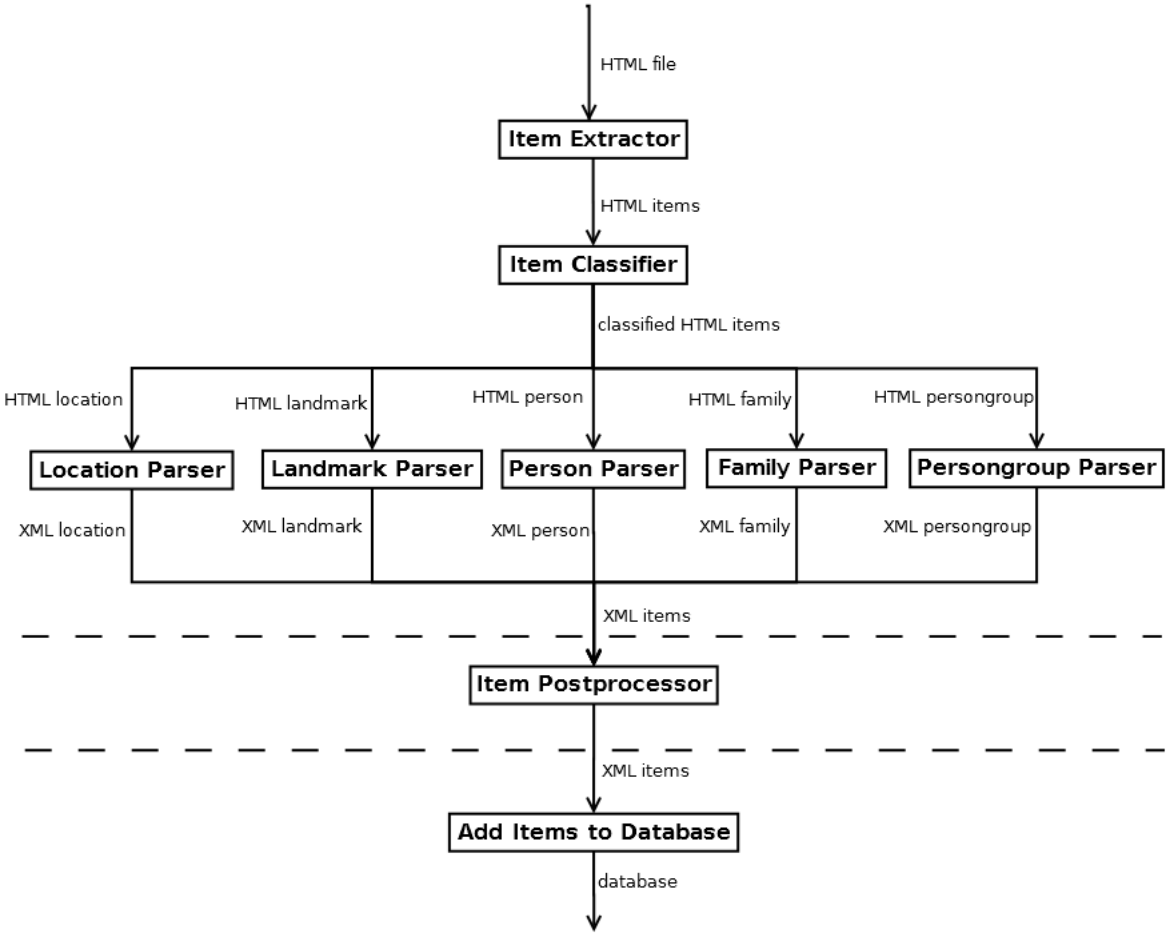

Figure 20: Simplified architecture of `index_extractor.py`

file into the database.
The `index_extractor.py` module is divided into three modules, all of which are stored in the directory `sbr-regesten/extraction/index_utils`:

- `index_to_xml.py` extracts the index from the HTML file and converts it into an almost complete XML
- `index_xml_postprocess.py` post-processes the XML solving the references to other index entries
- `index_to_db.py` extracts the index items from the XML and writes them into the database

The index extraction is implemented in two separate stages which convert the index into XML and store it in the database, respectively. This has two reasons: it structures the module and it allows a manual correction of the extracted XML before the items are added to the database.

4.3.1 Extracting, Classifying and Parsing Index Items

The module `index_to_xml.py` uses the python package *BeautifulSoup* to process the HTML file and convert it into its XML representation. One problem that arose was that due to the maximum recursion limit the very deep structured index entry *Saarbrücken, Burgsiedlung/Dorf/Stadt* could not be processed. Therefore the recursion limit had to be increased[8].
Figure 21 shows the call graph for the main function of `index_to_xml.py`. It is pruned excluding the preprocessing steps and the solving of underspecified index items (see Section 4.3.1 *ItemClassifier*).

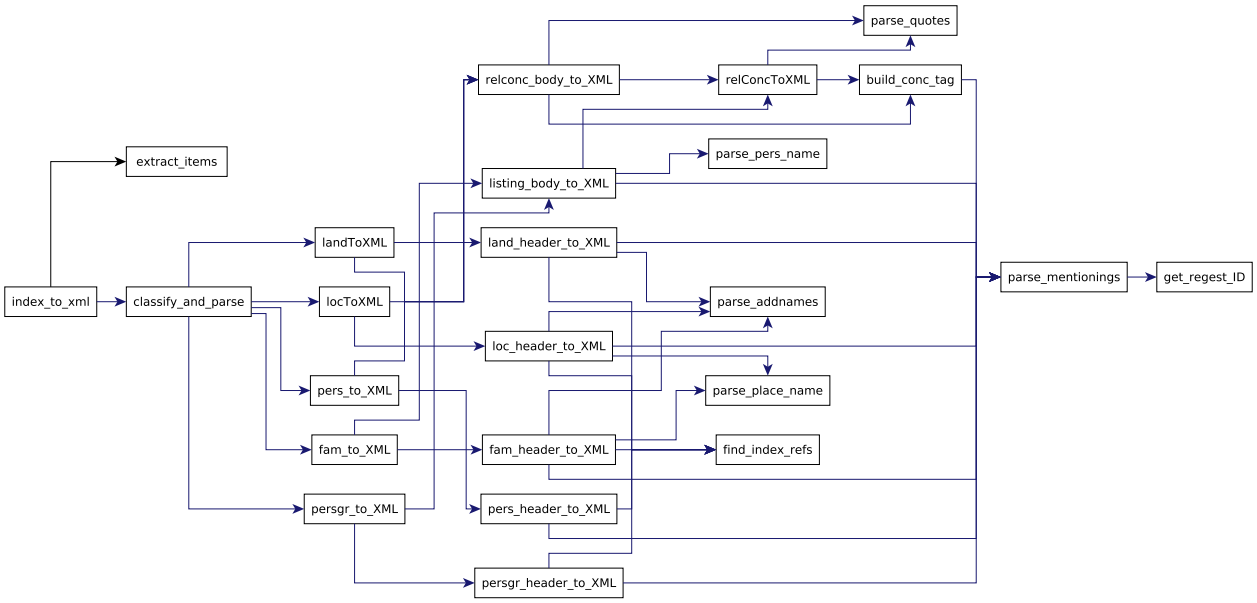

Figure 21: Pruned call graph of `index_to_xml.py`

.

ItemExtractor The ItemExtractor (implemented by the function `extract_items`) reads the HTML source and converts it into a `BeautifulSoup` instance. The detection of the individual index items is very straightforward, as the HTML provides tags for paragraphs (`<p>`). Each index item is a paragraph. In order to find the beginning of the index, the paragraph starting with the keyword *Index* is searched. Once it is found, the ItemExtractor tags the next non-empty paragraph as `<index-info>`. Each of the following non-empty paragraphs is recognized as index `item`. It obtains an id and is preprocessed. The preprocessing step deletes tags that are irrelevant from the HTML and overcomes some irregularities. An instance of `Index Item` is created, which divides the HTML paragraph into a header and body. The ItemExtractor finishes processing the paragraphs when reading more than ten empty paragraphs in a row. In that case the index section is assumed to be finished. The end of the index can not be identified directly via a keyword as for the beginning of the index and as in the extraction of the other parts of the books because the index is the last part of the book, only followed by the footnotes (in the HTML file). Finally, all extracted item objects are stored in a list and are in the next step given to the ItemClassifier.

[8]This seems to be a common problem when using BeautifulSoup. See this blog entry on maximum-recursion-limit-in-python.

ItemClassifier The ItemClassifier is implemented by the function `classify_and_parse`. It classifies the items and forwards them to their corresponding parser functions for processing. The ItemClassifier obtains the list of index items and searches the header of each item for a set of keys and regular expressions. These were retrieved by a manual bootstrapping process and are displayed in Figure 22 [9]. According to the keys, the ItemClassifier decides for each item whether it belongs to the group *location, landmark, person, family, persongroup* or *siehe*. It forwards the item to the corresponding parser (`loc_to_XML, land_to_XML, pers_to_XML, fam_to_XML` or `persgr_to_XML` respectively). The items stored in the *siehe* list contain no direct information about the group they belong to. But they contain references to other index entries. Assuming that referring and referenced entry have the same type, they are forwarded to the corresponding parser after having solved the reference. Items that do not contain any of the keywords are stored in the list *unclassified* and are not further processed. This is the case for the entry *Saarbrücken, Gliederung* as this is a rather unusual entry (see Section 4.1).

ItemParser When an item is classified it is passed to its corresponding ItemParser. There is an ItemParser for each of the five groups. It forwards the header to the specific header parser of its group and the body to one of the two body parsers. It then adds the parsed item to an itemList. The HeaderParsers return also the name of the index entry, which is taken as `value` attribute of `item`. For example, the LocationParser passes the header to the Location-Header-Parser, which returns an XML `location-header` and the name of the item. This name is taken as `value` attribute. The id obtained by the ItemExtractor is taken as `id` attribute and the `type` (in this case `location`) is added. The body is passed to the Concept-Body-Parser. The LandmarkParser, PersonParser, FamilyParser and PersongroupParser proceed analogously. The following two sections describe the parsers for the headers and the bodies.

4.3.2 Header Parsers

This section describes the five parser functions for the different item headers: *Location-Header-Parser, Landmark-Header-Parser, Person-Header-Parser, Family-Header-Parser* and *Persongroup-Header-Parser*. Functionalities that are used by several parsers are presented at the end of this section: references to the index, references to the regest and parsing of additional names.

Location-Header-Parser The parser for the location headers (`loc_header_to_XML`) builds a `<location-header>`, which contains a `<placeName>`, which again includes a `<settlement>`. The latter gets the attributes `abandoned-village` and `av-ref`. The parser searches the header for the keyword *W.stung* in order to decide whether the location is an abandoned village and assigns `abandoned village` the corresponding value (`true/false`). The regular expression

```
Staerk, W.stungen Nr. [0-9]{1,2}
```

is used to extract the reference to Staerk's catalogue[10] `av-ref`. The name of the location is extracted from the HTML via the `<b>` tag and added to the `<settlement>` tag. The additional names are parsed by the AddNamesParser (see Section 4.3.2 *Parsing of Additional Names*) exploiting existing parenthesis and `<i>` tags. From the rest of the HTML location-header, all HTML tags are deleted before the rest is processed. Then references to the regests and to other index entries are parsed

[9] Note, that the order of the keys and the matches are relevant, as some are more reliable than others.
[10] Dieter Staerk, Die Wüstungen des Saarlandes, Minerva-Verlag, Saarbrücken, 1974.

```
famMatch = re.search('[Ff]amilie|Adelsgeschlecht', header)

locMatch = re.search('[Ss]tadt,|Stadtteil|Dorf|Burg |Hof |Hofgut|'\
                     'Gemeinde |Ort |.rtlichkeit |Kloster|Schloss|'\
                     'Herrschaft|Gft\.|Kgr\.|Region|Gebiet|Abtei|'\
                     'Land |Kgr\.|Herzogtum|Hzgt\.|[Gg]rafschaft|'\
                     'F.rstentum|Deutschordenskommende|RLP|Gde\.|'\
                     'Bistum|Vogtei|Regierungssitz|Hochstift|'\
                     'Pfarrei|W.stung|F\)|Erzstift|, Erzbistum|'\
                     'Dekanat|Domstift|Reichsland|Deutschordensballei'\
                     '|M.hle|Wallfahrt|Land |Reise|lothr. Amt|'\
                     'Deutschordenshaus|[Ss]tadt (?!S)', header)

grpMatch = re.search('Notare|, Grafen|, Markgrafen|[Hh]erz.ge|'\
                     '[Bb]isch.fe|Edelknechte|Herrn von|[Ff].rsten|'\
                     'Personen|K.nige|Ritter von|Einwohner|P.pste|'\
                     'Wildgrafen|Herren|(?<!, )Dominikaner', header)

persMatch = re.search('Bischof|Pastor|Graf |Papst |II\.|I\.|III\.|'\
                      'IV\.|V\.|Hzg\.|Bf\.|Adliger|Herr |Frau |Kg\.|'\
                      'Elekt|meister|Ritter|, Schulthei.|, Herzogin|'\
                      'Amtmann|Lehensmann|Vetter von|Markgraf |'\
                      'Pfalzgraf|Ebf\.|, Herzog|, Dominikaner|Hans|'\
                      'Erzpriester|[dD]iakon|Provinzial|r.m\. K.nig|'\
                      'Kammermajor|Witwe|Junker|Stephan|Jacob|Klaus|'\
                      'Elisabeth|Fabricio|Nikolaus|Alheim|Gerbod', \
                      header)

landMatch = re.search('Fluss|Berg|gau[ ,]|Gau|Bach|Tal|Landschaft|'\
                      'Wald|Waldung|Gemeindewald|Au|furt|Engenberg', \
                      header)
```

Figure 22: Regular expressions used to classify the index items

by the respective parsing functions (see Section 4.3.2 and 4.3.2). The remaining string is searched for the type of the settlement (for the whole list see Figure 4). If one of the keywords is found it is taken as value for the attribute `type` of `settlement`. The rest of the string is parsed by the function `parse_place_name` (see Section 4.3.2). Afterwards, the placeNameTag is added to the location-header, as well as the references to the regests and the references to the index. The header is now parsed.

