MultiLA Web API

Markus Konrad

Jul 26, 2023

Contents

Markus Konrad <markus.konrad@htw-berlin.de>, April 2023

1 Technical requirements

- Docker with Docker Compose v2 (recommended: run Docker in *rootless* mode)
  - all you need is to install the Docker Engine for your operating system (Docker Desktop is optional)
  - it is recommended to set up Docker in rootless mode if your operating system supports it
- Python 3.11 if not running the web application in a Docker container (see *Option 1: Using a venv on the local machine*)

2 Software and frameworks used in this project

- Python 3.11
- Django 4.2 as web framework with Django REST framework extension package
- PostgreSQL database

3 Relevant documentation parts in used frameworks

Django:

- Models and databases (tutorial / topic guide)
- Views (tutorial / topic guide)
- Automated admin interface (tutorial / documentation)
- Testing (topic guide)

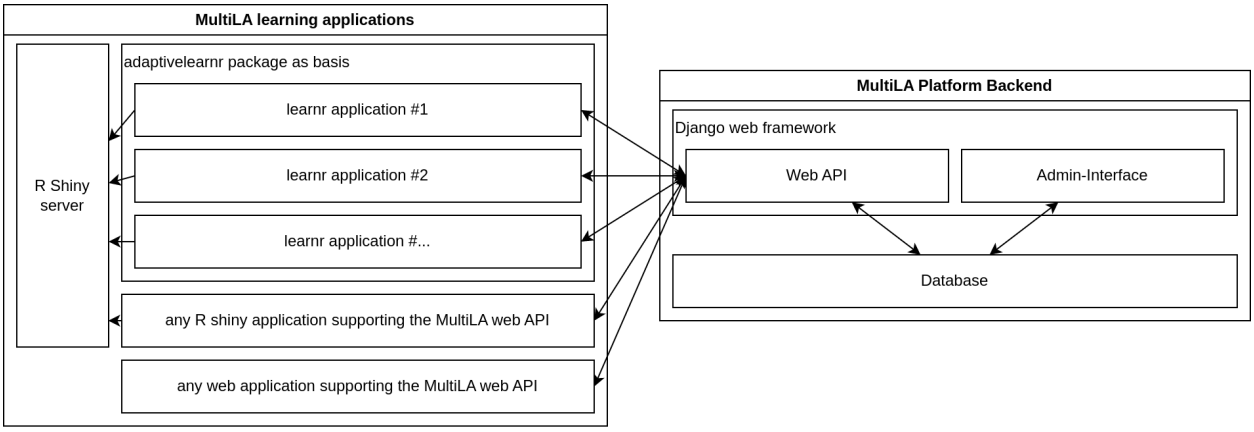
Django REST framework:

- Serialization
- Requests and Responses
- Testing

4 Software components

4.1 Overview

The following image show an overview of the MultiLA platform components:



- the web API is central and provides a common platform for setting up client applications, configuring and sharing them, and tracking user data and feedback
- all data – user generated or operational – is stored in the database
  - only the web API service has direct access to the database – client applications cannot access the database directly
- for *learnr* based client applications, there is a package *adaptivelearnr* that provides all necessary (JavaScript) code to interact with the web API and to make client applications *configurable*
  - this allows to quickly create several client applications that share the same code for interfacing with the web API and that can be configured in some details (e.g. including/excluding certain sections, aesthetic changes, etc.)
- the R Shiny server doesn't communicate with the MultiLA web API, only the JavaScript code on the client side implements the communication
- in general, any (web) application can use the MultiLA web API, which means for example R Shiny applications or Jupyter Notebook applications
- it may be possible to connect external services for authentication (e.g. Moodle)

4.2 Code repositories overview

- Web API and database: https://github.com/IFAFMultiLA/webapi
- *adaptivelearnr* R package: https://github.com/IFAFMultiLA/adaptivelearnr
  - uses adapted (forked) mus.js code for (mouse) tracking: https://github.com/IFAFMultiLA/musjs (all adaptions currently in branch *record-current-elem*)
- R learning applications (using *adaptivelearnr*):
  - TestgenauigkeitBayes: https://github.com/IFAFMultiLA/TestgenauigkeitBayes

