Universitätsstraße 38
70569 Stuttgart
Germany

System Description

Damast

v1.1.3

Max Franke

May 23, 2022

Institute for Visualization and Interactive Systems
University of Stuttgart
Germany

Contents

1. Introduction

Damast is a comprehensive visual analysis system for the collection, analysis, and export of historical data regarding peaceful coexistence of religious groups. Damast was developed for the digital humanities project "Dhimmis & Muslims — Analysing Multireligious Spaces in the Medieval Muslim World". The project was funded by the VolkswagenFoundation within the scope of the "Mixed Methods" initiative. The project was a collaboration between the Institute for Medieval History II of the Goethe University in Frankfurt/Main, Germany, and the Institute for Visualization and Interactive Systems at the University of Stuttgart, and took place there from 2018 to 2021.

The objective of this joint project was to develop a novel visualization approach in order to gain new insights on the multi-religious landscapes of the Middle East under Muslim rule during the Middle Ages (7th to 14th century). In particular, information on multi-religious communities were researched and made available in a database accessible through interactive visualization as well as through a pilot web-based geo-temporal multi-view system to analyze and compare information from multiple sources. A publicly explorable version[1] of the research is available at Humboldt-Universität zu Berlin. An export of the data collected in the project can be found in the data repository of the University of Stuttgart (DaRUS) [4].

Damast consists of a Flask server backend, which also communicates with a PostgreSQL database, as well as a web-based frontend. The frontend consists of multiple pages with various functions, including: an interactive visual analysis component, a table-based database editing interface, a document-based annotation interface for data entry, and various smaller utilities. Damast also supports generating textual reports based on subsets of the historical data.

The rest of this document is structured as follows: The structure and contents of the main PostgreSQL database, as well as the user and report databases, are discussed in chapter 2; the structure and functionalities of the server backend are discussed in chapter 3; and the various front-end facilities are discussed in chapter 4.

[1] `https://damast.geschichte.hu-berlin.de/`

2. Database

The main database is a PostgreSQL 10 database. Additionally, the PostGIS plugin is installed into the database. An easy way to obtain a base database system into which the schema can just be imported is to use the `postgis/postgis:10-3.1` Docker image.

2.1. Table Structure

Figure 1 shows the schematic structure of the PostgreSQL database. Tables are represented as boxes consisting of three parts, with the table name in the first part, the primary key (if it exists) in the second part, and the remaining columns in the third part. Foreign key references are indicated by $\diamond\!\!\longrightarrow$ arrows with a diamond at the starting point. Weak references are indicated with $\circ\!\!\longrightarrow$ open circles at the starting point, and one-to-many references[2] $*\!\!\longrightarrow$ are indicated with six rays at the starting point. The tables are grouped by function. The individual groups and tables are described in more detail in sections 2.1.1 and 2.1.2.

2.1.1. Data Tables

Table 1.: Confidence values.

Value	Comment
`false`	We know the information is not true.
`uncertain`	We are mostly convinced that the information is not true.
`contested`	We are unsure whether the information is true.
`probable`	We are mostly convinced that the information is true.
`certain`	We are as sure as one can be with historical information that the information is true.

Data types. Most data columns have standard PostgreSQL data types such as `text`, `integer`, or `boolean`. Time and text ranges are stored as the PostgreSQL `int4range` type, and geographical locations are stored in the PostgreSQL `point` type. To qualify the historical data, most tables also have a *confidence* column, which contains a `NULL`-able confidence value. These are stored as an `enum`, the contents of which are explained in table 1. For more details on our choice of levels of confidence and the confidence data model, please refer to our 2019 publication[3]. The aspects of confidence, and where they are stored, are explained in table 2. To accommodate for unstructured metadata and notes, most tables also have a `NULL`-able `comment` column.

Base data. Places, or cities, are stored in the `place` table. This contains the primary name for that place, its geographical location, if known, and a boolean flag indicating whether the place should be included in the visualization. A place also has a place type (stored in the `place_type` table). Place types also have a visibility flag, which affects the visibility of all places with that type in the visualization. Alternative names for places are stored as entries in the `name_var` table. An alternative name has a primary name, an

[2] This type of one-to-many references are realized with PostgreSQL arrays, and are weak references. Strong one-to-many references in relational databases are handled via intermediary tables, which is done, for example, with the `tag_evidence` or `time_group` table (although the latter, and all instance tables, serve additional purposes).

[3] Max Franke et al. "Confidence as First-class Attribute in Digital Humanities Data". In: *Proceedings of the 4^{th} VIS4DH Workshop* (Oct. 2019). URL: `http://vis4dh.dbvis.de/papers/2019/VIS4DH2019_paper_1.pdf`.

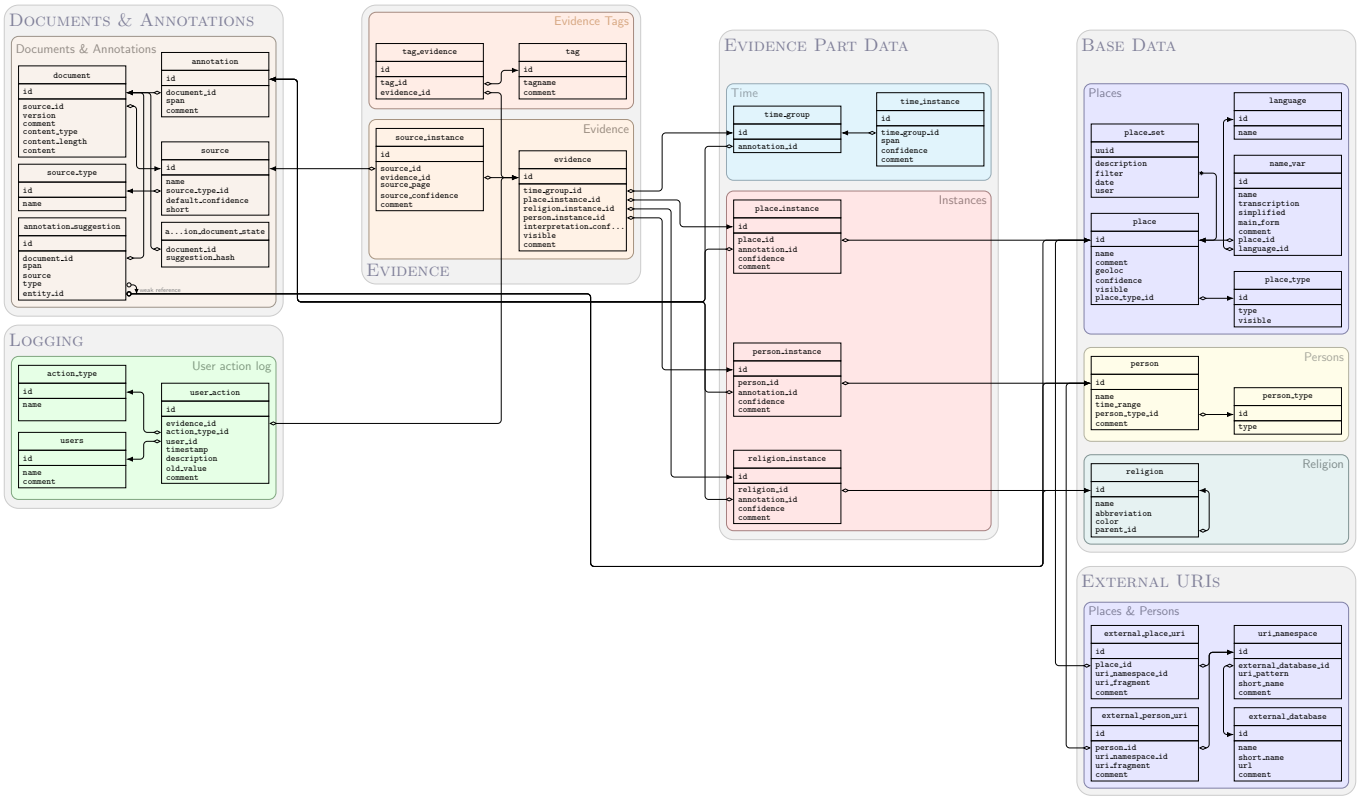

Figure 1.: Schematic structure of the PostgreSQL database. The tables are represented by boxes, which in turn are grouped by function. Relationships between tables are indicated by arrows.

optional transcription of the name (for example, if the name is in Arabic script), and optionally one or more simplified forms that can be used for full-text search. For an alternative name, we also store which language (stored in the `language` table) it is a name in, and whether it is a main form of the name that should be displayed to visitors. Persons are stored in the `person` table. They have a type (stored in the `person_type` table). We also store a time range for persons, which is a free-text field. The name combined with the time range must be unique; for instance, there can be multiple persons with the name *"Marcus"*, but each must have a different time range; which could, for example, consist of a year range, or a qualifier such as "the Third." Finally, religions are stored in the `religion` table, with their name, abbreviation, and color used in the visualization. Religions are hierarchical, so a religion can optionally reference a parent religion.

External URIs. Part of the data collection effort was put into aligning our base data entities with those of other databases for historical data, such as Syriaca.org, the Digital Atlas of the Roman Empire (DARE), or Pleiades. Such external databases are stored in the `external_database` table with a long and short name, a URL, and a comment. Namespaces for URIs in those databases are then stored in the `uri_namespace` table, with a reference to the external database entry, a name, and a comment. The URI namespace entry further contains a *URI pattern,* which is a `printf(3)`-style string with one `%s` placeholder. External URI references for places and persons are then stored in the `external_place_uri` and `external_person_uri` tables. These reference the respective base datum and the URI namespace the URI is based on, and contain a comment. They further contain a *URI fragment*, which is the part of the URI that is represented by the placeholder in the URI namespace entry. For example, the place *Edessa* on Syriaca.org[4] could be represented by a URI namespace with the URI pattern `http://syriaca.org/place/%s` and a external place URI entry with the URI fragment `78`. The structuring of these tables means that (1) an external database can have

[4] `http://syriaca.org/place/78`

Table 2.: Aspects of confidence.

Aspect	Table	Description
Religion confidence	`religion_instance`	How confident are we that this is the correct religion for the evidence?
Person confidence	`person_instance`	How confident are we that this is the correct person for the evidence?
Location confidence	`place`	How confident are we about the geographical location of this place?
Place attribution confidence	`place_instance`	How confident are we that this is the correct place for the evidence?
Time confidence	`time_instance`	How confident are we that this is the correct time span for the evidence?
Interpretation confidence	`evidence`	How confident are we in the general interpretation of this evidence?
Source confidence	`source_instance`	How confident are we that the information about the evidence we extracted from that source is correct?

more than one URI namespace, (2) only storing the fragment in the URI references reduces data entry errors, and (3) it is easier to update all URIs of a database in case it moves hosts[5].

Documents and annotations. Historical sources are stored in the `source` table with a long and short version of their name, where the long version is a proper citation. A source also has a source type, which is a reference to an entry in the `source_type` table, as well as a default interpretation confidence value, which is suggested for new source instances (see paragraph "Evidence" on p. 7). Digital versions of sources can be stored in the `document` table. A document references its source, has a version number, a comment, content type and content length fields, and the content itself, which is a byte array. Documents can be annotated, and these annotations are then stored in the `annotation` table with a reference to the document, the start and end position of the annotation as an `int4range` range, and a comment. Annotations are used in evidence parts (see paragraph "Evidence part data" on p. 6), and based on data already in the database, new annotations are suggested (see also section 4.4). These are stored in the `annotation_suggestion` table with some metadata, alongside a *weak reference* to the base datum (place, person, or religion) they refer to. The weak reference consists of an ID and a type string, which can be one of `'place'`, `'person'`, or `'religion'`. Because recalculating the suggestions is expensive, intermediate hashes that only change if there might be new suggestion results are stored for each document in the `annotation_suggestion_document_state` table, and the annotation suggestions are only recalculated if the calculated hashes do not match the stored ones.

Evidence part data. For each piece of historical evidence, *instances* of the base data are created. An instance contains a reference to the respective datum, a confidence value, and a comment. Additionally, it may contain a reference to an annotation. Instances for places, persons, and religions are stored in the `place_instance`, `person_instance`, and `religion_instance` tables, respectively. Time spans are grouped via the `time_group` table, which consists of only an ID column, and an optional reference to an annotation. Individual time spans are stored in the `time_instance` table, with a confidence value, a comment, and a reference to the time group. The time span itself is stored as an `int4range`; that is, time is stored in years.

[5]This happened with DARE in 2019, when the old site at `http://dare.ht.lu.se` was deactivated and moved to `https://dh.gu.se/dare/`. The new site, however, was not as functional. The creator Johan Åhlfeldt then re-hosted everything at `https://imperium.ahlfeldt.se`, but all already-entered URIs had to be updated.

Evidence. Pieces of historical evidence are stored in the `evidence` table. Evidence consists of a time group, a place instance, a religion instance, and optionally a person instance. Instances may be part of multiple evidences, which is a typical use case with the annotation system (see section 4.4), but are often connected to only one evidence. Each evidence tuple also has a visibility flag, as well as a confidence value and a comment. Evidence tuples can be *tagged* with zero to many tags to categorize them. Tags are stored in the `tag` table, with a short name and a comment. Evidence is attributed to tags via entries in the `tag_evidence` table. Evidence can also be attributed to zero to many sources via the `source_instance` table. Each source instance references a source and an evidence and also contains a free-text page reference in the source, the interpretation confidence value for that evidence, and a comment.

2.1.2. Provenance

For provenance, data edits relating to evidence tuples are recorded in the `user_action` table. This stores a reference to the evidence and the type of action (a reference to an entry in the `action_type` table, but typically one of CREATE, UPDATE, or DELETE), alongside a timestamp, a short description of the action, and the old data entry value before the change, if it exists. Further, the user that did the change is recorded, which is a reference to an entry in the `users` table. This table needs to contain an entry for each user that can log in to the front-end and has the `writedb` role (see also section 3.2).

2.2. Roles

Each user has a set of roles they are part of. Individual Flask endpoints in turn are set to allow a certain set of roles. A user is then allowed to use an endpoint if these two sets of roles overlap.

There is a basic `user` role that gives access to basic functionality behind the login, but just being part of that role is not enough if the endpoint wants a more specific role. Further, there is a `dev` role for endpoints that are important for developing, but not for users, such as the REST API documentation. Finally, there is an `admin` role, which should provide access to *all* endpoints.

For the REST API, there is the distinction between users without REST API access, those that can read, and those that can read and write. For this, there are two roles, `readdb` and `writedb`, and a user has either none, only `readdb`, or `readdb` and `writedb`. `writedb` provides access to changing endpoints in the REST API, which will modify database state, while `readdb` allows to query, but not modify. In the backend, this is controlled mainly by differentiating HTTP verbs: `GET` requests[6] are read-only, `PATCH`, `PUT` and `DELETE` are writing[7]. Read-only users are also passed a read-only cursor to the endpoint code.

For different functionalities of the site, there are more specific roles, where users can just be part of one or a few:

`annotator` This role grants the user access to the annotator. The user however requires at least the `readdb` role to be able to see the annotator interface, populated with data, properly, as the annotations and documents are loaded via the REST API. If the user has the `writedb` role as well, they can annotate and modify annotations and evidences.

`geodb` This role grants the user access to the GeoDB-Editor. Similarly to the annotator, the `readdb` role is required as well to see the contents of the tables, and the `writedb` role is required to make changes.

`pgadmin` This role is historical from when the pgAdmin4 server was still reverse-proxied behind the Damast server. Now (if there is one on the host system at all), the pgAdmin4 server is reverse-proxied by the

[6] `HEAD` requests, which are handled similarly to `GET` requests by most servers, are also read-only.

[7] The default behavior is to distinguish the HTTP verbs that way. However, the method decorator that provides the database cursor and distinguishes between read-only access and writing access (see `rest_endpoint` in `damast/postgres_rest_api/decorators.py`) can take an optional argument controlling which verbs are considered read-only. This is used for the `/rest/find-alternative-names` endpoint, which requires a payload and therefore is a `POST` request, but can be used with only the `readdb` role.

```
CREATE TABLE users (
  id          TEXT PRIMARY KEY NOT NULL CHECK(id <> 'visitor'),
  password    TEXT NOT NULL,
  expires     DATE DEFAULT NULL,
  roles       TEXT NOT NULL DEFAULT '',
  comment     TEXT DEFAULT NULL
);
```

Listing 1: The database schema of the SQLite3 user database.

web-facing reverse proxy server (e.g., NGINX, see fig. 2). If the user has the `pgadmin` role, a link to `${DAMAST_PROXYPREFIX}/pgadmin/` is shown in the header.

`reporting` This role grants the user access to the reporting functionality. Because report generation requires database access, the `readdb` role is needed as well. In the report list, users with the `reporting` role can only see the reports they generated themselves[8]. Administrators (`admin` role) can see all reports.

`vis` This role grants the user access to the visualization. To populate the visualization with the data from the database, the `readdb` role is required as well.

`visitor` This role is given to visitors alongside those in `${DAMAST_VISITOR_ROLES}`, if visitor handling is enabled. There is not yet any use for that role, but it could be used in future to grant visitors access to an endpoint even if they do not have any of the other roles required for it.

2.3. Other Databases

Beside the main PostgreSQL database, two other databases exist for configuration and storage purposes. These are SQLite3 database files that are placed in the configuration directory (see section 3.1.2). The first is the user database, the second stores reports and their metadata.

2.3.1. User Database

The SQLite3 user database (see listing 1) contains only one table, `users`. The file is stored in the runtime configuration directory (see section 3.1.2), and its location can be configured using the `DAMAST_USER_FILE` environment variable (`/data/users.db` by default).

Each user is listed here as a separate record, with their user name as the unique `id` field. Passwords are stored as `htpasswd(1)` hashes. The backend supports SHA256 (`$5$`), SHA512 (`$6$`) and Blowfish `bcrypt` (`$2b$` and `$2y$`) hashes, but `bcrypt` is preferred. The user's roles are stored in the `roles` field as comma-separated text (for example, `'user,readdb,vis'`). Optionally, each user can have a `comment`.

Account expiry is handled by the `expires` field, where a `YYYY-mm-dd` date string is stored. This uses an extension to SQLite3 for the `DATE` column format. If the field is empty, the user does not expire. If it is filled, the user cannot login from the specified day on, *inclusively.*

2.3.2. Report Database

This SQLite3 database (see listing 2) contains the reports and their metadata (see section 4.6). The file is stored in the runtime configuration directory (see section 3.1.2), and its location can be configured using the `DAMAST_REPORT_FILE` environment variable (`/data/reports.db` by default).

The UUID of the report is used as a primary key. Regarding metadata, the user provisioning the report, the start time, the (possibly empty) end time, the and server version are stored. For the completed report, the number of evidences contained are also stored in the `evidence_count` field.

[8] The exception here being visitors with the `reporting` role. Those cannot see the report list, but only have access to a report they know the UUID of.

```
CREATE TABLE reports (
  uuid TEXT NOT NULL PRIMARY KEY,
  user TEXT NOT NULL,
  started DATETIME NOT NULL,
  completed DATETIME DEFAULT NULL,
  report_state TEXT NOT NULL DEFAULT 'started',
  server_version TEXT NOT NULL,
  filter BLOB DEFAULT NULL,
  content BLOB DEFAULT NULL,
  pdf_map BLOB DEFAULT NULL,
  pdf_report BLOB DEFAULT NULL,
  evidence_count INTEGER NOT NULL DEFAULT 0
);
```

Listing 2: The database schema of the SQLite3 report database.

The state of the report is stored in the `report_state` field. When the report has been provisioned and generation is still under way, this is `'started'`. If an error occurs during report generation, the value is `'failed'`, otherwise it is `'completed'`.

The report contents themselves are stored in four fields. These fields are of the datatype `BLOB`, meaning SQLite3 stores them as a bytestring. Each of the fields' contents is GZIP-compressed. The `filter` field contains the filter and additional metadata that were used for the report generation. The format of this is a subset of the visualization state (see section 4.2.4.1) used for report generation (see section 4.6.1). The content of the HTML report (see section 4.6.3) is stored in the `content` field. The content of the PDF report, which is produced using LaTeX during report generation (see section 4.6.4), is stored in the `pdf_report` field. A PDF version of the map shown both in the HTML and PDF reports is stored separately in the `pdf_map` field so it can be included in other documents more easily.

2.4. Backups

Backups need to be arranged for from outside of the host system. The extent of backups depends on how the Damast system is used: If only the visualization and reporting functionalities are used, and no data entry is performed, it is not necessary to back up the PostgreSQL database, except for its initial state. If data entry is done, regular backups are advised. For the PostgreSQL database, `pg_dump(1)` can be used for regular, full dumps of the database, or `pg_rewind(1)` for duplication to a secondary database instance.

If user management is used (as opposed to mainly having visitors without login), backups of the user SQLite3 database (section 2.3.1) are also advisable, otherwise password changes are lost on data loss. Similarly, the report SQLite3 database (section 2.3.2) should be backed up if persistent reports are wanted. For the SQLite3 database, any type of backup can be used that can handle either file system backups or text dumps from databases. The other contents of the runtime configuration directory (section 3.1.2) need not be regularly backed up. However, it is advisable to keep an initial backup of the configuration.

3. Backend Structure

The server backend consists of a PostgreSQL database, discussed in chapter 2, and a Flask server. The rest of this chapter specifies the host system requirements, and the configuration and maintenance of the Flask server.

3.1. Server Host Specifications

In the usual setup, the PostgreSQL database runs on the same host as the Damast server. The host machine should therefore have resources to accommodate both, ideally at least two CPU cores, 8GB of RAM, and 200GB of hard drive space. The Damast server is deployed as a Docker image to encapsulate and pin down the dependencies. Hence, the host server should have Docker installed, and the Docker daemon running. It should be noted that the PostgreSQL server can also be easily set up using Docker, see chapter 2. The Damast server does not handle SSL encryption, so it is sensible to put a reverse proxy in front of the server which handles SSL, for example nginx.

3.1.1. Infrastructure

Figure 2 shows the structure of the backend on the host server. Here, running the PostgreSQL server in a Docker container as well is assumed. The Damast docker container expects an internal `/data` directory to exist, and to be readable and writable to the `www` user in the container. This directory should be mounted to a directory on the host, and contains runtime configuration (section 3.1.2), server logs (section 3.1.3), override blueprints (section 3.3.2), and the user (section 2.3.1) and report (section 2.3.2) databases. The directory (`/data`) in fig. 2) must be readable and writable to the `www` user in the container. To facilitate the mapping of the volume and the access rights, this `www` user should also exist on the host system, and the host directory should have the appropriate rights. The user ID and group ID of that user on the host system must then be passed to Docker when building the Docker image, using the `USER_ID` and `GROUP_ID` environment variables.

The deploy script (`deploy.sh`) in the repository provides more details on how to create the image. The repository also contains the components for the `Dockerfile`, the server run script, the `systemd` service file, and a sample nginx configuration. For development, a variant of the Docker image can also be built that mounts the local `damast/` directory into the Docker image. This way, the local source files and assets can be used while the dependencies pinned in the Docker image are available. This Docker image can be built using the `host.sh` script in the repository with the `-b` flag.

3.1.2. Runtime Configuration

Many aspects of the Damast behavior can be configured. Configuration is possible either via environment variables, some of which are already baked into the Docker image. These can be supplemented or overwritten by passing values to Docker either with the `--env` flag, or by passing a path to an environment variable file with the `--env-file` argument. The sample run script in the repository assumes that a file `docker.env` is present in the runtime configuration directory. Configuration values can also be passed via a JSON file, the path to which (in the Damast Docker container) is passed to Damast with the `DAMAST_CONFIG` environment variable. Table 3 lists the configuration options, their environment variables and JSON keys, as well as their default value and description.