Landmark-Header-Parser The function `land_to_XML` implements the parser for landmark headers. It builds a `BeautifulSoup` instance `<landmark-header>` and adds a `<geogName>` tag to it. The landmark's name is extracted via the `<b>` tags in the HTML and added to a `<name>` tag, which is appended to `geogName`. The remaining HTML is passed to the AddNameParser in order to extract additional names, possibly appearing in the beginning. The remaining part is flattened to a string, whose end is searched for index-refs and mentionings. The remaining middle part is thereafter searched for a set of keys for the type of landmark. See Figure 4. If none of the keys is found in the rest string, the name of the landmark is searched. This is because the type is often not explicitly specified if it is already contained in the name, e.g. *Dirminger Tal, Ensdorfer Au*.

Person-Header-Parser The function `pers_header_to_XML` parses person headers. It builds a `<person-header>` `BeautifulSoup` tag and adds a `<person>` tag to it. The latter obtains a continuous id with the pattern `person_[0-9]+`. A tag `<persName>` is added to the `person`. The `name` is extracted exploiting the `<b>` tags of the HTML. Thereafter, the HTML tags are aborted and index-refs and mentionings are extracted from the end of the remaining string (see Section 4.3.2 and 4.3.2). The remaining middle part is then used to extract `forename, surname, genName and roleName`. For that purpose, a set of keywords is exploited (e.g. *der Dritte, der Junge, Jr* for `genName` and *Kaiser, Herzogin, Graf* for `roleName`). A list of forenames is provided by the file `sbr-regesten/resources/forenames.txt` which is imported at the beginning of `index_to_xml.py` and augmented during the extraction by the Listing-Body-Parser. A set of regular expressions describes possible combinations of these different names. The first matching rule is chosen and the item header tagged accordingly. The remaining part of the string is tagged as `<description>`. At the end the extracted index-refs and mentionings are added to the `<person-header>` tag.

Family-Header-Parser The function `fam_header_to_XML` parses headers of family items. A `<family-header>` tag is created and a tag `<family-name>` is appended to it. The latter contains the name of the family extracted from the HTML via the `<b>` tags. Afterwards, the `<b>` tag is deleted and the rest of the HTML is searched for additional names. The rest of the HTML that does not contain any additional names is converted into a tag-less string. If this contains an opening parenthesis before the next word, the content of these parenthesis are detected to be a location, which is parsed by the placeName Parser (see Section 4.3.2), which also parses the corresponding part from the location headers. The part following the parenthesis (if there were such, elsewise, the whole remaining string) is searched for index-refs and mentionings.

Persongroup-Header-Parser The Parser for persongroup-headers is the least complex header-Parser, due to the simplicity of the actual headers and the XML schema. A `<persongroup-header>` tag is created. The HTML-tags are deleted and index-refs and mentionings are found at the end

of the string. The remaining part is tagged as `<group-name>`. `group-name, index-refs` and `mentionings` are added to `persongroup-header`.

References to the Index References to the index can be found at the very end of an index entry (regardless its type). They start with the keywords *siehe* or *siehe auch*, e.g. *siehe Lenterdingen, Volkersweiler*. The function `find_index_refs` therefore recognizes strings starting with one of these keyword and ending with the end of the string as index-refs. As the references can not be solved at the time of extracting the index (references can refer to future index entries), they are not further processed, but only tagged with `<index-refs>`. The further division into the individual index references and their solving does not take place until the post-processing step (see Section 4.3.4).

References to the Regests The references to the regests are given by the date of the regest, e.g. *1377-03-08 (nach), 1065-04-03, 1277-07-02 (b)*. The are not exactly the same as the titles of the regests. They usually contain only the date plus the information that is crucial for disambiguation. But this does not seem to be very consistent. They may occur at the end of `related-concepts`, and at the end of each type of index-header, only possibly followed by references to the index (`index-refs`). The function `parse_mentionings` recognizes the references to the regests in a given string (by using a regular expression). It builds a `<mentioned-in>` tag. It starts form the end of the string and tries to find one reference after another. It tags it as `<reg-ref>` and adds as attribute `regest` the id of the regest it refers to. This part is carried out by the function `get_regest_ID`. Once the precedent part was not recognized as reference, `parse_mentionings` finishes.

Parsing of Additional Names (`parse_addnames`) `parse_addnames` obtains a flat HTML element and extracts additional names. Additional names are written in italics and in parenthesis. Therefore the parser looks for `<i>` tags surrounded by parenthesis. The string inside contains the additional names, which are separated by commas. Each is tagged with `<addName>` and added to a `<addNames>` tag.

4.3.3 Body Parsers

This section describes the two parser functions for index bodies *Concept-Body-Parser* and *Listing-Body-Parser*.

Concept-Body-Parser The function `conc_body_to_XML` implements the Concept-Body-Parser. It splits the HTML item into lines, making use of the `<br>` tags. It created a `<concept-body>` tag, which contains `<related-concepts>` tag. Each non-indented HTML line is a `concept` and added to `<related-concepts>`. It is split into `name`, `description` and `mentionings`. `mentionings` are extracted according to paragraph 4.3.2 *References to the regests*. The remaining string is split at the first comma. The first part of the string will be the `name` of the concept and the second its `description`. The indented lines below a concept are added to a `related-concepts`. `related-concepts` again is split into concepts and processes recursively. `<name>`, `<description>`, `<mentionings>` and `<related-concepts>` are added to the `<concept>` tag.

Listing-Body-Parser The function `listing_body_to_XML` implements the parser for listing bodies. It builds a `<listing-body>` tag, which again contains a `<members>` tag. The Listing-Body-Parser splits the item HTML in lines exploiting the `<br>` tags. Each non-indented line is a person.

A <person> tag is created which contains a <persName> tag. In almost all cases, the string of the person starts with its forename (especially in family-headers, where the surname is already given in the header). The Listing-Body-Parser therefore tags the first word in the person as <forename> and adds it to the <persName> tag. These forenames are additionally used to extend the forenames list (which is provided by sbr-regesten/resources/forenames.txt and used to find forenames in person headers). From the remaining string additional names (addNames) are parsed and added to <persName>. <mentionings> are extracted as described in Section 4.3.2 *References to regests*). The remaining part is tagged as <description>. The indented lines below a person-line are related-concepts of that person. The latter are parsed recursively using the same process as described in the paragraph above(4.3.3). <persName>, <description>, <mentionings> and <related-concepts> are added to the <person> tag.

4.3.4 Post-Processing

ItemPostprocessor The post-processing step solves the references to other in index entries. This is only possible when the entire index is already parsed, as it searches all index items. The post-processing step is carried out by the module index_xml_postprocess.py. It takes the temporary file index.xml as input, which was produced by the module index_to_xml.py and deletes it afterwards. It builds a BeautifulSoup instance out of the index XML, which is split into single index items. It detects the yet unsolved <reg-refs> tags and splits the contained string into the single references (using comma as separator). The references are solved by matching the string to the value of each index item. If the reference can be solved, a <reg-ref> tag is added to <reg-refs>. The id of the referred index item is stored in the attribute regest. Finally, the parsed index is added to sbr-regesten/sbr-regesten.xml.

4.3.5 Writing the Index into the Database

index_to_db.py takes the sbr-regesten.xml as input and processes the index items. It creates an entry in the database, extracts the tags from the item and adds the information to the fields defined in the model (see Section 6.5).

4.3.6 Future Work

This section presents possible next steps for extending the XML schema for the index, extraction process and possible further uses. Depending on future demands, different extensions will have priority. Section 6.4.1 describes how to obtain a copy of the source code. After modifying the Sbr-Regesten schema or the extraction process, an XML Schema *Validator*, as the CoreFiling XML Schema Validator, can (and should) be used to validate the XML output against the schema.

References to the Regests At the time of developing the index extraction, the regest part of the book was not yet extracted or written into the database. Once this is the case, the first expansion of the current code would be to solve the references to the regests. The references are already recognized and tagged with reg-ref. Each <reg-ref> is supposed to store the id of the regest it refers to in the attribute regest. This is not yet the case. The function get_regest_ID always returns the dummy id regest_99999. We would recommend to re-implement the function get_regest_ID in

sbr-regesten/extraction/index_utils/index_to_xml.py

The references can be solved by searching the database, as the regests are added to the database before `index_to_xml.py` is called. To include the references also in the database, the function `ment_to_db` in `sbr-regesten/extraction/index_utils/index_to_db.py` has to be implemented.

Extracting Person Names The user can provide a list of forenames, which is used to extract the names of `person` items. The forenames have to be stored in the file

`sbr-regesten/resources/forenames.txt`

During the extraction of the index it uses this list to recognize forenames and expands it at the same time. Newly amended names are only considered from the point on they were added. This could be overcome by proceeding similar to the solving of references to other index items (see Section 4.3.2). In the extraction step persons would be recognized and parsed as such. At the same time the list of forenames would be augmented. In the post-processing step all extracted forenames could be used to parse the names. Additionally, a similar process could be conceivable for surnames, role names and generative names.

Quotes The HTML source provides information about which parts of the text are original text. Such sections are marked with italics (`<i>`). To mark original text the Sbr-Regesten schema introduces the element `quote`. This is mainly used in the regests. In the index, quotes are extracted from the description part of the bodies. The tag `<addNames>` implicitly encodes this information as they extract text in italics. Otherwise this information gets lost during the process of extraction. But we can imagine the usefulness of conserving information about original text also in other parts of the index, we propose to expand the schema to allow quotes.

Expand References to the Index References to other index entries are only found and solved if they appear in the header right after the mentionings. But such references can also appear in the bodies or in other parts of the headers, e.g. in location-headers before the district. The schema could be adapted to allow `reg-refs` there as well. They would have to be tagged by `index_to_xml.py` and would then (automatically) be extracted by the existing the `parse_siehe` function in `index_xml_postprocess`.

Recognize Persons in the Bodies Index item bodies often contain many persons which could be extracted using the `person` element defined in the Sbr-Regesten schema. The most prominent case, of course, are the bodies of families and persongroups. The current extraction process extracts persons at the outermost level of listing-bodies. But persons do also appear on deeper levels of listing-bodies or in concept bodies. These persons are currently parsed as `concepts`. The schema allows such extension, as it allows `persons` in the `related-concept` element.

```
<xs:complexType name="related-concepts" mixed="true">
  <xs:choice maxOccurs="unbounded">
    <xs:element name="concept" type="concept" />
    <xs:element name="person" type="person" />
  </xs:choice>
</xs:complexType>
```

Semantic Network The regests and the index do not only provide information about entities, but also about the relations among them. This becomes evident especially in the case of persons. The index provides information such as *married with* or *son of*. This explicit information could be extracted to build networks of relations between persons or other entities. Also, we can imagine to extract such relations directly from the regests. The relations could build the basis for semantic queries and easy visualization of entity relations.

5 Extraction Process: Other Parts of the Book

This chapter describes the extraction process for the remaining parts of the Sbr-Regesten. In terms of the complexity needed to represent and extract them, they are a lot less involved than the "Regesten" and "Index" parts.

5.1 Front Matter

The *front matter* section of the Sbr-Regesten contains the full title of the book, some information about the authors/editors and the cover art, as well as some acknowledgements about financial support.

XML Schema Aside from a `<frontmatter>` tag for wrapping the front matter section, the current version of the XML schema does not introduce any additional markup for representing the front matter of the Sbr-Regesten.

Extraction As contents of the front matter are interspersed with conditional comments in the HTML version of the Sbr-Regesten, the extraction process relies on textual cues that were manually determined to locate relevant lines in the file. When the `frontmatter_extractor.py` module is loaded, it first reads the contents of `html/sbr-regesten.html` and turns it into a `BeautifulSoup` instance. It then collects all relevant lines in a list. As a last step, the `extract_frontmatter()` function takes care of inserting the start tag of the root element (`<sbr-regesten>`) into the XML output file and appends the contents of the front matter.

5.2 Table of Contents (TOC)

The *Table of Contents* lists individual chapters of the Sbr-Regesten.

XML Schema Aside from a `<toc>` tag for wrapping the Table of Contents, the current version of the XML schema does not introduce any additional markup to represent individual entries.

Extraction In the HTML source, the *Inhaltsverzeichnis* key word indicates the start of the table of contents, while the title of the next chapter (*Vorwort*) indicates the end of the TOC section. These cues are used by the `toc_extractor.py` module to determine which lines in the HTML source are relevant for the table of contents. Aside from using different cues, the code for extracting the table of contents operates in exactly the same way as the extractor responsible for the front matter section: It reads the HTML source and turns it into a `BeautifulSoup` instance, collects all relevant lines from the textual content of the HTML in a list, and then writes these lines to the XML output file, wrapping it in the `<toc>` tag.

