https://www.netidee.at/detect

DETECT Technical Guide

Dieser Technical Guide beschreibt die Installation und Architektur der DETECT Applikation. Darüber hinaus beschreibt er die Schritte zum Aufsetzen der Entwicklungsumgebung, um die Applikation über das Projekt hinaus weiter entwickeln zu können. Wie in Opensource-Projekten mit internationalem Publikum üblich, ist diese Dokumentation auf Englisch verfasst.

1 Inhalt

2 Introduction 3

3 Overview 3

4 Local environment setup 4

4.1 Node and npm 4

4.2 Python environment setup 5

4.3 Local deployment 5

5 Docker setup 5

5.1 Steps to start from scratch: development 5

5.2 Cleaning up old containers and database content 6

5.3 Production deployment 6

5.4 Manage 6

6 Advanced Developer documentation 7

6.1 Application settings 7

6.2 Asset Management 7

6.3 DETECT Session 7

6.4 Concepts 7

Tasks 7

6.5 Database schema / Flask models 8

6.6 Testing 9

6.7 Running Tests in VSCode 10
6.8 Linting 10

7 Advanced Devops Documentation 10

7.1 Database migrations 10

2 Introduction

This guide explains how to set up, develop, and deploy the DETECT application "Fake-Shop Explorer".

Fake-Shop Explorer is a prototype of a browser-based gamified crowd-sourcing application (i.e. a game) that collects user/player input on potential fake shops.

Fake-Shop Explorer has been developed and tested on Ubuntu 20.04.4 LTS (native and WSL Windows Subsystem for Linux) for use in Firefox and Chrome.