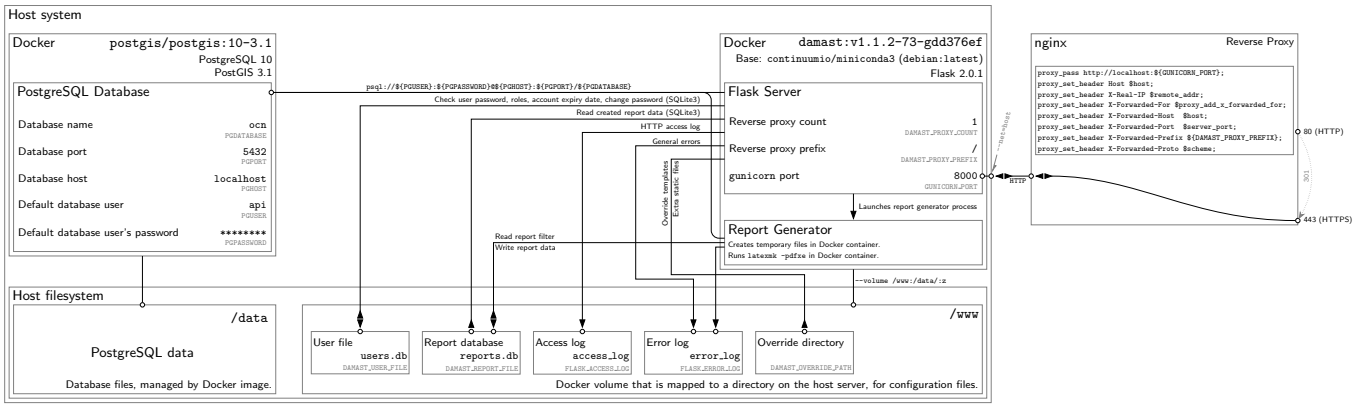

Figure 2.: Backend system structure: The Flask server runs in a Docker container, and a single directory from the host system is exposed to the container as a volume. This directory contains configuration and logging files. The PostgreSQL database can be anywhere, but usually runs on the same host as a Docker image. If the Flask server does not manage its own SSL certificates, an nginx server handles incoming HTTP and HTTPS requests as a reverse proxy.

Table 3.: Configuration options for Damast.

`DAMAST_CONFIG`
`-`
no value
	JSON file to load configuration from. All other configuration values in this table can be passed via this file as key-value entries in the root object, where the key is the "JSON key" column of this table.

`DAMAST_ENVIRONMENT`
`environment`
no value
	Server environment (`PRODUCTION`, `TESTING`, or `PYTEST`). This decides with which PostgreSQL database to connect (`ocn`, `testing`, and `pytest` (on Docker container) respectively. This is usually set via the Docker image.

`DAMAST_VERSION`
`version`
no value
	Software version. This is usually set via the Docker image.

`DAMAST_USER_FILE`
`user_file`
`/data/users.db`
	Path to SQLite3 file with users, passwords, roles.

`DAMAST_REPORT_FILE`
`report_file`
`/data/reports.db`
	File to which reports are stored during generation.

`DAMAST_SECRET_FILE`
`secret_file`
no value
	File with JWT and app secret keys. These are randomly generated if not passed, but that is impractical for testing with hot reload (user sessions do not persist). For a production server, this should be empty.

`DAMAST_PROXYCOUNT`
`proxycount`
`1`
	How many reverse proxies the server is behind. This is necessary for proper HTTP redirection and cookie paths.

Table 3 – *continued from page 11*

Environment variable **JSON key** **Default value**	**Description**
DAMAST_PROXYPREFIX proxyprefix /	Reverse proxy prefix.
DAMAST_OVERRIDE_PATH override_path *no value*	A directory path under which a `template/` and `static/` directory can be placed. Templates within the `template/` directory will be prioritized over the internal ones. This can be used to provide a different template for a certain page, such as the impressum.
DAMAST_VISITOR_ROLES visitor_roles *no value*	If not empty, contains a comma-separated list of roles a visitor (not logged-in) to the site will receive, which in turn governs which pages will be available without user account. If the variable does not exist, visitors will only see public pages.
DAMAST_MAP_STYLES map_styles *no value*	If not empty, a relative filename (under `/data`) on the Docker filesystem to a JSON with map styles. These will be used in the Leaflet map. If not provided, the default styles will be used.
DAMAST_REPORT_EVICTION_DEFERRAL report_eviction_deferral *no value*	If not empty, the number of days of not being accessed before a reports' contents (HTML, PDF, map) are *evicted.* Evicted reports can always be regenerated from their state and filter JSON. Eviction happens to save space and improve performance on systems where many reports are anticipated. *This should not be activated on systems with changing databases!*
DAMAST_REPORT_EVICTION_MAXSIZE report_eviction_maxsize *no value*	If not empty, the file size in megabytes (MB) of report contents (HTML, PDF, map) above which reports will be evicted. If this is set and the sum of content sizes in the report database *after deferral eviction* is above this number, additional reports are evicted until the sum of sizes is lower than this number. Reports are evicted in ascending order of last access date (the least-recently accessed first). The same rules as above apply.
DAMAST_ANNOTATION_SUGGESTION_REBUILD annotation_suggestion_rebuild *no value*	If not empty, the number of days between annotation suggestion rebuilds. In that case, the suggestions are recreated over night every X days. If empty, the annotation suggestions are never recreated, which might be favorable on a system with a static database.
FLASK_ACCESS_LOG access_log /data/access_log	Path to `access_log` (for logging).

Table 3 – *continued from page 11*

Environment variable **JSON key** **Default value**	**Description**
FLASK_ERROR_LOG error_log /data/error_log	Path to error_log (for logging).
DAMAST_PORT port 8000	Port at which gunicorn serves the content. **Note:** This is set via the Dockerfile, and also only used in the Dockerfile.
PGHOST pghost localhost	PostgreSQL hostname.
PGPASSWORD pgpassword *no value*	PostgreSQL password. This is important to set and depends on how the database is set up.
PGPORT pgport 5432	PostgreSQL port
PGUSER pguser api	PostgreSQL user

3.1.3. Logging

HTTP access to the Flask server is logged to an access log file (/data/access_log by default, see table 3). Server errors and miscellaneous information is logged to an error log file (/data/error_log by default). Exceptions in the server are also logged here alongside a UUID, and the UUID is displayed in the HTTP response. This avoids revealing internal functionalities on errors while still allowing to reconstruct errors from an issue with the UUID or a screenshot of the response. Logfiles are rotated daily, and old files (with a .YYYY-mm-dd suffix) kept for ten days. The access log saves the IP address and user name of the user requesting a resource, and also logs the blueprints handling the response.

3.2. User Authentication

Users log in using the login blueprint. On successful login, a JWT token is returned as a cookie, encrypted with the server secret. This cookie must be passed with every subsequent HTTP request. Hence, at least the *necessary cookies* must be allowed by users to be able to log in.

Role-based access. Access to all pages is restricted based on roles, not users. To be able to successfully make an HTTP request to a certain endpoint, the user's roles and the role's allowed for the endpoint-HTTP verb pairing must overlap. For the REST API (section 3.4), the endpoints also distinguish between reading access (with the readdb role, via GET requests), and writing access (with the writedb role, for PUT, PATCH, and DELETE requests).

Visitors. A special *visitor role* can be enabled by setting the DAMAST_VISITOR_ROLES variable (see table 3). If set, this contains a comma-separated list of roles (see section 2.2) that visitors are assigned. These cannot contain writedb, dev, admin, or any roles that do not exist. If this is enabled, the functionalities allowed

Figure 3.: Hierarchical list of blueprints in Damast.

to users with the respective roles will also be available to visitors *without logging in.* For example, setting `DAMAST_VISITOR_ROLES` to `readdb,user,vis` means that visitors can see the start page, the visual analysis component and the data, as well as some utility pages.

3.3. Flask Blueprints

The different functionalities are split up into separate Flask blueprints. Some blueprints, like the REST API (section 3.4), are hierarchically split up into sub-blueprints (see fig. 3).

3.3.1. Templates

HTML pages are created by Flask from Jinja2 templates[9]. Most page templates inherit from a *base template,* which has a header with navigation and user management, a main page area, and a footer with additional links and a copyright statement. The visual analysis component does not inherit from the base template, but instead defines a more compact layout without a footer.

[9] `https://palletsprojects.com/p/jinja/`

```
<style>
  :root {
    --home-bg-image: url({{ url_for('override.static', filename='bg.jpg') }});
  }
</style>
```

Listing 3: The contents of `templates/override/background-image-url.html` when providing a custom background image.

3.3.2. Overriding Templates and Static Files

When running the server, it might be necessary to override some pages; for example, one might want a different home page, or a different impressum. The server uses the `DAMAST_OVERRIDE_PATH` environment variable to load such overrides. If it is set, the Flask server creates an extra blueprint, and the directory `${DAMAST_OVERRIDE_PATH}/templates/` is added to the Jinja2 template search path with precedence. Further, the files in the `${DAMAST_OVERRIDE_PATH}/static/` directory will be served without requiring any authentication under the URL `${DAMAST_PROXYPREFIX}/override/static/<file path>` (use `url_for('override.static', filename='<file path>')` in templates).

Keep in mind that all contents of the static directory will be served without user authentication. For details on which paths templates must be put under for proper override, refer to the `template/` directories of the blueprints in the repository. For details on how to inherit from the base template, refer to the base template and other, inheriting templates. Adding static files while the server is running should work without problems, but templates are not hot-loaded in production systems, so the server needs to be restarted if the templates change.

Overriding the Start Page Background Image The background image intended for the default start page cannot be put in the repository. Hence, to use it, or a custom background image visible behind the content, the following instructions must be followed: The background image can be added as an override static file, or referenced via an external URL. For the first variant, suppose the image is stored in the override folder in `static/bg.jpg`. The contents shown in listing 3 must then be placed in the override folder in `templates/override/background-image-url.html`: Alternatively, for an external URL, the contents of the `url()` statement would be the URL of the image (e.g., `url(https://example.org/bg.jpg)`).

3.4. REST API

The REST API provides access to the data in the PostgreSQL database. The API uses JSON as the main data format for communication, and provides CRUD[10] functionality using the HTTP verbs `PUT`, `GET`, `PATCH`, and `DELETE`. Access to the REST API endpoints differentiates between read-only and read-write access, which are controlled by the `readdb` and `writedb` user roles (section 2.2).

The API endpoints are generally conservative in the input they accept, and provide feedback on what is wrong in the HTTP response via the status code and response body. The endpoints are documented in the respective Python function docstrings. The `docs.api-description` blueprint (section 3.3) collects all endpoints and renders a list with the path, parameters, allowed methods, and docstrings. This list is also attached in appendix A.

[10] Create, read, update, delete.

4. Frontend Structure

The web frontend pages all derive from a base template (section 3.3.1). This has a consistent styling, a header with all internal links visible to the user's roles, a main area that is populated by the individual page's template, and a footer with additional links, like the imprint.

4.1. Home Page

The home page introduces the project and the Damast system. This page is visible to everyone with the `user` role. The home page in the repository uses a background image that cannot be put into the repository for licensing reasons. For instructions on how to set a custom background image here, see section 3.3.2.

4.2. Visualization

This section then provides documentation on the different parts of the *visualization,* what they show, and how they can be interacted with. Here, the individual *views* and their respective functionalities are explained. These explanations can also be accessed individually from within the visualization by clicking on the ⍰ question mark icon in the upper right corner of each view. We first introduce the terms we used, and explain the visualization types.

4.2.1. Visualization Terminology

The explanations provided in the present documentation as well as in the info texts in the visualization use terminology and concepts from information visualization. We strove to make these texts as accessible as possible to a wider audience, but some basic understanding of the concepts used is still required.

4.2.1.1. Interactive Visualization

Card et al. [1] formalized the *data visualization pipeline*[11] in 1999. This pipeline specifies the steps that data goes through, from its source representation to the point where it is presented to the users using the visualization. The pipeline consists of four steps, in which certain operations are applied, transforming the data for the next step. After the last step, the data is present as pixels on a screen (or as ink on paper). The pipeline allows for viewers to *interact* with and influence the process during each of the four steps, changing the end result in different ways. In a first step, the data has been processed into a consistent, clean form, which is stored in the database. In general, user interaction only applies to the last three steps, and we will focus on these in what follows.

In a second step, through **data transformations,** the *source data* is transformed into *data tables* (i.e., suitably structured data). These transformations can include general data mapping, aggregation (such as counting or averaging), and filtering (e.g., include data after the year 1000 CE only). By *applying filters* in the visualization or changing, for example, the *display mode* in the settings (see section 4.2.2.3), viewers influence this second step of data transformation.

In the third step, called the **visual mapping,** the data tables are *mapped* to visual structures. For example, in a bar chart visualization, data items are mapped to rectangles, and the value is mapped to the height of the respective rectangle. In general, data attributes are mapped to *visual variables.* Visual variables include, but are not limited to: position, length (height, width, diameter), area, shape, color value,

[11] See `https://infovis-wiki.net/wiki/Visualization_Pipeline`

color hue, or texture. Examples for interaction with regard to the visual mappings are to switch between *qualitative* and *quantitative* mode for the timeline (see section 4.2.2.4). In this case, data is either mapped to rectangles of different colors, or to stacked area. *Note:* In this example, the *data transformation* step is also affected.

In the fourth and last step, the visual structures are then *rendered* to the views. In this step, the **view transformations,** the visual perspective on the data is also modified. This is done via geometric transformations: translation, scaling, and rotation (although the latter is used less frequently). Viewer interaction concerning this step includes, for example, zooming and panning (see section 4.2.1.2).

4.2.1.2. Zooming and Panning

Zooming and *panning* are two types of interaction taking place in the image space of the visualization. However, they possibly interact with other parts of the information visualization pipeline, rather than just with the *view transformation* step.

Zooming is the process of increasing or decreasing the scale of the visualized image. In a map, this means showing a smaller area of the map in more detail (zooming in), or showing a larger area in less detail (zooming out). In a timeline, this could mean showing a shorter time span in more detail, or a longer time span in less detail.

In most cases in information visualization, zooming is not merely a geometrical scaling operation (such as zooming in on a picture, simply enlarging the size in which pixels are shown). Rather, zooming in means that data can be displayed with more detail and less aggregated. Similarly, when zooming out, data needs to be aggregated more. Hence, zooming often involves not only the *view transformation* step, but also the *data transformation* and *visual mapping* steps as parts in the information visualization pipeline. This type of zooming is also called *semantic zooming,* as opposed to *geometric zooming,* which only affects the view transformation step.

Panning is the process of changing the *geometrical translation* of the visualized image. Panning does not affect *visual mapping*, but only *view transformation*. Examples for panning include:

- Moving a map's center around, such that an area to the east is now shown. In maps, panning is often possible by clicking, then dragging the mouse, then releasing. In this case, there is no zooming involved.
- Moving the visible area in a timeline. For example, the timeline first shows the time span from 600 CE to 800 CE. After panning, the time span shown covers the years 650 CE to 850 CE; the timeline was panned by 50 years.

4.2.1.3. Multiple Coordinated Views

A multiple coordinated views (MCV) visualization consists, as the name implies, of multiple *views.* These views are visually separated, either just by empty space between them or by borders. In Damast (see fig. 4), each view is displayed in a separate user interface (UI) element. These UI elements are called *panes* and can be resized, rearranged, or maximized, similar to the way windows can be interacted with in an operating system such as Microsoft Windows.

In an MCV visualization, each view shows a *different perspective* on the *same data;* that is, the view visualizes a specific aspect of the data. In Damast, one view visualizes the temporal aspect of the data (the timeline), another view the geospatial aspect (the map), and so on. However, the same underlying data (pieces of evidence) is shown in each view. Further, the views are *coordinated,* meaning that interaction with one of the views is reflected by changes in other views. Figure 4 provides an example: Filtering by time range in the timeline view also affects the data shown in the map view. After filtering, only places with evidence from that time range are shown. For more details on the different interactions; see sections 4.2.1.4 to 4.2.1.6.

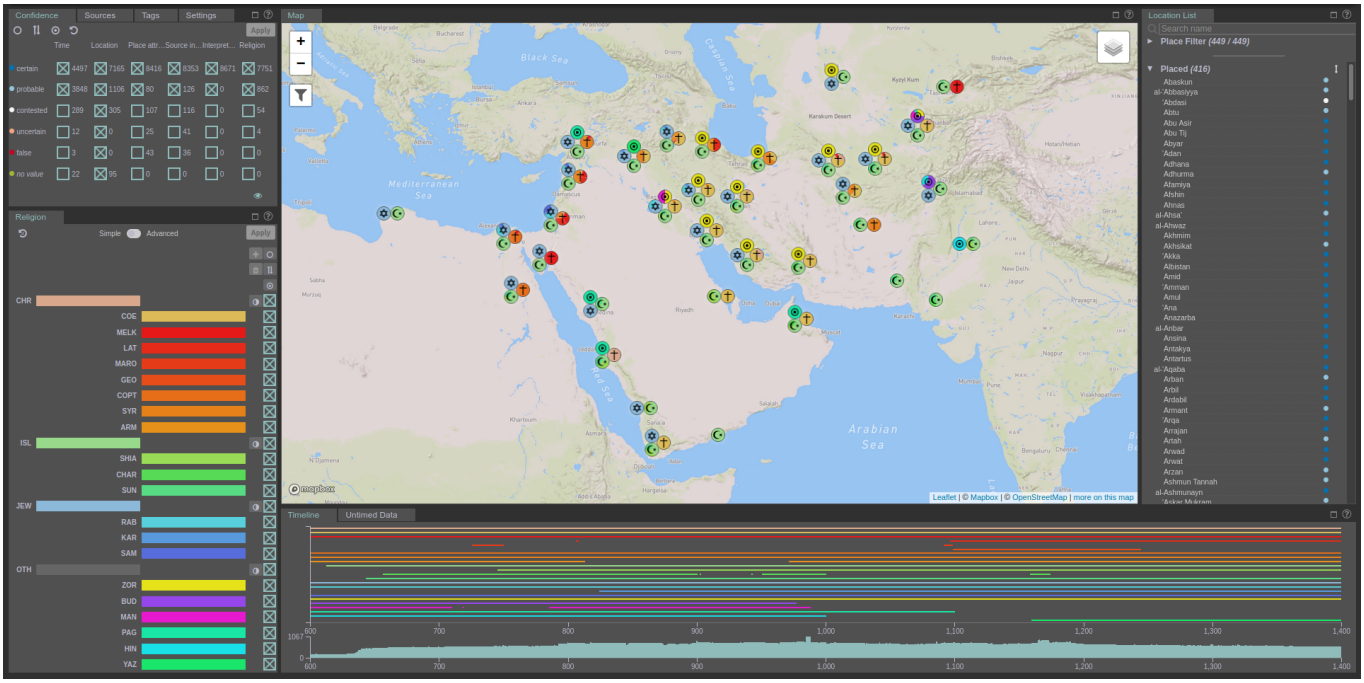

Figure 4.: Screenshot of the visualization, consisting of multiple coordinated views. Each view visualizes one aspect of the data, and interactions with one view are reflected in the others.

4.2.1.4. Filtering

Damast is a *top-down visualization,* meaning that initially, all data is shown, and users can then *drill down* into that data to find smaller, more specific subsets of the data. The drill-down is realized by applying *filters* to the data. A filter decides for each datum whether it matches specific criteria or not. Applying a filter to a dataset results in a subset of the dataset (not necessarily a *proper subset* in mathematical terms).

An example for a filter in Damast is to select a time span from the timeline. The filter then specifies the time span within which the evidence must lie to still be visualized. Another example is to specify one or more sources that the evidence must be attributed to; in this case, the data visualized stems from these sources only.

Our MCV visualization (section 4.2.1.3) implements *multi-faceted filtering.* That means that each view can have a separate filter active at the same time. Because the views show different aspects of the same data, the filters, too, apply to different aspects of the data: The timeline filter applies to the temporal aspect of the evidence, the map filter applies to the geospatial aspect of the evidence, and so on.

Generally, one should be aware of how these filters work *between* as opposed to *within* views. *Between views,* the filters are applied in conjuction; that is, a piece of evidence is shown only if it matches *all* filters. For example, if the map and timeline both have an active filter, evidence is only visualized if it is within the specified time span *and* within the specified geographical region.

Within views, the filters are applied in disjunction; that is, a piece of evidence matches the filter if it matches *any* of the criteria. For example, if a religion filter with two religions is active, it matches evidence that has *either* one *or* the other religion. This behavior is logical, in that there can be no evidence that has *both* religions at the same time, or is attributed to two places at once.

4.2.1.5. Selection

Selection is a user interaction with the data. In Damast, selection is done by *clicking* on some visual element with the computer mouse. For example, clicking a glyph (i.e., a symbol representing one or more places) in the map informs the visualization that the user selected that glyph. Note that what the visualization

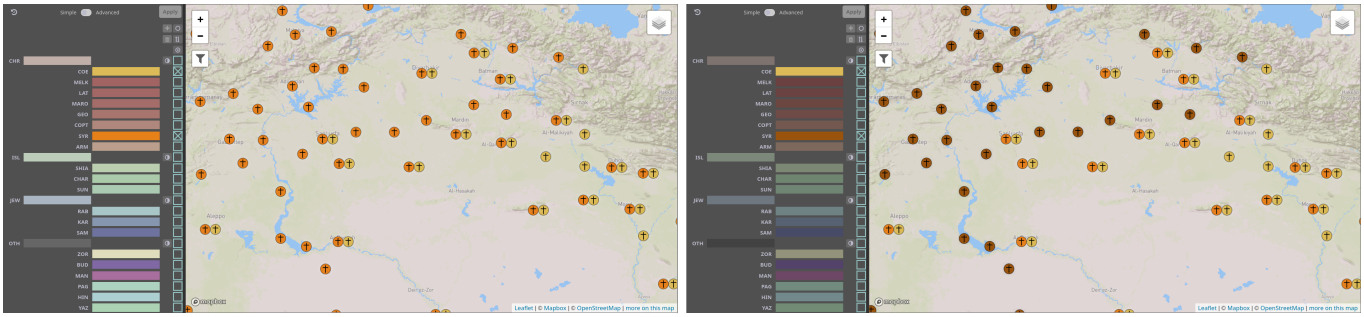

Figure 5.: Example for brushing and linking. Evidence of two religious groups is visualized (left). After selecting the Church of the East (COE) in the religion view, evidence of COE is *brushed,* and the respective visual representations of that data are *linked* in all views (right). In the religion view, all other religions' representations are desaturated and darkened. In the map, all map glyphs for places or clusters of places not containing COE evidence are darkened and desaturated.

does *in reaction to* clicking is no longer part of the selection itself. For the main purpose of selection, see section 4.2.1.6.

4.2.1.6. Brushing and Linking

Brushing and linking is a process used in interactive MCV visualizations to help users understand the connection between different views, or rather, between different aspects of the visualized data. First, the user selects (section 4.2.1.5) some visual element in the visualization. The visualization interprets that selection, and performs *brushing* on the underlying data that is represented by that element. Next, the visualization *links* the brushed data in *all views* by applying specific highlighting to them.

In Damast, brushing always happens on a subset of the visualized evidence. The linking is then done in all views, including the one the selection and brushing originated from. In most views, we display linked elements by keeping their saturation, while all elements that are *not* linked are desaturated and darkened. A notable exception to this behavior is the timeline, where linking will show the temporal data from the brushed subset only, while all other data will be hidden.

Clicking on the same selected element again will *revert* the selection and also clear the brushing and linking. Selecting any other visual element will instead *replace* the selection and brushing and linking will apply to the new subset accordingly. In the map, the brushing and linking can also be cleared by clicking in an empty place.

A source of confusion with the term *"brushing"* can be that it is interpreted in the sense of "paint brush;" in that sense, the brushing itself would be a change of the visual representation of the elements (e.g., their being highlighted). However, this change of the visual representation is the *linking* part. Rather, *brushing* should be understood as in *touching (or brushing) with one's fingers*: *Selecting* a subset of data touches (brushes) them, and the subset of data is highlighted in all views, thereby visually *linking* the data to the selection and providing context.

Figure 5 shows an example of brushing and linking in Damast: Evidence from two religions is visualized. Clicking on one of these religions in the religion view *selects* it and *brushes* the respective evidence. The visual representations of this subset of evidence is then *linked* in all views by desaturating the elements that are not part of the subset.

4.2.1.7. Visualization Types

A number of visualization types are used in Damast. The proper scientific terms are used in the description below, and are introduced here.

Bar Chart and Stacked Bar Chart In a **bar chart,** categories are represented by rectangular bars, usually with a common baseline in one dimension, and a common width. The height of each bar encodes a value associated with the respective category. Bar charts can be horizontal as well, in which case the height of the bars is constant, and the width encodes value instead. In Damast, the bar charts used are horizontal, for example in the source view (section 4.2.12).

A special case of bar charts are **stacked bar charts,** in which each category, or bar, is further divided. Each segment of the bar encodes a sub-category's value. In Damast, stacked bar charts are used in the untimed data view (section 4.2.10), where each general religious affiliation is represented by a bar, and its specific religious groups by segments of that bar. In *all data mode,* all regular bar charts turn into stacked bar charts, with one segment for active data, and one for filtered-out data. Similarly, in *confidence mode,* a segment for each represented level of confidence is shown.

