MultiLA Web API

Markus Konrad

Apr 04, 2023

CONTENTS

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

CHAPTER

ONE

REQUIREMENTS

- Docker with Docker Compose v2 (recommended: run Docker in *rootless* mode)
- recommended: IDE with Docker Compose support (e.g. PyCharm Professional, VSCode)
- Python 3.11 if not running the web application in a docker container (see *Option 1: Using a venv on the local machine*)

CHAPTER

TWO

SOFTWARE AND FRAMEWORKS USED IN THIS PROJECT

- Python 3.11
- Django 4.1 as web framework with djangorestframework extension package
- PostgreSQL database

CHAPTER

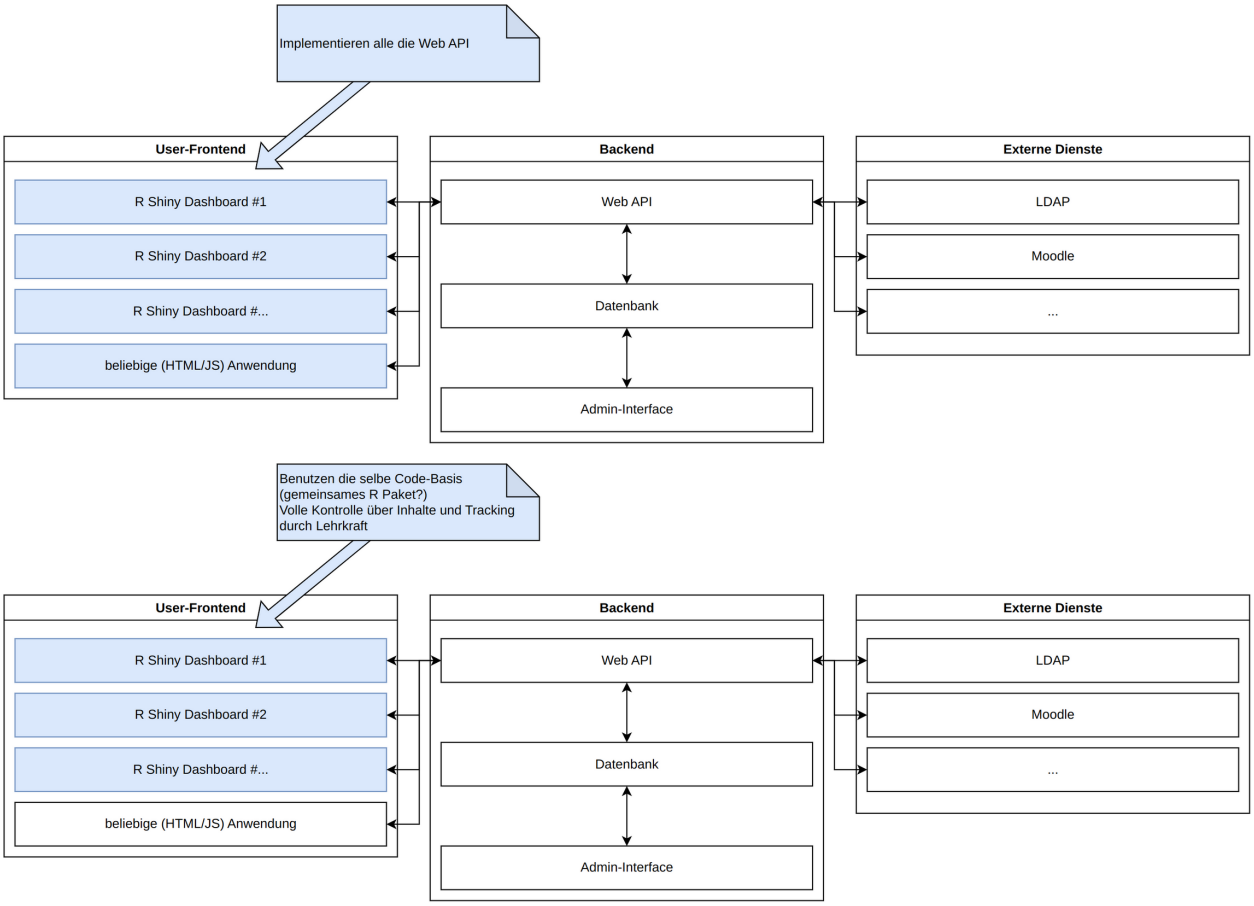
THREE

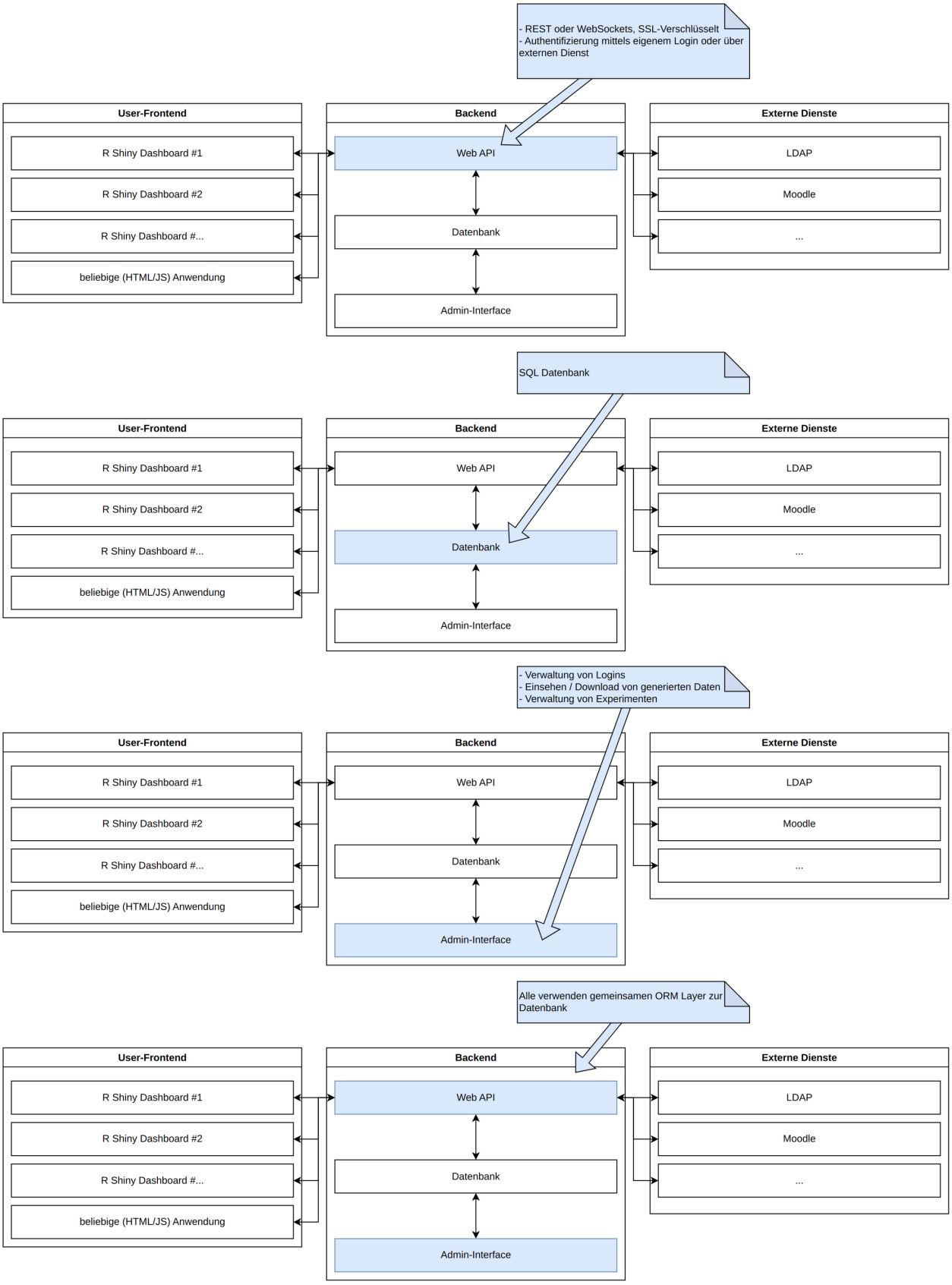
TABLE OF CONTENTS

3.1 Software components

3.1.1 Overview

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




- REST oder WebSockets, SSL-Verschlüsselt
- Authentifizierung mittels eigenem Login oder über externen Dienst
User-Frontend
R Shiny Dashboard #1
R Shiny Dashboard #2
R Shiny Dashboard #...
beliebige (HTML/JS) Anwendung
Backend
Web API
Datenbank
Admin-Interface
Externe Dienste
LDAP
Moodle
...
SQL Datenbank
- Verwaltung von Logins
- Einsehen / Download von generierten Daten
- Verwaltung von Experimenten
Alle verwenden gemeinsamen ORM Layer zur Datenbank

3.1.2 Further information on 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.)
- external services are optional so far

