Text Repository

Sep 20, 2023

Contents

Backend repository to store and index corpora with metadata and versions.

CHAPTER 1

Features

- **Store texts of a corpus in a uniform domain model:**
  - Keep track of file versions
  - Link all file types to the same source document
  - Store metadata about documents, files and versions
- Use Rest API to create, read, update and delete texts
- Search files using stock and custom elasticsearch indices
- Explore API with concordion and swagger

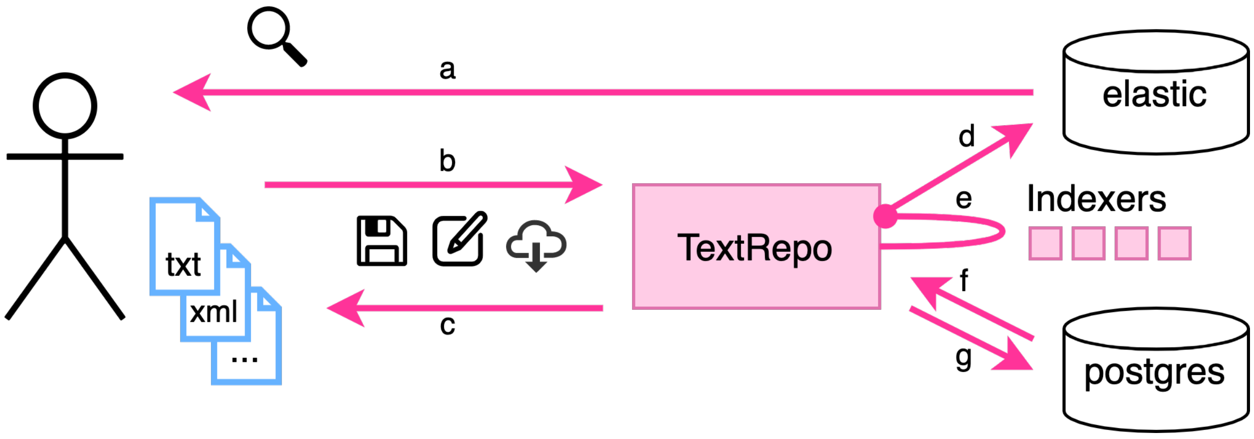

Fig. 1: Schematic overview of the Text Repository: **a)** search in corpus; **b+c)** manage file types, versions and metadata of corpus; **d+e)** build custom indices that are automatically kept in sync with corpus; **f+g)** store all data in a unified database model.

CHAPTER 2

Installation

Prerequisites: docker-compose.

To run the Text Repository locally, run in a new directory:

```
git clone https://github.com/knaw-huc/textrepo .
cd examples/production
curl -o scripts/wait-for-it.sh https://raw.githubusercontent.com/vishnubob/wait-for-
↪it/master/wait-for-it.sh
chmod +x scripts/wait-for-it.sh
./start-prod.sh
```

Read more on basic usage

CHAPTER 3

Documentation

3.1 Basic Usage

3.1.1 Run locally

Production setup

To start the Text Repository without building any containers, start it as described in the production example. This docker-compose setup does not build images but downloads the images from docker hub.

Development setup

When you want to start developing or debugging, you can use the 'dev' setup, as found in `examples/development`. When starting, docker-compose will build all Text Repository images from scratch. When stopping, all containers, volumes and network are removed.

See `examples/development/README.md` for installation details.

3.1.2 Explore locally

After running the docker-compose setup, you can:

- Find basic use cases in the integration test results
- Add some test data: `./scripts/populate.sh`
- Explore REST-API of Text Repository using swaggger
- Search in full-text and autocomplete indices

3.1.3 FAQ

Elasticsearch

If you run into this Elasticsearch warning:

```
max virtual memory areas vm.max_map_count [65530] is too low, increase to at least␣
↪[262144]
```

you may have to:

```
sysctl -w vm.max_map_count=262144
```

(Source: https://github.com/docker-library/elasticsearch/issues/111)

If you run into this Elasticsearch error:

```
max file descriptors [4096] for elasticsearch process is too low, increase to at␣
↪least [65535]
```

you may have to increase the ulimits.nofile of elasticsearch in your docker-compose.yml:

```
services:
  elasticsearch:
    ulimits:
      nofile:
        soft: 65535
        hard: 262144
```

(Source: https://stackoverflow.com/a/58024178)

3.2 Domain model

The Text Repository was born out of the desire to archive and unlock text corpora and their various files and formats in a durable and consistent way.