An even more special case is the **normalized** stacked bar chart, where the width (or height) of the bars is constant across bars as well. Hence, the width of individual segments encodes their *relative* proportion within the parent category. Normalized stacked bar charts appear in the source view (section 4.2.12), where they show the distribution of religious groups (or confidence levels) within each source.

Stacked Histogram A **histogram** shows aggregated values of one value in dependence of another value, usually time, which is split up into bins.. A **stacked** histogram, similar to a *stacked bar chart,* is a chart where multiple such areas are stacked on top of each other for each bin. This representation makes it harder to read individual values, but provides a better understanding of the sum over all categories and general trends. In Damast, stacked histograms are used in the *quantitative mode* of the timeline (section 4.2.9).

Indented Tree An **indented tree** is a visualization for hierarchies. Nodes of the hierarchy are represented as elements placed in individual rows, or columns. To signify parent-child relationships, children are *indented* further than their parents. Indented trees are often used in file managers to show directory structures, or in mail programs to display e-mail threads. In Damast, an indented tree visualization is used to show the religion hierarchy (section 4.2.8).

4.2.2. Settings Pane

This pane shows different **settings** pertaining to the visualization. It also offers some **functionalities** to store or load these settings and to generate reports from the currently shown data.

4.2.2.1. Visualization Settings

These are settings that directly affect the visualization. In particular, they control which data is shown and in which way.

4.2.2.2. Show filtered data

This switch controls if data is visualized although it **does not match the currently applied filters.** If *All data* is selected, non-matched data is shown in less saturated colors. Otherwise, it is not shown at all.

4.2.2.3. Display mode

This switch controls **which aspect of the data** is mapped to color. If *Religion* is selected, religion is used for coloring, using the color scheme in the *religion view.*. If *Confidence* is selected, the level of confidence is used, using the color scheme in the *confidence view.* The *aspect of confidence* used for the visualization can be selected in the *confidence view;* there, the currently used aspect is indicated with an eye symbol below the column.

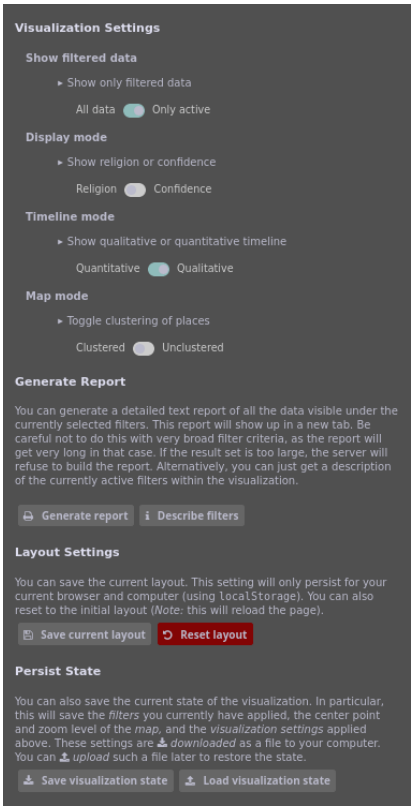

Figure 6.: The settings pane.

4.2.2.4. Timeline mode

This switch controls whether the *timeline* shows **a qualitative summary or quantitative information**. If *Quantitative* is selected, the number of pieces of evidence of that type in that year is represented by the height of the area. The *timeline* then looks like a *stacked histogram.* If *Qualitative* is selected, the *timeline* only shows *whether* there are pieces of evidence of that type each year.

4.2.2.5. Map mode

This switch controls whether map glyphs are ***clustered* or *unclustered.*** By default, glyphs are *clustered:* zooming out will lead to religions attributed to different places being *clustered* ("summarized"). (Note that for these clusters, all places are treated equally; for example, the distribution of religions of place A and place B is combined without taking into account any further aspects of A or B.)

If glyphs are *unclustered,* one small symbol per place and religion is shown. *Note:* This mode may lead to overlap of symbols when zooming out even to a relative small scale and should, thus, be used with due caution when interpreting the results.

4.2.3. Generate Report

With this feature, a detailed **text report** can be generated, which presents all data matched by the currently applied filters. The report will also contain information on the filter criteria that led to the selection of data. After clicking *Generate report,* a new tab in the browser will open. A report can also be created by uploading a *visualization state* that was saved and downloaded beforehand (section 4.2.4.1).

Note: It is not advised to create a report with very broad filter criteria, as it will become very long and thus difficult to handle.

This section also has a button labeled *Describe filters.* Clicking this button will open a small window (a so-called modal window) that describes the currently active filters in text form. This description is the

same as the one at the top of a report.

4.2.4. Layout Settings

The **current layout** of the visualization; i.e., the arrangement of the views, can be saved. This feature is only available if the option *All cookies* is accepted in the *Cookie preferences* (accessible through the menu bar on top). The *layout settings* will only persist for the currently used browser and computer (using `localStorage`). You can also reset the layout to the initial state. Note that this will reload the page.

Note: Depending on your browser settings, cookies may be deleted after closing the browser, which leads to the loss of the saved *layout settings.*

4.2.4.1. Persist State

With this feature, the current **overall state** of the visualization can be saved. In particular, this will save

- the *filters* that are currently applied,
- the center point, zoom level, and visible layers of the *map,* and
- the *visualization settings* (section 4.2.2.1).

These settings are *downloaded* as a file to the computer. Later, such a file can be *uploaded* to restore the state. The file can also be shared to either present your results or to (re-)generate a report based on the filters.

4.2.5. History Tree

These controls located in the header bar of the page control the *interaction history* of the visualization. This allows users to undo, or redo, interactions. Each applied filter counts as an interaction, as do mode changes in the settings pane (section 4.2.2), and moving or zooming the map.

Buttons The control bar itself has three buttons: The *undo button* reverts the last interaction (i.e., goes up one level in the history tree). The *redo button* re-applies the last undone interaction (i.e., goes down one level in the history tree). It is disabled if there are no recent undone actions. The last button opens the *history tree* in a modal window.

History Tree Window The history tree window shows the interaction tree as a *node-link diagram.* Each state is represented by a circle, next to which the age of the state is written. Child states are connected to their parent by a link, which is straight for the first child, and makes a step for subsequent children. Children are placed to the right of their parent. The current state's node is filled. Clicking on a node will put the visualization in the respective state. Hovering over a node will give additional information in a tooltip.

Three additional buttons in the history tree window allow to manage the visualization:

Clear: This will remove all states but the initial state from the tree, and revert the visualization to the initial state.

Prune: This will prune all states from the history tree that are not the current state, or direct ancestors thereof. The result is a linear history from the initial state to the current state.

Prune and condense: This will prune all states from the history tree but the initial state and the current state. The current state is then the direct child of the initial state, and no other states exist. If the current state is the initial state, the result will be the same as if clicking **Clear.**

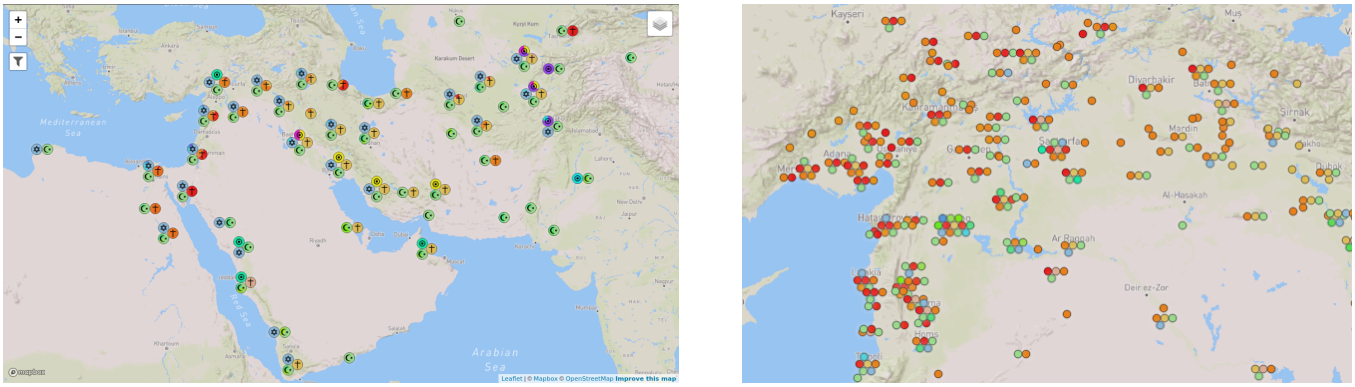

Figure 7.: The map pane in *clustered* (left) and *unclustered* (right) *map mode.*

4.2.6. Map

The ***map*** visualizes the geographical aspect of the data, i.e., evidence for the presence of religious groups at different locations.

4.2.6.1. Content

Map Glyphs. Pieces of evidence are represented by so-called map glyphs. By default, these map glyphs are **clustered** (i.e., **aggregated,** or "summarized").

This aggregation depends on the data currently active and on the current zoom level of the map; that is, once the zoom level or filters are changed, the map is populated with glyphs according to the aggregation rules; simply *panning* the map will not affect the map glyphs. More details are given below.

This *map mode* is called *clustered.* It can be changed in the settings (sections 4.2.2.5 and 4.2.6.1).

Aggregation of Locations. Locations are *aggregated* to eliminate overlap, and so, by default, each map glyph represents **one or more** cities, depending on the zoom level. *Note:* For this aggregation, all places are treated equally; that is, the distribution of religions of place A and place B is combined without taking into account any further aspects of A or B.

When *hovering* over a glyph, it loses opacity and the aggregated locations are indicated on the map as small turquoise dots (maybe partially covered by the glyph). While hovering, a tooltip is also shown, which provides information on the number of pieces of evidence, of religions, and of places related to the glyph. As long as the glyph does not aggregate more than five places, the toponym, geographical coordinates, and more information on the pieces of evidence belonging to that place are provided for each place. Even more details are provided if the glyph only represents one place.

Aggregation of Religions. A map glyph consists of up to four circles, each representing a general religious affiliation with a distinct symbol and color:

- Christianity: cross, red/orange
- Islam: crescent, green
- Judaism: Star of David, blue
- Other: dot, varying colors

Generally, multiple religious groups belonging to the same general religious affiliation are aggregated to one circle. This circle then functions like a pie chart: in *religion mode* (set by default), each piece of the pie chart represents a different religious group in its respective color and the piece's size shows the amount of

pieces of evidence of that religious group relative to the overall evidence of its general religious affiliation. In *confidence mode,* the division of the circle or pie chart is based on the pieces of evidence attributed with different levels of confidence and is colored accordingly.

However, if, regarding *all* map glyphs currently displayed, no more than four religous groups are to be represented, there can be more than one circle representing a general religious affiliation. For instance, if only COE (Church of the East), SYR (Syriac Orthodox Church), and SUN (Sunni Islam) are filtered using the *religion view,* any given map glyph cannot have more than three circles. Accordingly, both COE and SYR will be represented by an individual circle with a cross and the respective color. Note that this aggregation may result from other filters or interactions with the map, not just filtering using the *religion view,* as the map glyphs are dynamically altered based on many different criteria.

Note that placement and aggregation of map glyphs depends on the data currently active as well as the zoom level—not, however, on the current center of the map. In other words, "all map glyphs *currently displayed*" include map glyphs outside the current scope of the map that will appear when panning the map.

If in at least one glyph currently displayed multiple religious groups are aggregated into one circle, this same aggregation will apply to *all* other glyphs as well, even if, in sum, they have less than four circles and would not be aggregated. Otherwise, the perceived variety and distribution of religions would be skewed; for example, a location with only two Christian groups would be represented by a map glyph with two circles for Christianity, while a location with many Christian groups as well as other groups would be displayed with only one such circle.

In *only active* mode, data that has been filtered out disappears from the map. Note that this can lead to glyphs having less circles, if, for example, all Christian data of a glyph has been filtered out. In *all data* mode, data that has been filtered out is indicated by less saturated colors.

Unclustered Mode. The aggregation or clustering of places can be turned off in the *settings pane.* In this *map mode,* called *unclustered,* each location is represented by an individual glyph. Here, the glyphs consist of smaller circles, one for each religious group. These circles are arranged in a hexagonal pattern. Overlap can and will happen in this mode, even when zooming out to a relative small scale. Z-ordering of the glyphs ensures that glyphs with fewer religious groups appear in front of larger glyphs with more religious groups.

Note: For both modes, the specific way of representing the data should be considered when interpreting the results.

Layers. The map provides multiple layers, which can be controlled from the layer control in the upper right corner. One of two *base layers* is always selected and shown: by default, a layer based on a custom map from MapBox[12] is selected, which shows topological features but no geo-political borders (i.e., borders of modern nation states). As an alternative, a map provided by the Digital Atlas of the Roman Empire[13] (DARE) can be selected as the base layer.

In addition, one or more overlay layers can be shown:

Markers consists of the clustered or unclustered map glyphs. It is shown by default.

Diversity Markers displays all locations, without clustering, each colored according to its religious diversity (i.e., the number of distinct religions present in each place). The color scale Viridis[14] is used, where low values are mapped to violet, and high values to yellow.

Diversity Distribution shows an estimation of the religious diversity and is colored according to the same scale as the *diversity markers.*

Distribution shows an estimation of the density of pieces of evidence.

[12] `https://www.mapbox.com/about/maps`
[13] `https://imperium.ahlfeldt.se/`
[14] `https://github.com/d3/d3-scale-chromatic#interpolateViridis`

Note: The two layers for diversity (one displaying markers, the other displaying a heatmap) are *alternative* representations of the *same* data. Thus, they should not be both displayed at the same time. Similarly, the two layers using markers (the default one displaying map glyphs and the one displaying markers of diversity) should not be shown together.

What the Map is not Showing. Damast visualizes religious constellations in cities and towns of the Islamicate world with static Non-Muslim communities. The map does *not* depict a representation of the population density in the medieval Middle East. In other words, an area with no or only few map glyphs is not necesarrily less populated than other areas. The map makes no claim to be complete, nor does it show the general distribution of religions in a given area.

Empty areas on the map can have multiple reasons:

- The area is outside the geographical scope of Damast, e.g., Europe.
- No data for a city was collected.
- No data on non-Muslim communities was available.

Furthermore, in *clustered* mode, the overall size of the map glyphs (i.e., the number of circles) does *not* directly correlate with the number of religions or pieces of evidence; for instance, a map glyph with three circles does not necessarily represent more pieces of evidence than one with only two circles.

4.2.6.2. Interaction

The map can be interactively zoomed and panned (i.e., the center of the map is moved).

Selection. Clicking on a map glyph will *select* the represented places, *brush* the represented data, and *link* the respective data in the rest of the views. Likewise, brushing data in other views will link the respective places in the map. Also, selecting a place in the location list will pan the map to center on that place. Map glyphs that are not linked will be displayed in less saturated colors. Linking persists when zooming, even if clustered glyphs split up or merge. Note, however, that a map glyph often represents more than one location. In this case, *all* of the circles belonging to the glyph are highlighted, even if the linking only refers to part of the evidence. For instance, a place from the *location list,* which only has pieces of evidence of the general religious affiliation "Christian", may be selected. If this place is aggregated with other places that additionally have, for instance, Islamic pieces of evidence, both the circle with the crescent as well as the one with the cross are highlighted.

Geographical Filtering. Evidence can be filtered by geographical location. This is done by drawing the respective bounding shapes into the map. For this, the Leaflet plugin for Geoman[15] is used. The respective tools to control the filters are available by clicking on the button with the funnel in the upper left corner of the map, below the zoom buttons. The button then expands to a set of controls, arranged in three blocks: the first block for adding elements, the second for editing them, and the third to apply, remove or revert the filters. These are described in detail below. Tooltips are shown when hovering over the individual buttons.

The first block is for *adding* new elements to the bounds:

- A rectangular area can be added, by first clicking in the map to select one corner, then moving the mouse, and clicking again to select the opposite corner.
- The second option is to add a polygon. Here, a new point is appended to the polygon each time you click in the map. To complete the polygon, click on the first node.

[15] `https://geoman.io/leaflet-geoman`

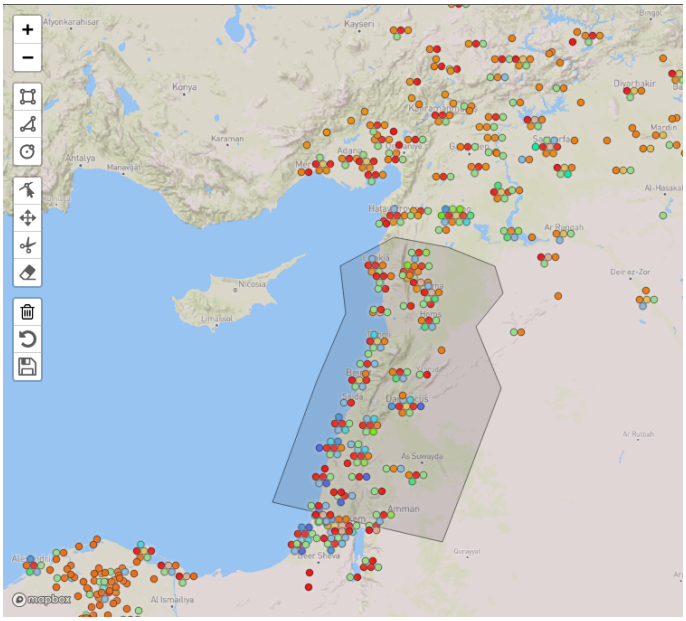

Figure 8.: The map pane with the Geoman editor open, and a polygon being edited.

- The last option is to add a circle. Click once to select the center, move the mouse, then click again once the circle has the appropriate radius. Note that the circle is converted to a GeoJSON polygon when saving.

The second block contains controls for *editing* existing elements. It is possible to

- move points on existing shapes,
- move entire shapes,
- cut (subtract) a polygon from an existing shape, and
- remove entire shapes.

Note that polygons can only be moved or removed individually *before* they have been applied as filters.
The last block contains controls to *apply* the created bounds to the dataset filter:

- The trash can button removes all existing shapes.
- The undo button reverts the bounds back to the state of the currently active filter, i.e., the geographical filters applied last. It then collapses the menu.
- The save button applies the new bounds to the dataset filter, and then closes the editor.

4.2.7. Location List

In this view, **all locations** in the data are listed, using their *main toponym* as place name. Places can be searched, selected and filtered.

4.2.7.1. Contents

The view consists of several sections:

- a *search* field;

Figure 9.: The location list. The *confidence of location* is indicated by a colored marker behind the place name. Places that match the search term at the top are highlighted in orange if their primary name matches, or in yellow if an alternative name matches. Names are shown in italics, darker, and desaturated if their level regarding confidence of location is not currently checked.

- a *place filter;* and
- the actual *location list,* consisting of two lists, *Placed* and *Unplaced,* representing locations with and without geo-coordinates, respectively.

Detailed descriptions how to use the *search* field and the *place filter* are found below under *"Interaction"* (section 4.2.7.2). In what follows, the two lists are described.

The list *Placed* contains all locations for which a geographical position is known. The list *Unplaced* contains all locations for which a geographical position is not (yet) known; in this case, the *confidence of location* is attributed with *no value* and colored accordingly.

Note: Missing data is normal during research and while entering data is still in progress. However, missing data can severely affect confidence in the visualization if not properly communicated. We have therefore chosen to make missing geographical locations and time information explicit in separate views of this interface (apart from the section *Unplaced* in the *location list,* see the untimed data view (section 4.2.10). This also allows for searches directed at data in need of improvement.

The position of the two lists can be swapped by clicking on the swap button. The following is true for both lists: Each line in the lists represents one location. Locations are listed using their *main toponym* as place name. Accordingly, looking through the lists with a specific place name in mind, if this place name is not the *main toponym,* the location is not found and should be searched in the *Search field.* There, *alternative names* are considered as well (section 4.2.7.2).

The place names are sorted alphabetically, disregarding a prefix of apostrophes (e.g., for the letter *'ain*) or Arabic definite articles; for instance, "Amid" comes before "'Amman", and "Jubayl" comes before "al-Juma". In *only active* mode (section 4.2.2.2), only places with pieces of evidence matching the current filters are listed. When *hovering* over the line with the mouse, a *tooltip* (fig. 10) with additional information is shown, that is:

- place type,
- geographical location (i.e., coordinates),
- confidence of location,
- alternative names,

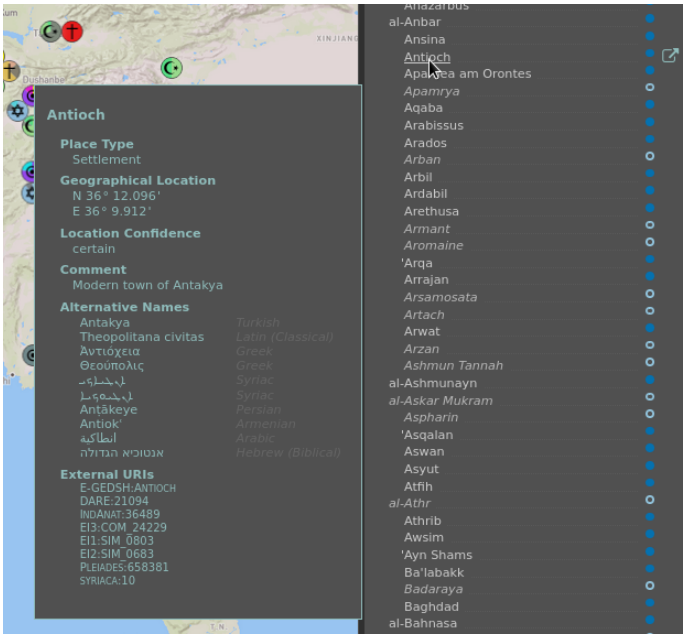

Figure 10.: A tooltip in the location list, for the place Antioch.

- external URIs referencing the same place.

Also, while hovering, a link symbol appears at the very end of the line (fig. 10). Clicking the symbol opens an overview for this place, the so-called *place URI page* (section 4.5). *Note:* If the user has rights to edit the database, clicking the symbol leads to the respective entry of the location in the database instead.

Behind the place name, the *confidence of location* of each place is shown as a small, colored circle. The color scale is the same as in the *confidence view.* In *all data* mode (section 4.2.2.2), when the filter set for *confidence of location* does not match the place's confidence of location, this confidence circle is not filled. Also, the name is displayed in italics, darker, and in less saturated colors in that case.

4.2.7.2. Interaction

Selection. Individual locations can be selected by clicking on the respective line of the list. This will highlight the location, bring it to the top of the list and show a short vertical line to the left of the place name. Data related to the selected location will be *linked* in all other views, while non-selected data will be displayed in less-saturated colors.

Searching for Toponyms. It is possible to **search for a specific place using different toponyms** in the search field at the top. Non-Latin scripts can be used for searching, too. Typing a search query in that field will highlight the search results in bold orange and sort them to the top of the list. The search matches not only the *main toponym* but also *alternative names;* for example, "Edessa" will find the respective place under its *main toponym* "al-Ruha". Matches to *alternative names* that do not match the *main toponym* are sorted after *main toponym* matches and highlighted in bold yellow.

Apart from toponyms, **external URIs** can be entered in the *search field.* For example, `syriaca:10` or `syriaca.org/place/10` finds Antioch. The search is partially case-sensitive: `ask` finds both "Daskara" as well as "al-Askar Mukram", while `Ask` only finds "al-Askar Mukram". The search field additionally supports JavaScript-style **regular expressions.** For example, searching for `Bagh?dad` would find "Bagdad" as well as "Baghdad", because `h` followed by `?` matches no "h" or exactly one "h". For further reference, refer to the MDN documentation[16].

[16] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions`

Filtering. The expandable section *Place Filter* at the top of the view allows to **filter evidence by place.** When expanded, a list of all places in the database is shown. If, however, something has been entered into the *search field,* only places matching the search are listed.

This list is used to include or exclude places from the so-called *place set,* that is, the set of places currently active. For instance, if Baghdad is excluded from the *place set,* all pieces of evidence attributed to Baghdad are filtered out. Keep in mind that, when *all data* mode is active, pieces of evidence attributed to places excluded from the *place set* are visible but displayed in less saturated colors.

A place can be removed from the place set by clicking on the *red minus* on its right, or added to the place set by clicking on the *green plus.* Note that the selection is not applied until the *Apply* button is clicked. This button is disabled if the selected filters are matching the data currently visualized.

A symbol to the left of the place name indicates the current state of the place:

- a turquoise check mark if the place is in the currently active *place set;*
- no mark if the place is not in the currently active *place set;*
- a grey cross if it will be excluded from a new *place set* which has not been applied yet; and
- a green check mark if it will be added to a *new place* set which has not been applied yet.