5 Client-server communication

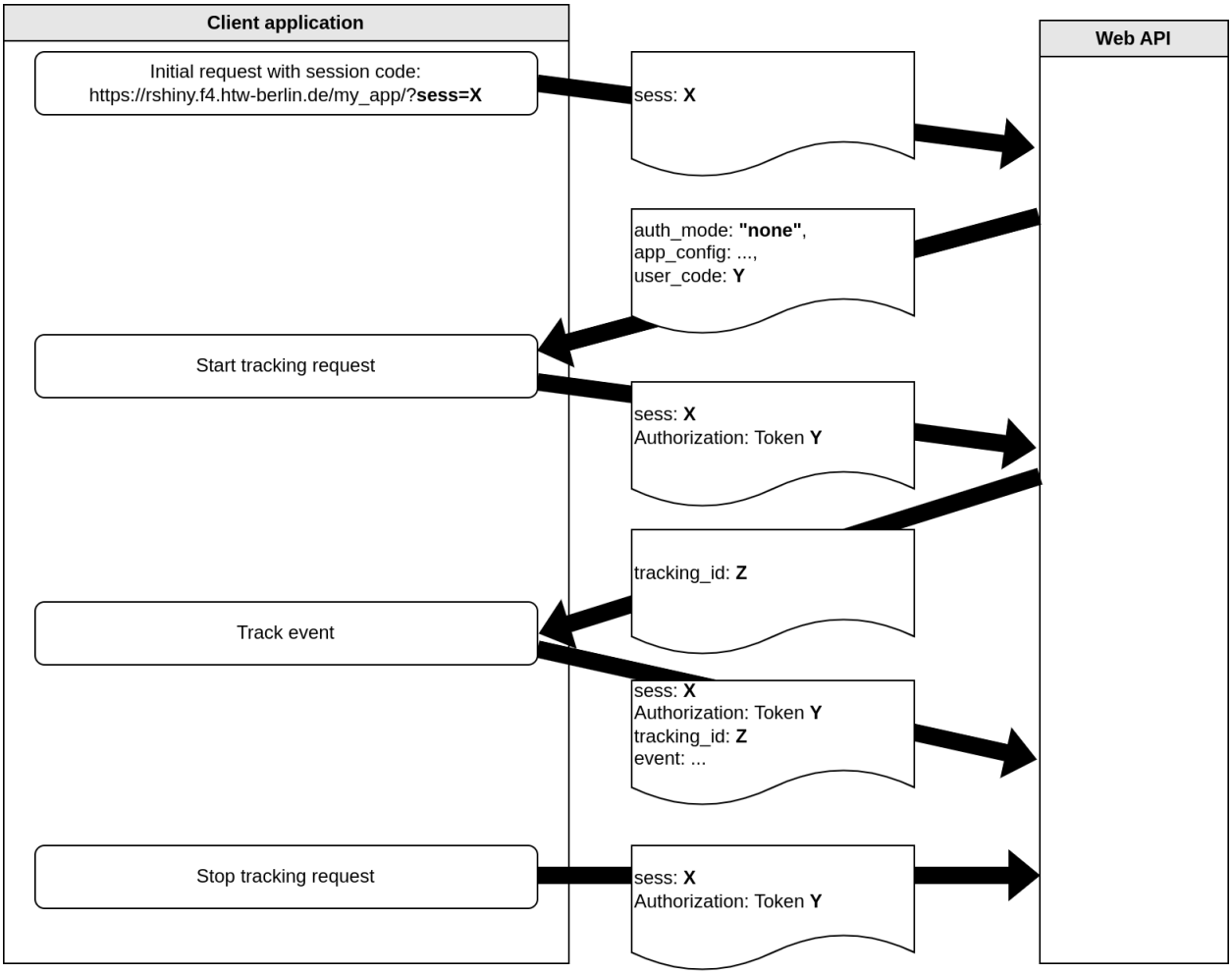
- client-server communication happens on the basis of a RESTful web API implemented in this repository
- the implementation is done in `api/views.py`
- the API exposes an OpenAPI schema under the URL `http[s]://<HOST>/openapi` when `settings.DEBUG` is `True`