5.3 Preface

Author: Conrad Steffens

5.4 Bibliography

The *bibliography* section of the Sbr-Regesten lists all references that are quoted multiple times throughout the text. Entries in the bibliography section look like this:

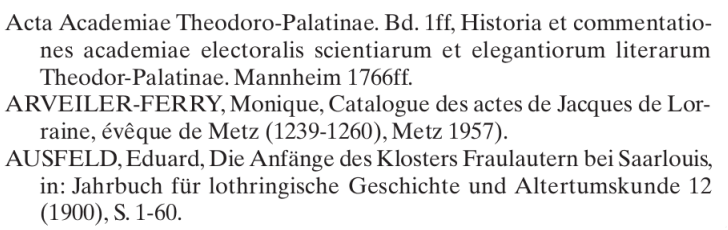
Acta Academiae Theodoro-Palatinae. Bd. 1ff, Historia et commentationes academiae electoralis scientiarum et elegantiorum literarum Theodor-Palatinae. Mannheim 1766ff.
ARVEILER-FERRY, Monique, Catalogue des actes de Jacques de Lorraine, évêque de Metz (1239-1260), Metz 1957).
AUSFELD, Eduard, Die Anfänge des Klosters Fraulautern bei Saarlouis, in: Jahrbuch für lothringische Geschichte und Altertumskunde 12 (1900), S. 1-60.

Figure 23: Example entries from the bibliography section of the Sbr-Regesten

XML Schema The parts of the XML schema for representing the bibliography section of the Sbr-Regesten are a bit more involved than those for representing the front matter and TOC sections. As a whole, the section is wrapped using the `<listBibl>` tag as suggested by the TEI guidelines. The title of the bibliography section (*Literaturverzeichnis*) is taken over from the original text as is; the XML schema does not define a separate element for it. After the title, the bibliography section contains a small amount of free text. In the XML output, this text is wrapped with a tag called `<listBibl-info>`. Lastly, individual entries of the bibliography are annotated using the `<bibl>` tag whose name was borrowed from the TEI guidelines and which defines a single attribute called `biblidtype` for storing a unique ID for each entry in the bibliography.

Extraction The bibliography is extracted by the module `bibliography_extractor.py`. The module reads the entire HTML source and turns it into a `BeautifulSoup` instance. It then proceeds to split the document up into paragraphs exploiting the `<p>` tag in the HTML source. Then, the `bibliography_extractor.py` creates a `<listBibl>` tag. In order to find the paragraphs relevant to the bibliography it searches for the keyword *Literaturverzeichnis*. Once this is found, it starts extracting the bibliography. The first paragraph following the keyword *Literaturverzeichnis* is tagged as `<listBibl-info>` and appended to `<listBibl>`. Each of the following non-empty paragraphs is tagged as `<bibl>` and also added to `<listBibl>`. A continuous ID is assigned to each `<bibl>` and stored as attribute. The `bibliography_extractor.py` stops when finding the keyword *Abk*, which is the beginning of the next section of the book. The created `BeautifulSoup` instance is prettified and written into a temporary file. The latter is post-processed for the purpose of indentation, before the bibliography is finally added to the `sbr-regesten.xml`.

5.5 Abbreviations

The *abbreviations* section of the Sbr-Regesten lists abbreviations used throughout the book, along with their expansions:
The XML schema described below for representing this part of the original text is heavily based on recommendations from the TEI guidelines.

XML Schema For wrapping the abbreviations section as a whole, the XML schema defines the `<abbrev-list>` tag. The title of the abbreviations section (*Abkürzungen*) is inserted into the `<abbrev-list>` node as is; the schema does not define a separate tag for it. By contrast, the

A.	Aussteller
Abt.	Abteilung
anh.	anhängend
aufgedr.	aufgedrückt
Ausf.	Ausfertigung
begl.	beglaubigt
besch.	beschädigt

Figure 24: Excerpt from the abbreviations section of the Sbr-Regesten

free text following the title is wrapped using the `<list-info>` tag.[11] Individual entries of the abbreviations section are represented by `<entry>` nodes. These nodes have two child elements each, one for tagging the abbreviation itself, called `abbr`, and one for tagging its expansion, called `expan`.

Extraction The `abbrev_extractor.py` module uses the titles of the abbreviations section and the section which succeeds it (*Abkürzungen* and *Siglen*, respectively) to identify relevant lines in the HTML source; any textual content between these two keywords is considered to belong to the abbreviations section. The extraction algorithm is similar to the algorithms used for extracting front matter and TOC sections: First, using the cues described above and ignoring empty lines, a list of relevant lines is accumulated from a `BeautifulSoup` instance representing the HTML source. Then, abbreviations and their expansions are annotated with the corresponding tags and written to the XML output file one by one.

5.6 Initials

The *initials* section of the Sbr-Regesten lists author initials that are used throughout the book, along with their expansions:

Ed	Irmtraut Eder-Stein, Koblenz
He	Hans-Walter Herrmann, Saarbrücken
Jac	Fritz Jakob, Saarbrücken (†)
Kl	Hanns Klein, Saarbrücken / Wellesweiler (†)

Figure 25: Initials section of the Sbr-Regesten

XML Schema For wrapping the initials section as a whole, the XML schema defines an element called `initials-list`. The *type* of this element is the same as the one used for the abbreviations section. As a result, the inner structure of the `<initials-list>` node in the XML output is identical to that of the `<abbrev-list>` node discussed in Section 5.5, and does not need to be discussed again here.

Extraction The module uses the title of the initials section (*Siglen*) and the title of the first regest to detect content belonging to the initials section in the HTML source. Aside from some adjustments that had to be made due to the fact that the initials section lacks an info section, the algorithm for extracting and annotating initials and their expansions is virtually identical to the one used for the abbreviations section.

[11] The name of this tag has been kept generic on purpose; it is reused in the list of initials described shortly.

5.7 Archives

Author: Conrad Steffens

5.8 Future Work: DRYing Out Extractor Modules

Due to time constraints it was not possible to refactor the modules responsible for extracting the parts of the Sbr-Regesten discussed in this chapter. As a result, there is a fair amount of code duplication within and between individual extractor modules. Since they contain fairly small amounts of code, this is not a critical issue, but if the extraction process is to be extended in the future, it would be desirable to DRY out the existing code beforehand.

6 The Sbr-Regesten Web Application

Author: Tim Krones

This chapter describes the Sbr-Regesten Web Application. To provide the minimal amount of theoretical background necessary to productively work with and extend the application, we first give an overview of the Django Web Application Framework which was used to develop the application, and how it implements the *MVC pattern*. We then describe in detail how to install, run, and use the application, and provide pointers for deploying and extending it. In the last section of this chapter, we discuss the data model that was designed for the web application, focusing mainly on how it differs from the XML schema presented in previous chapters.
Django-related information provided in this chapter is based on the official Django documentation, which provides comprehensive information about the framework and can be found here. Please note that this chapter is not to be understood as a replacement for reading external documentation: Its main purpose is to give the reader a *basic* understanding of all aspects involved in dealing with Django applications in general, and the Sbr-Regesten Web Application in particular. As a result, some Django-related concepts are touched on only briefly. We do, however, give pointers to external documentation wherever necessary.

6.1 The Django Web Framework

Django is a Python framework for rapid prototyping and development of interactive web applications. It uses the *MVC pattern* to separate the different tasks that are involved in creating interactive web applications.

6.1.1 The MVC Pattern

Model-View-Controller, or MVC for short, is a Software Design Pattern commonly used by Web Frameworks such as Django and Ruby on Rails. The basic idea of MVC is to divide application logic into three layers. The *Model* layer is responsible for storing and operating on data. This usually involves at least the basic CRUD operations *Create*, *Read*, *Update*, and *Delete*. The *View* layer takes care of presenting available data to end users. *Controllers* are responsible for handling user requests. Depending on the type of request, this usually involves querying the model layer for data, manipulating this data in various ways (if necessary), and sending it off to the view layer for presentation.
Different frameworks interpret MVC in different ways; the next chapter describes Django's implementation of this pattern.

6.1.2 How Django Implements MVC

This chapter presents an overview of how Django interprets and implements the MVC pattern. For an in-depth treatment of the individual components, please consult the documentation at `https://docs.djangoproject.com/`.
While Django's separation of concerns is heavily influenced by the MVC pattern conceptually, the framework uses a different terminology to distinguish the individual components for dealing with user requests, data, and presentation. The terminological differences tend to confuse users that are new to Django or to working with MVC frameworks in general, which makes it all the more important to understand these differences before delving into Django development.

Django distinguishes between *models*, *templates*, and *views*, which is why the framework is commonly referred to as an "MTV" framework. The model layer in Django corresponds to the concept of a model layer as it is defined (or at least commonly understood) in the context of MVC. Django templates correspond to views in MVC, and the responsibilities of Django views are similar to those of controllers in MVC.
From an architectural point of view, a Django *project* usually consists of one or more Django *apps*. Among other things, each app includes a dedicated Python module for the model layer called `models.py`, two Python modules that are jointly responsible for handling user requests called `views.py` and `urls.py`, and a hierarchy of templates written in Django's template language.
The `models.py` module contains specialized Python classes (called *models*) which define the data model of a given Django app. Each class corresponds to a table in the database of the project, with additional tables being created as necessary to represent relationships between different models.
The `views.py` module contains specialized Python functions (called *view functions*) for handling user requests. These functions are responsible for querying the database for information, manipulating that information if necessary, and rendering the appropriate templates back to the user, filled with the information that was requested. In this context, the `urls.py` module acts as a kind of *dispatcher*: It contains a mapping from URLs (or, generally speaking, URL patterns) to appropriate view functions, allowing Django to identify the actions it needs to take based on the URL that was requested by the user.

6.2 Installation and Usage

This chapter explains how to install, run, and use the Sbr-Regesten Web Application.

6.2.1 Installation

Python and Django The Sbr-Regesten Web Application was developed using Python 2.7.3 and Django 1.4.3.
Python binaries and source code for all major operating systems can be obtained from `http://python.org/download/`. Python binaries are usually pre-installed on Linux distributions, and different versions can be obtained from standard repositories using a package manager: Please note that at the time of this writing, Django is **not** compatible with Python 3, so in order to run the app successfully, Python 2.7.* needs to be installed.
The easiest way to install specific versions of Django is using the pip-installer which is a tool for installing and managing Python packages. `pip` should be available in the standard repositories of most Linux distributions as a package called `python-pip`. For generic installation instructions, visit `http://www.pip-installer.org/en/latest/installing.html`.
Once `pip` has been installed, version 1.4.3 of Django can be installed using the following command:

```
$ pip install Django==1.4.3
```

For instructions on how to install Django manually, consult this part of the Django documentation.

BeautifulSoup The process of extracting and annotating information from the Sbr-Regesten makes heavy use of a tool called *BeautifulSoup*, which needs to be installed in order to reproduce the extraction process locally.
Like Django, BeautifulSoup is pip-installable:

```
$ pip install beautifulsoup4
```

For the purpose of improving or extending the extraction process, detailed information about BeautifulSoup can be found in its official documentation.

Further Recommendations In addition to the hard dependencies described in the previous sections, we recommend installing the *IPython* interpreter as it provides a lot of features not included in the standard python interpreter and thus makes interacting with the database from Django's development shell a lot easier. The latest version of IPython can be installed using pip as follows:

```
$ pip install ipython
```

6.2.2 Running the Application

Once all necessary dependencies are installed, you are ready to run a local instance of the application. Extract the contents of the source archive to an appropriate folder in your file system and `cd` into the root folder of the project. This folder is called `sbr-regesten`. Look for a file called `sbr-regesten.db`. If it's there, this means that the source package you have received includes a pre-populated database, and that you can run the application right away by typing

```
$ python manage.py runserver
```

If you find that the database file is missing from the source archive, you need to proceed as follows: First, initialize the database by running

```
$ python manage.py syncdb
```

At some point before this command finishes, you will be asked whether or not you would like to create a superuser for the database. Type `yes` and press Enter, then provide a username, email address and password. For development purposes it is both convenient and acceptable to simply set username and password to `admin`.
When the `syncdb` command finishes, you can either start using the application with an empty database by typing the `runserver` command listed above, or you can go ahead and populate the database with information from the Sbr-Regesten. Since the extraction process was implemented as a Django *management command*, you can trigger it using the following command:

```
$ python manage.py extract
```