Place sets can be saved using the *Save* button under the list. For users with access to the database, this will store a place set in the database under the name entered by the user. For users without access to the database, the place set will only be saved to `localStorage`. This feature is only available if the option *all cookies* is accepted in the *cookie preferences* (accessible through the menu bar on top). By clicking the *Load* button, a saved *place set* can be loaded back into the filter. *Note::* Depending on your browser settings, cookies may be deleted after closing the browser, which leads to the loss of the saved *place sets.*

Buttons Facilitating Creating Place Sets. Buttons in the top left corner help in editing the *place set* on a larger scale:

The revert button will *revert* the changes to the *place set;* that is, the *place set* will match the current filters.
Note: Because of the functionalities detailed below, the *place set* will **not** update from the database when other filters change; it changes only with the initial load and loads of the visualization state (section 4.2.4.1). Instead, resetting the *place set* is left to the user, which allows to build up a place set incrementally as illustrated by the example below.

The empty circle button will *uncheck* all places; that is, no place would be included in the intended *place set* after application.

The exchange button will *invert* the current marks; that is, all places that were marked are unmarked, and vice versa.

The circle button with dot inside will *check* all places; that is, all places would be included in the intended *place set* after application.

The set union button will *extend* the *place set* (PS) by all places currently shown in the *location list* (LL) (section 4.2.7.1). That is, all places that were in the *place set* before are still there, and *additionally* all places from the *location list* are checked. The result is the *set union* of the previous *place set* and the current *location list:*

$$\mathrm{PS}_{\mathrm{new}} = \mathrm{PS}_{\mathrm{old}} \cup \mathrm{LL}$$

The set intersection button will *restrict* the *place set* to contain only a subset of its current contents, namely those places that are *also* in the *location list.* The result is the *set intersection* of the *place set* set and the current *location list:*

$$\mathrm{PS}_{\mathrm{new}} = \mathrm{PS}_{\mathrm{old}} \cap \mathrm{LL}$$

The set subtraction button will *remove* all places currently in the *location list* from the *place set.* The result is the *set difference* of the previous *place set* and the current *location list:*

$$\mathrm{PS}_{\mathrm{new}} = \mathrm{PS}_{\mathrm{old}} \setminus \mathrm{LL}$$

These last three operations can be used to quickly create a complex *place set* from a number of criteria. *Note:* Since they use the contents of the *location list,* they only make sense when the *only active* visualization mode is active.

To illustrate the possibilities provided by these operations, consider the following case: We want to explore all pieces of evidence of *Christianity* in cities between 800 and 900, where there is evidence for *Muslims* but **not** for *Jews:*

1. *Clear* all filters (everything is visible).
2. *Filter* by time range (800–900) in the time line.
3. *Filter* by religion using the *religion view,* choosing *Islamic* groups only, then click *Apply.*
4. *Restrict* the *place set* to only those places currently displayed by clicking the *set intersection* button. The *place set* now contains all places where there is evidence of Islam between 800 and 900.
   Important: Do *not yet* apply the *place set.* This would affect the other views.
5. *Filter* by religion using the *religion view,* choosing *Jewish* groups only, then click *Apply.*
6. *Remove* the shown places from the *place set* by clicking the *set subtraction* button. The *place set* now only shows places where there are pieces of evidence of Islam between 800 and 900, but *not* of Judaism.
7. *Apply* the place filter by clicking on the *Apply* button. *Note:* The map is blank; this is normal as you have just filtered out all places with presence of Judaism while the *religion view* is still set to show Judaism only.
8. *Filter* by religion using the *religion view,* choosing *Christian* groups only, then click *Apply.*

4.2.8. Religion Hierarchy

This view shows **all religious groups** contained in the database and allows for selecting and filtering the data according to religion.

4.2.8.1. Content

The list of religious groups is hierarchical and represented by a tree visualization: each religious group occupies a line, which is indented according to its rank in the hierarchy. In technical terms, the higher level is conceptualized as *parent,* the respective lower levels as *children.*

Each line consists of an *abbreviation* for the religious group, a *node,* displayed as a bar, with the color associated with the group, and a checkbox used for filtering. Hovering with the mouse over the line displays a tooltip with the full name of the religious group and the number of pieces of evidence for the group based on the currently applied filters.

Figure 11.: **Left:** The religion hierarchy in its default state. At the top are the controls to revert the filter, the switch between simple and advanced filter mode, and the apply button. The religions are visualized as an indented tree, and are filtered by enabling the respective checkbox.
Right: The religion hierarchy in advanced mode. The filter limits pieces of evidence to those from places where two or more of the Syriac Orthodox Church (SYR), the Armenian Church (ARM), and the Church of the East (COE) are present.

The general religious affiliation (e.g., Christianity) is on the first level of the hierarchy. For other religions beside Christianity, Islam, and Judaism, we made the pragmatic choice to group them under the category *Other*.

The religious groups each have a distinct color. For Christianity (red), Islam (green), and Judaism (blue), religious groups belonging to these general religious affiliations have hues of the same color. However, the difference between colors is limited because of the number of religious groups. In *confidence mode* (section 4.2.2.3), the bars of the groups are colored based on the average confidence of the represented data.

Details Regarding Coloring and Saturation. Depending on the options *show filtered data* (*all data* or *only active*) as well as *display mode* (*religion* or *confidence* mode) in the *settings,* the coloring of the bars changes:

1. *Only active, religion mode*
   A bar has only one color. If *any* data related to a religious group is active, the bar has saturated colors; if all data related to a religious group is filtered out, the bar is shown in a less saturated color.

2. *All data, religion mode*
   A bar can be divided into two parts, a saturated and a less saturated part. The less saturated part represents the relative amount of data currently filtered out.

3. *Only active, confidence mode*
   A bar can be divided into several sections. Each section represents the relative amount of data with a certain level of confidence.

4. *All data, confidence mode*
   A bar can be divided into two general parts: a saturated and a less saturated part, representing the

data currently active or filtered out, respectively. Each part can have several sections, which each represent the relative amount of data with a certain level of confidence.

The *aspect of confidence* used for the visualization can be selected in the *confidence view.* There, the currently used aspect is indicated with an eye symbol below the column.

4.2.8.2. Interaction

Selection. Clicking on a line will *select* the religious group. This group will be highlighted, and other groups will be displayed in less saturated colors. The data represented by the selection will be *brushed,* and is *linked* to related data in all views. For instance, map glyphs in the *map* representing pieces of evidence with the selected religion will be highlighted, other locations will be represented by less saturated colors.

Filtering. The checkbox of a religious group can be unchecked, which leads to filtering out this group throughout the entire visualization. Depending on whether *only active* or *all data* mode is selected in the *settings,* data that has been filtered out is either hidden or displayed in less saturated colors.

There are two basic modes for filtering by religion (more details further below): In *Simple* mode, pieces of evidence matching *any* of the checked religions are shown. In *Advanced* mode, only pieces of evidence from places are shown, where checked *combinations* of religions were present. Note that, in both modes, the selection is not applied until the *Apply* button is clicked, which is enabled once there are changes to the filter.

Simple Mode. In *simple* mode, data is filtered simply according to the checked or unchecked religions. Initially, all religions are checked. To filter out a religion, it can be unchecked in the checkbox column.

Advanced Mode. In *advanced* mode, it is possible to filter not individual religions but different *combinations* of religions. For instance, analysis may require to show places where two or more of the following three religious groups exist together: the Syriac Orthodox Church (SYR), the Armenian Church (ARM), and the Church of the East (COE)—but not those places where only one of the three is present. In this case, pieces of evidence are active if, for any combination, all religions of that combination are present at a place.

Different from *simple* mode, there are multiple columns of checkboxes. Each column represents a set of religious groups as described above. To *control* the filters in *advanced* mode, columns can be added by using the plus button above each column. Every column (except the last remaining one) can be deleted by pressing the delete button above the column.

The following procedure creates a filter corresponding to the analysis described above (fig. 11 right):

- Switch to *advanced* mode.
- Create two additional columns by pressing the plus button.
- In the first column, check SYR and ARM.
- In the second column, check SYR and COE.
- In the third column, check ARM and COE.
- Click the *Apply* button.

Note: If only places should match the filter where *all* of these three religious groups are present, only one column in *advanced* mode is necessary with all three groups checked. This must be understood as "pieces of evidence with COE **and** ARM **and** SYR." In turn, this differs from checking the three religious groups in *simple* mode, which equals "pieces of evidence with COE **or** ARM **or** SYR."

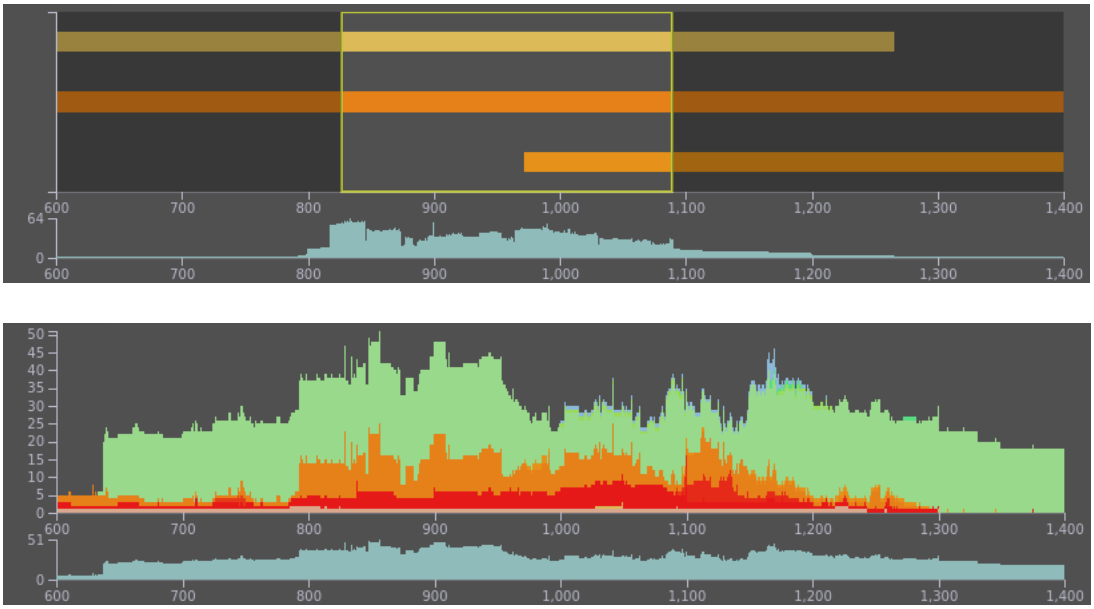

Figure 12.: **Top:** The timeline in qualitative mode, showing three religious groups. A time range between 830 and 1090 is set as the filter.
Bottom: The timeline in qualitative mode.

Filter Controls. A few additional utilities are available for managing the checkbox columns. The controls for adding (plus button) and removing (trash can button) are already described in the section 4.2.8.2. Importantly, columns can only be added in advanced mode, and the last remaining column cannot be removed. The additional controls in the view are:

Revert filters This button is found in the top left corner of the view. Clicking it reverts the filters back to the state that is currently shown in the visualization; that is, all changes that are not *applied* yet are discarded.

Uncheck all boxes in this column This button is found at the top of each checkbox column. Clicking it will uncheck all boxes in the column.

Invert all boxes in this column This button is found at the top of each checkbox column. Clicking it will invert all boxes in the column; that is, boxes that were checked are unchecked, and vice versa.

Check all boxes in this column This button is found at the top of each checkbox column. Clicking it will check all boxes in the column.

Toggle subtree This button is present to the *left* of some of the checkboxes in each column; namely regarding religions that have *children* in the hierarchy. Clicking the button will check or uncheck this religion and all its *children* in the respective column. Whether clicking checks or unchecks depends on whether the majority of checkboxes in the subtree are checked.

4.2.9. Timeline

The *Timeline* shows the distribution of the different religious groups **along the overall time period:** 600–1400 CE. In *confidence mode,* the level of confidence is shown instead of religious groups (sections 4.2.2 and 4.2.11).

4.2.9.1. Content

The *timeline* consists of two graphs: the ***main timeline*** and a smaller ***overview.*** The graphs are described in the following.

Also, the view changes depending on the choice of *show filtered data* in the *settings* (section 4.2.2): in *all data* mode, data that has been filtered out is indicated by less saturated colors. In *only active* mode, data that has been filtered out is hidden from the visualization in all views, including the timeline.

Main Timeline. The *main timeline* shows either a qualitative summary or quantitative information, depending on the *timeline mode* selected in the *settings pane.* If *quantitative* is selected, the number of pieces of evidence of that type in that year is encoded into the height of the area as a *stacked histogram.* If *qualitative* is selected, the timeline only shows *whether* there is evidence of that type each year. The colors used in the graph match the ones used in the *religion view* or *confidence view,* depending on the *display mode* selected in the *settings pane.* Importantly, the *main timeline* can be used to filter the data according to time (section 4.2.9.2).

Overview. The smaller **overview** serves as a *minimap* for the larger *main timeline:* it always shows the entire time range contained in the database. By clicking and dragging in the *overview,* a rectangle can be drawn which represents a time range. The *main timeline zooms* to this time range and shows data in greater detail. The overview represented by the lower graph does not change during this *zooming* in the upper graph. Vice versa, if the data is filtered by clicking and dragging in the *main timeline* (see "Filtering" (section 4.2.9.2), the selected range is indicated as a thick bar below the *overview.*

The rectangle representing the time range for zooming can be moved by clicking and dragging or altered by changing the width of the rectangle. Note the mouse cursor in form of a grabbing hand (or a four-headed arrow) when hovering over the rectangle or the double-headed arrow when hovering the left and right borders of the rectangle, respectively. Clicking anywhere in the *overview* outside the rectangle will deactivate the zooming.

Tooltips. When **hovering** over any of the graphs, a tooltip with a summary of pieces of evidence for that year is displayed. If the *main timeline* shows a smaller time range than the *overview,* the year in the tooltip depends on whether the mouse cursor is on the *main timeline* or the *overview.*

When **selecting** data in other views, through *brushing and linking,* only the selected data shows in the timeline. *Note:* Selecting data in the timeline itself is not possible.

4.2.9.2. Interaction

Filtering. By *clicking and dragging* in the *main timeline,* a rectangle can be drawn which represents a time range. Data outside the time range is filtered out; accordingly, the area outside the rectangle is displayed in less saturated colors. The time range serves as a time filter for the data across all views.

The rectangle representing the time range for filtering can be moved by clicking and dragging or altered by changing the width of the rectangle. Note the mouse cursor in form of a grabbing hand when hovering over the rectangle or the double-headed arrow when hovering the left and right borders of the rectangle, respectively. Clicking anywhere in the *main timeline* resets the filter.

4.2.10. Untimed Data

Data with **missing temporal information** cannot be included in the timeline. Accordingly, this data is visualized in the *untimed data view* as a stacked bar chart.

4.2.10.1. Content

The chart is divided by the four general religious affiliations along the x-axis. All pieces of evidence that have no temporal information and belong to one of these religious affiliations are stacked, with the number of pieces of evidence on the y-axis. The pieces of evidence are grouped and colored according to the religious group or the level of confidence, depending on the *display mode* selected in the *settings pane* (section 4.2.2).

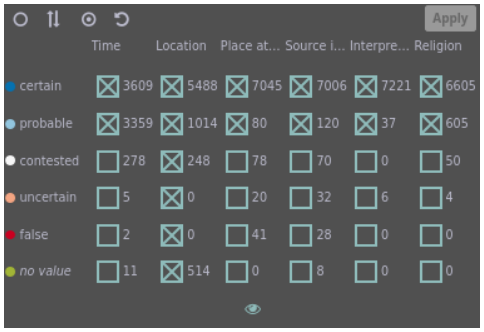

Figure 13.: The confidence view. Aspects of confidence are listed as columns, and confidence levels ordered into rows. Checkboxes control which confidence levels should be shown for each aspect, and the number of pieces of evidence for each confidence level and aspect are given.

Note: Missing data is normal during research and while entering data is still in progress. However, missing data can severely affect confidence in the visualization if not properly communicated. We have therefore chosen to make missing geographical locations and time information explicit in separate views of this interface (apart from *untimed data,* see *"unplaced data"* in the *location list* (section 4.2.7.1)). This also allows for searches directed at data in need of improvement.

4.2.11. Confidence View

This view shows the different **levels of confidence** attributed to the different elements of a piece of evidence (i.e., place, religious group, time span, etc.).

4.2.11.1. Contents

The view consists of a table, in which each row represents one **level of confidence** and each column an **aspect of confidence.**

For each cell in the table, a **checkbox** for filtering (section 4.2.11.2) is displayed as well as the **number of pieces of evidence** with the respective confidence and aspect. The rows are sorted by descending level of confidence.

Levels of Confidence. Generally, the confidence of any given information was categorized according to these 5 degrees:

- *certain* (blue),
- *probable,*
- *contested,*
- *uncertain,* and
- *false* (red).

In addition, a confidence can have *no value* at all (green).

Note: The attribution with *no value* is used while data is still being collected and usually does not appear in a published instance of this visualization. A notable exception are places, especially regions, with no geographical coordinates. In that case, no attribution of confidence except *no value* is suitable.

Aspects of Confidence. Confidence is attributed to different aspects of the data:

Timespan confidence signifies the confidence in the veracity of the timespan stored in a piece of evidence. Pieces of evidence without a time attribute are assigned the confidence *no value.*

Confidence of location signifies the confidence in the geographical location of a place. Some places, especially regions, have *no value* (section 4.2.11.1).

Confidence of place attribution signifies the confidence in assigning a location from the database to a toponym from a source.

Confidence regarding the source signifies the confidence in the veracity and objectivity of the source itself regarding a piece of evidence.

Confidence of interpretation signifies the confidence in the correct interpretation and recording of the source when entering a piece of evidence into the database.

Confidence regarding religion signifies the confidence in the veracity of the presence of a particular religious group at a given time and place.

4.2.11.2. Interaction

Filtering. For each aspect of confidence and each confidence level, a checkbox is shown in the respective cell. Unchecking a checkbox will filter out pieces of evidence with this confidence level for that aspect of confidence. This allows for a very fine-grained control over what data to visualize.

Note that the selection is not applied until the *Apply* button is clicked. This button is disabled if the selected filters are matching the data currently visualized.

The filters work in *logical conjunction* across different aspects of confidence; that is, having filters in place for multiple aspects of confidence will only show pieces of evidence that match *all* filters.

Four additional buttons in the top left corner make filtering more convenient:

1. Uncheck all checkboxes.
2. Invert all checkboxes. Checked boxes are unchecked, unchecked boxes are checked.
3. Check all checkboxes.
4. Revert the selection of checkboxes to the default. By default, *all* confidence levels are selected for confidence of location, and only *certain* and *probable* for the other confidence aspects.

In addition, entire *rows* or *columns* of checkboxes can be set to either checked or unchecked by clicking the *row header* (the confidence levels) or the *column header* (the aspects of confidence). Here, the new value for the new row (or column) is the inverse of what the majority of values (3 or more) was before; for example, if 4 checkboxes were checked and 2 unchecked for the *uncertain* row, clicking on the row header would *uncheck* all checkboxes in the row.

Visualized Aspect of Confidence. By clicking on the eye in the bottom row of each column, the aspect of confidence that is visualized in *Confidence* mode (section 4.2.2.3) can be selected. The currently selected aspect is indicated by a colored eye. By default, *confidence regarding religion* is used.

4.2.12. Source View

This view shows the **sources** from which the pieces of evidence (i.e., sets of data concerning a place, religious group, time span, and the respective confidences) were gathered.

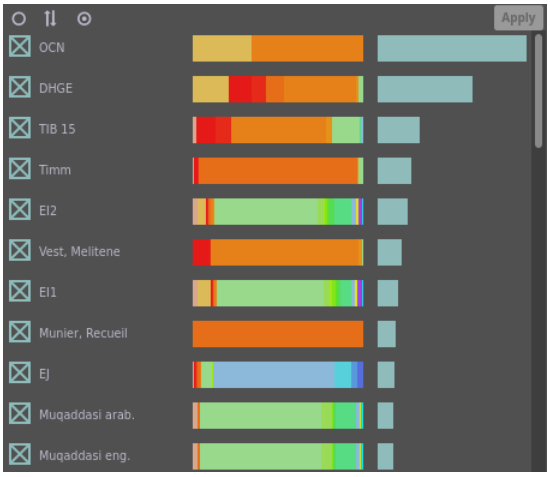

Figure 14.: The source view.

4.2.12.1. Contents

The view consists of a table, in which each row represents one source. The rows are sorted by descending number of pieces of evidence. Note that a piece of evidence can stem from more than one source; therefore, the sum total of all references to sources can be larger than the sum total of pieces of evidence. In what follows, the respective columns of one given row are described.

Checkboxes. The first column contains a *checkbox* that toggles the visibility of evidence from a source. Keep in mind that a piece of evidence can stem from more than one source: as long as *any* source associated with a piece of evidence is checked in this view, the piece of evidence will be active.

Short Names. The second column shows the **short name** of the source, generally an abbreviation. The full name of the source with bibliographical details is available via a tooltip; that is, when hovering over the name with the mouse.

Religion/Confidence Segmentation Visualization. The third column visualizes the segmentation of the pieces of evidence on the religions or on the levels of confidence, depending on the chosen *display mode* (section 4.2.2.3); in *confidence mode,* instead of religion, the currently selected aspect of confidence is used). The visualization used is a normalized stacked bar chart. Because the total width of the bars is the same regardless of the amount of pieces of evidence, the width of each segment signifies the number of pieces of evidence with that religion or confidence level *in relation to* the total evidence count for that source.

Evidence Count Visualization. The fourth column visualizes the count of pieces of evidence for each source. This column uses the same scaling in all rows: the longer the bar, the more pieces of evidence stem from the respective source. This effectively creates a vertical histogram across the rows.

4.2.12.2. Interaction

Selection. Clicking on a row will *select* the source, *brush* the represented data, and *link* related data in all views. For instance, locations in the *map* with pieces of evidence from the selected source will be highlighted, other locations will be represented by less saturated colors. Clicking on the same row again will reset the selection. Selecting a different row, or an element in a different view, will replace the selection; that is, only one source can be selected at a time.

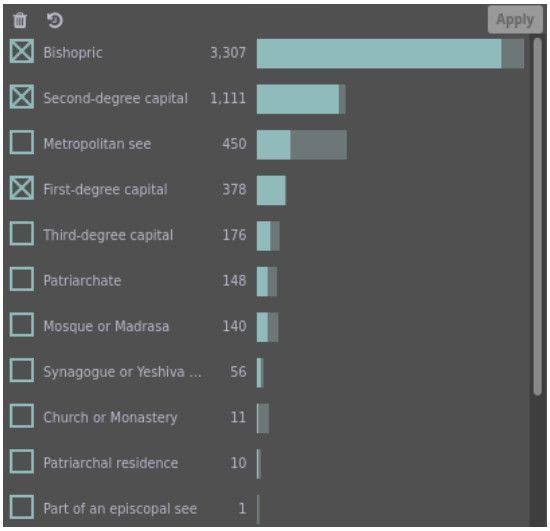

Figure 15.: The tags view. Three tags were selected.

Filtering. A source can be filtered out from the visualized data in all views by (un-)checking the respective checkbox. Note that the new filter is not applied until the *Apply* button is clicked. This button is disabled if the selected filters are matching the data currently visualized.

Checking multiple (or all) sources works as a *logical disjunction:* a piece of evidence is matched if it stems from *any* of the checked sources.

The source filter works together with filters from other views in *logical conjunction;* for instance, if only source X and religion Y are set active in the respective views, only pieces of evidence with source X *and* religion Y are matched.

Three additional buttons in the top left corner make filtering more convenient:

1. Uncheck all checkboxes.
2. Invert all checkboxes. Checked boxes are unchecked, unchecked boxes are checked.
3. Check all checkboxes.

Sorting. The order of the shown sources can be changed using the sorting options switch in the center of the header of the view. The two options are:

1. Sort the sources first alphabetically, in ascending order, by their short name, which is shown in the second column of the list.
2. Sort the sources first in descending order by the number of visible pieces of evidence derived from them.

For both sorting modes, the other sorting criterion is considered as secondary criterion. More specifically, if the sources are ordered by count, and two sources have the same count, the source that would come first alphabetically is listed first of the two.

4.2.13. Tag View

This view shows **tags** which are associated with pieces of evidence (i.e., sets of data concerning a place, religious group, time span, and the respective confidences). Tags were implemented to include additional information and to make further distinctions possible. For instance, the tag *Bishopric* is attributed to pieces of evidence referring to bishoprics as distinguished from *metropolitan sees.* This way, the visualized data can be filtered according to additional criteria. Note that, by default, all tags are *deselected.* This means that no filters based on tags are active ((section 4.2.13.2).