5.1 Client-server communication flowchart

- an application session may either require a login or not – this can be configured in the administration backend for each application session as "authentication mode"
- all API endpoints except for `session/` and `session_login/` require an HTTP authorization token, a.k.a "user token", even when no login is required
- this makes sure that each request to the API is linked to a user – either to a registered user (when a login is required) or to an anonymous user that is only identified with a unique code (when no login is required)

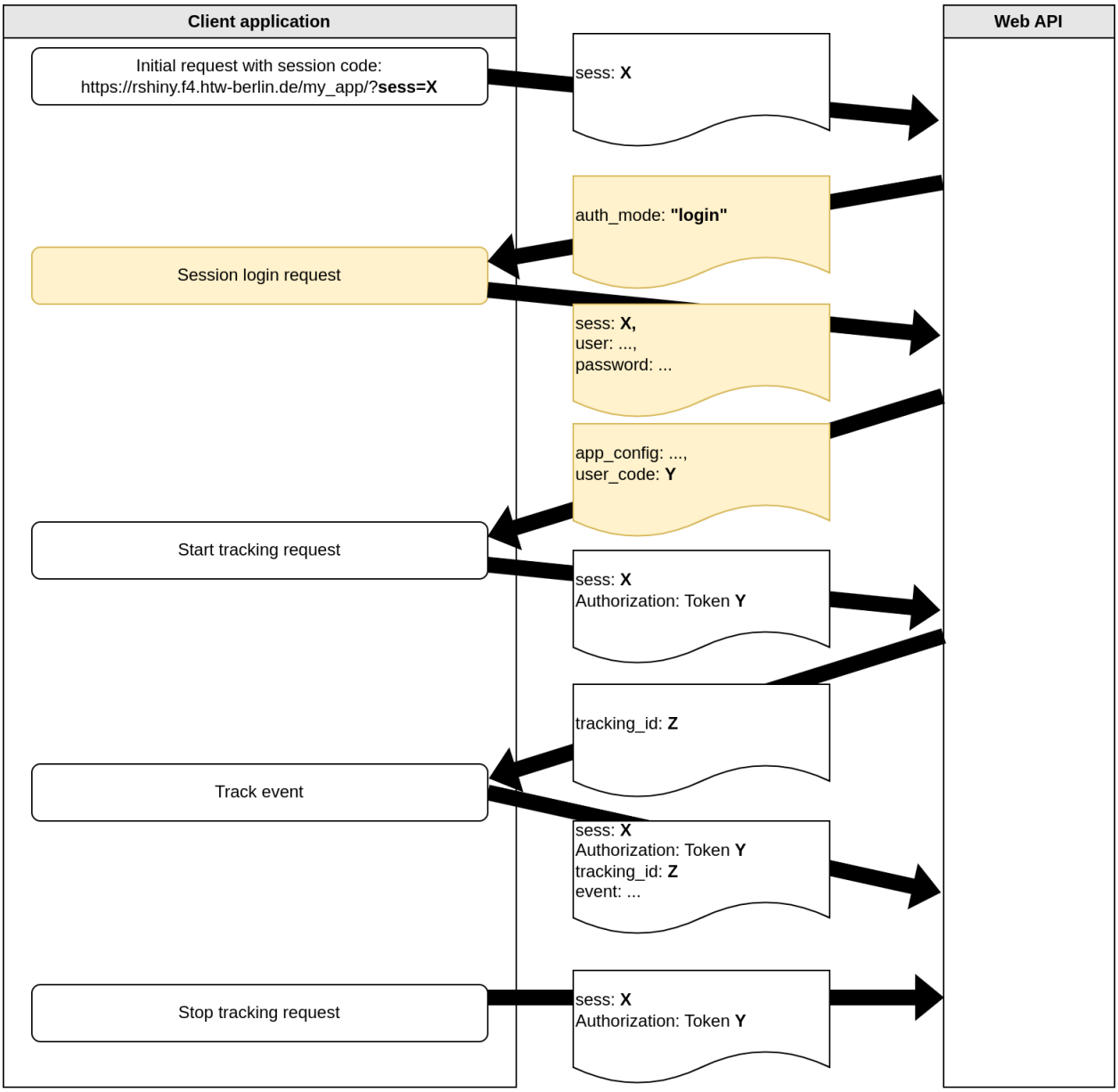
Without login ("anonymous")