To represent text corpora in a generic way, the Text Repository is build around the following core concepts:

- **document**: top level object which represents the core physical entity of a digitized corpus (e.g. a page) that resulted in scans, xml-files, text files and other file types. A document contains a list of files, unique by file type
- **file**: as found on your computer, including a file type but *without* its contents. A file contains a list of versions
- **version**: version of a file. A version contains the bytes of a file and a timestamp. A file can have a number of different versions
- **metadata**: documents, files and versions can contain metadata in the form of list of key-value pairs

---

3.2.1 File types

The Text Repository is built to contain 'human readable' file types that can be processed by elasticsearch, like plain text, json, and xml.

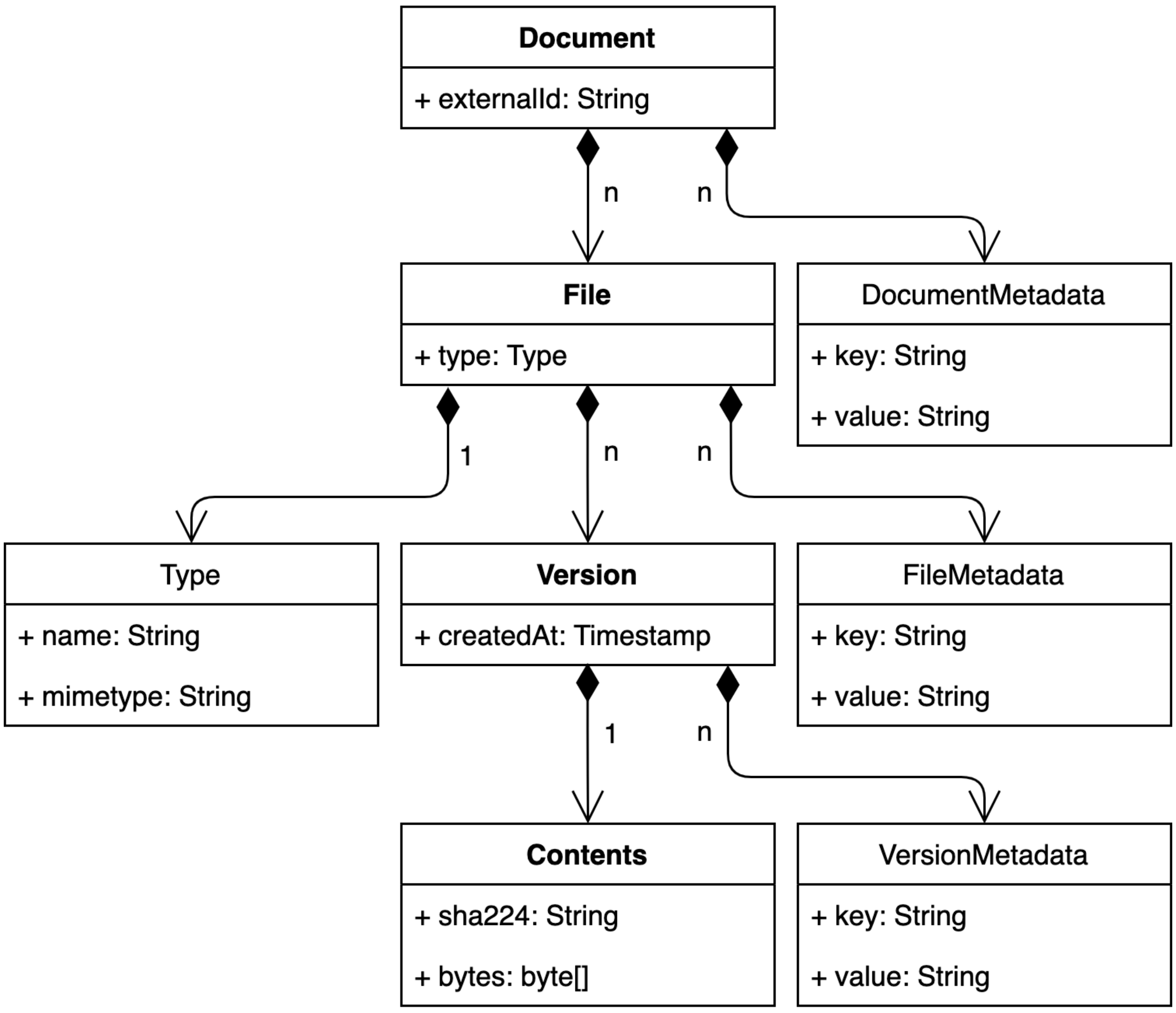

Fig. 1: Graphical representation of the Text Repository domain model

3.2.2 Work in progress

Note that this project is work in progress. The Text Repository model of a text corpus will improve and expand as the project progresses.

3.3 Components

3.3.1 Endpoints

- Text Repository API
- Text Repository healtch checks
- Integration test results
- Elasticsearch indices
- Stub indexer

3.3.2 Services

The Text Repository application is bundled with a couple of services:

- Text Repository, core application
- Postgres as database
- Elasticsearch as search engine
- Docker and docker-compose for running and connecting services
- Nginx as a reverse proxy

Text Repository

The Text Repository is a java application build with dropwizard. It contains a REST-API to create, retrieve, update and delete documents, files, versions and metadata. It uses the postgres database to store its domain model. It keeps the elastic search indices in sync with the postgres database.

Postgres

Postgres is used as a 'vault' which contains all the data or 'the golden standard', from which all indices are generated.

Postgres data can be modified using the Text Repository REST-API.

Elasticsearch

See the indexing page.

Nginx

As a reverse proxy it exposes the relevant parts of all the service endpoints. See exposed endpoints below.

