MultiLA Web API

Markus Konrad

Apr 18, 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)
- 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*)

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)

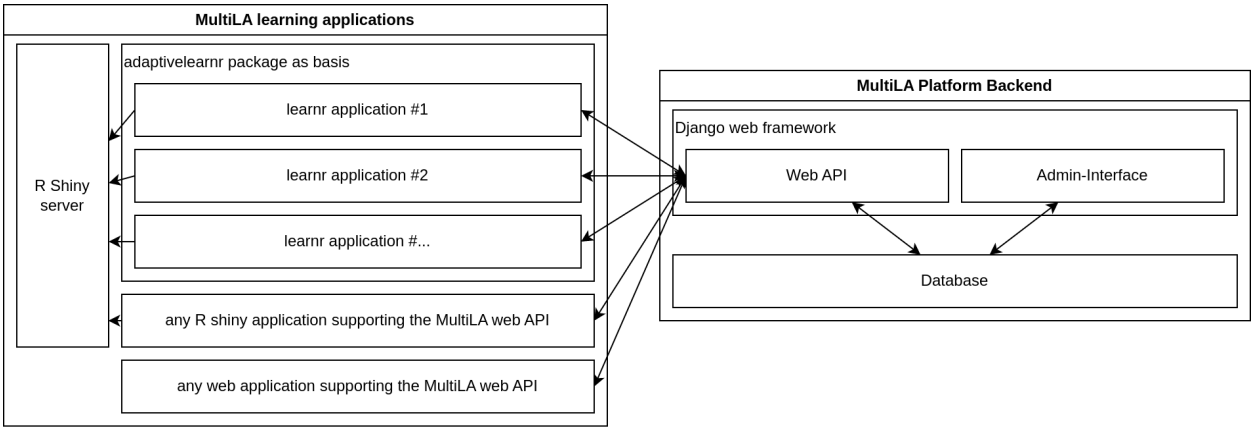
Django REST framework:

- 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)

5 Client-server communication

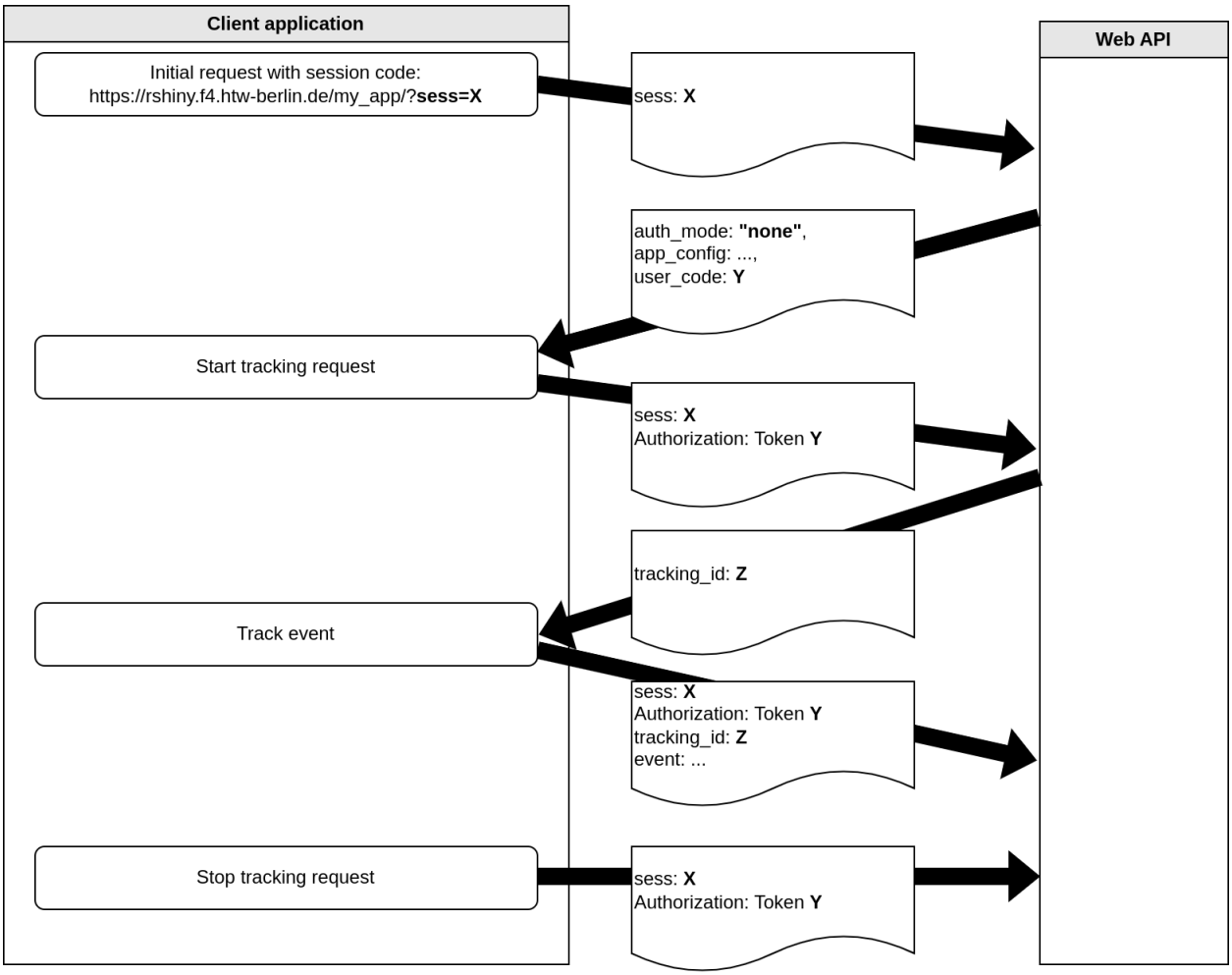
- client-server communication happens on the basis of a RESTful web API implemented in this repository
- the implementation is done in `api/views.py`
- the API exposes an OpenAPI schema under the URL `http[s]://<HOST>/openapi` when `settings.DEBUG` is `True`