- doesn't require an account
- user authentication is based on a user token that is generated on first visit and then stored to cookies for re-use



With login

- requires that the user has registered an account with email and password



5.2 Application configuration

- to each application, a configuration can be passed when starting the application session
- the API sends the configuration as JSON object after the initial session request (anonymous session) or after log in (session that requires login) – this is displayed as `app_config` key in the above figures
- on the client side, the *adaptivelearnr* R package handles reading the configuration and setting up the application accordingly
- the application configuration JSON object has the following format and options:

```
{
  "exclude": [<HTML element IDs to exclude>],
  "js": [<additional JavaScript files to load>],
```

(continues on next page)

(continued from previous page)

```
  "css": [<additional CSS files to load>],
  "feedback": {       # enable/disable specific user feedback features
    "quantitative": <bool>,  # point based feedback (5-star assessment)
    "qualitative": <bool>    # free form text based feedback
  },
  "tracking": {       # enable/disable specific tracking features
    "mouse": <bool>,    # mouse tracking w/ mus.js
    "inputs": <bool>,   # tracking of input changes
    "chapters": <bool>  # tracking of switching betw. chapters
  }
}
```

6 Local development setup

There are two ways to set up a local development environment: either by using a Python virtual environment *(venv)* on the local machine to run the Python interpreter or by using a Python interpreter inside a Docker container. The latter is currently harder to set up in conjunction with an IDE.

6.1 Option 1: Using a venv on the local machine

- create a Python 3.11 virtual environment and activate it (e.g. via `python3 -m venv venv` in the project root directory and then activating it via `source venv/bin/activate`)
- install the required packages via pip: `pip install -r requirements.txt`
- create a project in your IDE, set up the Python interpreter as the one you just created in the virtual environment
- copy `docker/compose_dev_db_only.yml` to `docker/compose_dev.yml`
- start the docker services for the first time via `make up` or via your IDE's docker interface
  - **note:** the first start of the "web" service may fail, since the database is initialized in parallel and may not be ready yet when "web" is started – simply starting the services as second time should solve the problem
- optional: create a launch configuration for Django in your IDE
- start the web application using the launch configuration in your IDE or use `python src/manage.py runserver`

6.2 Option 2: Using a Python interpreter inside a docker container

- copy `docker/compose_dev_full.yml` to `docker/compose_dev.yml`
- create a project in your IDE, set up a connection to Docker and set up to use the Python interpreter inside the `multila-web` service
  - for set up with PyCharm Professional, see here
- start all services for a first time
  - **note:** the first start of the "web" service may fail, since the database is initialized in parallel and may not be ready yet when "web" is started – simply starting the services as second time should solve the problem
- alternatively, to manually control the docker services outside your IDE, use the commands specified in the Makefile:

  - `make create` to create the containers
  - `make up` to launch all services

6.3 Common set up steps for both options

- when all services were started successfully, run `make migrate` to run the initial database migrations
- run `make superuser` to create a backend admin user
- the web application is then available under `http://localhost:8000`
- a simple database administration web interface is then available under `http://localhost:8080/admin`

6.4 Generating the documentation

- all documentation is written reStructuredText using the Python documentation system Sphinx
- the documentation source files are located under `docs/source`
- different output formats can be produced using the Makefile in `docs`, e.g. via `make html`
- the generated documentation is then available under `docs/build/<output_format>`
- a shortcut is available in the Makefile in the project root directory – you can run `make docs` from here
- note that generating a PDF of the documentation requires that the packages *texlive*, *texlive-latex-extra* and *latexmk* are installed

7 Codebook for MultiLA web API data export

This data export archive contains three files in CSV format, all of which can be joined via common identifiers that are documented below and highlighted in **bold.**

7.1 File `app_sessions.csv`

Contains data on application sessions, i.e. information on applications and the configured sessions that can be visited by users.