3.4 Text Repository API

The Text Repository has a REST-full api to retrieve, create, update and delete resources as depicted in its domain model. For certain complex activities a `task`-endpoint has been created.

To explore the API, start up the Text Repository locally, and checkout swagger or the integration *Endpoints*.

3.4.1 Tasks

The REST API can be a bit laborious for certain activities. For example, to retrieve the latest contents of a document by its external document ID and file type, a user would have to perform a requests for each of the following steps:

- find a document ID by its external ID;
- find a file ID by its parent document ID and file type;
- find the latest version of a file by its parent file ID;
- get the contents of the latest version by its parent version ID or its hash.

To simplify such a workflow, the Text Repository offers `task`-endpoints to perform a complex task within a single request. The advantage is simplicity and ease of use. However do not expect these tasks to be 'REST-compliant'. Eg, the list of requests above can be replaced with a single request to: `/task/find/{externalId}/document/metadata`.

3.5 Indexing

The Text Repository creates and updates ElasticSearch (ES) indices as defined in `docker-compose.env`.

Elasticsearch documents are automatically created and updated when new file versions are added to the Text Repository.

More on: searching in ES.

3.5.1 Indexer

To convert new file versions into a format that ES understands, the Text Repository uses 'indexer' services. An indexer is a service with three endpoints:

- `GET mapping` returns json mapping used by the Text Repository to create an ES index.
- `POST fields` converts the contents of new file version into a json document that matches the json mapping
- `GET types` returns mimetypes that the indexer accepts, including the 'subtypes' of a mimetype, which will handled in the same way as their parent mimetype.
  - See JSON schema below.
  - For example: `application/xml` could have as a subtype `application/vnd.prima.page+xml`, meaning that the indexer handles pagexml just like ordinary xml.
  - An indexer that returns no types (status `204 No Content`) is assumed to handle all file types

More on: ES mappings.