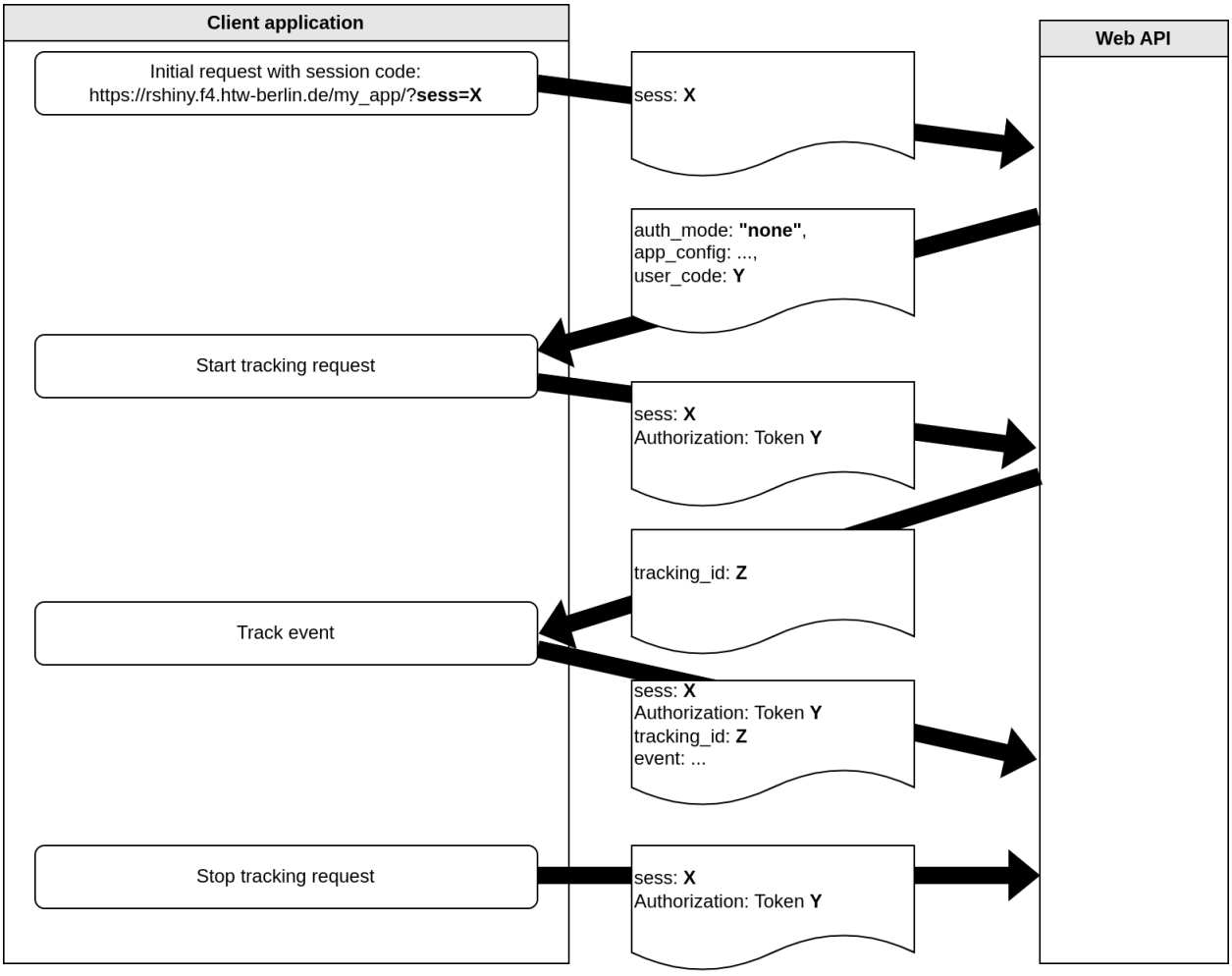
3.2 Client-server communication

- client-server communication happens on the basis of a RESTful web API implemented in this repository
- the API is self-documented using an OpenAPI schema (TODO)