Note that this process might take a long time to finish. Also, please make sure that there is a file called `sbr-regesten.xml` in the top-level directory of the project (`sbr-regesten`) with the following contents:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
```

For best results, the XML declaration should be followed by a newline.

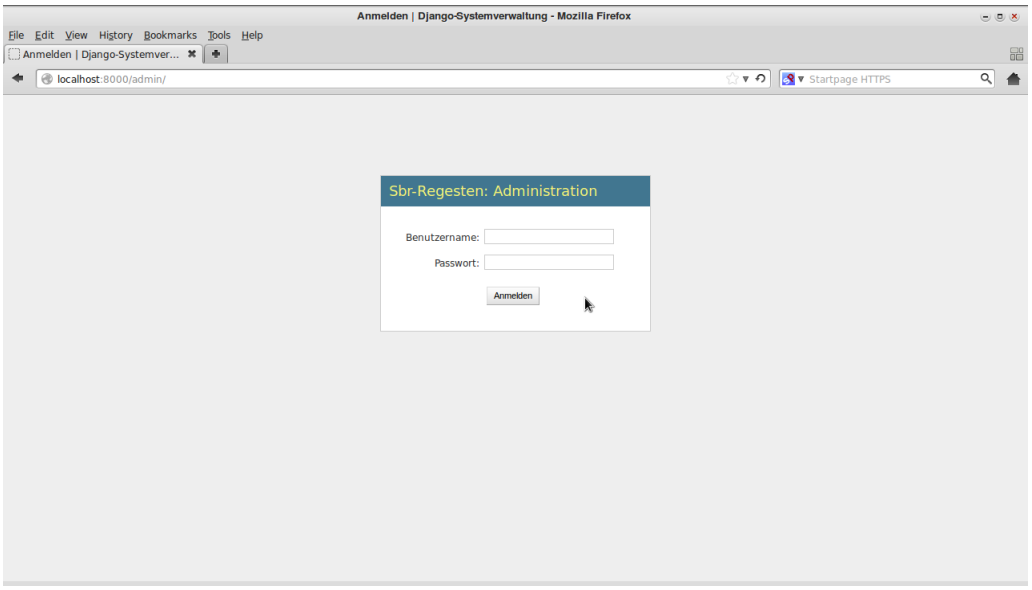

Figure 26: Login screen for the Django admin interface

6.2.3 Using the Application

After starting the development server with the `runserver` command, direct your browser to `http://127.0.0.1:8000/admin` to access the application. You will be greeted by a login form, shown in Figure 26.
If you are working with a pre-populated database, enter `admin` in both the *Benutzername* and the *Passwort* field. If you initialized the database yourself, use the credentials you specified in the `syncdb` step explained in the previous section. Clicking on *Anmelden* will take you to the main page of the *Django Admin Interface* which is shown in Figure 27.
The admin interface allows you to browse, search and manipulate (i.e. add, change, and delete) the data that is stored in the database from the convenience of your browser.

Browsing and Searching Data From the main page of the admin interface you can get listings of database entries for the following entities that can be found in the Sbr-Regesten[12]:

- Archives (listed as *Archive*)
- Families (listed as *Familien*)
- Landmarks (listed as *Flurnamen*)
- Concepts (listed as *Konzepte*)
- Locations (listed as *Orte*)
- Persons (listed as *Personen*)
- Person groups (listed as *Personengruppen*)

[12]Consult chapter 6.5 for more information about these entities.

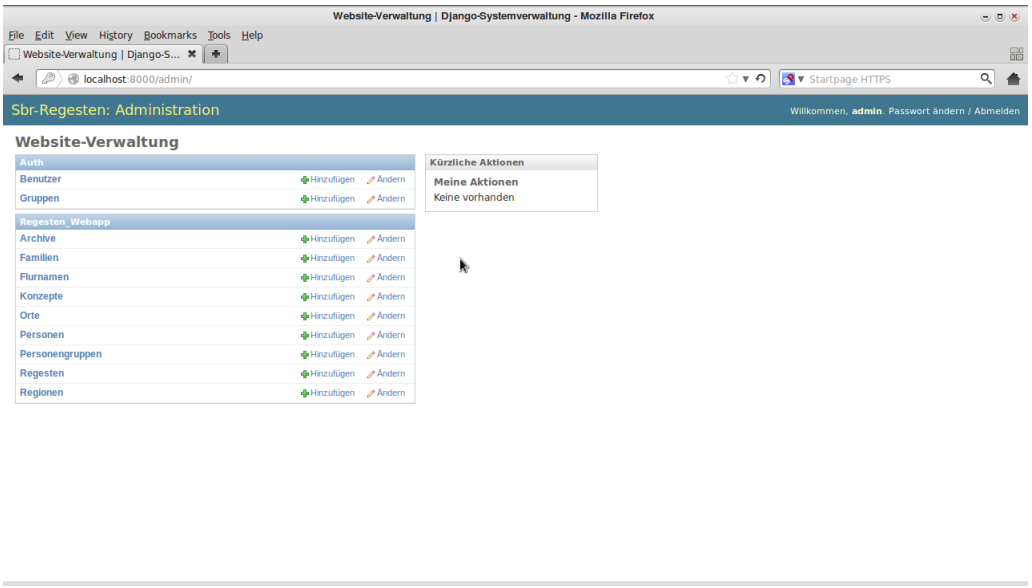

Figure 27: Main page of the admin interface, starting point for browsing and manipulating data

- Regests (listed as *Regesten*)
- Regions (listed as *Regionen*)

Click on the name of a specific entity to get to the corresponding listing of database entries. The listing for locations (*Orte*) looks like this:
To sort the table by a specific column, click on its header. Clicking more than once will toggle between ascending and descending order for that column.
If you are looking for a specific entry, you can narrow down the list by entering appropriate search terms in the search field and clicking the *Suchen* button. On the result page, click on the link next to the information about the number search results to get back to the full listing.
Please note that as of this writing, search is configured to consult only a limited number of model fields for each entity for efficiency reasons.

Adding Data Starting from the main page of the admin interface, you can manipulate the data in the database in various ways. For each entity/model that has been configured to be editable via the admin interface, Django displays two buttons; one for adding a new database entry for a specific model (*Hinzufügen*), and another one for changing information associated with existing entries (*Ändern*).
To add a new database entry for a specific model, click on the corresponding *Hinzufügen* button. This brings up the appropriate input form for the model. As an example, consider the form for adding a new Regest entry to the database, shown in Figure 29.
In any input form, fields with labels in bold font are mandatory. This means that Django will not allow you to save a new entry to the database without filling them in. Instead, it will annotate the fields you failed to fill in with appropriate error messages and redisplay the input form[13]; see Figure 30 for an example.

[13]The interface behaves in exactly the same way if you **edit** the information for an existing entry and accidentally delete any mandatory information.

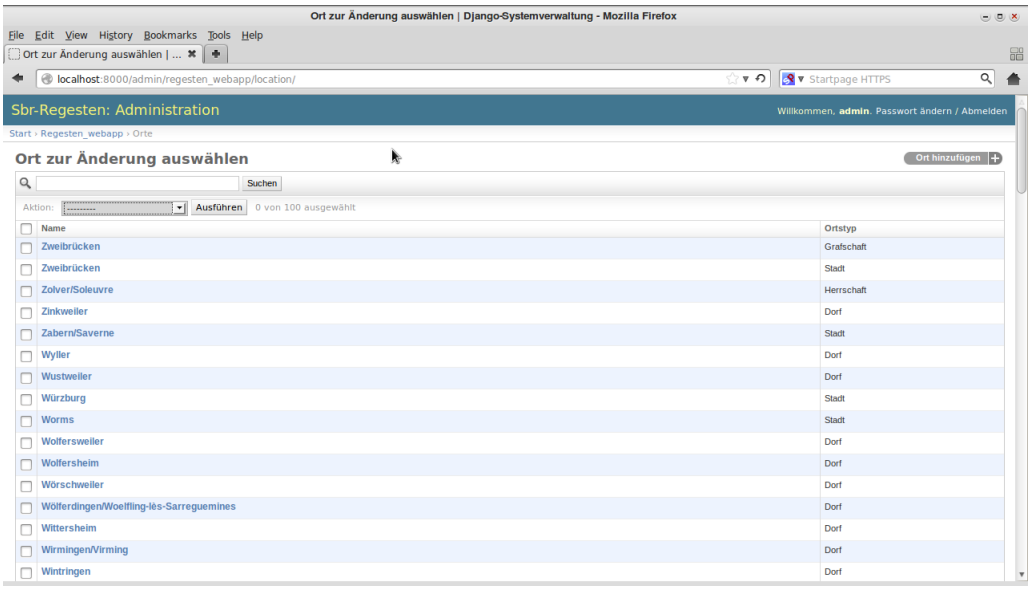

Figure 28: Example listing of database entries

When you are done filling in the information that you would like to store as a new entry in the database, click the *Sichern* button at the very bottom of the input form. This will take you back to the listing of existing entries for the model that you were working on. Alternatively, if you want to add another database entry for the same model, click the *Sichern und neu hinzufügen* button. This will bring up a new input form for the same model. If you want to save the entry and continue working on it afterwards, click *Sichern und weiter bearbeiten.*[14]
If you are familiar with the Sbr-Regesten, most of the fields included in the input forms for the different models should be self-explanatory. To help you along, some of the less obvious fields are annotated with examples of the type of input that they expect. However, before starting to add or edit information from the admin interface, please refer to chapter 6.5 to get a more detailed understanding of the type of information represented by any given field.

Manipulating Data The steps that are involved in editing existing data are very similar to the ones that are required for adding new entries. On the main page of the admin interface, click on the *Ändern* button that corresponds to the type of model you would like to edit one or more entries for. This will bring up the already familiar listing of database entries for the model. If necessary, sort or narrow down the list as described above, then click on the name of the entry you would like to edit. The fields of the input form that comes up will be pre-populated with the information that is available for this entry. Other than that the input form looks and behaves exactly like the form you are already used to for adding new entries.
To get a list of all changes that were made to a specific database entry via the admin interface, click on the button that says *Geschichte* in the top right corner of the input form.

Removing Data If you want to remove a specific entry from the database, navigate to its input form as described above. Then, on the very bottom of the page, click the *Löschen* button. This will bring up a confirmation page, shown in Figure 31.

[14]Note that this won't work if any of the mandatory fields are empty.

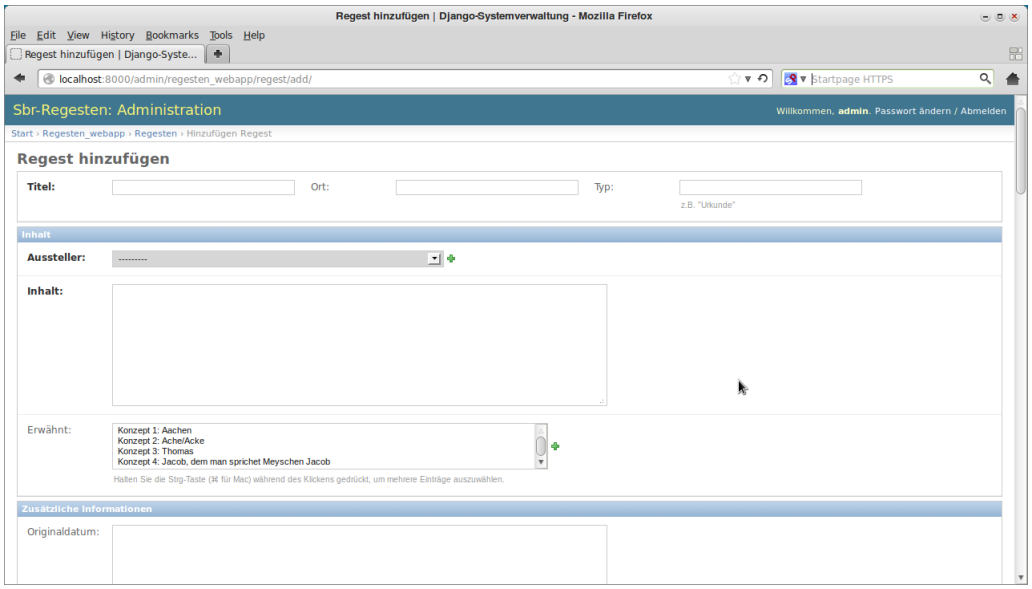

Figure 29: Top part of the form for adding a new Regest entry to the database

Click on the *Ja, ich bin sicher* button to finalize the action. If you change your mind, you can use e.g. the breadcrumbs to navigate somewhere else.
It is also possible to delete multiple entries at once. On the page that lists all existing entries for a given model, mark the entries you would like to delete by clicking on the check boxes next to the names of the entries. Then choose *Ausgewählte MODEL_NAME löschen* from the drop-down menu labeled *Aktion* at the top and click the *Ausführen* button. Figure 32 shows the page right before finalizing the delete action.

6.3 Deployment

As mentioned in Section 6.2.3, the `runserver` command runs the *development* server for the Sbr-Regesten Web Application. There are some additional steps you need to take to get the application set up and running on a production server. Depending on the type of server you are running, there are several ways to achieve this, so when you are ready to deploy the application, please consult the Deploying Django section of the Django documentation for detailed information on different ways to get the application running on your server.

A Note on Debug Mode Despite not going into detail here about how to deploy the Sbr-Regesten Web Application, there is one important security issue that needs to be addressed: For the purpose of making applications easy to develop and debug, Django provides a *debug mode*. If this mode is turned on, Django displays a *debug page* for any errors or exceptions that occur while interacting with the application in the browser. This page includes a full error trace-back containing extensive information about the environment the application is running in. This poses a high security risk, as it provides users (who might have malicious intentions) with all kinds of information they should not have access to. Therefore, before deploying the Sbr-Regesten Web Application in a production environment, make sure that the `DEBUG` variable (which is used to turn debug mode on and off) is set to `False` in the `settings.py` modules located in the `sbr-regesten/sbr_regesten/` directory.

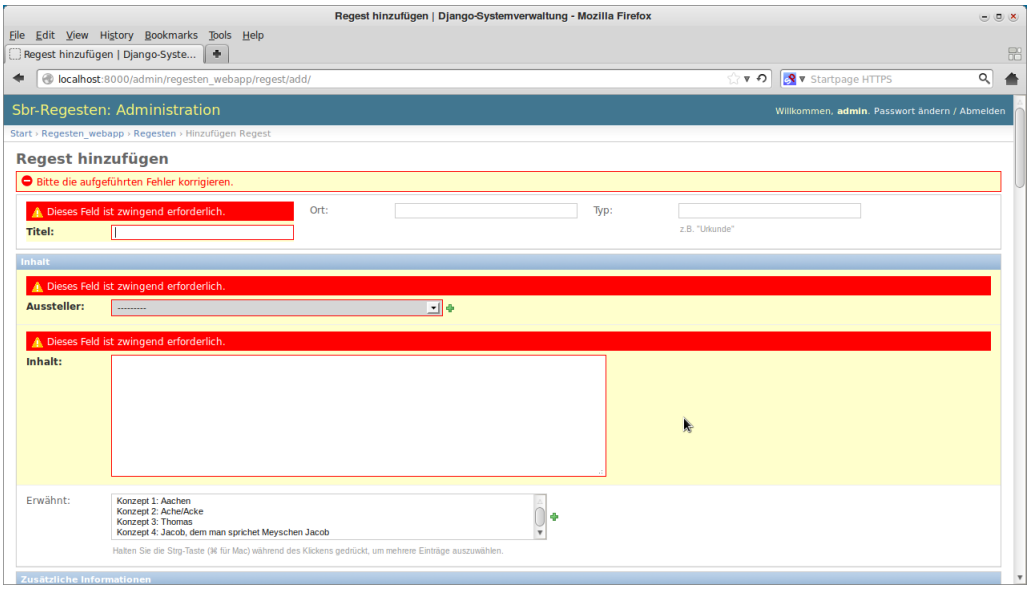

Figure 30: Input form with errors

6.4 Extending the Web Application

This section provides information about how to obtain a copy of the source code for development purposes that includes the complete history of changes made since the beginning of the project. It also includes information about testing and translating the application, as well as some suggestions about missing features that should be implemented next.

6.4.1 Version Control with Git

The Sbr-Regesten Web Application was developed using the Git Distributed Version Control System. For the purpose of extending the application we recommend you continue using this system, as this will enable you to make use of the development history in various ways.

Installing Git Git is available for all major operating systems. On most major Linux distributions it can be installed from the standard repositories using a package manager. For detailed platform-specific instructions and/or to download graphical installers for Mac OS X and Windows, go to `http://git-scm.com/downloads`.

Using Git to Download the Source Code The source code repository for the Sbr-Regesten is hosted on GitHub at `https://github.com/itsjeyd/sbr-regesten`. To obtain a copy of the most recent version of the source code, `cd` to the directory in which you want to store it and enter the following command:

```
$ git clone https://github.com/itsjeyd/sbr-regesten.git
```

This will download (*clone*) the repository to a folder called `sbr-regesten` in your current working directory.
Note that doing this will **not** give you *push access* to the GitHub repository: Git lets you commit any future changes you make to the source code *locally*; hosting code on a remote server is not

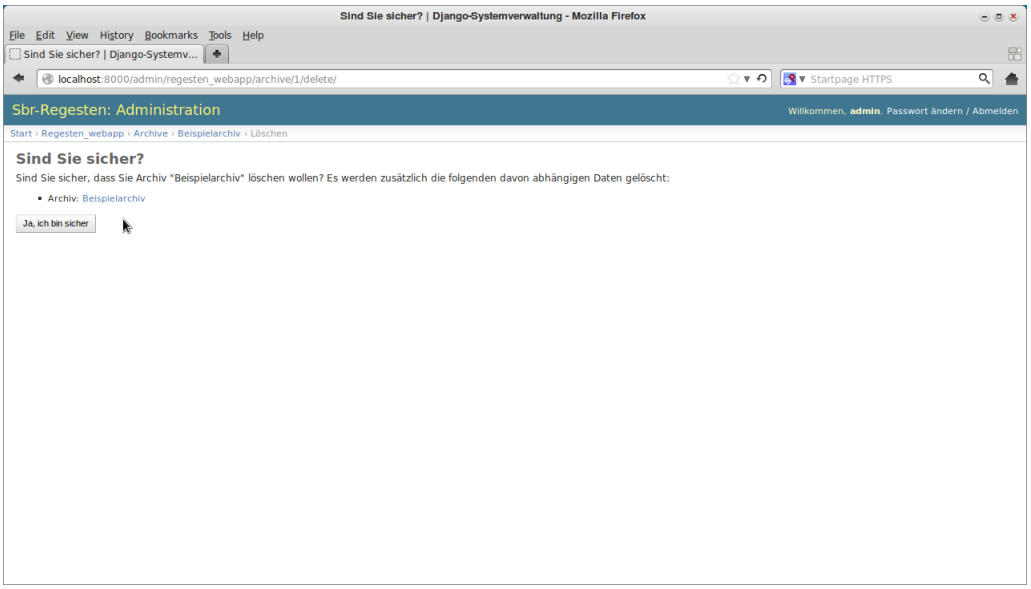

Figure 31: Confirmation page for deleting items

necessary in order to successfully keep it under version control with Git. However, unless you are granted permission by the owner of the GitHub repository, you will not be able to *push* your changes back to the server.
If you would like to host your own copy of the source code on GitHub (e.g. to facilitate collaboration among multiple developers), instead of cloning the code from the original repository, you should follow these steps:

1. Create a GitHub account
2. Fork the original repository
3. Clone the repository from your own fork to your local machine, using the `clone` command above with the appropriate URL.

A Note About the XML Schema The XML schema for the Sbr-Regesten was developed using a separate Git repository called *regesten-schemas*.[15] This repository is integrated into the main repository as a *submodule*. When you clone the `sbr-regesten` repository using the `clone` command listed above, Git will not download any files associated with the submodule: Its folder (`sbr-regesten/regesten-schemas`) will be empty. To download the contents of the regesten-schemas repository to this folder, you need to run

```
$ git submodule init
```

followed by

```
$ git submodule update
```

For more information on Git submodules, refer to this section of the "ProGit" book listed below.

[15] In case you are not interested in the Sbr-Regesten Web Application and only want to modify or extend the XML schema, you can `clone` the regesten-schemas repository directly from the following URL: `https://github.com/itsjeyd/regesten-schemas.git`

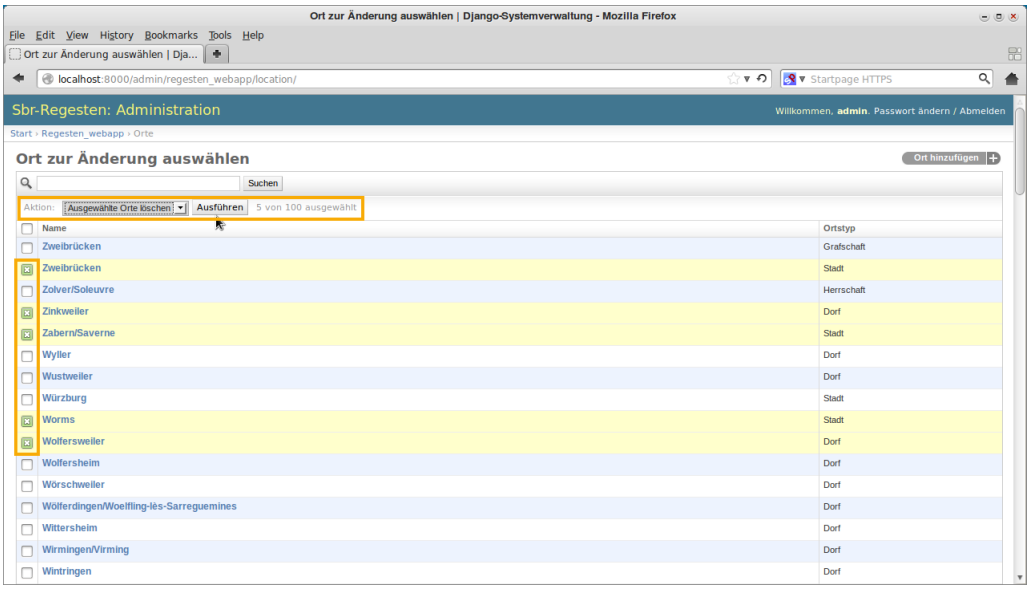

Figure 32: Deleting multiple items at once

Git Resources There is a wealth of information about how to use Git available online, so it does not make sense to duplicate this information here. If you are new to Version Control with Git, the following resources will help you get started:

- tryGit, a free, *no-setup-required* CodeSchool[16] course that lets you practise working with Git in the comfort of your browser
- Pro Git by Scott Chacon, a comprehensive book about Git that is available both in print and online; the link references the online version which is searchable and continuously receives updates
- The Git Reference,

  > "a quick reference for learning and remembering the most important and commonly used Git commands",

  which also provides links to relevant sections of the Pro Git book.

6.4.2 Next Steps: Suggestions for Future Extensions

Before creating a full-fledged web application for the Sbr-Regesten, a data model to represent the contents of the original document had to be created. This task was the main focus of our project, along with extracting, annotating and storing the information contained in the Sbr-Regesten based on this new model. The purpose of this section is to point out functionality that was outside the scope of this project but should most likely be implemented next.

[16] `http://www.codeschool.com/`

Changing the Database Backend Django supports a number of different database engines. By default it uses `sqlite3`, as this involves virtually no additional setup, because Python 2.7 includes sqlite connector modules in its standard library. For development purposes, sqlite is an acceptable choice. In terms of performance, however, it leaves a lot to be desired and is therefore not recommended for use in a production environment. MySQL and PostgreSQL are much more suited for these kinds of environments.
There are several steps involved in changing the database backend:

1. If you haven't already, install the database engine of your choice on the server you are planning to deploy on.

2. Install the appropriate Python bindings for the type of database you picked.

3. Grant Django the necessary rights to create and alter tables; for testing purposes, Django will also need permission to create test databases.

4. Back up the data currently stored in the `sqlite3` database using the following command:

   ```
   $ python manage.py dumpdata > datadump.json
   ```

5. Adapt the `DATABASES` setting in `settings.py`:

   - Change the `ENGINE` option to the value corresponding to the new engine (see instructions in the module for a list of possible values).
   - Change the `NAME` option to the name of the database you would like to use with MySQL.
   - Provide values for the `USER` and `PASSWORD` options.
   - (Optional) Provide values for the `HOST` and `PORT` options.

6. If you haven't added any additional information to the database, create a new database and re-run the extraction process, following the steps listed in Section 6.2.2. Alternatively, you can load the data you backed up in step 4 into the new database with this command:

   ```
   $ python manage.py loaddata datadump.json
   ```

For additional information, refer to the Get your database running section of the Django documentation.

XML Export As new information gets added to the database via the admin interface of the web application, the XML version of the Sbr-Regesten will become more and more outdated. In order to prevent this, a feature that allows for XML export of database entries needs to be implemented. In the spirit of CRUD, this could e.g. be done by adding entity-specific `to_xml` methods to each of the relevant models. Additionally, for convenience sake, this functionality could be integrated into the admin interface by writing a custom Admin Action. Admin actions are useful for performing the same task on multiple database entries at once; for example, deleting multiple database entries is an instance of an admin action. They are available from the drop-down menu labeled *Actions* on the pages that list existing database entries for individual models (see Section 6.2.3, "Browsing and Searching Data").

Public Pages for End-Users In terms of actual functionality, the web application so far "only" provides an interface for *administrating* the information stored in the database. Regular users will most likely not be given access to this interface, and the Django Admin Interface was also not designed to be used by a large number of end users. The application is lacking functionalities for presenting the information stored in the database to end users, which will have to be implemented before releasing the application to the public. In more technical terms, this means that future work will focus heavily on the *view* and *template* layers of the application.
The following sections of the Django documentation provide detailed information about these layers:

- Handling HTTP requests
- Working with forms
- The Django template language

6.4.3 Adding Tests for New Features

Django comes with a built-in test execution framework which extends the `unittest` module included the Python Standard Library for web development-specific needs. By default, when using the `startapp` management command to create the basic structure for a new Django app, Django creates a single module for the tests associated with the app called `tests.py`. This module is located in the top-level directory of the app. However, as individual apps grow in size, it might become necessary to split up their tests into several modules to keep the overall architecture modular. In order to accomplish this without confusing the default test runner you should follow these steps:

1. In the directory that holds the default `tests.py` module, create a directory called `tests`.
2. `cd` into this directory and turn it into a *package* by creating a file called `__init__.py`.
3. Move the default `tests.py` module to the `tests` directory and rename it according to the types of tests it contains.[17]
4. Add any number of additional test modules to the directory.
5. As a last step, add a separate `import` statement for each test module to the `__init__.py` module. For instance, if `tests` contains three modules `model_tests.py`, `view_tests.py`, `extraction_tests.py`, the contents of `__init__.py` will have to look like this:

```
from model_tests import *
from view_tests import *
from extraction_tests import *
```

The current version of the Sbr-Regesten Web Application already includes tests for non-trivial methods of the model layer. They are located in the default `tests.py` module under

```
sbr-regesten/regesten_webapp/
```

To run them, issue the following command from the top-level directory of the project:

[17] If you haven't added to or changed the default test module, you could e.g. rename it to `model_tests.py`, since at the time of this writing it only contains tests for the model layer.