- `app_id`: ID of the application – integer
- `app_name`: name of the application – character string
- `app_url`: URL where the application was served – character string
- `app_config_id`: ID of the application configuration – integer
- `app_config_label`: name of the application configuration – character string
- **``app_sess_code``: session code of the application session (session code for a configured application that was shared to the users) – character string**
- `app_sess_auth_mode`: authentication mode of the application session – categorical; `"none"` or `"login"`

7.2 File `tracking_sessions.csv`

Contains data on user and tracking sessions, i.e. information on users and their interaction sessions with the applications starting with the first visit of an application session and ending with closing the browser window.

- **``app_sess_code``: session code of the application session (session code for a configured application that was shared to the users) – character string**
- `user_app_sess_code`: user application session code (session code for an individual anonymous or registered user interacting with a specific application session) – character string
- `user_app_sess_user_id`: user ID for registered users; no further data on individual users is provided in this dataset – integer for registered users or NA for anonymous users
- **``track_sess_id``: tracking session ID (ID indicating for a continuous interaction of a user with the application session on a single device) – integer**
- `track_sess_start`: start of the tracking session (first visit of a user on this device for this application session) – UTC date and time in format `Y-M-D H:M:S`
- `track_sess_end`: end of the tracking session (user closes the browser window of logs out) – UTC date and time in format `Y-M-D H:M:S`
- `track_sess_device_info`: information on the device used by the user in this tracking session – JSON with the following information:
  - `user_agent`: "user agent" string from the browser – character string
  - `form_factor`: categorical; `"desktop"`, `"tablet"` or `"phone"`
  - `window_size`: array with two elements as integers: `[window width, window height]`

7.3 File `tracking_events.csv`

Contains data on events produced by users within a tracking session.

- **``track_sess_id``: tracking session ID (ID indicating for a continuous interaction of a user with the application session on a single device) – integer**
- `event_time`: time when the event took place – UTC date and time in format `Y-M-D H:M:S`
- `event_type`: type of the event – categorical; `"device_info_update"`, `"input_change"`, `"learnr_event_*"` (see below for possible *learnr* events in * placeholder) or `"mouse"`
- `event_value`: event data – JSON; depends on `event_type`:
  - for `"device_info_update"`: changed window size as `{"window_size": [width, height]}`
  - for `"learnr_event_*"`:
  - for `"input_change"`: an object with the following keys and values –
    - `id`: HTML ID of the tracked input element – may be empty
    - `xpath`: XPath to the tracked input element
    - `value`: new value of the input element
  - for `"mouse"`: raw mouse tracking data as collected with mus.js; data is collected in chunks and must be concatenated to form the trace for the whole tracking session
    - `frames`: array with mouse interactions; each item is an array `[type, x, y, xpath, timestamp]`

        - `type` can be: `"m"` – move; `"c"` – click; `"s"` – scroll; `"i"` – key input; `"o"` – input value change (sliders, checkboxes, etc.)
        - `x` and `y` are cursor positions within the window
        - `xpath` is the XPath for the current element or `null` if the element is the same as in the previous record
        - `timestamp` is the time in ms
    * `window`: window size
    * `timeElapsed`: time in ms since mouse tracking started

Learnr events

- `exercise_hint`: User requested a hint or solution for an exercise.
- `exercise_submitted`: User submitted an answer for an exercise.
- `exercise_result`: The evaluation of an exercise has completed.
- `question_submission`: User submitted an answer for a multiple-choice question.
- `video_progress`: User watched a segment of a video.
- `section_skipped`: A section of the tutorial was skipped.
- `section_viewed`: A section of the tutorial became visible.