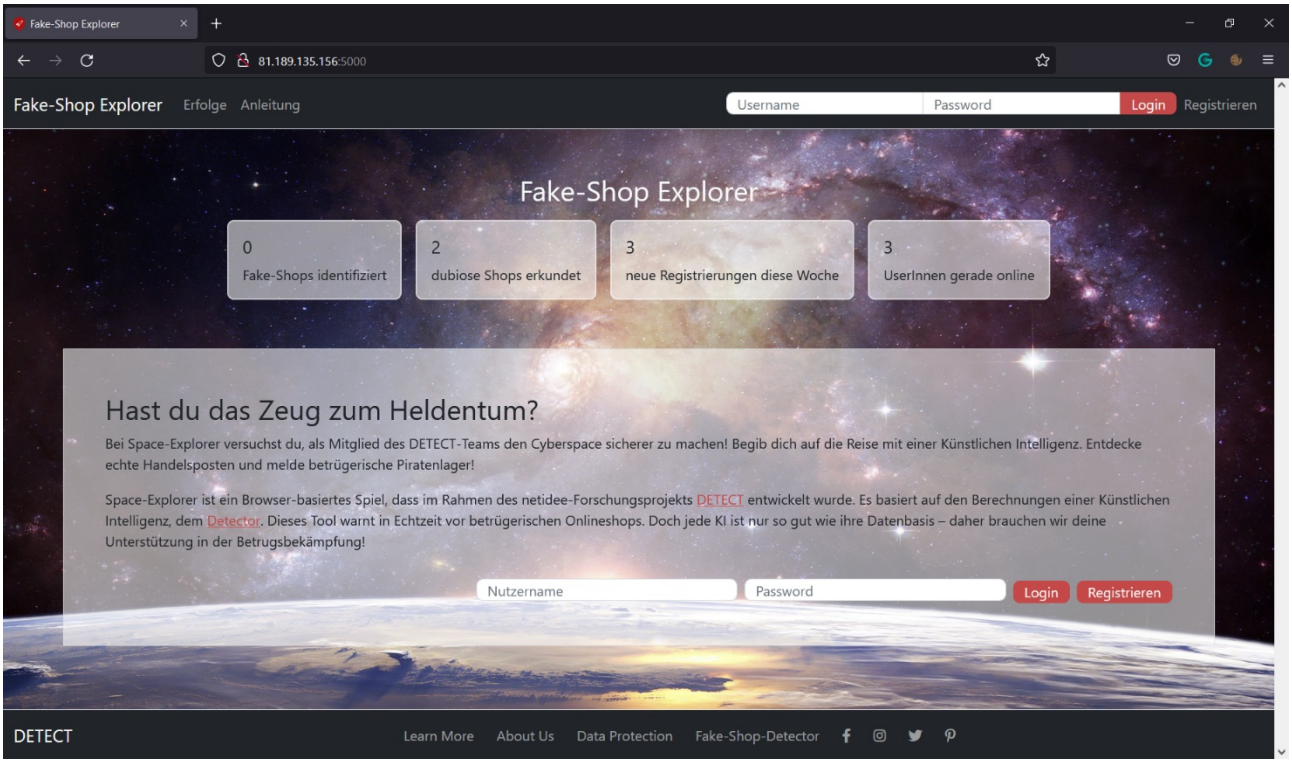

Figure 1: DETECT starting screen with introduction, overview statistics, and logins

3 Overview

Fake-Shop Explorer is implemented as a Flask application. It can be run in a containerised setup (using Docker and docker-compose) or locally without Docker.

In production deployment, Fake-Shop Explorer uses two web services:

1. **Fake shop detector** which has a public API: https://api.fakeshop.at/fake-shop-detector/api/1.2/ui/ and
2. **Fake shop database** https://db.malzwei.at/api/ which requires an access token to use. If you want to use the fake shop database, obtain a token and place it in the path that is defined in detect/settings.py for DEV_DB_API_KEY_LOC. As a **fallback** (without token) sample URLs are read from a **local** resource file.

Depending on the game level, players are asked different questions about the websites (potential fake shops). For this purpose, the websites are embedded into the Fake-Shop Explorer interface:

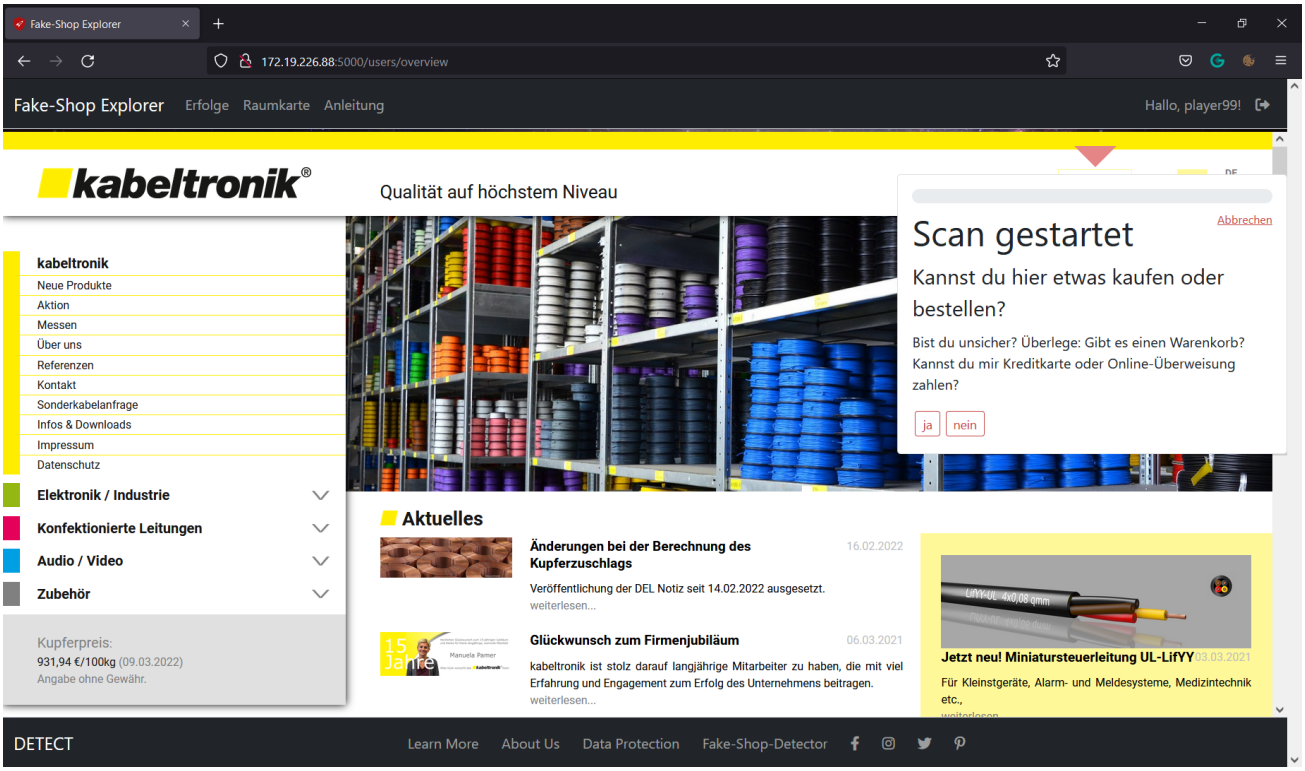

Figure 2: Fake-Shop Explorer interface displayed on top of a shop website

Player inputs are saved to a database (MySQL in production, Sqlite for development).

Since Fake-Shop Explorer can be run locally or using Docker, the following instructions describe both options.

4 Local environment setup

To get started, download the sources:

```
git clone https://git-service.ait.ac.at/dsai-public/detect
```

and change your working directory to the download folder.

4.1 Node and npm

Fake-Shop Explorer development requires **node 16**. (Don't install node 17+ because of https://stackoverflow.com/questions/69719601/getting-error-digital-envelope-routines-reason-unsupported-code-err-oss)

It is recommended to install `nvm`. Based on https://askubuntu.com/a/1009527: To avoid conflicts, remove existing installs:

```
sudo apt purge nodejs npm
```

then install nvm LTS and use it

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.8/install.sh |
bash
nvm install --lts
```

Finally, run `npm install` to install the dependencies specified in package.json.

4.2 Python environment setup

Fake-Shop Explorer development requires **Python >= 3.9**. A good option to create a dedicated `detect` Python environment for development purposes is to use conda:

```
conda create --name detect python=3
```

and - once the `detect` environment is created - activate it and install the development dependencies:

```
conda activate detect
pip install -r requirements/dev.txt
```

4.3 Local deployment

After installing node, npm, and Python, the Fake-Shop Explorer can be built and run using:

```
npm run-script build # use build-debug for non-minimized javascript
npm start  # run the webpack dev server and flask server using concurrently
```

When the app starts, the log will show where it is launching, something like `http://172.30.212.54:5000/`.

The local version of this app uses Sqlite. You can access the database using:

```
sqlite3 /tmp/dev.db
```

5 Docker setup

The containerized version of Fake-Shop Explorer consists of three main services: `flask-dev`, `flask-prod`, and `manage`.

Services and images are defined in `docker-compose.yml` resp. `Dockerfile`.

5.1 Steps to start from scratch: development

To get started, clone the repo, build and start the **dev** service:

```
git clone https://git-service.ait.ac.at/dsai-public/detect.git
cd detect-prototype
docker-compose build --no-cache flask-dev
docker-compose up -d flask-dev
```

Then build the app resources inside the dev container:

```
docker exec -it detect-prototype_flask-dev_1 /bin/bash
npm run-script build
exit

docker-compose down
```

Note: A docker volume `node-modules` is created to store NPM packages and is reused across the dev and prod containers. For the purposes of DB testing with `sqlite`, the file `dev.db` is mounted to all containers. This volume mount should be removed from `docker-compose.yml` if a production DB server is used.

Note: The list of `environment:` variables in the `docker-compose.yml` file takes precedence over any variables specified in `.env`.

5.2 Cleaning up old containers and database content

In development, it is often easier to start from scratch with an empty database and new containers when the schema or initial table content change. You can remove the old containers and database content using:

```
docker image remove detect-development:latest
docker image remove detect-manage:latest
sudo rm -r .mysql-data/
```

5.3 Production deployment

supervisord is used for process management and gunicorn is used as HTTP server. Settings for supervisord are defined in `supervisord-conf`, settings for gunicorn are defined in `supervisord_programs/gunicorn.conf`,

Build the production container:

```
docker-compose build --no-cache flask-prod
```

Start the container in detached mode:

```
docker-compose up -d flask-prod
```

Check if detect-prototype_flask-prod_1 and detect-prototype_db_1 run:

```
docker ps
```

To access the logs of a background container (started with -d) use:

```
docker logs detect-prototype_flask-dev_1
```

To access the containerized MySQL database, connect to the container:

```
docker exec -it detect-prototype_db_1 /bin/bash
```

and run

```
mysql -u detect -p detect
```

5.4 Manage

The manage container is used to run commands using the `Flask CLI`:

```
docker-compose run --rm manage <<COMMAND>>
```

Therefore, to initialize a database you can run:

```
docker-compose run --rm manage db init
docker-compose run --rm manage db migrate
```

```
docker-compose run --rm manage db upgrade
```

Similarly, to open the interactive Flask shell, run

```
docker-compose run --rm manage db shell
```

6 Advanced Developer documentation

6.1 Application settings

Settings can be modified in `detect/settings.py`

6.2 Asset Management

Files placed inside the `assets` directory and its subdirectories (excluding `js` and `css`) will be copied by webpack's `file-loader` into the `static/build` directory. In production, the plugin `Flask-Static-Digest` zips the webpack content and tags them with a MD5 hash. As a result, you must use the `static_url_for` function when including static content, as it resolves the correct file name, including the MD5 hash. For example

```
<link rel="shortcut icon" href="{{static_url_for('static',
filename='build/img/favicon.ico') }}">
```

If all of your static files are managed this way, then their filenames will change whenever their contents do, and you can ask Flask to tell web browsers that they should cache all your assets forever by including the following line in `.env`:

```
SEND_FILE_MAX_AGE_DEFAULT=31556926  # one year
```

6.3 DETECT Session

`detect/detect_session.py` wraps the Flask session object and should be used instead of the session object to store user-specific information.

6.4 Concepts

In the game, users travel through space and visit unknown planets.

Tasks

Tasks have to be solved to collect fuel for the spaceship, and to map/categorize new planets. Thus, two task types exist:

- **fuel**/**simple** tasks (level 0): replenish energy for the spaceship by answering a simple yes/no decision "shop or not?"
- **planet** tasks (level 1-7): answer detailed questions about potential web/fake shop sites. Planet tasks become increasingly complicated with level progress.

In the data model, `task` is used synonymously for website url, while `subtask` is synonymous for answering questions about the website url.

6.5 Database schema / Flask models

`detect/user/models.py` contains the app-specific database tables and view definitions. The full schema is shown in the ER diagram on the right. The tables contain the following information

- `events`: logs user actions, such as visiting planets or gathering fuel
- `options`: answer options for game level questions
- `roles`: user roles, not used yet
- `subtasks`: game level questions
- `tasks`: URLs of the websites / fake-shops
- `user_task_link`: logs user guess regarding shop / no-shop / fake shop classification
- `user_task_subtask_link`: logs user answer to game level questions (e.g. if "Impressum" (site notice) exists or looks plausible)

The views `v_agreement`, `v_accuracy`, and `v_accuracy_monthly` are used to compute how well a user's answers agree with the majority opinion of other users.

`detect/user/models.py` also contains two functions that are called from `app.py` before the database is first queried:

- `create_subtasks_and_options` parses the subtasks (game level questions) defined in `detect/resources/subtasks.json` and inserts them into the the `subtasks` table.
- `create_users` creates some test users for easier testing and should be removed in production.

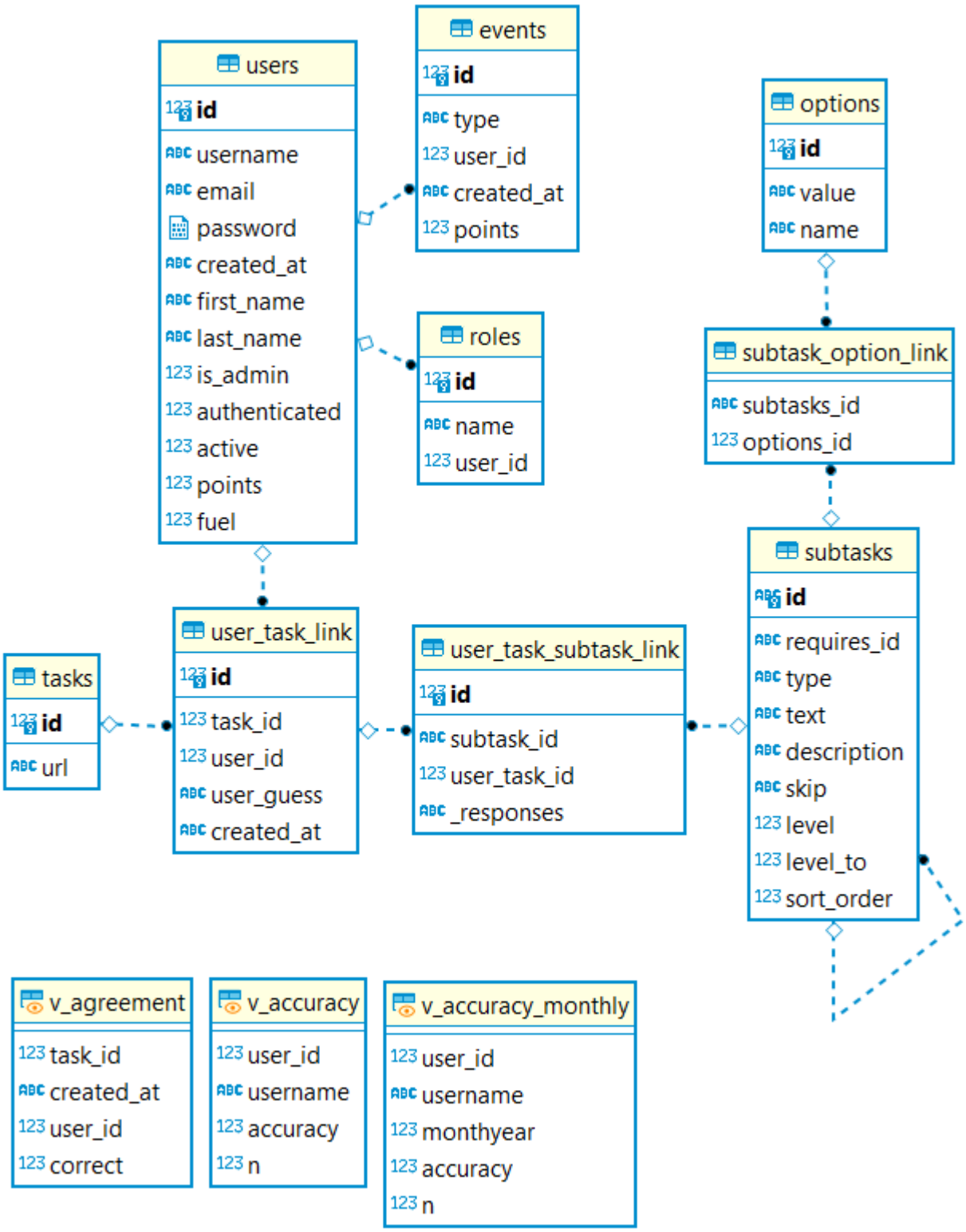

Figure 3: DETECT database schema

6.6 Testing

Fake-Shop Explorer comes with an extensive suit of tests.

To run all tests with Docker run:

```
docker-compose run --rm manage test
```

Alternatively, without Docker, run:

```
flask test
```

6.7 Running Tests in VSCode

In VSCode, the following launch.json can be used:

```
{
    "configurations": [
        {
            "name": "DETECT local debugger",
            "type": "python",
            "request": "launch",
            "program": "main.py",
            "console": "integratedTerminal",
            "cwd": "/<your-project-path-here>/detect-prototype",
            "python": "/<your-python-path-
here>/anaconda3/envs/detect/bin/python",
        }
    ]
}
```

6.8 Linting

The `lint` command will attempt to fix any linting/style errors in the code. If you only want to know if the code will pass CI and do not wish for the linter to make changes, add the `--check` argument.

To run the linter, run:

```
docker-compose run --rm manage lint
flask lint # If running locally without Docker
```

7 Advanced Devops Documentation

7.1 Database migrations

Whenever a database migration needs to be made. To generate a new migration script, run:

```
docker-compose run --rm manage db migrate
flask db migrate # If running locally without Docker
```

Then apply the migration:

```
docker-compose run --rm manage db upgrade
flask db upgrade # If running locally without Docker
```

For a full migration command reference, run `docker-compose run --rm manage db --help`.

If you will deploy your application remotely (e.g on Heroku) you should add the `migrations` folder to version control. You can do this after `flask db migrate` by running the following commands

```
git add migrations/*
git commit -m "Add migrations"
```

Make sure folder `migrations/versions` is not empty.