```
$ python manage.py test regesten_webapp
```

The Django framework itself also comes with a large number of tests. To run tests for all built-in Django apps that the Sbr-Regesten project uses, along with any custom tests currently defined in the `tests.py` module, you can use the following command:

```
$ python manage.py test
```

When extending the Sbr-Regesten Web Application, tests for any new features of the model layer (e.g. XML export) can be added to the existing `tests.py` module. As soon as you start implementing views and templates for end users, however, we recommend that you switch to using a dedicated `tests` directory and multiple test modules as described above.
To learn more about writing and running tests for Django applications as well as using alternative testing frameworks with Django, consult the Testing Django applications section of the official Django documentation.

6.4.4 Internationalization: Translating the Application

As the Sbr-Regesten Web Application is targeted at users whose native language is German, any new functionality associated with user-facing features will require translation. Django provides comprehensive support for internationalization and localization that can be leveraged to make this process easy.
The basic procedure for translating relevant parts of a Django application consists of the following steps:

1. Add translation hooks to the source code where necessary; these hooks are also called *translation strings*
2. Extract translation strings into a *message file*
3. Add translations for the individual strings to the message file
4. Compile the message file

In regular Python code, translation strings are specified using the `ugettext()` function, which by convention is imported as `_`. As an example, consider the definition of the `title` attribute of the `Regest` model:

```
title = models.CharField(_('title'), max_length=70)
```

The first argument to the `models.CharField` constructor is the *verbose name* of the attribute to use for presentation purposes. By default, Django uses the name of the attribute (with underscores converted to spaces) as its verbose name, but for the purpose of telling Django that we want it to translate the name of this attribute, we have to specify it in the constructor, and surround it with a call to `ugettext()` / `_`. Looking at the remaining definition of the Regest model and other model definitions in `models.py`, you will notice many more examples for translation strings.
To update the message file for German translations of the code base when you are done implementing a new feature, run the following command from the `sbr-regesten/regesten_webapp` directory:

```
$ django-admin.py makemessages -l de
```

The message file for German translations is located in the directory

```
sbr-regesten/regesten_webapp/locale/de/LC_MESSAGES
```

and it is called `django.po`. To add translations for new translation strings, open this file and fill all empty `msgstr` strings with the appropriate translations. The structure of message files is explained in detail in the Django documentation; see below for a link.
As a last step, when you (or your translators) are done editing the message file, you need to compile it by running the following command from the same directory you ran the `makemessages` command from earlier:

```
$ django-admin.py compilemessages
```

This command outputs a compiled message file called `django.mo` to the `LC_MESSAGES` directory that also holds the plain text version of the message file.
Finally, please note that Django uses the GNU `gettext` utilities for creating and compiling message files. If these utilities are not available, the files generated by the `makemessages` command will be empty.
To learn more about Django's features for translating applications, consult the Internationalization and localization section of the official documentation.

6.5 Django Data Model for the Sbr-Regesten

The Django data model for the Sbr-Regesten was derived from the XML schema described in previous chapters. As a result, the level of correspondence between the two models is high, but they are not completely identical. The main purpose of this section is to provide an overview of the Django data model and to highlight and motivate the differences that exist between the two models. We only go into detail about models and model attributes that do not correspond directly to elements or attributes present in the XML schema. If a specific model or attribute is not addressed here, it is intended to be understood as serving the same purpose as its XML equivalent.

6.5.1 Overview

The UML class diagram in Figure 33 shows the Django data model in its entirety.[18] Please note that:

1. Model methods were omitted to keep the focus on how information from the Sbr-Regesten is represented in the database.

2. Every entity represented by a model class has to have a unique ID that can be used as a *Primary Key* for the corresponding table in the database. This ID is available on instances of each of the models shown in the diagram, in the form of an attribute called `id`. For brevity, and because the models inherit this attribute from their common super class (`models.Model`) rather than defining it themselves, the attribute is omitted in the diagram.

[18] The diagram was created using Dia, a cross-platform drawing application with support for a large variety of diagram types.

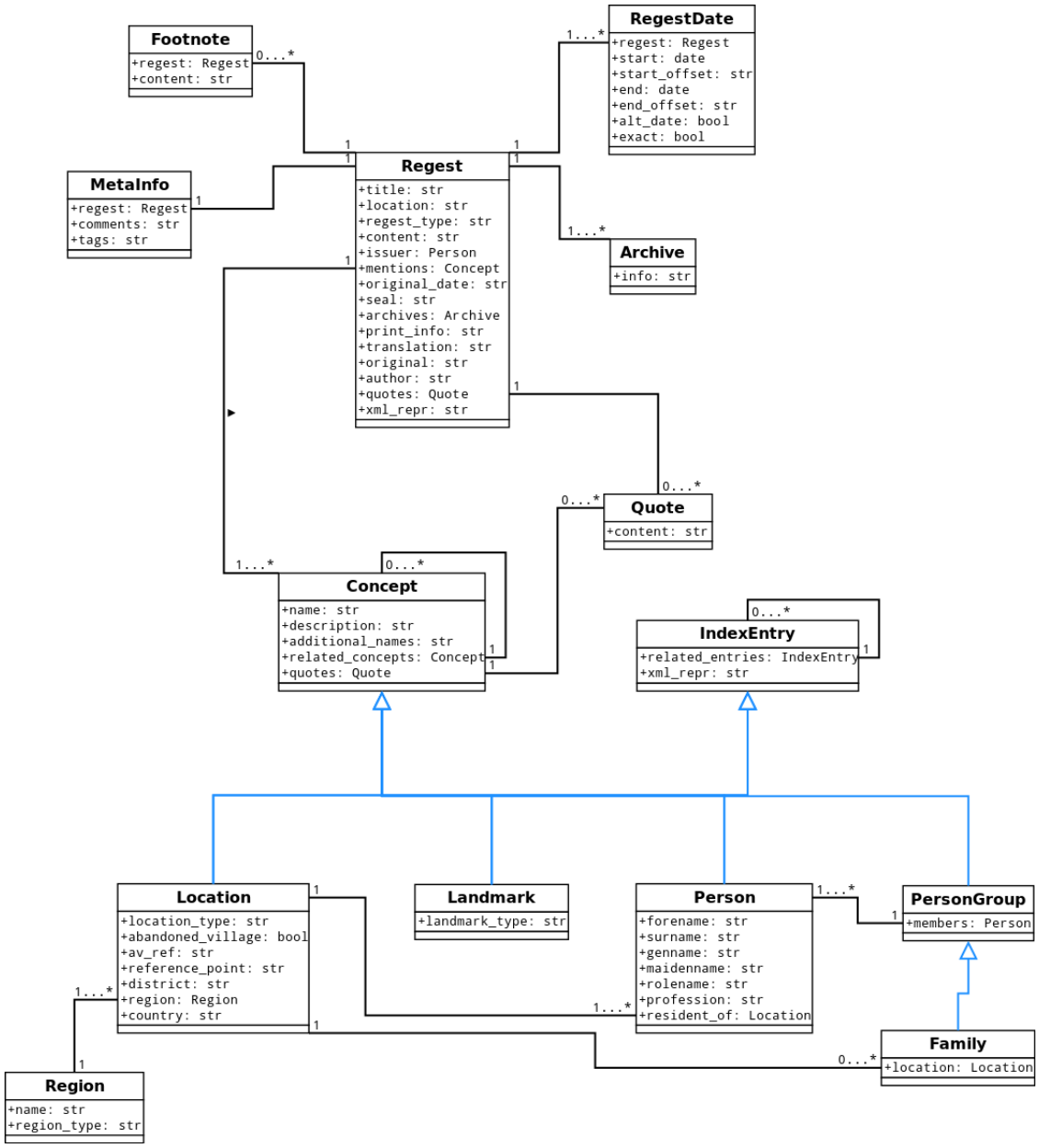

Figure 33: The Django data model of the Sbr-Regesten

Following standard UML notation, boxes denote model classes in the diagram. Names of individual models are given in bold in the top section of each box; attributes are listed in the middle section.[19] Blue arrows with hollow heads denote *inheritance* relationships.[20] Classes with incoming arrows are super classes of classes with outgoing arrows; super classes pass their attributes on to their subclasses. Relationships between different types of models – *associations* – are denoted by black lines connecting their boxes. Associations are annotated with *cardinalities*. For instance, *one* (`1`)

[19]Bottom sections of classes are empty because they normally hold class methods which, as mentioned above, were omitted in the diagram.

[20]N.B. that the fact that these arrows are colored in blue in the class diagram has nothing to do with UML. They are colored differently to make them easily distinguishable from other types of lines present in the diagram.

location can have *one to many* (`1...*`) residents. Similarly, a single concept can be related to *zero or arbitrarily many* concepts.

6.5.2 Data Model vs. XML Schema: General Differences

Naming Schemes One of the main differences between the Django data model and the XML schema concerns naming schemes for *identifiers* (i.e. class and attribute names in Django, element names in XSD): In the XML schema, dashes are used to separate individual words that make up *compound identifiers* like `abbrev-list` and `original-date`. Corresponding Django attributes contain underscores instead of dashes, as dashes are not allowed in Python identifiers.
Identifiers in the XML schema are generally lowercase, except for those that were borrowed from the TEI guidelines, which were transferred to our schema *as is* to distinguish them from identifiers for custom types that were introduced specifically for the Sbr-Regesten. For the Django data model, we stuck to widely used Python naming conventions: `CamelCase` was used for class and model names, `snake_case` for everything else.
Additionally, some identifiers had to be affixed going from the XML schema to the Django data model due to clashes with names of Python 2.7 functions and keywords. Examples include `print`, which was renamed to `print_info`, and `type`, which was renamed for individual models using the following pattern: `<model_name>_type`.

Storing Information About High-Level Entities The XML schema distinguishes between *elements* and *attributes* for storing information about high-level entities. Attributes are generally employed to represent *names of entities* (examples include regest titles and names of index entries) and different kinds of *meta information*, which may or may not be provided explicitly for a specific entity. For example, author information for individual regests is usually provided explicitly, while the type of any given index entry is deduced in the extraction process. The Django data model, by contrast, makes no such distinction: It uniformly stores information about entities in model attributes, the majority of which are textual types.

Relationships Between Entities In the XML schema, relationships between entities are modeled in two different ways: Entities that belong together logically and are close to each other in the original text are most often connected via (transitive) *parent-child* relationships. On a conceptual level, for instance, every regest has one or more dates associated with it, each of which is extracted from its title. To make this relationship explicit, `date` nodes are nested into `regest` nodes as direct child nodes in the XML version of the Sbr-Regesten.[21]
Connections between related entities that are further removed from each other structurally can not be modeled in the same way. For these cases, the XML schema uses explicit references to unique entity IDs to indicate links between specific types of entities. As an example, consider the `mentioned-in` lists associated with entities listed in the index: Along with the titles of the regests a given entity is mentioned in, they reference their IDs via the `regest` attribute of the `reg-ref` node.
In Django there is only one way to model relationships between high-level entities: To express that there is a connection between two distinct models, **one** of the two models needs to define an appropriately named attribute which "points" to the other model. The type of "pointer" that is appropriate for a given relationship (`ForeignKey` vs. `OneToOneField` vs. `ManyToManyField`) and

[21] As an additional example for a transitive case, consider the relationship between a regest and its issuer (*Aussteller*): `<issuer>` is a child node of `<content>`, which in turn is a child node of `<regest>`.

where (i.e., on which model) the attribute representing it has to be defined depends on the direction and cardinality of the relationship: In some cases, the attribute had to be set on the *component* model instead of the main one. The `RegestDate` model, for instance, defines a `regest` attribute (of type `ForeignKey` since it encodes a *many-to-one* relationship), instead of the `Regest` model defining a `regest_date` attribute like in the XML schema.[22]

Representing Structure vs. Content The XML schema created for the Sbr-Regesten mixes structural information about the original text with markup used to identify semantic units and entities within the text. By contrast, the Django data model was designed to serve as a starting point for a comprehensive knowledge base independent of a particular text, and therefore strongly emphasizes *content*. However, because high-level entities (namely `Regest` and `IndexEntry`) store their XML representations in an attribute called `xml_repr`, which for obvious reasons has no equivalent in the XML schema, structural information about the original text **can** be derived from the database using not only relationships between different models, but also the values of this attribute. Another side effect of having to preserve and represent structure in addition to content is that the XML schema requires multiple levels of *nesting* of elements in a lot of cases. Compared to that, the Django data model is a lot "flatter": Most pieces of information associated with a given entity are stored simply as text, as opposed to a more complex data structure.

6.5.3 Regesten-Specific Differences

In addition to the general differences between the XML schema and the Django data model listed above, there are some differences specific to the representation of regests:

- In the XML schema, the `issuer` element is not associated directly with the `regest` element; instead it is nested into the `content` element. The Django data model, by contrast, associates the issuer of a given regest with the regest as a whole.
- For tagging proper names in the `content` of individual regests, the XML schema provides the `name` element. Django collects all concepts that are mentioned in a given regest in the `mentions` attribute of the `Regest` model.
- Similarly, in the XML schema `footnote` elements are nested into the different content types representing individual parts of a regest, whereas the Django data model only goes as far as associating `Footnote` objects with whole regests.
- The `MetaInfo` model has no corresponding XML entity, and does not represent information contained in the original text. It is meant to be used for storing administrator-assigned comments and tags for individual regests.

6.5.4 Index-Specific Differences

With regards to representing the index of the Sbr-Regesten, the following differences between the XML schema and the Django data model are of note:

[22] This does not mean that associated `RegestDate` instances are not accessible from instances of the `Regest` model. For information about how to access related objects from models that do not explicitly define "relationship attributes" themselves, see the "Related objects" part of the Making queries section in the official Django documentation.

- Being that it does not aim to be a structural representation of the Sbr-Regesten, the Django data model does not distinguish between `header` and `body` of an index entry.
- As can be seen in Figure 33, the Django data model defines two classes – `Concept` and `IndexEntry` – grouping features that are common to all types of concepts and index entries. While XSD does support a limited form of inheritance via the `<xs:extension>` tag for both simple and complex types, this is restricted to adding attributes and **appending** additional elements to sequences defined by complex types. Somewhat unfortunately, however, elements common to headers of different types of index entries **follow** those that are unique:

```
<xs:complexType name="landmark-header" mixed="true">
  <xs:sequence>
    <xs:element name="geogName" type="geogName" />
    <xs:element name="mentioned-in" type="mentionings" minOccurs="0" />
    <xs:element name="index-refs" type="index-refs" minOccurs="0" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="location-header" mixed="true">
  <xs:sequence>
    <xs:element name="placeName" type="placeName" />
    <xs:element name="mentioned-in" type="mentionings" minOccurs="0" />
    <xs:element name="index-refs" type="index-refs" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

  Therefore, in the XML schema for the Sbr-Regesten we were not able to make use of inheritance to reduce code duplication like we did with the Django data model.
- The type of any given index entry is made explicit by the name of the model used to represent it in the database. As a result, there is no need for a dedicated `type` attribute on the `IndexEntry` model. This is in contrast to the XML schema which states that each index `item` must be assigned a valid value for its `type` attribute.
- The XML schema currently does not define elements corresponding to the `profession` and `resident_of` attributes of the `Person` model. These attributes were added to the Django data model upon request of some of the stakeholders of the Sbr-Regesten Web Application. Within the scope of our project it was not possible to implement functionality to capture this kind of information during the extraction process. However, the application does provide support for adding this information manually via the Django admin interface.