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)

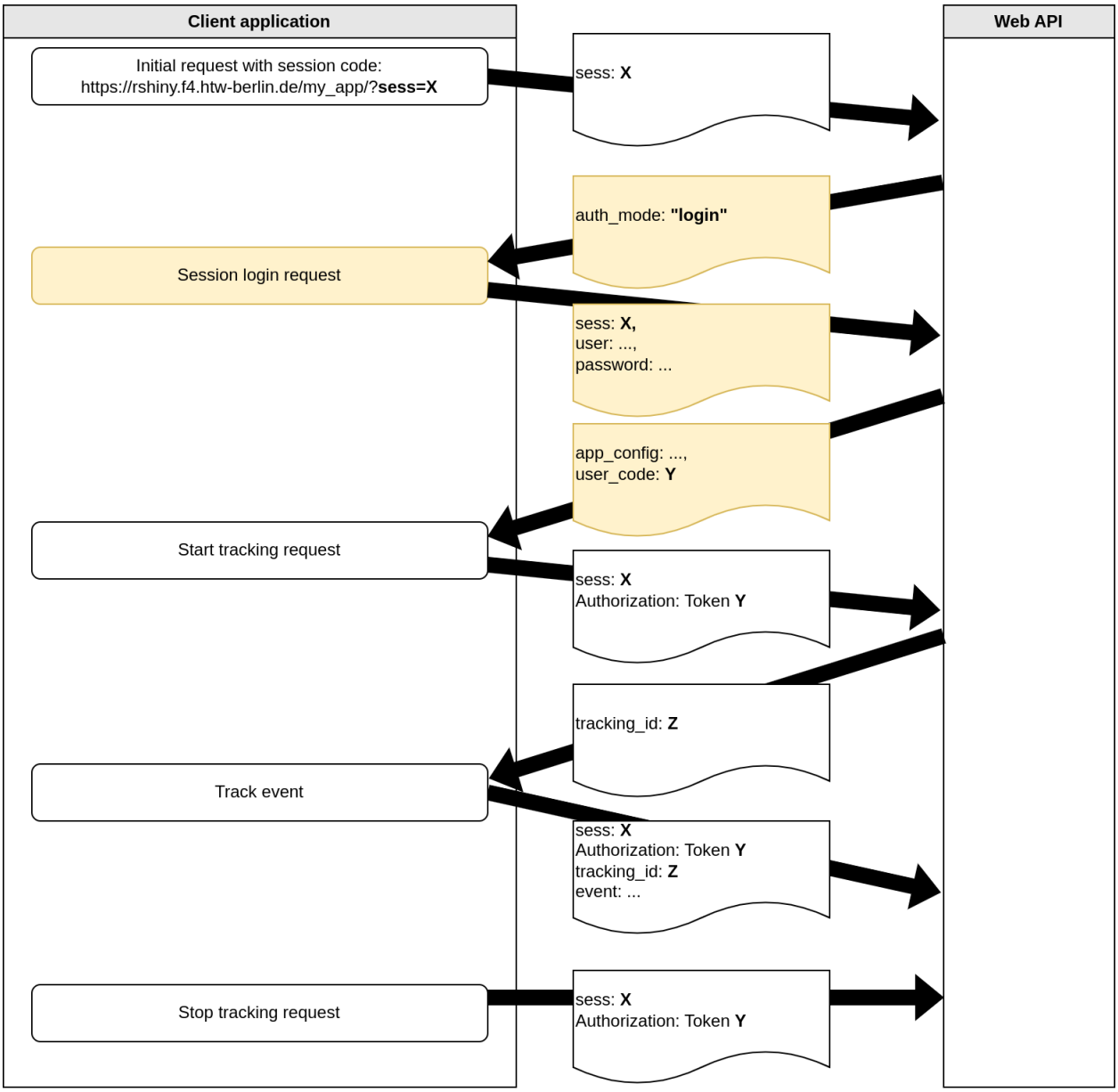
Without login (“anonymous”)

- 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



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 to by using a Python interpreter inside a docker container.

6.1 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`

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`

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 Server deployment

7.1 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!**

7.2 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 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.

7.3 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
- 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 Indices and tables

- genindex
- modindex
- search