3.2.1 Client-server communication flowchart

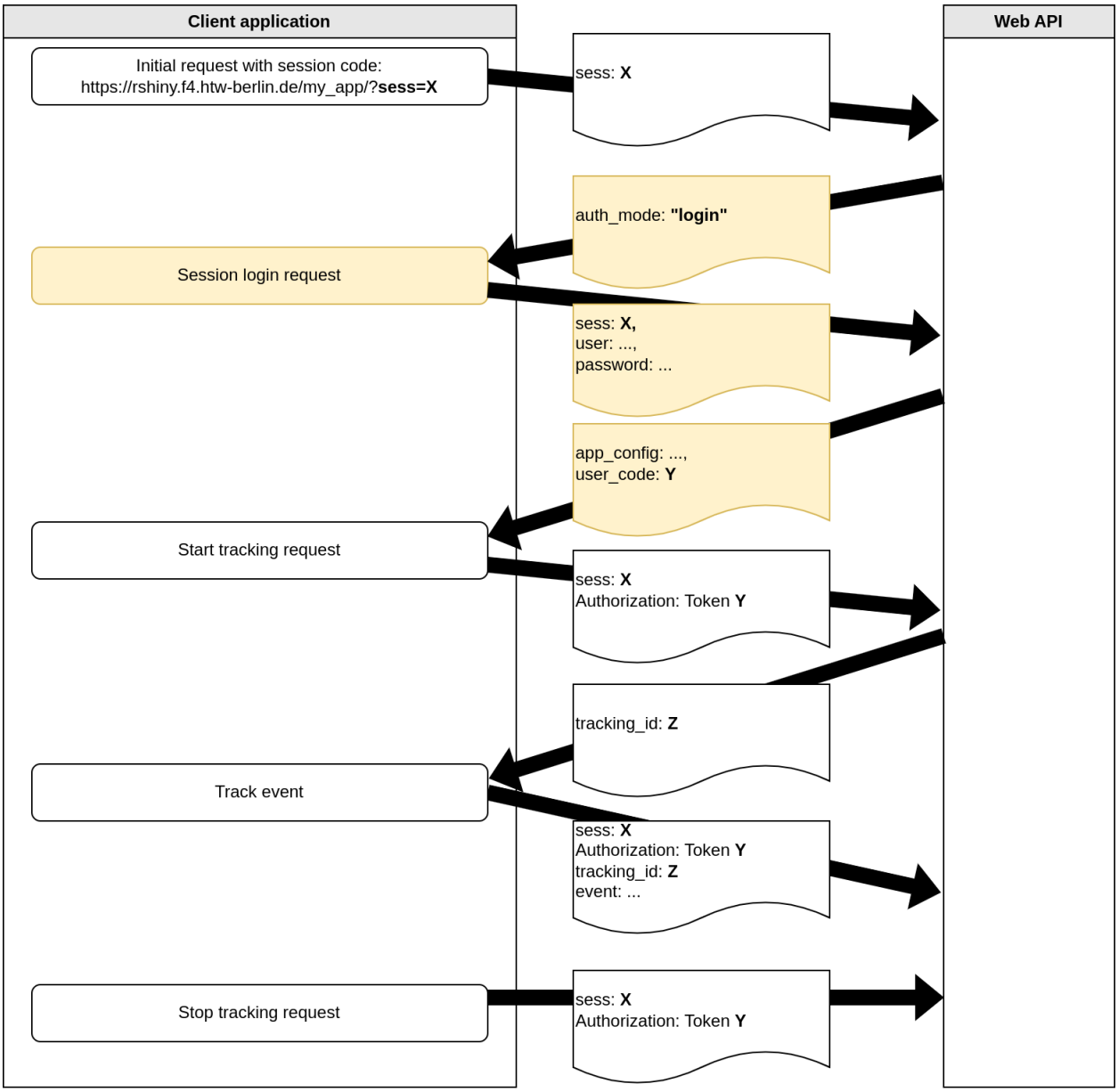
Without login (“anonymous”)

- 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



3.3 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 to by using a Python interpreter inside a docker container.

3.3.1 Option 1: Using a venv on the local machine

- create a Python 3.11 virtual environment and activate it
- 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 or
- start the web application using the launch configuration in your IDE or use `python src/manage.py runserver`

3.3.2 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

3.3.3 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`

3.3.4 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

3.4 Server deployment

3.4.1 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!**

3.4.2 Initial deployment

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

```
version: '2'

services:
  db:
    image: postgres
    volumes:
      - '../data/db:/var/lib/postgresql/data'
    environment:
      - 'POSTGRES_USER=admin'
      - 'POSTGRES_PASSWORD=<CHANGE_THIS>'
      - 'POSTGRES_DB=multila'
  web:
    build:
      context: ..
      dockerfile: ./docker/Dockerfile_prod
    command: python -m uvicorn --host 0.0.0.0 --port 8000 multila.asgi:application
    volumes:
      - ../src:/code
    ports:
      - "8000:8000"
    environment:
      - 'POSTGRES_USER=admin'
      - 'POSTGRES_PASSWORD=<CHANGE_THIS>'
      - 'POSTGRES_DB=multila'
      - 'DJANGO_SETTINGS_MODULE=multila.settings_prod'
```

(continues on next page)

(continued from previous page)

```
    - 'SECRET_KEY=<CHANGE_THIS>'
  depends_on:
    - db
```

2. Make sure the correct server and directory is entered in `Makefile` under `SERVER` and `APPDIR`. Then run `make sync` to upload all files to the server.
3. On the server, do the following:
   - 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
   - run `make check` to check the deployment
   - 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 would use the following:

```
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.

3.4.3 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

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search