4.2.13.1. Contents

The view consists of a table, in which each row represents one tag. The rows are sorted by descending number of pieces of evidence. In what follows, the respective columns of one given row are described.

Checkboxes The first column contains a *checkbox* used for filtering.

Tag Names The second column shows the name of the tag. Generally, the name should be self-explanatory. Further information is displayed when hovering with the mouse.

Evidence Count Visualization The third column visualizes the count of pieces of evidence for each source. The less saturated part of the bar represents pieces of evidence that are not active (i.e., do not match the current filters).

4.2.13.2. Interaction

Selection. Clicking on a row will *select* the tag, *brush* the represented data, and *link* related data in all views. For instance, locations in the *map* with pieces of evidence with the selected tag will be highlighted, other locations will be represented by less saturated colors. Clicking on the same row again will reset the selection. Selecting a different row, or an element in a different view, will replace the selection. Only one tag at a time can be selected.

Filtering. The visualized data can be filtered according to one tag or multiple tags in all views by (un-)checking the appropriate checkbox or checkboxes. Keep in mind that a piece of evidence can have no tag, one tag, or multiple tags. If, for instance, it has no tag, its visibility is not affected by the checkboxes. If, on the other hand, it has multiple tags, its visibility is only affected by (un-)checking all respective checkboxes. Note that the selection is not applied until the *Apply* button is clicked.

Checking multiple (or all) tags works as a *logical disjunction:* a piece of evidence is matched if it is assigned *any* of the checked tags. The tag filter works together with filters from other views in *logical conjunction:* for instance, if only tag X and religion Y are selected in the respective views, only pieces of evidence with tag X *and* religion Y are matched.

Two additional buttons in the top left corner make filtering more convenient:

1. Uncheck all checkboxes; that is, remove all filters based on tags. Note that this is not applied until the *Apply* button is clicked.
2. Revert all checkboxes to represent the state of filters applied last.

4.3. GeoDB-Editor

The GeoDB-Editor is a tabular data entry interface for data entry into the PostgreSQL database. Not all, but the most commonly edited tables, are represented here. The GeoDB-Editor is split into two pages, one for place-related tables, one for person-related tables.

The tables are hierarchically connected as shown in fig. 16: A row in each table can be *selected,* which is indicated by a hand symbol at the start of the row. All tables hierarchically dependent on that table will then show entries for that selected row. For example, selecting place P in the place table will only show external place URIs, alternative names, and evidences for P. Selecting the first evidence E in that table then shows the time instances and source instances for E in the respective tables. For the evidence table, a `place_instance` entry implicitly exists that links the evidence to the place, and columns from the `place_instance`, `religion_instance`, and `person_instance` tables are shown directly in the evidence table.

Columns with text type can be edited as text. Columns that reference foreign keys (e.g., religions in the evidence table) or enums (e.g., levels of confidence) will show a drop-down menu instead. Changes in a

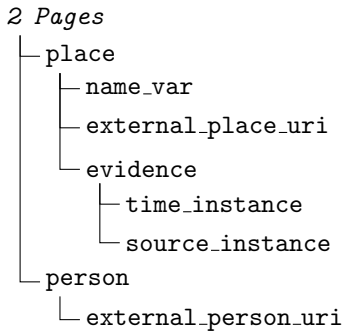

Figure 16.: Hierarchical dependence of table editors in the GeoDB-Editor.

row are indicated by a light green background. Clicking the save (floppy disk) icon at the end of the row persists the changes to the database. Clicking the revert (left-pointing curved arrow) icon at the end of the row reverts the row contents to the database state. Clicking the delete (trash can) icon at the end of the row deletes the row from the database.

The column order can be changed by drag and drop in the column header. Here, some columns also allow to filter, and to sort. At the top of each table, the columns shown can be toggled individually. The currently visible rows can also be downloaded as CSV, and the table filters cleared.

A last button at the top allows to create a new entry. If the table is dependent, the respective foreign key references are implicitly added. The new row will appear at the top of the table, have a yellow background, and have the word "NEW" in the ID column. The delete button at the end is also replaced by a cancel button (striked-through circle), and the save button by an upload button (cloud with arrow pointing upwards). For deletion, reverting changes, and switching entry when there are changes in depending tables, a confimation dialog is shown first.

4.4. Annotator

The annotator provides a facility to upload and then annotate digital versions of documents. For now, only digital text (in the form of plaintext and HTML) are supported, meaning that optical character recognition (OCR) is required as a preprocessing step. The digital texts are then stored in the PostgreSQL database. The annotator allows to create annotations in these texts and to attach historical instances (persons, places, religions, or time information) to individual annotations. By linking the annotations of different types, evidences can be created.

4.4.1. Database Schema

The data is stored in a relational database. These are exceptionally efficient in the lookup and querying of specific attributes and subsets of data, as well as complex joins across attributes and tables, at the cost of the readability and intuitivity of the storage format. In particular, storing relationships such as an *optional* relationship, or a *one-to-many* relationship, cannot be modeled in a straightforward way, but require intermediate tables. The database structure is, therefore, a bit more complex and fragmented than the mental data model it represents. Still, it is valuable to understand the data model, because the way evidence is generated from annotations and groups of annotations is based very closely on that model.

Figure 17 shows a simplified *entity-relationship diagram* of the tables in the database involved in evidence creation and annotations. One piece of evidence *must* contain a place and religion, and *may* contain one person and one or more time spans. In the database, this is represented by instance tuples, where each instance also stores meta-information specific to that tuple, such as a comment and confidence. The instance tuples then point to the base entities (places, religions, persons), which only exist once. The one or more time spans (time instances) are grouped by a singular time group. The instances *may* be linked to one annotation, such that an annotation can *either* be for a place, religion, person, or set of time spans. Each

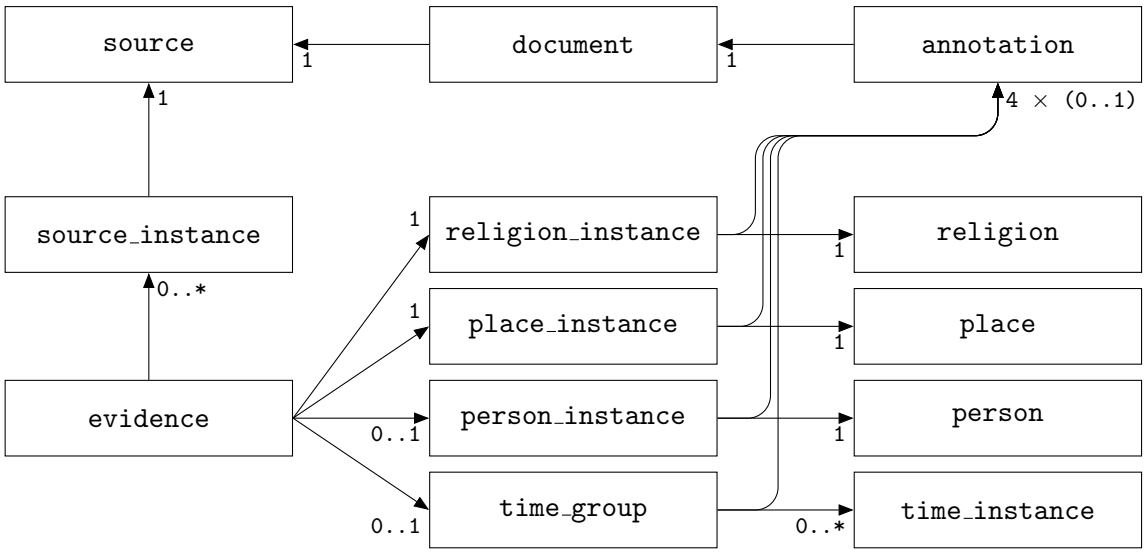

Figure 17.: Database structure of tables directly involved in storing historical evidence and annotations.

annotation is attributed clearly to one document, and each document to one source. An evidence tuple can be derived from zero or more sources, which is coded via the source instance table. In cases where an evidence was created using the annotator, there is only one source instance, whose source is the same as that of the document the annotations belong to.

An evidence created from the annotator, therefore, is a *group* of two to four annotations, where one belongs to a place instance, one to a religion instance, zero or one to a person instance, and zero or one to a time group. Evidences created, for example, using the GeoDB-Editor look exactly the same, except that their instances do not have a connected annotation. For more information on the database structure, please reference fig. 1.

4.4.2. Document Selection

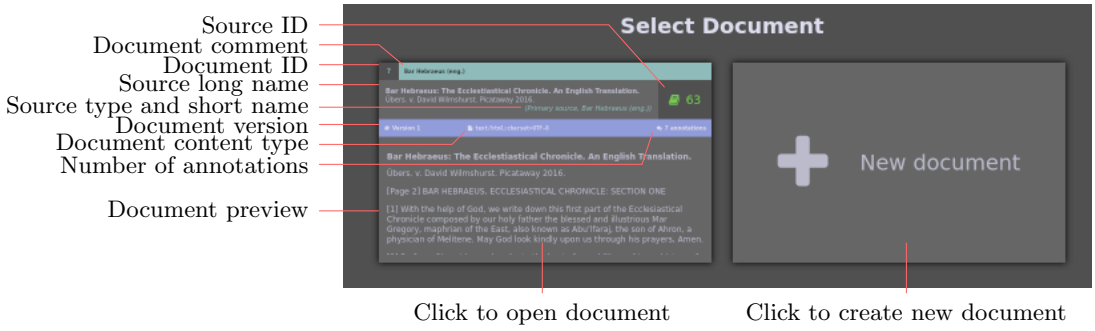

Figure 18.: The initial screen of the annotator shows a *document selection,* where the document to be annotated can be selected. The documents are shown as cards, displaying additional metadata about the document and its underlying source. A separate link leads to a form for creating new documents.

When opening the annotator, the first screen shows a list of all available documents, shown in fig. 18. Each document is represented by a card, which lists some metadata about the document and its source:

- the ID of the source,
- the ID of the document,

- the name of the document (its comment field),
- the full name of the source,
- the type of the source (primary source, literature, ...),
- the short name of the source (e.g., "OCN"),
- the document version,
- the document content type,
- the number of annotations in that document,
- and a short preview of the document.

The goal of all that metadata is to make perfectly clear which document this is; for example, there could be multiple *versions* of one source, each being its own *document.* This could contain differences in spelling, whitespace, and other aspects, which in turn affect the positioning of existing annotations. Therefore, creating new versions of existing documents when the textual content changes is favorable to updating the existing document, as that might create issues with those existing annotations.

To open a document in the annotator, simply *click* on the card in question. Besides one card for each existing document, there is also one card at the very end of the list to create a new document. This will link to the document creation form, described in section 4.4.3.

4.4.3. Creating a new Document

Creating a new document is fairly simple: This will show a page with a form, the contents of which will then populate a new record in the `document` table. The form has five fields, all of them mandatory, described below:

Source: Each document is attributed to a source, which can be selected here using a drop-down menu. The source entry must already exist when creating a new document.

Version: A source can have multiple versions of itself as documents, and so each document must have a version number. This number **must be** unique for each source (i.e., no two documents of the same source may have the same version), and only numerical values are allowed.

Comment: This field is the title of the document. Currently, this is a bit badly named and configured on my part, as *"comment"* does not really fit the purpose of the field: providing a short and ideally unique and recognizable label for a document. On that note, the comment field is currently not enforced to be unique, but it might be favorable to give each document a unique comment in any case. I will at some point clean this up.

Content type: For the moment, annotatable documents can only have two *content types:* either, they are just plain text, or they are marked-up hypertext (HTML). There are subtle differences in how to calculate text positions between plain text (where each character in the file is visible) and HTML. In HTML, a character in the file might have a special meaning, such as making the following text bold, or forcing a line break. This field is a drop-down menu with those two options. It is important to select the correct one here to avoid strange effects during annotation.

Content: This is a file selection form field. Select the file containing the document text from your hard drive, and its contents will be uploaded and put in the database when submitting the form. By default, the file selection dialog should only show text and HTML files.
Note: The file you select here will be read by the browser and sent to the server. This is one of the few occasions where the browser is allowed to read your files, or at least one of them. Be careful not to select any confidential or private file by accident, as its contents will be visible to other users of the annotator as well.

After filling out all fields properly, a new document is created by clicking on the green *Submit* button. This will put the document in the database and navigate back to the document selection page (see section 4.4.2). Before clicking the button, no data will be sent to the server, and you can clear all form fields by clicking on the gray *Reset form* button.


```
...
<p>
  <strong data-virtual-text="Page 2 "></strong>Elit...
  Dolore explicabo excepturi hic nam odit.
  ...
</p>
...
<h2 data-virtual-text="Section 2"></h2>
<p>
  Sit minima ex dignissimos ea nam ullam!
  ...
</p>
...
```

Figure 19.: (l) Virtual text in a document cannot be selected, and is not part of annotations. Virtual text can be placed inline, or as a block element. (r) The HTML source code to produce the virtual text on the left.

In theory, all HTML files should be fine to use for annotation purposes. For security purposes, many tag types and tag attributes are stripped from the document before upload[17]. The resulting document in the database will consist of semantically relevant, but unstyled, HTML tags, and text. However, there might be special considerations to be made about how you want the text to be represented. Some questions to think about before creating a HTML document for annotation are:

- Are there sections of the text (e.g., page numbers) that are *"not really"* part of the text? That means: those sections should not be selectable or annotatable. Some information on how to incorporate virtual text is given in the paragraph below.
- Is there text that requires special highlighting or styling? Would such styling interfere with the styling of annotated text or the evidence links?
- Is the text left-to-right (e.g., latin scripts), right-to-left (e.g., Arabic), or even top-to-bottom, right-to-left (e.g., Chinese)? Is the writing direction mixed (e.g., English headings in Arabic text)?

To add *virtual* text to a document, we use HTML *pseudo-elements,* which are generated by the browser and are not part of the textual content of the document. Take note that virtual text, therefore, is only possible to add when using HTML, not plain text. Virtual text has the advantage of being skipped by the offset calculations and not being selected, therefore also not being part of the text selection and the annotation content. For adding virtual text, add the respective HTML element (e.g., `<h2>` or `<span>`) *without* any text between the tags. The virtual text itself needs to be passed as an *element attribute* to the text. The attribute name used is `data-virtual-text`. The appearance of the virtual text depends on which type of HTML element you use to represent it. Block-level elements, like headings (`<h1>`, `<h2>`, etc.), will be represented thus, and inline elements, like `<span>`, `<strong>`, or `<em>`, will be placed inline, which might

[17] The entirety of `<script>`, `<style>`, and `<head>` tags are discarded, both their content and the DOM nodes themselves. Disallowed tags are removed, but their contents are kept as plain text and child nodes. Allowed tags are: `<a>`, `<abbr>`, `<address>`, `<article>`, `<aside>`, `<b>`, `<bdi>`, `<bdo>`, `<blockquote>`, `<br>`, `<caption>`, `<cite>`, `<code>`, `<col>`, `<colgroup>`, `<dd>`, `<del>`, `<details>`, `<dfn>`, `<div>`, `<dl>`, `<dt>`, `<em>`, `<figcaption>`, `<figure>`, `<footer>`, `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, `<h6>`, `<header>`, `<hr>`, `<i>`, `<img>`, `<ins>`, `<kbd>`, `<li>`, `<main>`, `<mark>`, `<nav>`, `<ol>`, `<p>`, `<picture>`, `<pre>`, `<q>`, `<rp>`, `<rt>`, `<ruby>`, `<s>`, `<samp>`, `<section>`, `<small>`, `<span>`, `<strong>`, `<sub>`, `<summary>`, `<sup>`, `<table>`, `<tbody>`, `<td>`, `<tfoot>`, `<th>`, `<thead>`, `<time>`, `<title>`, `<tr>`, `<u>`, `<ul>`, and `<wbr>`. All attributes are removed from tags, except for `href`, `title` and `hreflang` for `<a>` tags; and `src`, `width`, `height`, `alt`, and `title` for `<img>` tags. In addition, the `data-virtual-text` attribute can be placed on every tag.

be useful if the virtual text should be within a paragraph (e.g., for sentence numbering). Figure 19 shows an example document with two instances of virtual text: One inline `<strong>` element, and one block-level `<h2>` element. The HTML content of the document to produce this effect looks as listed in fig. 19.

4.4.4. Annotations

An *annotation,* in general, consists of a *start* and *end* position within a *document,* as well as an *instance* (see section 4.4.1 for more details). Annotations are represented in the text by colored background behind the annotated text passage. In the following, the user controls of the annotator are described, as well as how to create, edit, and delete annotations.

4.4.4.1. The Annotator Interface

Figure 20.: The annotator consists of of two main components: the *document area* on the left, and the *editor pane* on the right. A section on the top displays information about the current document. Annotations are displayed by colored background behind the annotated text. Evidence is represented by links connecting multiple annotations. The evidence is ordered into vertical pathways on the left of the document area, so-called *swimlanes.*

After having selected a document in the document selection screen (section 4.4.2), it and its annotations and evidences are displayed in the annotator. The annotator interface, shown also in fig. 20, consists of two main views. The *document area* on the left shows the document, annotations, and evidences; and the *editor pane* on the right shows annotation and evidence editors, when open.

The document area is scrollable, and the document text is displayed here with a large line height to accomodate links between the rows. A scrollbar on the left of the document area shows the current position in the document, and the positions of annotations are also indicated here. Annotations within the text are indicated by a colored background, where different colors signify different types of annotations (place, person, religion, time group).

Evidences are groups of one place annotation, one religion annotation, zero or one person annotations, and zero or one time group annotations (see section 4.4.1). In the annotator, they are represented by a line connecting all annotations that are part of the evidence. If the annotations are in different lines of the text, the line takes a detour via the left margin of the document to avoid crossing text. In the margin, the evidence links are horizonally distributed into *swimlanes* to avoid overdrawing. For multiple evidence links going into the same line of text, they are also vertically distributed in the same fashion.

The editor pane is where the annotation or evidence editor is displayed when creating or editing an annotation. This is described in more detail below. Initially, the editor pane is empty, as no annotation or evidence is being edited.

4.4.4.2. Creating an Annotation

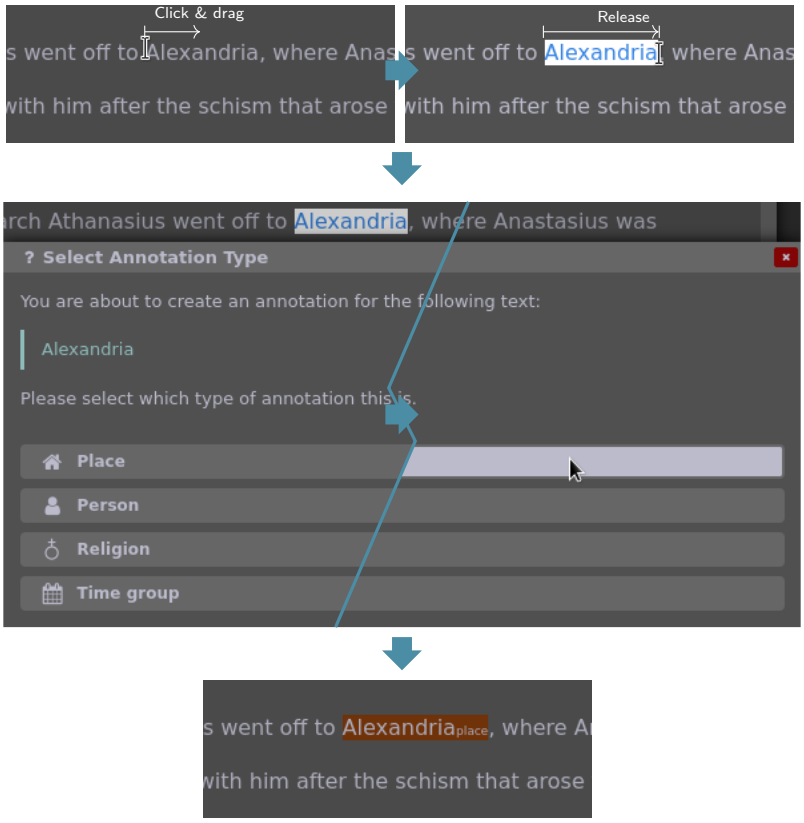

Figure 21.: Annotation creation starts by selecting a text passage using the mouse. Then, a dialog window pops up, where the type of annotation is selected. After editing and saving the new annotation, it appears in the text.

The process for creating a new annotation is shown in detail in fig. 21. Initially, the text passage that is to be annotated needs to be selected using *regular text selection;* that is, going over the start of the text passage with the mouse, push and hold down the left mouse button, drag the mouse until the end of the text passage, and release the left mouse button. The selection should become highlighted while doing this.

As soon as you release the left mouse button, the selection of the text passage is finished. The selected text is what will become the annotation. Because there are four types of annotations, depending on which type of instance is represented (see section 4.4.1), next a pop-up window appears, where the type of annotation needs to be selected. The window also shows the content of the annotation again, and there are four buttons, one for each type of annotation. By clicking one of the buttons, that type is selected, the pop-up window is closed, and the *annotation editor* is opened. If you want to re-select the text passage or abort the creation of the annotation for any other reason, you can either click on the red cross in the top right, or anywhere outside of the pop-up window.

Next, in the *editor pane,* an annotation editor appears, where you can fill out the data for that annotation and the connected instance. This editor window is described in more detail in section 4.4.4.4, as the process for creating and editing existing annotations is quite similar. The only difference is that:

1. the button for closing the editor will cancel creation without a prompt,
2. the delete button at the bottom is instead labeled *"Cancel creation"* and serves the same function,
3. the save button at the bottom is instead labeled *"Create"* and clicking it will persist the new annotation and instance to the database, and

4. the editor row for changing the text extent of the annotation is missing.

Finally, after clicking *"Create"* in the editor, the editor will close, and the new annotation will appear in the document. Annotations *may* overlap, and may even cover exactly the same text passage. It is completely fine to do a text selection over an existing annotation in the text. When annotations overlap, the parts where they do are highlighted in gray instead of the normal annotation colors.

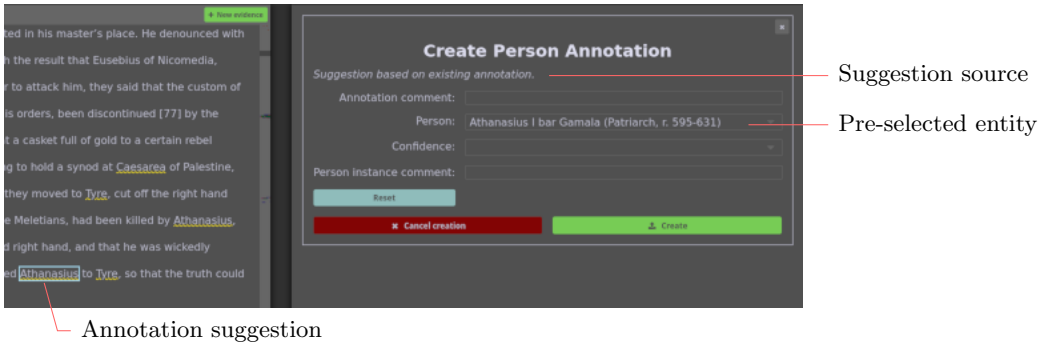

Figure 22.: Suggestions for new annotations are indicated by yellow curly underlining. These suggestions are based on known names for entities from the database, as well as existing annotation content.

The server will also generate suggestions for annotations. These are based on known names (place names, alternative names from other languages, religion and person names) from the database, as well as on existing annotations in the current document. Figure 22 shows an example: The person *Athanasius I bar Gamala* has been annotated in the text previously (under the name *"Athanasius"*). Based on that, the suggestion for a person annotation at this position is suggested. *Clicking* on the annotation suggestion, which is indicated by a yellow curly underline, will open an annotation editor of the respective type, with the suggested entity (place, person, or religion) already selected. The editor now also indicates where the suggestion stems from. Clicking *Create* will commit the annotation to the database, replacing the suggestion.

4.4.4.3. Selecting an Annotation

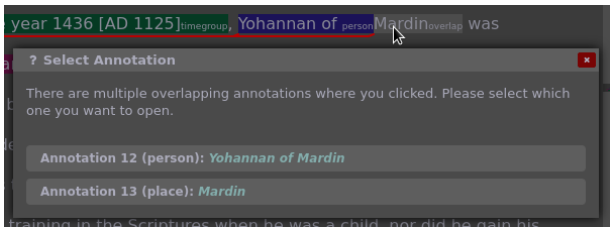

Figure 23.: When clicking on an overlap between two or more annotations, a pop-up window appears, where you can select which annotation you meant to click on.