6.5.5 Other Parts of the Book

Unlike the XML schema, the Django data model does not extend to other parts of the book. The reasoning behind this is that the database holding the information extracted from the Sbr-Regesten is not intended to be a static representation of one specific book. As mentioned in multiple places throughout this document, its main purpose is to act as a knowledge base which can be constantly expanded with new information.

A HTML Differences

This appendix lists the differences between the originally extracted HTML file and the manually adapted version. Lines starting with < show the extracted HTML and lines starting with > the adapted version.

```
< - Agnes, oo Heinrich III. 1065-04-03<o:p></o:p></span></p>
---
> Agnes, oo Heinrich III. 1065-04-03<o:p></o:p></span></p>


< <i style='mso-bidi-font-style:normal'>Oster, </i>Wirt und Bürger zu
---
> - <i style='mso-bidi-font-style:normal'>Oster, </i>Wirt und Bürger zu


< - - Kinder 1513-12-27<br>
---
> - Kinder 1513-12-27<br>


< - Johann, Graf zu Nassau-Saarbrücken, Sohn von Graf Philipp II., geboren.
---
> Johann, Graf zu Nassau-Saarbrücken, Sohn von Graf Philipp II., geboren.


< Haus, Hof, Hofstatt 1306-06-11, 1316, 1385-01-03, 1425-03-01, 1493-04-30<br>
---
> - Haus, Hof, Hofstatt 1306-06-11, 1316, 1385-01-03, 1425-03-01, 1493-04-30<br>


< normal'>bij der capellen an der <u>s<o:p></o:p></u></i></span></p>
<
< <p class=MsoBodyText style='margin-top:0cm;margin-right:0cm;margin-bottom:0cm;
< margin-left:35.45pt;margin-bottom:.0001pt;text-indent:-35.45pt'><i
< style='mso-bidi-font-style:normal'><u><span style='color:windowtext'><o:p><span
<  style='text-decoration:none'> </span></o:p></span></u></i></p>
<
< <p class=MsoBodyText style='margin-top:0cm;margin-right:0cm;margin-bottom:0cm;
< margin-left:35.45pt;margin-bottom:.0001pt;text-indent:-35.45pt'><i
< style='mso-bidi-font-style:normal'><u><span style='color:windowtext'><o:p><span
<  style='text-decoration:none'> </span></o:p></span></u></i></p>
<
< <p class=MsoBodyText style='margin-top:0cm;margin-right:0cm;margin-bottom:0cm;
< margin-left:35.45pt;margin-bottom:.0001pt;text-indent:-35.45pt'><i
< style='mso-bidi-font-style:normal'><u><span style='color:windowtext'>tegen</span></u>
```

```
---
> normal'>bij der capellen an der <u>stegen</span></u>


< - - Nikolaus von Saarbrücken, Kaplan 1415-11-04<br>
< <i style='mso-bidi-font-style:normal'><span style='mso-spacerun:yes'> </span>
< </i>1416-03-29<br>
---
> - - Nikolaus von Saarbrücken, Kaplan 1415-11-04, 1416-03-29<br>


< stoißet oben an den wegk, der umb die stat get</i> 1508-11-12<br>
< 1461-07-25, 1502-10-09, 1526-05-03<br>
---
> stoißet oben an den wegk, der umb die stat get</i> 1508-11-12,
> 1461-07-25, 1502-10-09, 1526-05-03<br>
```

B XML Examples

This appendix lists sample index entries, extracted by the Sbr-Regesten Web Application.

B.1 Locations

```
<item id="item_271" type="location" value="Frauenberg">
   <location-header>
      <placeName>
         <settlement type="Dorf" w="false">Frauenberg</settlement>
         <addNames> (
            <addName>Frauwenberg</addName>)
         </addNames>, Dorf (
         <region type="Departement">Dep. Moselle</region>,
         <country>F</country>)
      </placeName>
      <mentioned-in>
         <reg-ref regest="regest_99999">1452-12-26</reg-ref>,
         <reg-ref regest="regest_99999">1453-01-12</reg-ref>,
         <reg-ref regest="regest_99999">1459-02-19</reg-ref>
      </mentioned-in> siehe
      <index-refs>
         <index-ref itemid="item_534">Lenterdingen</index-ref>,
         <index-ref itemid="item_955"> Volkersweiler</index-ref>
      </index-refs>
   </location-header>
   <concept-body>
      <related-concepts>
         <concept>
            <name> Beutedepot </name>
            <mentioned-in>
               <reg-ref regest="regest_99999">1460-05-04</reg-ref>
            </mentioned-in>
         </concept>
      </related-concepts>
   </concept-body>
</item>

<item id="item_23" type="location" value="Arschofen">
   <location-header>
      <placeName>
         <settlement type="Dorf" w="true" w-ref="Staerk, Wüstungen Nr. 19">
            Arschofen
         </settlement>
         <addNames> (
            <addName>Arßhoffen</addName>)
         </addNames>, Dorf
         <reference-point>im Köllertal</reference-point> (Wüstung,
```

```
         <district>Gde. Gersweiler, Stadtverband Sb.</district>,
         <region type="Bundesland">SL</region>; Staerk, Wüstungen Nr. 19)
      </placeName>
   </location-header>
   <concept-body>
      <related-concepts>
         <concept>
            <name> Zins </name>
            <mentioned-in>
               <reg-ref regest="regest_99999">1441-11-08</reg-ref>,
               <reg-ref regest="regest_99999">1512-01-12</reg-ref>
            </mentioned-in>
         </concept>
      </related-concepts>
   </concept-body>
</item>
```

B.2 Landmarks

```
<item id="item_770" type="landmark" value="Rossel">
    <landmark-header>
        <geogName type="Fluss">
            <name>Rossel </name>
            <addNames>(
                <addName>Russel</addName>)
            </addNames>
        </geogName>, Fluss
    </landmark-header>
    <concept-body>
        <related-concepts>
            <concept>
                <name> Fischerei</name>
                <description>, </description>
                <mentioned-in>
                    <reg-ref regest="regest_99999">1444-12-03 (a)</reg-ref>
                </mentioned-in>
            </concept>
        </related-concepts>
    </concept-body>
</item>
```

B.3 Persons

```
<item id="item_441" type="person" value="Johann I.">
    <person-header>
        <person id="person_412">
            <persName>
```

```
                <forename>Johann</forename>
                <genName>I.</genName>,
                <roleName>Graf</roleName>
            </persName>
            <description> von Saarbrücken-Commercy (1307-1341),
                        oo Mathilde von Apremont </description>
        </person>
        <mentioned-in>
            <reg-ref regest="regest_99999">1310</reg-ref>,
            <reg-ref regest="regest_99999">1310-10-21</reg-ref>,
            <reg-ref regest="regest_99999">1310-12-01</reg-ref>,
            <reg-ref regest="regest_99999">1313-02-23</reg-ref>,
            <reg-ref regest="regest_99999">1313-06-03</reg-ref>,
            <reg-ref regest="regest_99999">1313-08</reg-ref>,
            <reg-ref regest="regest_99999">1316</reg-ref>,
            <reg-ref regest="regest_99999">1316-03-20</reg-ref>,
            <reg-ref regest="regest_99999">1316-04-07</reg-ref>,
            <reg-ref regest="regest_99999">1318-04-20</reg-ref>,
            <reg-ref regest="regest_99999">1319-10-03</reg-ref>,
            <reg-ref regest="regest_99999">1322-03</reg-ref>,
            <reg-ref regest="regest_99999">1322-08-17</reg-ref>,
            <reg-ref regest="regest_99999">1322-12-15</reg-ref>,
            <reg-ref regest="regest_99999">1324-03-20</reg-ref>,
            <reg-ref regest="regest_99999">1324-04-08</reg-ref>,
            <reg-ref regest="regest_99999">1324-08-06</reg-ref>,
            <reg-ref regest="regest_99999">1324-08-10</reg-ref>,
            <reg-ref regest="regest_99999">1325-03-30</reg-ref>,
            <reg-ref regest="regest_99999">1327-04-23</reg-ref>,
            <reg-ref regest="regest_99999">1328-02-02</reg-ref>,
            <reg-ref regest="regest_99999">1328-02-24</reg-ref>,
            <reg-ref regest="regest_99999">1332-11-06</reg-ref>,
            <reg-ref regest="regest_99999">1333-12-11</reg-ref>,
            <reg-ref regest="regest_99999">1334-10-03</reg-ref>,
            <reg-ref regest="regest_99999">1334-10-14</reg-ref>,
            <reg-ref regest="regest_99999">1335-04-06</reg-ref>,
            <reg-ref regest="regest_99999">1336-01-02</reg-ref>,
            <reg-ref regest="regest_99999">1336-03-21</reg-ref>,
            <reg-ref regest="regest_99999">1337-03-29</reg-ref>,
            <reg-ref regest="regest_99999">1337-12-05</reg-ref>,
            <reg-ref regest="regest_99999">1339-02-08</reg-ref>
        </mentioned-in>
    </person-header>
    <concept-body></concept-body>
</item>
```

B.4 Families

```
<item id="item_81" type="family" value="Blankenberg">
   <family-header>
      <family-name>
         <name>Blankenberg</name>
         <addNames> (
            <addName>Blanckemberg</addName>)
         </addNames>
       </family-name>, Familie von
    </family-header>
    <listing-body>
       <members>
          <person id="person_64">
             <persName>
                <forename> Ulrich</forename>
             </persName>
             <description>, Gubernator des Stiftes Metz </description>
             <mentioned-in>
                <reg-ref regest="regest_99999">1474-05-26</reg-ref>,
                <reg-ref regest="regest_99999">1474-06-03</reg-ref>
             </mentioned-in>
          </person>
          <person id="person_65">
             <persName>
                <forename> Johann </forename>
             </persName>
             <mentioned-in>
                <reg-ref regest="regest_99999">1480-05-20 (kurz nach)</reg-ref>
             </mentioned-in>
          </person>
       </members>
    </listing-body>
 </item>
```

B.5 Persongroups

```
<item id="item_661" type="persongroup" value="Notare">
    <persongroup-header>
        <group-name>Notare </group-name>
    </persongroup-header>
    <listing-body>
        <members>
            <person id="person_659">
                <persName>
                    <forename> Nikolaus Wolff</forename>
                </persName>
                <description>
```

```
                , Notar in Zweibrücken und Kanoniker in St. Arnual
            </description>
            <mentioned-in>
                <reg-ref regest="regest_99999">1372-02-12</reg-ref>
            </mentioned-in>
        </person>
        <person id="person_660">
            <persName>
                <forename> Johannes</forename>
            </persName>
            <description>, Notar in Metz </description>
            <mentioned-in>
                <reg-ref regest="regest_99999">1415-11-04</reg-ref>
            </mentioned-in>
        </person>
    </members>
</listing-body>
</item>
```