JSON Schema of types endpoint:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "array",
  "items": [
    {
      "type": "object",
      "properties": {
        "mimetype": {"type": "string"},
        "subtypes": {"type": "array", "items": [{"type": "string"}]}
      },
      "required": ["mimetype", "subtypes"]
    }
  ]
}
```

3.5.2 Indexing moments

When are changes made to the ES indices?

- `POST /rest/files` -> Creating a new file resource will create ES docs with empty body
- `PUT /rest/files` -> Updating a file resource will update ES docs with latest version contents or an empty body when no latest version contents available
- `DELETE /rest/files` -> Deleting a file will delete the corresponding ES docs
- `POST /rest/versions` -> Updating a version resource will update ES docs with latest version contents
- `DELETE /rest/versions` -> Deleting the latest version of a file will update ES docs with the pre-latest version contents

Which tasks change indices?

- `POST /task/index` -> Multiple endpoints for reindexing a subset of files
- `POST /task/import/documents/{externalId}/{typeName}` -> index the imported file (optional, default)
- `DELETE /task/delete/documents/{externalId}` -> delete files of document (optional, default)
- `DELETE /task/index/deleted-files` -> new, delete all ES-docs of deletes files, meaning: delete all ES doc IDs not present in Text Repository database

Note: when calling any other endpoints (e.g. mutating metadata or external IDs), reindexing should done by calling one of the indexing tasks.

3.5.3 Indexing workflow

When a new file or file version is added, the Text Repository has to determine which ES document in which index needs to be created or updated. The current workflow consists of the following steps:

- Retrieve file type of new file version
- For each indexer: does it handle this file mimetype?
  - No? Skip
  - Yes? Continue
- Convert file version contents to ES document using appropriate indexer `fields` endpoint.

- Is this file version the first version?
  - Yes: create new ES document
  - No: update ES document by file ID
- Use response of `fields`-endpoint to create or update ES documents

3.5.4 Fields endpoint

The fields-endpoint of an indexer converts a file into an ES document. The Text Repository can call this endpoint in two ways: as `multipart` or `original`.

- When using **multipart**, files will be send to the fields endpoint using `Content-Type: multipart/form-data` with a body part named `file` which contains the file contents and and a `Content-Type` header with the file mimetype.
- When using **original**, files will be send to the fields endpoint with a `Content-Type` header and a body containing the file contents.

3.5.5 Indexer configuration

Indexers and their ES indices can be configured with `$TR_INDEXERS` in `docker-compose.env`. An (empty) indexer configuration consists of the following elements:

```
indexers:
- name:        # string, name of indexer
  mapping:     # string, url of GET mapping endpoint
  fields:
    url:       # string, url of POST fields endpoint
    type:      # string, 'multipart' or 'original'
  mimetypes:
    -          # list of strings, supported mimetypes
  elasticsearch:
    index:     # string, name of index
    hosts:
      -        # list of strings, host urls
```

3.5.6 Default indexers

The textrepo contains a number of default indexers:

- Full-text indexer: for basic full-text search queries
- Autocomplete indexer: for autocomplete suggestions
- File indexer: for searching the metadata of documents, files and versions

3.6 Dashboard

The Text Repository offers a, currently minimal, dashboard API which can be used to get some ‘high level’ overview statistics. We welcome ideas and use cases to improve and expand the dashboard API. You can try out the dashboard using swagger.

3.6.1 Get document counts

At `/dashboard`, Text Repository gives counts of:

- how many documents are registered;
- how many documents have at least a single file;
- how many documents have at least one item of metadata;
- how many documents have both a file and metadata;
- how many documents have neither a file, nor any metadata.

The latter, in particular, is interesting when it is non-zero. These are documents that have been registered with Text Repository with their external identifier, but are otherwise content-free. This is probably not what you want.

3.6.2 Find orphaned documents

If `/dashboard` indicates there are some documents with no content associated with them ("orphans"), you can find out which these are at `/dashboard/orphans`.

3.6.3 Get metadata counts

At `/dashboard/metadata` you can find out how many documents have / share a given metadata key. It returns documents broken down by metadata key.

3.6.4 Get metadata value counts

At `/dashboard/metadata/{key}` you can find, for a specific metadata key, documents counts grouped by metadata value for that key.

3.7 Production Setup Example

In production we want to:

- backup and restore production data
- run integration tests without touching production data

See: `./example/production/`.

3.7.1 Backing Up Data

We can backup Postgres by archiving its data directory.

Elasticsearch is a bit more complicated: we cannot simply copy the data from its data dir. It would result in corrupt indices and missing data. Instead we will be using the 'snapshot' api of ES to create and restore backups.

See also: backing up elasticsearch, an example.

Example of backing up and restoring production data:

```
cd ./examples/production

# start production:
./start-prod.sh

# show logging:
./log.sh prod

# add some docs, files and versions:
(cd ../../ && ./scripts/populate.sh)

# check added documents exist:
curl localhost:8080/textrepo/rest/documents
curl localhost:8080/index/full-text/_search

# we might want to backup our data?
mkdir ~/backup
./stop-prod.sh
./backup-prod.sh
ls ~/backup/

# disaster strikes!
source docker-compose.env
docker-compose -f docker-compose-prod.yml down -v

# nothing left!
./start-prod.sh && ./log.sh prod
curl localhost:8080/textrepo/rest/documents
curl localhost:8080/index/full-text/_search

# restore volumes:
docker-compose -f docker-compose-prod.yml down -v
./restore-prod.sh

# check docs have been restored:
./start-prod.sh && ./log.sh prod
curl localhost:8080/textrepo/rest/documents
curl localhost:8080/index/full-text/_search
```

See: `./backup-prod.sh` and `./restore-prod.sh` in `./examples/production/`.

3.7.2 Running Integration Tests

In production we want to show the integration test results from the concordiondata volume, but we do not want concordion to wipe out our production data while running its tests. This can be achieved by creating separate postgres and elasticsearch volumes for the production and integration environment. However, we only have one concordion data volume, in order to share the concordion test results of the integration setup with the production setup.

Example of running integration tests on production server (i.e. after upgrading) without interfering with production data:

```
cd ./examples/production

# for starting and populating production, see example above.

# stop production:
```

(continues on next page)

(continued from previous page)

```
./stop-prod.sh

# run integration tests with different volumes:
./start-integration.sh && ./log.sh integration

# check that no documents exist (because populate.sh was not run):
curl localhost:8080/textrepo/rest/documents

# when tests have run, stop integration test setup:
./stop-integration.sh

# start production again...
./start-prod.sh && ./log.sh prod

# check that documents created by ./scripts/populate.sh still exist:
curl localhost:8080/textrepo/rest/documents
curl localhost:8080/index/full-text/_search

# check concordion results from integration setup are available through nginx:
open http://localhost:8080
```

See: `docker-compose-integration.yml` and `docker-compose-prod.yml` in `./examples/production/`.

When rerunning an example, make sure all previous data is gone:

```
cd ./examples/production

source docker-compose.env
docker-compose -f docker-compose-prod.yml down -v
docker-compose -f docker-compose-integration.yml down -v
```

3.8 Backing Up Elasticsearch Example

We cannot simply copy the data from es data dirs, because it could result in corrupt indices and missing data. Instead we will be using the 'snapshot' api of ES to create and restore backups (i.e. snapshots).

More on: elasticsearch snapshot api.

3.8.1 Testing ES Snapshot

See also the production example, `backup-prod.sh` and `restore-prod.sh`.