Selecting an annotation is as simple as clicking on it in the document area. When hovering over the annotation with the mouse, it is already outlined. Especially in cases where annotations are very long, or there are overlaps with other annotations, this outline can be helpful for understanding which annotation is currently under the mouse cursor. When annotations overlap and you click on the gray section (i.e., the overlapping part), it is not immediately clear which annotation you want to select. In that case, a pop-up window appears, listing the different annotation candidates with their type and content (see fig. 23). By clicking on the intended annotation in the list here, it is selected. The selection process can be cancelled by clicking on the red cross, or anywhere outside of the pop-up window.

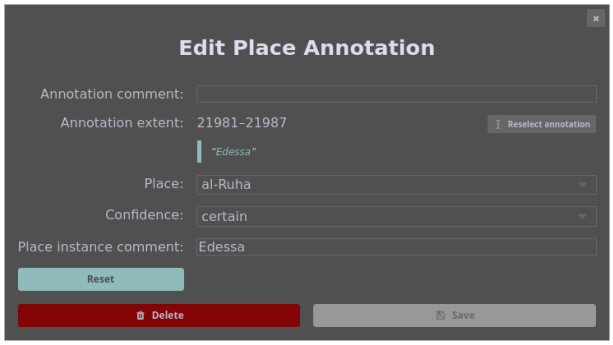

Figure 24.: The annotation editor for a place annotation.

4.4.4.4. Editing an Annotation

In the annotation editor, you can edit the *annotation comment,* which is the `comment` field of the record in the `annotation` table. This might be useful if there is something special about the placement of the annotation in the text, or some useful context. You can also see and edit the textual extent of the annotation (more details in the next paragraph). Further, you can edit the *instance* data. The editors for the different types of annotations, therefore, look slightly different. Figure 24 shows an editor for a place annotation: Here, the annotation comment field is empty. For the place instance, the place itself, the location confidence, and the comment in the `place_instance` table can be edited. The place is selected via a drop-down menu, as is the confidence. The comments are entered using text fields. For person and religion annotations, the editors look quite similar, but the first drop-down menu lists persons or religions, respectively.

An annotation has a start and end position in the text, which are stored in the `annotation` table of the database. As the placement of the annotation could need to be changed, the editors provide a way to see the current extent, and to edit it. Under the title *"Annotation extent,"* three elements are visible: A representation of the start and end position of the annotation, a button to start editing, and a text area where the textual content of the annotation is shown. To change the extent, click on the button, which is initially labeled *"Reselect annotation"* (see fig. 24) The button now turns red and the text says *"Cancel"* (see fig. 25), and clicking it again will go back to the previous state. By now selecting a text passage in the document area, the textual extent of the annotation will be updated. As with all other attributes of the annotation and instance, the changes will only be put into the database when clicking on the save button. The annotation's textual extent can only be edited for existing annotations, and therefore this facility is not displayed when creating a new annotation.

For the time group annotations, the editor looks a bit different because of the way time groups work: One time group can have zero or more time instances, and all of them would be attributed to the annotation. Figure 25 shows an editor for a time group annotation. Besides the annotation comment and textual extent, time instances are shown as separate items, where the comment, confidence, start time, and end time can be edited. In this case, start and end time must be numbers, and the end time must be greater than or equal to the start time. Each time instance can separately be removed by clicking on the *"Delete instance"* button in the respective box, and new time instances can be added via the large *"New time instance"* button. All changes, additions and deletions are only persisted to the database when the entire time group annotation is saved with the save button in the lower right, and are not persisted if the editor is closed, discarding the edits.

Clicking on the *reset* button in the lower left of the editor will revert the values to their initial state, as if the editor was freshly opened. The editor can be closed by clicking on the cross in the upper right. If there are unsaved changes, a prompt will appear to confirm that those changes should be discarded. The green *save* button in the lower right will persist all changes to the database. The button will be greyed out and disabled if there are no changes to the annotation yet. The red *delete* button will delete the instance and annotation (see section 4.4.4.5).

All form data in the editor is validated. If a field is empty *and* mandatory (e.g., no place is selected for a

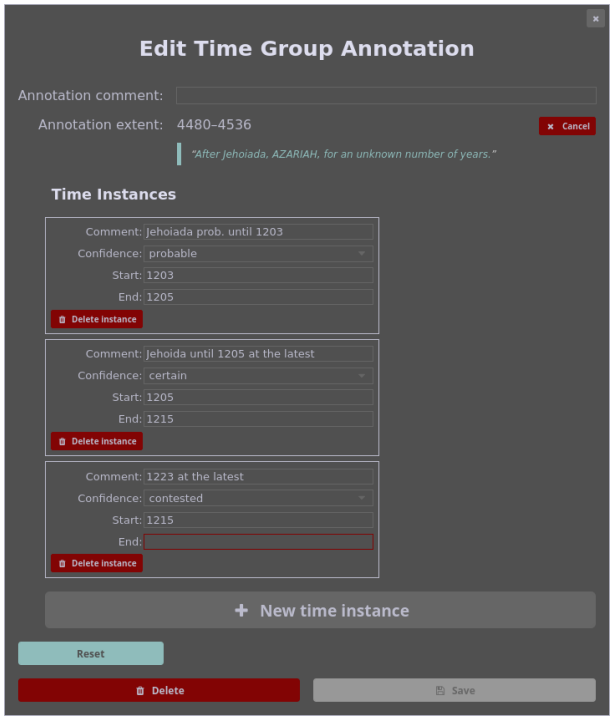

Figure 25.: The annotation editor for a time group annotation. For the third time instance, the mandatory end time field is empty, and the input is therefore outlined in red. The user is currently editing the textual extent of the annotation, and the button in that section of the editor therefore reads *"Cancel."*

new place annotation), the field will be outlined in red to signify that. Similarly, if the content of an input field is invalid (e.g., end time before start time for a time instance), it will be outlined in red as well. In both cases, the save or create button will be disabled and greyed out.

4.4.4.5. Deleting an Annotation

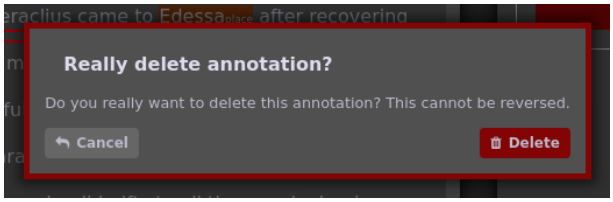

Figure 26.: A confirmation dialog appears when deleting an annotation.

The red *delete* button in the bottom left of the annotation editor will delete the instance and annotation. Deleting an instance is only possible if the instance is not part of an evidence, and therefore this button is greyed out and disabled if that is the case. When clicking on the button, a confirmation dialog will first appear to make sure that this is the intended action, see fig. 26. When clicking *cancel,* the deletion is not performed. When clicking *delete,* the annotation and the instance will be removed from the database, and the annotation will disappear from the document area.

4.4.5. Evidences

An *evidence,* in general, is a grouping of a *place instance,* a *religion instance,* and optionally a *person instance* and a *time group* (see section 4.4.1). In addition, the evidence has a comment field, the *interpretation*

confidence which specifies how confident you are in the interpretation of the source when creating the evidence, and a visibility flag that controls whether the evidence will appear in the visualization or not.

Each evidence also has a *source instance,* which for annotator-generated evidences is created automatically based on the document's source. Here, the *source confidence,* which specifies the source's trustworthiness for that specific evidence, can also be set. Last, evidences can be *tagged* with zero or more tags.

4.4.5.1. Selecting an Evidence

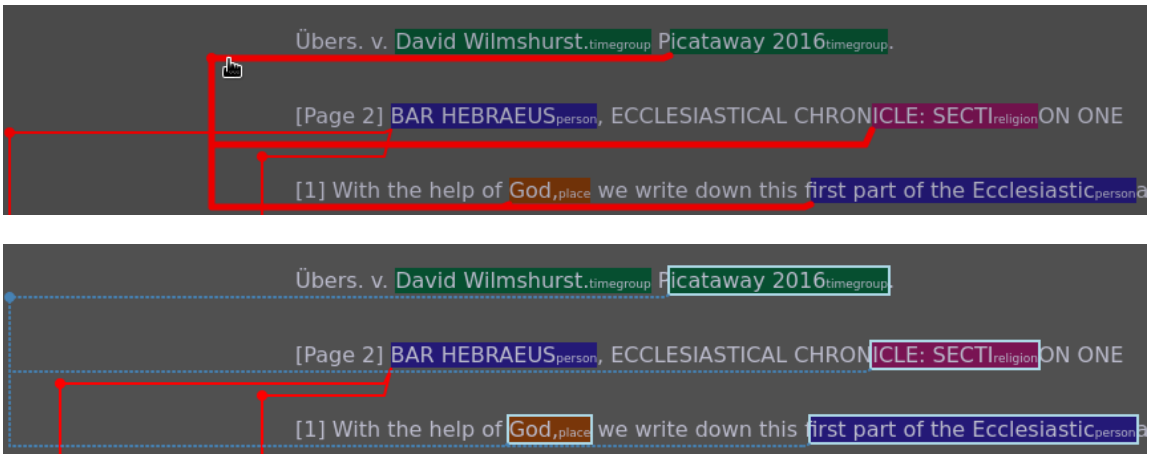

Figure 27.: To select an evidence, hover on the link with the mouse. This will already make the link bolder. Then clicking the link will select the evidence. The link will then turn blue and start to be animated, and the contained annotations are also outlined and animated.

When selecting an evidence, it is opened in the evidence editor. When creating a new evidence, that new evidence is automatically selected for as long as it is edited. An existing evidence is selected by clicking anywhere on the link. The links are layed out in a way that they overlap as little as possible. To further distinguish which evidence is currently under the cursor, the link gets bolder when the mouse hovers on it. Clicking on a link will *select* that evidence. It is then opened in the evidence editor. Further, the evidence link and the connected annotations are highlighted differently, with blue color and animation, as shown in fig. 27. Selecting a different evidence while the editor is opened will switch to editing that evidence; however, if there are unsaved changes, a confirmation prompt is shown first.

4.4.5.2. Creating an Evidence

To create an evidence, simply click on the *"New evidence"* button in the top right of the document area (see fig. 20). This will open the evidence editor with a new evidence (see fig. 28). While the evidence editor is opened, the button is greyed out and disabled. As with annotations, creating a new or editing an existing evidence is very similar, and so the description of the editor itself is described below, in section 4.4.5.3. And again, there are slight differences in the three buttons (compare also fig. 28 and fig. 29).

1. the button for closing the editor will cancel creation without a prompt,
2. the delete button at the bottom is instead labeled *"Cancel creation"* and serves the same function, and
3. the save button at the bottom is instead labeled *"Create"* and clicking it will persist the new evidence to the database.

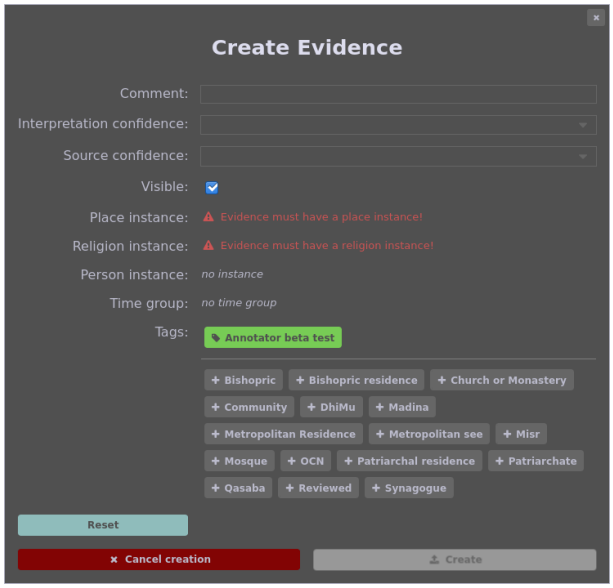

Figure 28.: Evidence editor for a new evidence. The mandatory place and religion instances are empty, and the evidence can therefore not be created yet.

4.4.5.3. Editing an Evidence

The evidence editor, shown in fig. 28 and fig. 29, contains four traditional form fields, which are all optional. These form fields can be edited in a straightforward manner:

- a text field for the `comment` field of the evidence,
- a drop-down menu for the evidence's *interpretation confidence,*
- a drop-down menu for the source instance's *source confidence,* and
- a checkbox to toggle the visibility of the evidence in the visualization[18].

The next four rows represent the place instance, religion instance, person instance, and time group that are part of the evidence. Here, as the place and religion instance are mandatory, these will show up as red when empty (see fig. 28). These four fields cannot be directly edited (i.e., by clicking or typing in them), but instead are controlled via the *document area.* While the editor is opened, the annotations that are part of the evidence are outlined in blue and animated. Changing membership of annotations works as follows (see also fig. 30): To add an annotation and its instance to the evidence, *click* on the annotation. To remove an annotation and its instance that are already part of the evidence, also *click* on the annotation (clicking *toggles* membership). If the evidence already contains an instance and annotation of a certain type, clicking on a *different* annotation of that type *replaces* the previous instance and annotation with the new ones; for example, if there are two place annotations in the text, one for **Edessa** and one for **Damascus,** with **Edessa** being part of the evidence, clicking on the **Damascus** annotation would replace the place instance in the evidence, and the evidence would now be related to the place Damascus. An instance and annotation can be associated with multiple evidences.

The fields in the editor displaying the instances cannot be interacted with directly. They show more information on the instances themselves and update automatically. In particular, they show the instance ID, the name and ID of the entity the instance refers to (place, religion, or person), and the respective instance confidence. For the time group, a comma-separated list of all time instances, with start time, end

[18] Evidence tuples that do not have the `visible` flag set are *never loaded* from the database. While most other filters in the visualization will hide or show evidence dynamically, evidences that are not *visible* will never show up. Of course, the visibility can be changed later.

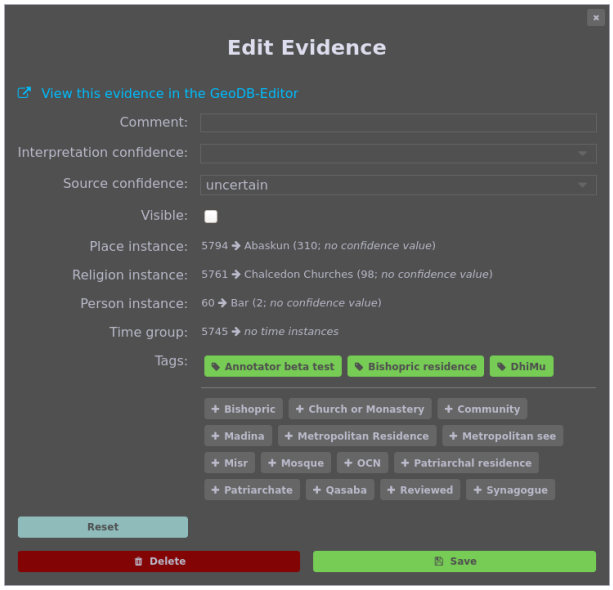

Figure 29.: Evidence editor for an existing evidence.

time, and confidence is shown instead. The data about the instances must be fetched from the database when it changes, so directly after opening the editor, or when toggling membership of an annotation and instance, the data is not available for a short while, and a loading indicator is shown instead.

If no instance of that type is connected, this is indicated instead, as shown in fig. 28. As the place and religion instances are mandatory, the absence of those is highlighted in red with more urgency.

The last part of the evidence editor is the evidence tags. Evidence can be tagged to create specific groups of evidences; for example, evidences that refer to Bishopric residences, or evidences that have been thoroughly reviewed. An evidence can have zero or more tags. Tags that are associated with the evidence are displayed at the top in green, tags that are not are displayed below in grey, with a *plus* symbol instead of the *tag* symbol. To remove an associated tag, click on it, and it moves to the bottom. Similarly, to add a tag, click on it, and it moves up and becomes green.

As with the annotation editor, all changes made are only local until you click the *save* button at the bottom (or, for new evidences, the *create* button). Using the *reset* button, the initial state from the database can be restored for all fields, discarding all changes. Clicking on the cross in the top right closes the editor if there are no unsaved changes, otherwise a confirmation prompt is shown first. The same prompt is also shown when trying to open a different evidence by clicking on the respective link in the document area. The *delete* button deletes the evidence, see section 4.4.5.4. The *save* button will persist all changes to the database. This button will only be enabled if it is currently possible to save: If there are no changes, it is disabled and greyed out. Further, if there is invalid input (i.e., no place or religion instance selected), saving is also not possible.

For evidence that is already saved in the database (i.e., when editing evidence, but not during creation), the evidence editor also shows a link at the top labeled *"View this evidence in the GeoDB-Editor."* Clicking this link will open the GeoDB-Editor, and there select the place of the evidence, scroll down to the evidence table, and select the evidence there as well. For evidence created using the annotator, a similar link exists in the evidence table of the GeoDB-Editor, which opens the appropriate document in the annotator and opens the respective evidence in the evidence editor.

4.4.5.4. Deleting an Evidence

Clicking on the red *delete* button in the evidence editor will delete the evidence from the database. A confirmation dialog (see fig. 31) will appear first to avoid accidental deletions. In the case of new evidences that have not been saved yet, the button will instead be labeled *"Cancel creation."*

Figure 30.: To toggle membership of an annotation, and its associated *instance,* to an evidence, click on the annotation while the evidence editor is opened. The link representing the evidence will update instantly.

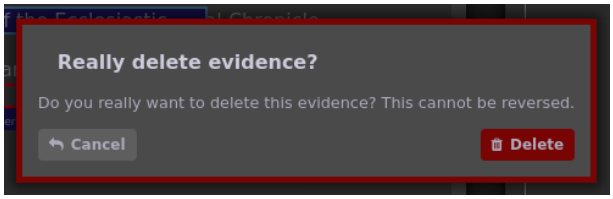

Figure 31.: A confirmation dialog appears when deleting an evidence.

Deleting an evidence will *not* delete the connected annotations and instances. Those will remain in the database and the document area. It will only delete the evidence itself, the source instance, and all tag associations. The deletion will be reflected at once in the document area, where the respective link will also disappear. After deletion, the evidence editor is closed.

4.5. Place URI Page

A page for place search is linked from the header. Here, place names, alternative place names, external URIs for places, and place comment content can be used to query the database for places. This returns a list, each of whose entries is a link to the respective place URI page. This page shows details about the place from the database in a textual form, and the URL (URI) of the page is deterministic and depends on the place's ID in the database. This is meant to be a citeable resource by others.

The place URI page shows the following:

- place name
- geographical position
- place type
- location confidence

- place comment
- alternative names and their language
- external URIs for the same place
- evidences for religions in that place, with time information
- evidences for persons in that place, with time information
- sources referencing this place
- a section on how to cite the place

4.6. Reporting

Damast provides the option to generate a *report* from a specific subset of historical evidence. This subset is derived from (1) the data contained in the database, and (2) a set of *filters* concerning different attributes of that data. The format and options of these filters are formalized as JSONSchema, and they are a part of the visualization state (see section 4.2.4.1). An appropriate file of filters can be obtained (downloaded) from the settings pane of the visualization, or from an existing report. Report generation can also be triggered directly from the visualization, in which case the state is passed to the report generation software behind the scenes. In both cases, the filters used are those that are currently active in the visualization as well, meaning that the report will contain the evidences currently shown in the visualization.

The report consists of multiple sections. First, metadata about the report, such as who provisioned it and when, as well as a textual description of the filters applied within the report, and the cardinality of data entities matching these filters. Then, the evidences are listed, and their contents are represented as serialized texts. The entities contained in these evidences (places, religions, and persons) are then listed individually with the relevant information (such as alternative names for the places), and an overall timeline is shown. Finally, all sources these evidences were extracted from are listed.

All information in the individual sections is cross-referenced. So, there is a reference from an evidence mentioning a place to the entry for that place in the report; and vice versa, the places, religions, and persons reference all evidences they are mentioned in. Further, sources are referenced from the evidences, and the evidences are again referenced as a list from the source section.

In the section of places, the report also shows a map where they are all marked together with the religions mentioned in those places. This map is generated using `matplotlib` [3] and is saved both as an SVG file for the HTML version of the report, and as a standalone PDF document for the PDF version. The PDF map is also stored separately for download. The map uses Natural Earth [19] vector map data, which is licensed under CC0 (public domain).

4.6.1. Generating a Report

Reports are generated by a separate process within the Docker container that is launched by the server. Figure 32 shows the lifecycle of reports. First, the Flask server generates a UUID for the report; and writes that, the metadata, and the filter into the report database. At that point, the state of the report is `started`. Then, it launches the report generation process, passing it the UUID. The report generation process then reads the filter from the database, collects the data from the PostgreSQL database, and generates the report content. The content is put together via Jinja2 templates into a HTML page (section 4.6.3) and a TEX file, the latter of which is is then compiled into a PDF file (section 4.6.4).

Once the contents are generated, they are GZIP-compressed and written back into the SQLite3 database. The state of the report is then set to `completed`. If an error occurs during report generation, instead an error message is written to the HTML file content in the SQLite3 database, and the state is set to `failed`.

[19] `https://www.naturalearthdata.com`

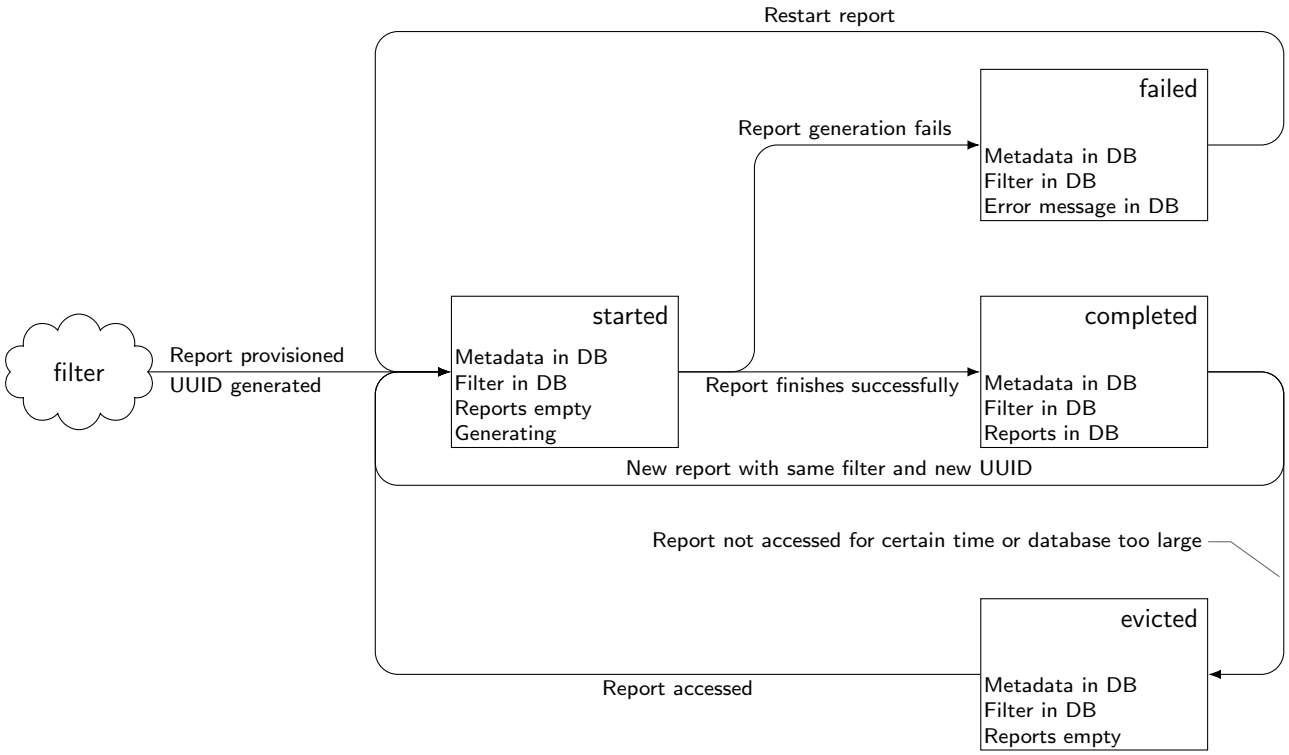

Figure 32.: The lifecycle of report records in the database. After a report gets provisioned, it is in `started` state while the report generation code runs. If that fails, it enters `failed` state, otherwise it is `completed`. After a while of not being accessed, the finished report is `evicted` and only the filter and metadata remain. On the next access, the report contents are re-generated from the database.