8 Server deployment

8.1 Prerequisites

- Docker with Docker Compose v2 (recommended: run Docker in *rootless* mode)
- an HTTP server such as Apache or nginx used as proxy
- a valid SSL certificate – **only run this service via HTTPS in production!**

8.2 Initial deployment

1. Create a Docker Compose configuration like the following as `docker/compose_prod.yml`:

```
version: '2'

services:
  # # optional: DB admin web interface accessible on local port 8081
  # adminer:
  #  image: adminer
  #  ports:
  #    - 127.0.0.1:8081:8080
  #  restart: always

  db:
    image: postgres
```

(continues on next page)

(continued from previous page)

```
    volumes:
      - '../data/db:/var/lib/postgresql/data'
      - '../data/backups:/data_backup'
    environment:
      - 'POSTGRES_USER=admin'
      - 'POSTGRES_PASSWORD=<CHANGE_THIS>'
      - 'POSTGRES_DB=multila'
    restart: always

  web:
    build:
      context: ..
      dockerfile: ./docker/Dockerfile_prod
    command: python -m uvicorn --host 0.0.0.0 --port 8000 multila.asgi:application
    volumes:
      - '../src:/code'
      - '../data/export:/data_export'
    ports:
      - "8000:8000"
    environment:
      - 'POSTGRES_USER=admin'
      - 'POSTGRES_PASSWORD=<CHANGE_THIS>'
      - 'POSTGRES_DB=multila'
      - 'DJANGO_SETTINGS_MODULE=multila.settings_prod'
      - 'SECRET_KEY=<CHANGE_THIS>'
    depends_on:
      - db
    restart: always
```

2. Make sure the correct server and directory is entered in `Makefile` under `SERVER` and `APPDIR`. Then run:
    - `make collectstatic` to copy all static files to the `static_files` directory
    - `make sync` to upload all files to the server
3. On the server, do the following:
    - run `make copy_static` to copy the static files to the directory `/var/www/api_static_files/` (you must have the permissions to do so)
    - run `make build` to build the web application
    - run `make create` to create the docker containers
    - run `make up` to launch the containers
    - run `make migrate` to initialize the DB
    - run `make superuser` to create a backend admin user – **use a secure password**
    - run `make check` to check the deployment
    - run `make test` to run the tests in the deployment environment
    - you may run `make logs` and/or `curl http://0.0.0.0:8000/` to check if the web server is running
4. On the server, create an HTTP proxy to forward HTTP requests to the server to the docker container running the web application. For example, a configuration for the Apache webserver that forwards all requests to `https://<HOST>/api/` would use the following:

```
# setup static files (and prevent them to be passed through the proxy)
ProxyPass /api_static_files !
Alias /api_static_files /var/www/api_static_files

# setup proxy for API
ProxyPass /api/ http://0.0.0.0:8000/
ProxyPassReverse /api/ http://0.0.0.0:8000/
```

All requests to `https://<SERVER>/api/` should then be forwarded to the web application.

8.3 Publishing updates

- locally, run `make testsync` and `make sync` to publish updated files to the server
- on the server, optional run `make migrate` to update the database and run `make restart_web` to restart the web application (there is a shortcut `make server_restart_web` that you can run *locally* in order to restart the web application on the server)
- if there are changes in the static files, you should run `make collectstatic` before `make sync` and then run `make copy_static` on the server
- if there are changes in the dependencies, you need to rebuild the container as decribed above under *Initial deployment*, point (3)

8.4 Optional DB administration interface

If you have enabled the `adminer` service in the docker compose file above, a small DB administration web interface is running on port 8081 on the server. For security reasons, it is only accessible from localhost, i.e. you need to set up an SSH tunnel to make it available remotely from your machine. You can do so on your machine by running:

```
ssh -N -L 8081:localhost:8081 <USER>@<SERVER>
```

, where `<USER>@<SERVER>` are the login name and the host name of the server, where docker containers are running. A shortcut is available in the Makefile as `adminer_tunnel`. You can then go to `http://localhost:8081/` in your browser and login to the Postgres server (not MySQL!) using the `POSTGRES_USER` and `POSTGRES_PASSWORD` listed in the environment variabless of the docker compose file.

8.5 DB backups

You can use `make dbbackup` on the server to generate a PostgreSQL database dump with the current timestamp under `data/backups/`. It's advisable to run this command regularly, e.g. via a cronjob, and then copy the database dumps to a backup destination e.g. via `make download_dbbackup`.

9 Indices and tables

- genindex
- modindex
- search