C Regest Title Patterns

This appendix lists regest title patterns that the `RegestTitleAnalyzer` and `RegestDateExtractor` classes can understand.

```
1009
1009-10
1009-10-20

1009 Diedenhofen
1009-10 Frankfurt am Main
1009-10-20 St. Arnual

1009 (vor)
1009-10 (kurz nach)
1009-10-20 (ca.)
1009 (ca. Mitte 15. Jh.)

1009 (?)
1009-10 (?)
1009-10-20 (?)

1009 (vor) Diedenhofen
1009-10 (nach) Frankfurt am Main
1009-10-20 (um) St. Arnual
1009 Diedenhofen (kurz nach)
1009-10 Frankfurt am Main (post)
1009-10-20 St. Arnual (ca.)
1009-10 St. Arnual (ca. Mitte 15. Jh.)
1009-10 (ca. Mitte 15. Jh.) St. Arnual
1507-12-27 (?) Diedenhofen
1507-12-27 Diedenhofen (?)

1273 (a)
1270-07 (b)
1377-03-05 (c)
1424-06-03 (a) und (b)

1442 (a) Diedenhofen
1270-07 (b) Frankfurt am Main
1354-04-01 (c) St. Arnual
1424-06-03 (a) und (b) Tull

1200 (vor) (a)
1200 (a) (vor)
1200-03 (kurz nach) (b)
1200-03 (b) (kurz nach)
```

```
1200-03-12 (ca.) (c)
1200-03-12 (c) (ca.)
1009-10-20 (ca. Mitte 15. Jh.) (d)
1009-10-20 (d) (ca. Mitte 15. Jh.)

1024-1030
1024 - 1030
1419-05 - 1419-06
1482-07-16 - 1499-01-03

0935-1000 (ca.)
1431 - 1459 (zwischen)
1419-05 - 1419-06 (um)
1419-05 (nach) - 1419-06 (vor)
1482-07-16 - 1499-01-08 (ca.)
1482-07-16 (nach) - 1499-01-08 (vor)

1460-1466 (a) ca.
1460-1466 (ca.) (b)
1460-1466 (c) (ca.)
1419-05 - 1419-06 (a) um
1419-05 - 1419-06 (um) (b)
1419-05 - 1419-06 (c) (um)
1482-07-16 - 1499-01-08 (a) ca.
1482-07-16 - 1499-01-08 (ca.) (b)
1482-07-16 - 1499-01-08 (c) (ca.)
1482-07-16 (nach) - 1499-01-08 (vor) (a)

1419-05 bis 06
1419-05-10 bis 20

1419-05 bis 06 (kurz nach)
1419-05-10 bis 20 (ca.)
1419-05-10 bis 06-20 (ca.)
1419-05 (nach) bis 06 (vor)
1419-05-10 (nach) bis 20 (vor)
1419-05-10 (nach) bis 06-20 (vor)

1419-05 bis 06 (a) kurz nach
1419-05 bis 06 (kurz nach) (b)
1419-05 bis 06 (c) (kurz nach)
1419-05-10 bis 20 (a) ca.
1419-05-10 bis 20 (ca.) (b)
1419-05-10 bis 20 (c) (ca.)

1524/1525
1524 / 1525
```

1419-05/1419-06
1419-05 / 1419-06
1421-10-05/1422-10-04
1421-10-05 / 1422-10-04
1502 (1503)
1502-11 (1503-02)
1502-11-22 (1503-02-07)
1520 [bzw. 1519]
1520-02 [bzw. 1519-03]
1520-02-18 [bzw. 1519-03-06]
1520 bzw. 1519
1520-02 bzw. 1519-03
1520-02-18 bzw. 1519-03-06
1520 bzw. 1519 bzw. 1518
1520-02 bzw. 1519-03 bzw. 1518-04
1520-02-18 bzw. 1519-03-06 bzw. 1518-04-23

1524/1525 Diedenhofen
1524 / 1525 Frankfurt am Main
1419-05/1419-06 St. Arnual
1419-05 / 1419-06 Diedenhofen
1421-10-05/1422-10-04 Frankfurt am Main
1421-10-05 / 1422-10-04 St. Arnual
1502 (1503) Diedenhofen
1502-11 (1503-02) Frankfurt am Main
1502-11-22 (1503-02-07) St. Arnual
1520 [bzw. 1519] Diedenhofen
1520-02 [bzw. 1519-03] Frankfurt am Main
1520-02-18 [bzw. 1519-03-06] St. Arnual
1520 bzw. 1519 Diedenhofen
1520-02 bzw. 1519-03 Frankfurt am Main
1520-02-18 bzw. 1519-03-06 St. Arnual
1520 bzw. 1519 bzw. 1518 Diedenhofen
1520-02 bzw. 1519-03 bzw. 1518-04 Frankfurt am Main
1520-02-18 bzw. 1519-03-06 bzw. 1518-04-23 St. Arnual

1524/1525 (a)
1524 / 1525 (b)
1419-05/1419-06 (c)
1419-05 / 1419-06 (a)
1421-10-05/1422-10-04 (b)
1421-10-05 / 1422-10-04 (c)
1502 (1503) (a)
1502-11 (1503-02) (b)
1502-11-22 (1503-02-07) (c)
1520 [bzw. 1519] (a)
1520-02 [bzw. 1519-03] (b)

1520-02-18 [bzw. 1519-03-06] (c)
1520 bzw. 1519 (a)
1520-02 bzw. 1519-03 (b)
1520-02-18 bzw. 1519-03-06 (c)
1520 bzw. 1519 bzw. 1518 (a)
1520-02 bzw. 1519-03 bzw. 1518-04 (b)
1520-02-18 bzw. 1519-03-06 bzw. 1518-04-23 (c)

1524/1525 (a) Diedenhofen
1524 / 1525 (b) Frankfurt am Main
1419-05/1419-06 (c) St. Arnual
1419-05 / 1419-06 (a) Diedenhofen
1421-10-05/1422-10-04 (b) Frankfurt am Main
1421-10-05 / 1422-10-04 (c) St. Arnual
1502 (1503) (a) Diedenhofen
1502-11 (1503-02) (b) Frankfurt am Main
1502-11-22 (1503-02-07) (c) St. Arnual
1520 [bzw. 1519] (a) Diedenhofen
1520-02 [bzw. 1519-03] (b) Frankfurt am Main
1520-02-18 [bzw. 1519-03-06] (c) St. Arnual
1520 bzw. 1519 (a) Diedenhofen
1520-02 bzw. 1519-03 (b) Frankfurt am Main
1520-02-18 bzw. 1519-03-06 (c) St. Arnual
1520 bzw. 1519 bzw. 1518 (a) Diedenhofen
1520-02 bzw. 1519-03 bzw. 1518-04 (b) Frankfurt am Main
1520-02-18 bzw. 1519-03-06 bzw. 1518-04-23 (c) St. Arnual

1524/1525 (vor)
1524 / 1525 (nach)
1419-05/1419-06 (um)
1419-05 / 1419-06 (ca.)
1421-10-05/1422-10-04 (kurz nach)
1421-10-05 / 1422-10-04 (post)
1502 (1503) (vor)
1502-11 (1503-02) (nach)
1502-11-22 (1503-02-07) (um)
1520 [bzw. 1519] (ca.)
1520-02 [bzw. 1519-03] (kurz nach)
1520-02-18 [bzw. 1519-03-06] (post)
1520 bzw. 1519 (um)
1520-02 bzw. 1519-03 (ca.)
1520-02-18 bzw. 1519-03-06 (kurz nach)
1520 bzw. 1519 bzw. 1518 (um)
1520-02 bzw. 1519-03 bzw. 1518-04 (ca.)
1520-02-18 bzw. 1519-03-06 bzw. 1518-04-23 (kurz nach)

1270-04/05

1270-04 / 05
1440-11-12/17
1440-11-12 / 17
1270-04-27/05-28
1270-04-27 / 05-28

1466 [04/05]
1466 [04 / 05]
1466-04 [28/29]
1466-04 [28 / 29]
1466 [04-28/05-01]
1466 [04-28 / 05-01]

1506-05 bzw. 11
1506-05-12 bzw. 10
1506-05-12 bzw. 11-10
1506-05 bzw. 11 bzw. 12
1506-05-12 bzw. 10 bzw. 01
1506-05-12 bzw. 11-10 bzw. 12-01

1343-04 oder 05
1343-04-12 oder 19
1343-04-12 oder 05-19

1270-04/05 Diedenhofen
1270-04 / 05 Frankfurt am Main
1440-11-12/17 St. Arnual
1440-11-12 / 17 Diedenhofen
1270-04-27/05-28 Frankfurt am Main
1270-04-27 / 05-28 St. Arnual

1466 [04/05] Diedenhofen
1466 [04 / 05] Frankfurt am Main
1466-04 [28/29] St. Arnual
1466-04 [28 / 29] Diedenhofen
1466 [04-28/05-01] Frankfurt am Main
1466 [04-28 / 05-01] St. Arnual

1506-05 bzw. 11 Diedenhofen
1506-05-12 bzw. 10 Frankfurt am Main
1506-05-12 bzw. 11-10 St. Arnual
1506-05 bzw. 11 bzw. 12 Diedenhofen
1506-05-12 bzw. 10 bzw. 01 Frankfurt am Main
1506-05-12 bzw. 11-10 bzw. 12-01 St. Arnual

1343-04 oder 05 Diedenhofen
1343-04-12 oder 19 Frankfurt am Main

1343-04-12 oder 05-19 St. Arnual

1270-04/05 (a)
1270-04 / 05 (b)
1440-11-12/17 (c)
1440-11-12 / 17 (a)
1270-04-27/05-28 (b)
1270-04-27 / 05-28 (c)

1466 [04/05] (a)
1466 [04 / 05] (b)
1466-04 [28/29] (c)
1466-04 [28 / 29] (a)
1466 [04-28/05-01] (b)
1466 [04-28 / 05-01] (c)

1506-05 bzw. 11 (a)
1506-05-12 bzw. 10 (b)
1506-05-12 bzw. 11-10 (c)
1506-05 bzw. 11 bzw. 12 (a)
1506-05-12 bzw. 10 bzw. 01 (b)
1506-05-12 bzw. 11-10 bzw. 12-01 (c)

1343-04 oder 05 (a)
1343-04-12 oder 19 (b)
1343-04-12 oder 05-19 (c)

1270-04/05 (a) Diedenhofen
1270-04 / 05 (b) Frankfurt am Main
1440-11-12/17 (c) St. Arnual
1440-11-12 / 17 (a) Diedenhofen
1270-04-27/05-28 (b) Frankfurt am Main
1270-04-27 / 05-28 (c) St. Arnual

1466 [04/05] (a) Diedenhofen
1466 [04 / 05] (b) Frankfurt am Main
1466-04 [28/29] (c) St. Arnual
1466-04 [28 / 29] (a) Diedenhofen
1466 [04-28/05-01] (b) Frankfurt am Main
1466 [04-28 / 05-01] (c) St. Arnual

1506-05 bzw. 11 (a) Diedenhofen
1506-05-12 bzw. 10 (b) Frankfurt am Main
1506-05-12 bzw. 11-10 (c) St. Arnual
1506-05 bzw. 11 bzw. 12 (a) Diedenhofen
1506-05-12 bzw. 10 bzw. 01 (b) Frankfurt am Main
1506-05-12 bzw. 11-10 bzw. 12-01 (c) St. Arnual

1343-04 oder 05 (a) Diedenhofen
1343-04-12 oder 19 (b) Frankfurt am Main
1343-04-12 oder 05-19 (c) St. Arnual

1270-04/05 (vor)
1270-04 / 05 (nach)
1440-11-12/17 (um)
1440-11-12 / 17 (ca.)
1270-04-27/05-28 (kurz nach)
1270-04-27 / 05-28 (post)

1466 [04/05] (vor)
1466 [04 / 05] (nach)
1466-04 [28/29] (um)
1466-04 [28 / 29] (ca.)
1466 [04-28/05-01] (kurz nach)
1466 [04-28 / 05-01] (post)

1506-05 bzw. 11 (vor)
1506-05-12 bzw. 10 (nach)
1506-05-12 bzw. 11-10 (um)
1506-05 bzw. 11 bzw. 12 (ca.)
1506-05-12 bzw. 10 bzw. 01 (kurz nach)
1506-05-12 bzw. 11-10 bzw. 12-01 (post)

1343-04 oder 05 (um)
1343-04-12 oder 19 (ca.)
1343-04-12 oder 05-19 (kurz nach)

1524 und 1525
1419-05 und 1419-06
1421-10-05 und 1422-10-04

1524 und 1525 Diedenhofen
1419-05 und 1419-06 Frankfurt am Main
1421-10-05 und 1422-10-04 St. Arnual

1524 und 1525 (a)
1419-05 und 1419-06 (b)
1421-10-05 und 1422-10-04 (c)

1524 und 1525 (a) Diedenhofen
1419-05 und 1419-06 (b) Frankfurt am Main
1421-10-05 und 1422-10-04 (c) St. Arnual

1524 und 1525 (um)

```
1419-05 und 1419-06 (ca.)
1421-10-05 und 1422-10-04 (kurz nach)

1270-04 und 05       (month different, no day)
1440-11-12 und 17    (day different)
1270-04-27 und 05-28 (month *and* day different)

1270-04 und 05 Diedenhofen
1440-11-12 und 17 Frankfurt am Main
1270-04-27 und 05-28 St. Arnual

1270-04 und 05 (a)
1440-11-12 und 17 (b)
1270-04-27 und 05-28 (c)

1270-04 und 05 (a) Diedenhofen
1440-11-12 und 17 (b) Frankfurt am Main
1270-04-27 und 05-28 (c) St. Arnual

1270-04 und 05 (um)
1440-11-12 und 17 (ca.)
1270-04-27 und 05-28 (kurz nach)

1337-12-
1400 (15. Jh., Anfang)
1419-10-20 (und oefter)
1500 (a) (16. Jh., Anfang)
1500 (e) (15. Jh., Ende) Saarbruecken
1419-05 bis 06 (Mai bis Juli)
1024-1030 (ohne Datum)
```