The report contents can be several megabytes large, depending on the number of evidences contained. To limit storage space requirements on long-running installations, it might be desirable to clean up old, unused reports. This can always be done from the outside, for example with a `cron(8)` job, working directly with the SQLite3 database. However, this might not be an ideal solution if generated reports should be guaranteed to persist. For this, the special `evicted` report state exists, into which a report can be moved. In that state, the filters and metadata remain in the database, but the report contents are deleted. The next access to that report then trigger the re-generation of the report[20], and move it back to the `started` state. Report eviction can also be done manually by setting the content fields to `NULL` in the SQLite3 database, and the `state` field's value to `'evicted'`. There is also the option to configure the Damast system to handle this automatically via two environment variables:

`DAMAST_REPORT_EVICTION_DEFERRAL`: If this variable exists and is not empty, it specifies a number in days. A report is moved to the `evicted` state if it has not been *accessed*[21] for at least that many days. A good number of days to set here could be 90 (three months).

`DAMAST_REPORT_EVICTION_MAXSIZE`: If this variable exists and is not empty, it specifies a file size number in *megabytes*. The server then regularly checks whether the sum of report content sizes exceeds that size. If so, it will evict the least-accessed reports until that sum is less than that size. A good number to specify here could be 5,000 (5 GB).

4.6.2. Accessing a Report

Existing reports can be accessed via their UUID. When generating a report, the user is redirected to the URL where the HTML report (section 4.6.3) is later shown. While the report is in `started` state, a wait page is shown that reloads automatically from time to time. In `failed` state, the error message is shown, and in `evicted` state, opening the page triggers re-generation of the report, and the `started` state's wait page is shown again. Separate links exist to the PDF version of the report (section 4.6.4) and the standalone map. All links require knowledge of the report UUID, or access via the list of exiting reports (section 4.6.5).

4.6.3. HTML Report

The HTML report is rendered from a HTML Jinja2 template, which in turn uses many smaller templates and macros to create the contents. The HTML report content stored in the SQLite3 database is rendered on demand into a template in the website hosted by Damast. Besides its contents, it contains links to the PDF version, the standalone map, the filters used, a trigger to re-generate the report now, and a link to the visualization, which is then opened with these filters active.

4.6.4. PDF Report

The PDF report is generated using LaTeX, specifically XeLaTeX. In a first step, the TeX code is rendered from a Jinja2 template, which in turn uses many smaller templates and macros to create the contents. The `latexmk` utility is then used to compile the TeX code within the Docker container. The resulting PDF file is stored into the SQLite3 database. Accessing the PDF file[22] then offers the PDF file for download. The PDF version of the map is used in the PDF report and compiled into it by XeLaTeX. It can also be downloaded separately[23].

[20] **Important note** about report eviction: If the goal is to guarantee that reports are always recallable in that state, two other things need to be guarantee on that system: (1) The software version of the Damast system (or at least the report generation software and templates) must not change, and (2) the contents of the underlying PostgreSQL database must not change.

[21] Note that eviction deferral does not consider the time since the *creation* of the report, but rather the time since its *last access*. Reports might be generated and never again accessed, but they might also be accessed on regular basis, in which case eviction would be deferred.

[22] Via `${DAMAST_PROXY_PREFIX}/reporting/<uuid>/pdf`

[23] Via `${DAMAST_PROXY_PREFIX}/reporting/<uuid>/map`

4.6.5. List of Reports

In the miscellaneous pages (see section 4.7), a page that lists existing reports is also available. Users with the `admin` role see all reports in the database, other users only see their own. However, this page only exists for users that are logged in, not for anonymous visitors. This is because visitors (if they have the `reporting` role) all generate reports under the `visitor` pseudo-user, and there is no way to determine which reports a visitor has provisioned.

The list shows the user, UUID, server version, start time, run duration, and number of evidences. Further, there are links to the HTML and PDF versions, the standalone PDF map, and the filter file. Another link will trigger report re-generation with the same filters[24]. Lastly, a link to the visualization will open that with the filters of that report active.

4.7. Other Pages

A few other, miscellaneous pages exist on the `docs` blueprint. These also include an index page under `${DAMAST_PROXY_PREFIX}/docs/` that shows links to all of these pages that the user is allowed to visit.

4.7.1. Documents

The documents page contains links to a few smaller pages and utilities. Some of these are not available to all users, but only to developers or administrators. Besides the page links listed in the following, the documents page also has links to the report creation page and the list of existing reports detailed in section 4.6.

Database edit log. This page is visible only to administrators (role `admin`). It details the contents of the `user_action` table in reverse chronological format. This data can be reviewed for provenance, or to quickly retrieve an old value from a database entry after a mistake in an edit.

REST API documentation. This page is visible to all developers (role `dev`), and contains a list of the endpoints in the REST API (see section 3.4). They are the HTTP endpoints under `/rest/`, which are listed in the form they are defined in Flask. The URL and possible HTTP verbs are extracted from Flask runtime information, and the documentation string is taken directly from the Python *docstring* of that method. The REST API methods' docstring are all quite detailed and show request and response content types, possible values and data entries, and payload or response examples, where applicable.

Annotator user guide. The annotator user guide explains how the annotator works and is used in great detail, and with example images. This page is visible to all users with the `annotator` role. The annotator user guide is the basis for the text in section 4.4.

PostgreSQL Database Schema. Download link to a PDF of the database structure. This link is available to all developers (role `dev`) and to users with the `pgadmin` role. The PDF is the same as fig. 1, enclosed in a page with headers and footers.

Download database dump. Download link to a SQL dump of the entire database. The file is served as GZIP-ed SQL, and is generated on the fly. This functionality is available to all users. However, if the requesting user does *not* have the `admin` role, the dump is data-only, and also omits the `action_type`, `user`, and `user_action` provenance tables described in section 2.1.2.

[24] This will create a *new* report and UUID. Note that this does not make sense on a system where the PostgreSQL database contents never change.

4.7.2. GDPR, Imprint, Cookie Preferences

A separate blueprint serves pages for the GDPR[25] content. These pages are accessible even without login, and are available as links in the footer of each page[26]. A general data protection page states what data is collected from visitors, and what rights they have under GDPR. The imprint *(Impressum)* page contains information about how to get in touch with the website administrators. Finally, the websites use cookies and `localStorage` to store state and information on the user's computer. These are used for login, but also for functionalities such as storing table or visualization layout. In accordance to § 7 ¶ 2 GDPR, consent to store these cookies needs to be given explicitly by users. For this, they are shown a consent popup when they first visit the site, with the choices of (1) no cookies, (2) only necessary cookies, or (3) all cookies. Users must be able to amend or withdraw consent at any time (§ 7 ¶ 3 GDPR), so the consent settings shown in the popup are also available as a separate page.

Users must consent to necessary cookies (option (2)) to be able to login, as the session cookie must be stored. Further, the consent itself is stored as a cookie. Consequently, visitors who reject all cookies (option (1)) will be shown the cookie consent popup each time they open a page, as there is no way to store their choice. The "all cookies" option (3) is necessary to use additional features such as storing custom visualization and table layouts.

When hosting the Damast system somewhere else, a different imprint than the default one is needed if the administrators and hosting organization differ. Probably, the data protection page needs to be modified as well. Here, the blueprint override functionality discussed in section 3.3.2 can come in handy.

4.7.3. Login and Password Management

A blueprint provides login and logout functionality, as well as a page to change the current user's password. These are linked in the top right of the page, in the header. If a visitor is not logged in, a link to the login page is shown there. If they are logged in, the user's name, a link to the password change page and a link to log out is shown.

On the password change page, the user must enter their current and their new password, and then submit. The server backend will check that the old password matches, and that the new password has sufficient complexity (entropy). If that is the case, the password hash is updated in the user database and the session token is renewed.

[25] **GDPR:** General Data Protection Regulation of the European Union. Called DSGVO *(Datenschutzgrundverordnung)* in German.

[26] In the visualization, which has a tighter layout without a footer, these links are placed in the header instead.

Bibliography

[1] Stuart K Card, Jock D Mackinlay, and Ben Shneiderman. *Readings in information visualization: Using vision to think*. Morgan Kaufmann Publishers Inc., 1999.

[2] Max Franke, Ralph Barczok, Steffen Koch, and Dorothea Weltecke. "Confidence as First-class Attribute in Digital Humanities Data". In: *Proceedings of the 4th VIS4DH Workshop* (Oct. 2019). URL: http://vis4dh.dbvis.de/papers/2019/VIS4DH2019_paper_1.pdf.

[3] J D Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[4] Dorothea Weltecke, Steffen Koch, Ralph Barczok, Max Franke, and Bernd Andreas Vest. *Data Collected During the Digital Humanities Project 'Dhimmis & Muslims - Analysing Multireligious Spaces in the Medieval Muslim World'*. Version 1. DaRUS, Mar. 2022. DOI: 10.18419/darus-2318. URL: https://darus.uni-stuttgart.de/dataset.xhtml?persistentId=doi:10.18419/darus-2318.

A. REST API Endpoint Documentation

/rest/annotation-suggestion/<int:as_id>

DELETE OPTIONS

Delete an annotation suggestion.

/rest/annotation/<int:annotation_id>

DELETE GET HEAD OPTIONS PATCH PUT

CRUD endpoint to get document metadata for a document.

[all] @param annotation_id ID of annotation

GET @returns application/json

Get the annotation data for annotation ‘annotation_id‘. Example payload:

```
{
    "id": 1,
    "document_id": 3,
    "span": "(422,431)",
    "comment": "comment content"
}
```

PUT @returns application/json

Create a new annotation. Returns the created annotation tuple’s ID. Example payload:

```
{
    "document_id": 4,
    "span": "[0, 10]",
    "comment": "foo bar"
}
```

PATCH @returns 205 Reset Content; empty body

Modify an existing annotation. Example payload:

```
{
    "comment": "new comment"
}
```

DELETE @returns application/json

Delete an annotation, and its connected instance. This will fail if the instance is still in use by an evidence tuple. Returns the deleted annotation’s and instance’s IDs.

/rest/annotator-evidence-list

GET HEAD OPTIONS

This endpoint returns a list of ‘evidence_id‘, ‘document_id‘ tuples for all evidence that was created using the annotator; i.e., all evidence whose instances are connected to annotations.

Example return value excerpt:

```
[[8122, 1], [8125, 1], [8145, 2], ...]
```

/rest/confidence-values

GET HEAD OPTIONS

Get a list of all confidence values.

@returns application/json

Example return value:

```
['false', 'uncertain', 'contested', 'probable', 'certain']
```

/rest/document/<int:document_id>

GET HEAD OPTIONS

CRUD endpoint to get document content for a document.

GET @param document_id ID of document
@returns document.content_type

This endpoint gets the actual document data for the document with ID ‘document_id‘. Because those documents can be quite large, this endpoint supports the ‘Range‘ HTTP header, but only in the following forms:

For ‘text/plain‘ type documents: ‘Range: bytes=[n]-[m]‘
For ‘text/html‘ type documents: ‘Range: content-characters=[n]-[m]‘

where ‘[n]‘ and ‘[m]‘ are both optional numbers. A range header with more than one range is not supported at the moment. The return code of the request is either 200 or 206 for successful requests, 416, 400 or 404 for unsuccessful requests.

SIDENOTE: The ‘<unit>=-<suffix-length>‘ variant of the ‘Range‘ header is currently not supported correctly, but instead is interpreted as ‘<unit>=0-<suffix-length>‘.

/rest/document/<int:document_id>/annotation-list

GET HEAD OPTIONS

REST endpoint to get a list of annotations associated with a document.

This is a slice of the ‘annotation_overview‘ VIEW. It contains all the data from the ‘annotation‘ table, alongside the connected instance IDs and evidence ID. The ‘span‘ property is an array with the first and last inclusive index of the annotation. As an instance can be part of multiple evidence tuples (i.e., an annotation can be in multiple groups), the ‘evidence_ids‘ is an array.

Example return value for ‘/rest/document/1/annotation-list‘:

```
[
  {
    "id": 1,
    "document_id": 1,
```

```
      "span": [ 0, 4 ],
      "comment": "test comment",
      "place_instance_id": null,
      "person_instance_id": 1623,
      "religion_instance_id": null,
      "time_group_id": null,
      "evidence_ids": [ 3611 ]
    },
    ...
  ]
```

This endpoint can also be requested as in LD-JSON format. In this case, it is assumed to be required for the Recogito tool. NOTE that to properly show the annotations in Recogito, the ‘{ mode: ’pre’ }‘ configuration value must be set, as the DOM annotator we use internally respects all white-space in the document without collapsing it.

Example return value excerpt for

```
  GET /rest/document/7/annotation-list
  Accept: application/ld+json

  [
    {
      "@context": "http://www.w3.org/ns/anno.jsonld",
      "body": [
        {
          "purpose": "tagging",
          "type": "TextualBody",
          "value": "Religion"
        },
        {
          "purpose": "tagging",
          "type": "TextualBody",
          "value": "Christianity"
        }
      ],
      "id": "#31847",
      "target": {
        "selector": [
          {
            "end": 479,
            "start": 474,
            "type": "TextPositionSelector"
          },
          {
            "exact": "Amen.",
            "type": "TextQuoteSelector"
          }
        ]
      },
      "type": "Annotation"
    },
    ...
    {
      "@context": "http://www.w3.org/ns/anno.jsonld",
      "body": [
```

```
        {
          "purpose": "tagging",
          "type": "TextualBody",
          "value": "religion"
        }
      ],
      "id": "#5843_religion",
      "motivation": "linking",
      "target": [
        {
          "id": "#31826"
        },
        {
          "id": "#31840"
        }
      ],
      "type": "Annotation"
    },
    ...
  ]
```

/rest/document/<int:document_id>/annotation-suggestion-list GET HEAD OPTIONS

REST endpoint to get a list of annotation suggestions associated with a document.

@returns application/json

Exemplary return value except:

```
  [
     {
       "document_id": 3,
       "entity_id": 772,
       "id": 830681,
       "source": [
         "name"
       ],
       "span": [
         1372645,
         1372657
       ],
       "type": "place"
     },
     ...
  ]
```

/rest/document/<int:document_id>/evidence-list GET HEAD OPTIONS

Get a list of evidence tuples based on a document.

@param document_id ID of document
@returns application/json

An evidence tuple is based on a document if any of its instances'
annotations is located in that document.

Example return value excerpt:

```
  [
    {
        "comment": "Bischof nachgewiesen",
        "id": 1632,
        "interpretation_confidence": null,
        "person_instance_id": null,
        "place_instance_id": 1632,
        "religion_instance_id": 1632,
        "time_group_id": 1632,
        "visible": true
    },
    ...
  ]
```


/rest/document/<int:document_id>/metadata

GET HEAD OPTIONS

CRUD endpoint to get document metadata for a document.

```
GET       @param document_id            ID of document
          @returns                      application/json
```

Example return value for ‘/document/3/metadata‘:


```
  {
    "comment": "Michael Rabo, Chronography",
    "content_length": 2926379,
    "content_type": "text/html;charset=UTF-8",
    "default_source_confidence": null,
    "document_version": 1,
    "id": 3,
    "source_id": 67,
    "source_name": "Michael der Syrer; Moosa, Matti (2014): The Syriac ...",
    "source_type": "Primary source"
  }
```


/rest/document/list

GET HEAD OPTIONS

GET a list of all documents.

Example return value excerpt:


```
    [
      {
        "comment": "Michael Rabo, Chronography",
        "content_length": 3012062,
        "content_type": "text/html;charset=UTF-8",
        "default_source_confidence": null,
        "document_version": 1,
        "id": 3,
        "source_id": 67,
        "source_name": "Michael der Syrer; Moosa, Matti (2014): The Syriac ...",
        "source_type": "Primary source"
      },
      ...
    ]
```

/rest/dump/

GET HEAD OPTIONS

Get a database dump of the PostgreSQL database.

@returns application/sql

This dumps the database and returns the SQL. If the user requesting it is an administrator, the entire database is dumped, including the user and provenance tables.

/rest/evidence-list

GET HEAD OPTIONS

Get a list of compact evidence tuples from the view ‘place_religion_overview‘.

@returns application/json

This replaces the ‘/PlaceReligion‘ API endpoint of the old servlet implementation. Returns a JSON array of objects with a place ID, evidence tuple ID, religion ID, and a time span. Only evidences with ‘evidence.visible‘, ‘place.visible‘ and ‘place_type.visible‘ are listed!

Example return value excerpt:

```
[
  {
    "place_id": 1,
    "religion_id": 4,
    "time_span": {
      "end": 1200,
      "start": 800
    },
    "tuple_id": 1,
    "source_ids": [12]
  },
  ...
]
```

/rest/evidence/<int:evidence_id>

DELETE GET HEAD OPTIONS PATCH PUT

CRUD endpoint to manipulate evidence tuples.

[all]	@param evidence_id	ID of evidence tuple, 0 or ‘None‘ for PUT
C/PUT	@payload	application/json
	@returns	application/json

Create a new evidence tuple. ‘place_instance_id‘ and ‘religion_instance_id‘ are required fields, the rest (‘person_instance_id‘, ‘time_group_id‘, ‘comment‘, ‘interpretation_confidence‘, ‘visible‘) is optional. Returns the IDs for the created evidence.

Exemplary payload for ‘PUT /evidence‘:

```
{
  "place_instance_id": 202,
  "religion_instance_id": 7,
  "person_instance_id": 12,
  "time_group_id": 4,
  "interpretation_confidence": "probable",
  "visible": true,
```

```
    "comment": "evidence comment: test"
  }
```

R/GET @returns application/json

Get one evidence tuple, specified by ‘evidence_id‘. Request takes no payload and returns a JSON object with data from a bunch of tables (‘place‘, ‘religion‘, ‘time_span‘, ‘source_instance‘, ‘source‘, and intermediary tables).

Exemplary reply for ‘GET /evidence/64‘:


```
  {
      "evidence_id": 64,
      "interpretation_confidence": null,
      "location_confidence": null,
      "place_attribution_confidence": null,
      "place_comment": "",
      "place_geoloc": null,
      "place_id": 46,
      "place_instance_comment": null,
      "place_name": "Beth Sayda",
      "religion_confidence": null,
      "religion_id": 4,
      "religion_instance_comment": null,
      "religion_name": "Syriac Orthodox Church",
      "sources": [
          {
              "source_id": 1,
              "source_instance_comment": "",
              "source_name": "OCN = Fiey, [...]",
              "source_page": "179"
          },
          {
              "source_id": 2,
              "source_instance_comment": "Levenq, G. (1935). : s. v. Bêth Saida. [...],
              "source_name": "DHGE = Aubert, [...],
              "source_page": "8, 1239-1240"
          }
      ],
      "time_group_id": 64,
      "time_spans": [
          {
              "comment": null,
              "confidence": null,
              "end": 1277,
              "start": 1261
          }
      ]
  }
```


U/PATCH @payload application/json
@returns application/json

Update one or more of the fields ‘comment‘, ‘interpretation_confidence‘, ‘visible‘, ‘person_instance_id‘, or ‘time_group_id‘. The

‘religion_instance_id‘ and ‘place_instance_id‘ CANNOT be updated for an existing evidence tuple.

Exemplary payload for ‘PATCH /evidence/12345‘:

```
{
  "visible": false,
  "comment": "updated comment...",
  "person_instance_id": 1234
}
```

D/DELETE @param cascade=0|1
@returns application/json

Delete evidence. If the ‘cascade‘ parameter is 1, also delete all related entities. Write ‘user_action‘ log, return a JSON with all deleted IDs.

/rest/evidence/<int:evidence_id>/source-instances

GET HEAD OPTIONS

Get all source instance entries for evidence ‘evidence_id‘.

@param evidence_id ID of evidence tuple
@returns application/json

Exemplary return value for ‘GET /evidence/3662/source-instances‘:

```
[
  {
    "comment": "new comment",
    "evidence_id": 3662,
    "id": 4702,
    "source_id": 3,
    "source_page": null,
    "confidence": "contested"
  },
  {
    "comment": "new comment 2",
    "evidence_id": 3662,
    "id": 4703,
    "source_id": 1,
    "source_page": "test",
    "confidence": "probable"
  }
]
```

/rest/evidence/<int:evidence_id>/tags

GET HEAD OPTIONS PUT

Get or set tag set for evidence.

@param evidence_id ID of evidence

GET

Get list of tag IDs as array.

@returns application/json

Return value example for ‘GET /rest/evidence/1/tags‘:

```
      [1,4,6]
```

PUT

```
    Replace list of tag IDs. Takes a JSON array as payload.

    @returns            nothing

    Payload value example for ‘PUT /rest/evidence/1/tags‘:

      [1,2]
```

/rest/find-alternative-names

OPTIONS POST

```
Retrieve ‘place.id‘ where the alternative name matches the ‘regex‘.

@payload                application/json
  @param ‘regex‘        ECMA-Script regular expression
  @param ‘ignore_case‘  If ‘true‘, regex is case-insensitive

@returns                application/json

As alternative names are not loaded initially, this API endpoint is used to
retrieve places’ ‘id‘ for text search in the location list, where an
alternative name matches the search term ‘regex‘. This API endpoint then
returns a JSON array of ‘place.id‘ for matching places.

Example response for case-insensitive serarch of "ko[a-g]":

  > POST /rest/find-alternative-names HTTP/1.1
  > Content-Type: application/json
  >
  > {
  >   "ignore_case": true,
  >   "regex": "ko[a-g]"
  > }
  >
  ---
  < HTTP/1.1 200 OK
  < Content-Type: application/json
  <
  < [
  <   236,
  <   694,
  <   493
  < ]
```

/rest/languages-list

GET HEAD OPTIONS

```
Get a list of all religions.

@returns        application/json

This returns a JSON array of objects with ‘id‘ and ‘name‘ from table
‘language‘. This API endpoint replaces ‘/LanguagesList‘ in the old servlet
implementation.

Example return value excerpt:
```

```
  [
    {
      "id": 1,
      "name": "Arabic - DMG"
    },
    {
      "id": 2,
      "name": "Arabic - AS"
    },
    ...
  ]
```

/rest/person-instance/<int:person_instance_id> DELETE GET HEAD OPTIONS PATCH PUT

```
CRUD endpoint to manipulate person instances.

[all]     @param person_instance_id      ID of person instance, 0 or ‘None‘ for PUT

C/PUT     @payload               application/json
          @returns               application/json

Create a new person instance. ‘person_id‘ is a required field.
‘confidence‘, ‘comment‘, and ‘annotation_id‘ are optional. Returns the ID
for the created person instance.

Exemplary payload for ‘PUT /person-instance/0‘:

  {
    "person_id": 12,
    "confidence": "certain",
    "comment": "foo bar"
  }


R/GET     @returns               application/json

Get one person instance.

Exemplary reply for ‘GET /person-instance/64‘:

  {
      "id": 64,
      "confidence": "certain",
      "comment": "baz",
      "annotation_id": null
  }


U/PATCH   @payload               application/json
          @returns               application/json

Update one or more of the fields ‘person_id‘, ‘comment‘, ‘confidence‘, or
‘annotation_id‘.

Exemplary payload for ‘PATCH /person-instance/12345‘:

  {
```

```
    "comment": "updated comment...",
  }
```

```
D/DELETE  @returns                application/json

Delete person instance. Write ‘user_action‘ log, return a JSON with all
deleted IDs.
```

/rest/person-list

GET HEAD OPTIONS

```
Get content of table ‘person‘ as a list of dicts.

@return     application/json

Example return value excerpt:

    [
      {
        "comment": null,
        "id": 23,
        "name": "John bar Hebraye of Tarsus (appr. 667)",
        "person_type": 2,
        "time_range": ""
      },
      ...
    ]
```

/rest/person-type-list

GET HEAD OPTIONS

```
Get a list of all person types.

@returns        application/json

This returns a JSON array of objects with ‘id‘, and ‘type‘ from table
‘person_type‘.

Example return value excerpt:

  [
    {
      "id": 1,
      "type": "Bishop"
    },
    ...
  ]
```

/rest/person/<int:person_id>

DELETE GET HEAD OPTIONS PATCH PUT

```
CRUD endpoint to manipulate person tuples.

[all]     @param person_id        ID of person tuple, 0 or ‘None‘ for PUT

C/PUT     @payload                application/json
          @returns                application/json

Create a new person tuple. ‘name‘ is a required field, the rest is optional.
Returns the ID for the created entity. Fails if a person with that name
already exists.
```

```
Exemplary payload for ‘PUT /person/0‘:

  {
    "name": "Testperson",
    "comment": "Test comment",
    "time_range": "6th century",
    "person_type": 2
  }


R/GET     @returns                application/json

Get person data for the person with ID ‘person_id‘.

@param person_id     Integer, ‘id‘ in table ‘person‘
@returns            application/json

This returns the data from table ‘person‘ as a single JSON object.

Example return value:

  {
    "id": 12,
    "name": "Testperson",
    "comment": "Test comment",
    "time_range": "6th century",
    "person_type": 2
  }


U/PATCH   @payload                application/json
          @returns                application/json

Update one or more of the fields ’comment’, ’name’, ’time_range’,
’person_type’, or ’name’.

Exemplary payload for ‘PATCH /person/12345‘:

  {
    "comment": "updated comment...",
    "name": "updated name"
  }


D/DELETE  @returns                application/json

Delete a person if there are no conflicts. Otherwise, fail. Returns the ID
of the deleted tuple.
```

/rest/place-instance/<int:place_instance_id>

DELETE GET HEAD OPTIONS PATCH PUT

```
CRUD endpoint to manipulate place instances.

[all]     @param place_instance_id      ID of place instance, 0 or ‘None‘ for PUT

C/PUT     @payload                application/json
          @returns                application/json
```

```
Create a new place instance. ‘place_id‘ is a required field.
‘confidence‘, ‘comment‘, and ‘annotation_id‘ are optional. Returns the ID
for the created place instance.

Exemplary payload for ‘PUT /place-instance/0‘:

  {
    "place_id": 12,
    "confidence": "certain",
    "comment": "foo bar"
  }


R/GET     @returns                application/json

Get one place instance.

Exemplary reply for ‘GET /place-instance/64‘:

  {
      "id": 64,
      "confidence": "certain",
      "comment": "baz",
      "annotation_id": null
  }


U/PATCH   @payload                application/json
          @returns                application/json

Update one or more of the fields ‘place_id‘, ‘comment‘, ‘confidence‘, or
‘annotation_id‘.

Exemplary payload for ‘PATCH /place-instance/12345‘:

  {
    "comment": "updated comment...",
  }


D/DELETE  @returns                application/json

Delete place instance. Write ‘user_action‘ log, return a JSON with all
deleted IDs.
```

/rest/place-list

GET HEAD OPTIONS

```
Get a list of places.

@arg        filter  An array of arrays specifying an advanced filter
@returns            application/json

This returns all tuples from the view ‘place_overview‘ as a JSON array of
objects. The overview contains geographical location, place ID, location
confidence, place name, and place type name.
If ‘filter‘ is specified, the resulting values are restricted.

Example return value excerpt:
```

```
[
  {
    "geoloc": {
      "lat": 36.335,
      "lng": 43.11889
    },
    "id": 1,
    "location_confidence": null,
    "name": "Mosul",
    "place_type": "Settlement"
  },
  ...
]
```

/rest/place-list-detailed

GET HEAD OPTIONS

Get a list of places, with more details.