```
ES_URL=$(docker port tr_elasticsearch 9200)

# register a snapshot repository
curl -XPUT $ES_URL/_snapshot/backup \
  -H 'content-type:application/json' \
  -d '{"type":"fs","settings":{"location":"/backup"}}'

# create a snapshot:
curl -XPUT "$ES_URL/_snapshot/backup/snapshot_1?wait_for_completion=true" | jq
```

(continues on next page)

(continued from previous page)

```
# when starting with fresh es install, first restore snapshot repository:
curl -XPUT $ES_URL/_snapshot/backup \
  -H 'content-type:application/json' \
  -d '{"type":"fs","settings":{"location":"/backup"}}'

# and restore snapshot:
curl -XPOST "$ES_URL/_snapshot/backup/snapshot_1/_restore" \
  -H 'content-type:application/json' \
  -d '{}'
```

3.9 Content Coordinate System

Caveat lector: HC SVNT DRACONES

3.9.1 Addressing Content

Suppose a document (e.g., a `txt` file) is in the Text Repository, how can we then refer to portions of the text inside the document?

Let's try some URIs.

Suppose that:

```
curl $prefix/documents/$id
```

will get you all of the document's `txt` contents. It would be nice if we could address a *character range*, e.g., from characters `10-15` in a manner such as:

```
curl $prefix/documents/$id/text/chars?start=10&end=15
```

or:

```
curl $prefix/documents/$id/text/chars?start=10&length=6
```

Similarly we could be interested in *lines* `2..4` at address:

```
curl $prefix/documents/$id/text/lines?start=2&end=4
```

The idea being that the `text` part in `$prefix/documents/$id/text/chars?start=10&length=6` signifies that we are interested in interpreting the document from a `txt` perspective, looking at lines, words, characters, etc. (as opposed to, e.g., an XML / TEI context where we could be interested in getting the author from the metadata, . . .)

Things get interesting once we challenge ourselves to the idea that we could be interested in viewing a document from this `text` perspective, irrespective of the actual format that was used to upload the document. So, even if a `TEI` (or `PageXML`, `hOCR`, . . .) document is uploaded, we still want to address its *textual content* via `/text/chars/...` and we consider it a Text Repository responsibility to be able to (in this case) yield the requested *character range* (lines, words, . . .) from the `TEI` document.

3.9.2 Perspectives

In `$prefix/documents/$id/text/{chars,lines,words,...}` we will call `text` the *perspective*.

TODO: *conjure up terminology for the* `{chars,lines,words}` *part*, perhaps `selector`?

`$prefix/documents/$id/<perspective>/<selector>?params`

Sidestepping to current Text Repository implementation, `perspective` could be a Resource class in the WebApp stack, translating the URI / addressing scheme. Then a separate *Perspective* class hierarchy is responsible for mapping various file formats to text (flatten the tree and yield the characters for a generic XML file, do something more intelligent for a TEI file, use "Rutger's" implementation for PageXML/hOCR, etc.). Then a *Selector* class hierarchy works in the `txt` domain and selects the requested fragment.

TODO: diagrams :-)

3.10 FAQ

3.10.1 Elasticsearch

If you run into this Elasticsearch warning:

```
max virtual memory areas vm.max_map_count [65530] is too low, increase to at least␣
↪[262144]
```

you may have to:

```
sysctl -w vm.max_map_count=262144
```

(Source: https://github.com/docker-library/elasticsearch/issues/111)

If you run into this Elasticsearch error:

```
max file descriptors [4096] for elasticsearch process is too low, increase to at␣
↪least [65535]
```

you may have to increase the ulimits.nofile of elasticsearch in your docker-compose.yml:

```
services:
  elasticsearch:
    ulimits:
      nofile:
        soft: 65535
        hard: 262144
```

(Source: https://stackoverflow.com/a/58024178)

CHAPTER 4

Support

If you are having issues, please let us know at:

https://github.com/knaw-huc/textrepo/issues

CHAPTER 5

Contribute

Want to improve the Text Repository? Submit a pull request at:

https://github.com/knaw-huc/textrepo

CHAPTER 6

License