@returns application/json

Example return value excerpt:

```
[
  {
    "external_uris": [
      "IndAnat:37356",
      "https://nisanyanmap.com/?yer=37356",
      "syriaca:285",
      "https://syriaca.org/place/285",
      "EI2:SIM_0749",
      "http://dx.doi.org/10.1163/1573-3912_islam_SIM_0749",
      "EI1:SIM_0872",
      "http://dx.doi.org/10.1163/2214-871X_ei1_SIM_0872",
      "EI3:COM_23768",
      "http://dx.doi.org/10.1163/1573-3912_ei3_COM_23768"
    ],
    "name_vars": [
      "Arzun, Arzon",
      "Arzūn, Arzōn",
      "...",
      "Arzan",
      "..."
    ],
    "place_comment": "There are two Arzan. [...]",
    "place_id": 36,
    "place_name": "Arzan"
  },
  ...
]
```

/rest/place-set

GET HEAD OPTIONS POST

REST endpoint for place sets. Place sets are used as a filtering possibility in the visualization, and are stored in the database to be shared between users.

GET Returns a JSON list of all place sets in the database.
POST Accepts ONE JSON place set as a payload. Depending on whether the UUID exists in the database already, the entry is either

```
            overwritten, or a new entry is created.
```

/rest/place-type-list

GET HEAD OPTIONS

```
Get a list of all place types.

@returns        application/json

This returns a JSON array of objects with ‘id‘, ‘type‘, and ‘visible‘ from
table ‘place_type‘. This API endpoint replaces ‘/PlaceTypeList‘ in the old
servlet implementation.

Example return value excerpt:

  [
    {
      "id": 1,
      "type": "Unknown",
      "visible": true
    },
    ...
  ]
```

/rest/place/<int:place_id>

DELETE GET HEAD OPTIONS PATCH PUT

```
CRUD endpoint to manipulate place tuples.

[all]     @param place_id         ID of place tuple, 0 or ‘None‘ for PUT

C/PUT     @payload                application/json
          @returns                application/json

Create a new place tuple. ‘name‘ is a required field, the rest is optional.
Returns the ID for the created entity. Fails if a place with that name
already exists.

Exemplary payload for ‘PUT /place/0‘:

  {
    "name": "Testplace",
    "comment": "Test comment",
    "geoloc": "(48.2,9.6)",
    "confidence": "contested",
    "visible": true,
    "place_type_id": 2
  }


R/GET     @returns                application/json

Get place data for the place with ID ‘place_id‘.

@param place_id     Integer, ‘id‘ in table ‘place‘
@returns            application/json

This returns the data from table ‘place‘ as a single JSON object.

Example return value (2020-01-09) of ‘GET /place/12‘:
```

```
    {
        "comment": "Gesch\u00e4tzt nach Iraq and the Persian Gulf [...]",
        "confidence": null,
        "geoloc": "(33.542,44.3726)",
        "geonames": "...",
        "google": "...",
        "id": 12,
        "name": "al-Baradan",
        "place_type_id": 3,
        "syriaca": null,
        "visible": true
    }
```

```
U/PATCH   @payload               application/json
          @returns               application/json

Update one or more of the fields 'comment', 'confidence', 'visible',
'place_type_id', 'geoloc', or 'name'.

Exemplary payload for `PATCH /place/12345`:

  {
    "visible": false,
    "comment": "updated comment..."
  }
```

```
D/DELETE  @returns               application/json

Delete a place if there are no conflicts. Otherwise, fail. Returns the ID
of the deleted tuple.
```

/rest/place/<int:place_id>/alternative-name/<int:name_id> DELETE GET HEAD OPTIONS PATCH PUT

```
CRUD endpoint to manipulate alternative names.

[all]     @param place_id         ID of place tuple
[all]     @param name_id          ID of name_var tuple, 0 or `None` for PUT

C/PUT     @payload                application/json
          @returns                application/json

Create new alternative name. `name` and `language_id` are required

Exemplary payload for `PUT /place/1234/alternative-name/0`:

  {
    "name": "Testplace",
    "language_id": 2
  }
```

```
R/GET     @returns                application/json
```

Get alternative name tuple.

Example return value of ‘GET /place/1234/alternative-name/5678‘:

```
    {
      "id": 5678,
      "place_id": 1234,
      "name": "Dummy test name",
      "language_id": 12
    }
```

```
U/PATCH   @payload               application/json
          @returns               application/json
```

Update one or more of the fields ’name’ and ’language_id’.

Exemplary payload for ‘PATCH /place/alternative-name/5678‘:

```
  {
    "name": "New name",
    "language_id": 13
  }
```

```
D/DELETE  @returns               application/json
```

Delete an alternative name tuple. Returns the ID of the deleted tuple.

/rest/place/<int:place_id>/alternative-name/all

GET HEAD OPTIONS

Get a list of all alternative names for place with ID ‘place_id‘.

```
@returns                         application/json
```

Exemplary return value for ‘GET /place/1234/alternative-name/all‘:

```
  [
    {
      "id": 5678,
      "place_id": 1234,
      "name": "Dummy test name",
      "language_id": 12
    },
    {
      "id": 5679,
      "place_id": 1234,
      "name": "Dummy test name 2",
      "language_id": 4
    },
    ...
  ]
```

/rest/place/<int:place_id>/details

GET HEAD OPTIONS

Retrieve more details for the place with ID ‘place_id‘.

```
@param ‘place_id‘   ‘id‘ in table ‘place‘
@returns            application/json
```

This API endpoint is aimed at the location list of the visualization, where tooltips show more details for a place on hover. This information is not loaded from the server initially for efficiency reasons. Instead, it is queried when the tooltip is created.
As of now (2020-01-10), the call returns a JSON object containing the place ID, the ‘comment‘ field from table ‘place‘, and an array of alternative names for the place together with the respective language name. This will exclude alternative names that are not main forms.

Example return value excerpt for ‘GET /place/14/details‘:


```
  {
    "alternative_names": [
      {
        "language": "Arabic - AS",
        "name": "\u062d\u0644\u0628"
      },
      {
        "language": "Arabic - DMG",
        "name": "\u1e24alab"
      },
      {
        "language": "Syriac - SS",
        "name": "\u071a\u0720\u0712 "
      },
      ...
    ],
    "comment": "[dummy] Comments are strings or null",
    "place_id": 14,
    "external_uris": [
      "Syriaca:3055",
      ...
    ],
  }
```


/rest/place/<int:place_id>/evidence

GET HEAD OPTIONS

Retrieve religion evidence for the place with ID ‘place_id‘.

@param ‘place_id‘ ‘id‘ in table ‘place‘
@returns application/json

This replaces the ‘/Religions‘ endpoint in the old servlet implementation.

Returns a list of evidences, with comments and confidences, time spans, and sources.

Example return value excerpt for ‘GET /place/12/evidence‘:


```
  {
    "place_id": 12,
    "evidence": [
      {
        "evidence_id": 17,
        "interpretation_confidence": null,
        "place_attribution_confidence": null,
        "place_id": 12,
        "place_instance_comment": null,
```

```
        "religion_confidence": null,
        "religion_id": 5,
        "religion_instance_comment": null,
        "religion_name": "Church of the East",
        "sources": [
          {
            "source_id": 8,
            "source_instance_comment": "Neuer Eintrag",
            "source_name": "AKg = Jedin, Hubert; Martin, Jochen (Hg.) (1987): [...]",
            "source_page": "26"
          }
        ],
        "time_spans": [
          {
            "comment": null,
            "confidence": null,
            "end": 1200,
            "start": 800
          }
        ]
      },
      ...
    ]
  }
```

/rest/place/<int:place_id>/evidence-ids

GET HEAD OPTIONS

```
Get all evidence tuple IDs for the place with ID ‘place_id‘.

@returns application/json

Example return value exerpt for ‘/place/12/evidence-ids‘:

  [
    1,
    5,
    12,
    191,
    ...
  ]
```

/rest/place/<int:place_id>/external-uri-list

GET HEAD OPTIONS

```
Get a list of external URIs for a place.

@param   place_id   ID of place
@returns            application/json

Example return value excerpt for ‘GET /rest/place/12/external-uri-list‘:

    [
      {
        "id": 1,
        "place_id": 12,
        "uri_namespace_id": 1,
        "uri_fragment": "27223",
        "comment": null
      }
      ...
    ]
```

/rest/place/<int:place_id>/external-uri-list

GET HEAD OPTIONS

Get a list of external URIs for a place.

```
@param   place_id   ID of place
@returns            application/json
```

Example return value excerpt for ‘GET /rest/place/12/external-uri-list‘:

```
    [
      {
        "id": 1,
        "place_id": 12,
        "uri_namespace_id": 1,
        "uri_fragment": "27223",
        "comment": null
      }
      ...
    ]
```

/rest/place/all

GET HEAD OPTIONS

Get content of table ‘place‘ as a list of dicts.

```
@return     application/json
```

Example return value excerpt:

```
    [
      {
        "comment": "Unknown place",
        "confidence": null,
        "geoloc": null,
        "id": 448,
        "name": "Dirigh",
        "place_type_id": 4,
        "visible": true
      },
      ...
    ]
```

/rest/places

GET HEAD OPTIONS

Get a list of places and their IDs.

```
@returns            application/json
```

This returns a list of place names and IDs.

Example return value excerpt:

```
    [
      {
        "id": 1,
        "name": "Mosul",
      },
      ...
    ]
```

/rest/religion-instance/<int:religion_instance_id> DELETE GET HEAD OPTIONS PATCH PUT

CRUD endpoint to manipulate religion instances.

```
[all]     @param religion_instance_id      ID of religion instance, 0 or ‘None‘ for PUT

C/PUT     @payload                 application/json
          @returns                 application/json
```

Create a new religion instance. ‘religion_id‘ is a required field.
‘confidence‘, ‘comment‘, and ‘annotation_id‘ are optional. Returns the ID
for the created religion instance.

Exemplary payload for ‘PUT /religion-instance/0‘:

```
  {
    "religion_id": 12,
    "confidence": "certain",
    "comment": "foo bar"
  }
```

```
R/GET     @returns                 application/json
```

Get one religion instance.

Exemplary reply for ‘GET /religion-instance/64‘:

```
  {
      "id": 64,
      "confidence": "certain",
      "comment": "baz",
      "annotation_id": null
  }
```

```
U/PATCH   @payload                 application/json
          @returns                 application/json
```

Update one or more of the fields ‘religion_id‘, ‘comment‘, ‘confidence‘, or
‘annotation_id‘.

Exemplary payload for ‘PATCH /religion-instance/12345‘:

```
  {
    "comment": "updated comment...",
  }
```

```
D/DELETE  @returns                 application/json
```

Delete religion instance. Write ‘user_action‘ log, return a JSON with all
deleted IDs.

/rest/religion-list GET HEAD OPTIONS

Get a list of religion names and IDs.

```
@returns            application/json
```

Example return value excerpt:

```
    [
      {
        "id": 1,
        "name": "Christianity",
        "parent_id": null
      },
      {
        "id": 5,
        "name": "Church of the East",
        "parent_id": 1
      },
      ...
    ]
```

/rest/religions

GET HEAD OPTIONS

Get a hierarchy of religions.

```
@returns            application/json
```

This utilizes the ‘parent_id‘ attribute in the table ‘religion‘ to build a list of trees. The return value is an array of JSON objects. Each root node is a main religion, and one of its attributes is ‘children‘, holding an array of children, which may again hold children, and so on.

Each node of each tree has the following, exemplary (2020-01-09) structure:

```
    {
        "abbreviation": "MARO",
        "children": [ ... ],
        "color": "hsl(10, 80%, 50%)",
        "id": 3,
        "name": "Maronite Church",
        "parent_id": 98
    }
```

/rest/source-instance/<int:source_instance_id>

DELETE GET HEAD OPTIONS PATCH PUT

CRUD endpoint to manipulate source instance entries.

```
[all]     @param source_instance_id     ID of tuple, ignored for PUT (use 0)

C/PUT     @payload                      application/json
          @returns                      application/json
```

Create a new source instance tuple. ‘source_id‘ is required in the payload, ‘source_page‘ and ‘comment‘ are optional. Returns the ‘id‘ of the created tuple on success.

Exemplary payload for ‘PUT /source-instance/0‘:

```
  {
    "comment": "new instance via PUT, 2",
    "source_id": 3,
    "source_page": "1--2",
    "source_confidence": "contested",
    "evidence_id": 12
```

```
  }
```

```
R/GET     @returns                      application/json

Get one source instance tuple.

Exemplary return value for ‘GET /source-instance/4702‘:

  {
    "comment": "new comment",
    "evidence_id": 3662,
    "id": 4702,
    "source_id": 3,
    "source_page": null,
    "source_confidence": "contested"
  }


U/PATCH   @payload               application/json
          @returns               application/json

Update one or more of the fields ‘comment‘ or ‘source_page‘. All other fields or
connected entities must be modified via their respective endpoints.

Exemplary payload for ‘PATCH /source-instance/567‘:

  {
    "comment": "updated comment...",
    "source_page": "6--7"
  }

D/DELETE  @returns               application/json

Delete all related entities, write ‘user_action‘ log, return a JSON with all deleted IDs.

Exemplary return valye for ‘DELETE /source-instance/5678‘:

  {
    "deleted": {
      "source_instance": 5678
    }
  }
```

/rest/sources-list

GET HEAD OPTIONS

```
Get a list of all sources.

@returns        application/json

This returns a JSON array of objects with ‘id‘ and ‘name‘ from table
‘source‘. This API endpoint replaces ‘/SourcesList‘ in the old servlet
implementation.

Example return value excerpt:

  [
```

```
    {
      "id": 32,
      "name": "Bar Ebroyo / Wilmshurst",
      "short": "Bar Ebroyo / Wilmshurst",
      "default_confidence": "probable"
    },
    {
      "id": 1,
      "name": "Fiey, Jean Maurice (1993): Pour un Oriens [...]",
      "short": "OCN",
      "default_confidence": "certain"
    },
    ...
  ]
```

/rest/tag-list

GET HEAD OPTIONS

Get a list of tags.

```
@returns            application/json
```

Example return value excerpt:

```
  [
    {
      "id": 1,
      "tagname": "bishopric",
      "comment": "test comment"
    },
    ...
  ]
```

/rest/tag-sets

GET HEAD OPTIONS

Get the set of evidence IDs for each tag.

```
@returns            application/json
```

Example return value excerpt:

```
  [
    {
      "tag_id": 1,
      "evidence_ids": [1,2,3,6,37]
    },
    {
      "tag_id": 2,
      "evidence_ids": [1,3]
    },
    ...
  ]
```

/rest/time-group/<int:time_group_id>

DELETE GET HEAD OPTIONS PATCH PUT

CRUD endpoint to manipulate time_group entries.

```
[all]     @param time_span_id           ID of tuple, ignored for PUT (use 0)

C/PUT     @payload                      application/json
          @returns                      application/json
```

Create a new group tuple. Optionally takes a valid ‘annotation_id‘.
Returns the ‘id‘ of the created tuple on success.

Exemplary payload for ‘PUT /time-group/0‘:

```
{
  "annotation_id": 412
}
```

R/GET @returns application/json

Get one time_group tuple.

Exemplary return value for ‘GET /time-group/3658‘:

```
{
  "annotation_id": null,
  "time_group_id": 3658,
  "time_spans": [
    {
      "comment": null,
      "confidence": null,
      "end": 988,
      "start": 985,
      "time_instance_id": 3658
    }
  ]
}
```

U/PATCH @payload application/json
@returns 204 NO CONTENT

Update the ‘annotation_id‘ field to a valid value, or ‘null‘.

Exemplary payload for ‘PATCH /time-group/789‘:

```
{
  "annotation_id": 456
}
```

D/DELETE @returns application/json

Delete all related entities, write ‘user_action‘ log, return a JSON with all deleted IDs.

Exemplary return value for ‘DELETE /time-group/5678‘:

```
{
  "deleted": {
    "annotation": 4211,
    "time_group": 5678,
    "time_instance": [
      19221,
```

```
        19222,
        32115
      ]
  }
```

/rest/time-group/<int:time_group_id>/time-instance/<int:time_instance_id> DELETE GET HEAD OPTIONS PATCH PUT

```
CRUD endpoint to manipulate time_instance entries. The ‘time_group_id‘ and
‘time_instance.time_group_id‘ must match.

[all]     @param time_group_id          ID of time_group tuple
          @param time_instance_id       ID of time_instance tuple, ignored for PUT


C/PUT     @payload                      application/json
          @returns                      application/json

Create a new timespan tuple. Optionally takes start, end, comment and
confidence. Returns the ID of the created tuple.

Exemplary payload for ‘PUT /time-group/2/time-instance/0‘:

  {
    "start": 1200,
    "end": 1300,
    "comment": "New comment",
    "confidence": "contested"
  }



R/GET     @returns                      application/json

Get one time_instance tuple.

Exemplary return value for ‘GET /time-group/3659/time-instance/3670‘:

  {
    "comment": "new comment",
    "confidence": "probable",
    "end": 1299,
    "id": 3669,
    "start": null,
    "time_group_id": 3659
  }



U/PATCH   @payload                application/json
          @returns                204 NO CONTENT

Update the start, end, confidence, or comment fields.

Exemplary payload for ‘PATCH /time-group/789/time-instance/1000‘:

  {
    "start": 1238,
```

```
    "comment": "Start year now confirmed",
    "confidence": "probable"
  }
```

D/DELETE @returns application/json

Delete tuple, write ‘user_action‘ log, return a JSON with deleted ID.

Exemplary return value for ‘DELETE /time-group/5678/time-instance/1234‘:

```
  {
    "deleted": {
      "time_instance": 1234
    }
  }
```

/rest/uri/external-database-list

GET HEAD OPTIONS

Get a list of external databases registered.

@returns application/json

This returns a list of all external_database tuples in the database as an array.

Example return value excerpt:

```
    [
      {
        "id": 1,
        "name": "Foo: the foo database",
        "short_name": "Foo",
        "url": "http://foo.bar/",
        "comment": null
      },
      ...
    ]
```

/rest/uri/external-person-uri-list

GET HEAD OPTIONS

Get a list of external person URIs.

@returns application/json

Example return value excerpt for ‘GET /rest/uri/external-person-uri-list‘:

```
    [
      {
        "id": 1,
        "person_id": 12,
        "uri_namespace_id": 1,
        "uri_fragment": "27223",
        "comment": null
      }
      ...
    ]
```

/rest/uri/external-person-uri/<int:uri_id>

DELETE GET HEAD OPTIONS PATCH PUT

CRUD endpoint to manipulate external person URIs.

```
[all]     @param uri_id           ID of tuple, 0 or ‘None‘ for PUT

C/PUT     @payload                application/json
          @returns                application/json
```

Create a new person URI tuple.

Required fields: ‘person_id‘, ‘uri_namespace_id‘, ‘uri_fragment‘.
Optional fields: ‘comment‘.

Returns the ID for the created entity.

Exemplary payload for ‘PUT /uri/external-person-uri/0‘:

```
  {
    "person_id": 12,
    "uri_namespace_id": 1,
    "uri_fragment": "1234",
    "comment": "comment"
  }
```

```
R/GET     @returns                application/json
```

Get data for the person URI.

```
@returns            application/json
```

Example return value of ‘GET /uri/external-person-uri/1‘:

```
  {
    "id": 1,
    "person_id": 12,
    "uri_namespace_id": 1,
    "uri_fragment": "1234",
    "comment": "comment"
  }
```

```
U/PATCH   @payload                application/json
          @returns                205 RESET CONTENT
```

Update one or more of the fields ‘person_id‘, ‘uri_namespace_id‘,
‘uri_fragment‘, or ‘comment‘.

Exemplary payload for ‘PATCH /uri/external-person-uri/12345‘:

```
  {
    "uri_fragment": "1236",
    "comment": "updated comment..."
  }
```

```
D/DELETE  @returns                application/json
```

Delete the tuple and return the ID of the deleted tuple.

/rest/uri/external-place-uri/<int:uri_id>

DELETE GET HEAD OPTIONS PATCH PUT

CRUD endpoint to manipulate external place URIs.

```
[all]     @param uri_id           ID of tuple, 0 or ‘None‘ for PUT

C/PUT     @payload                application/json
          @returns                application/json
```

Create a new place URI tuple.

Required fields: ‘place_id‘, ‘uri_namespace_id‘, ‘uri_fragment‘.
Optional fields: ‘comment‘.

Returns the ID for the created entity.

Exemplary payload for ‘PUT /uri/external-place-uri/0‘:

```
  {
    "place_id": 12,
    "uri_namespace_id": 1,
    "uri_fragment": "1234",
    "comment": "comment"
  }
```

```
R/GET     @returns                application/json
```

Get data for the place URI.

```
@returns            application/json
```

Example return value of ‘GET /uri/external-place-uri/1‘:

```
  {
    "id": 1,
    "place_id": 12,
    "uri_namespace_id": 1,
    "uri_fragment": "1234",
    "comment": "comment"
  }
```

```
U/PATCH   @payload                application/json
          @returns                205 RESET CONTENT
```

Update one or more of the fields ‘place_id‘, ‘uri_namespace_id‘,
‘uri_fragment‘, or ‘comment‘.

Exemplary payload for ‘PATCH /uri/external-place-uri/12345‘:

```
  {
    "uri_fragment": "1236",
    "comment": "updated comment..."
  }
```

D/DELETE @returns application/json

Delete the tuple and return the ID of the deleted tuple.

/rest/uri/uri-namespace-list

GET HEAD OPTIONS

Get a list of URI namespaces registered.

@returns application/json

This returns a list of all uri_namespace tuples in the database as an array.

Example return value excerpt:

```
[
  {
    "id": 1,
    "external_database_id", 1,
    "uri_pattern": "https://first.db/place/%s",
    "short_name": "first:%s",
    "comment": null
  }
  ...
]
```