DjangifyLab Project

A sandbox to test Django app packages before integrating it into a production environment

Version: v1.3.0

LORENA GOLDONI

UniMoRe - Università degli Studi di Modena e Reggio Emilia

Cloud and Edge Computing Project

Prof. Francesco Faenza

July 30, 2025

Abstract

DjangifyLab is a sandbox environment that streamlines the testing of reusable Django app packages by providing a clean, isolated Django project with zero setup required. Developers can plug in, run migrations, and execute Django apps without relying on an existing host project, avoiding common pitfalls such as hidden dependencies, preconfigured environments, or missing configuration steps. The platform offers full control over environment variables, supports Docker-based deployment simulations, and allows the integration of external configuration files—ensuring that Django apps are self-contained, reusable, and production-ready.

For the Cloud Computing exam project, the focus was placed on **DevOps and CI/CD practices**. This included automating workflows with Docker and GitHub Actions, building isolated upgrade/install containers, managing environment configuration via *.env* files, and enabling reproducible testing pipelines to simulate real-world deployment scenarios.

Project Architecture

Below is the actual directory layout of the project:

```
├── .env                          # Environment variables for runtime and CI
├── .github/                      # GitHub Actions and linters configs
│   └── workflows/
│       └── pull_request.yml
├── app_config_files/             # Optional config files per apps
├── CHANGELOG.md
├── djangifylab_project/          # Core Django project with shared settings
├── docker-compose.database.yml
├── docker-compose.override.yml
├── docker-compose.search.yml     # Optional Elastic/Open search services
├── Dockerfile
├── entrypoint.py                 # CLI for install/upgrade workflows
├── example-apps/                 # Sample apps and fixtures for testing
│   ├── packages/
│   └── fixtures/
├── LICENSE
├── manage.py
├── README.md
├── requirements.txt              # DjangifyLab project requirements
├── requirements_dev.txt          # Development dependencies
├── requirements_linters.txt      # Linter requirements
├── tests/                        # Pytest-based test suite
└── upgrade_logs/                 # Logs from container runs
```

System Structure

The project is organized around a single Django host project (`djangifylab_project`) that acts as a neutral, isolated environment to test third-party or custom Django apps. The architecture includes:

- A unified `settings.py` file with dynamic DB selection via `.env`
- Apps installed as Python packages via `pip install`

- A `manage.py` entrypoint used to run migrations, tests, and fixture loading
- A flexible CLI interface via `entrypoint.py` to support app installation or upgrade flows

The use of environment variables allows dynamic configuration of the database backend (SQLite, postgreSQL, or others in the future).

Dockerized Components

The system uses Docker Compose to manage the following key containers:

- `app-installer`: installs and runs a Django app or collection of apps
- `upgrade-runner`: handles version upgrades and migrations for apps
- `postgres`: default database container, can be swapped via `.env` (`sqlite` or `mongo` also available)
- (Optional) other services if needed by the Django third-party apps such as `Elasticsearch` and `Opensearch`

Code Snippet

Example from `docker-compose.override.yml`:

```
services:
 upgrade-runner:
   build:
     context: .
     dockerfile: Dockerfile
   env_file:
     - .env
   environment:
     DJANGO_SETTINGS_MODULE: djangifylab_project.settings
   volumes:
     - ./djangifylab_project:/app/djangifylab_project
```

```
    - ./example-apps/packages:/app/packages
    - ./example-apps/fixtures:/app/fixtures
    - ./upgrade_logs:/app/logs
  command: >
    python entrypoint.py --mode upgrade
      --previous_version /app/packages/buffalogs-2.7.0.tar.gz
      --new_version /app/packages/buffalogs-2.8.0.tar.gz
      --fixture /app/fixtures/buffalogs_complete_fixture.json
  depends_on:
    - postgres
  networks:
    - djangifylab_network

 app-installer:
  build:
    context: .
    dockerfile: Dockerfile
  env_file:
    - .env
  environment:
    DJANGO_SETTINGS_MODULE: djangifylab_project.settings
  volumes:
    - ./example-apps/packages:/app/packages
  command: python entrypoint.py --mode install --target /app/packages
  depends_on:
    - postgres
  networks:
    - djangifylab_network

networks:
 djangifylab_network:
  driver: bridge
```

DevOps and CI/CD pipeline

DjangifyLab adopts an opinionated CI/CD strategy rooted in best practices of test automation, reproducibility, and pre-merge verification. All core DevOps steps are integrated into a GitHub Actions workflow triggered on pull requests.

Pull Request Workflow Overview

The `pull_request.yml` file defines the following phases:

Environment Setup

```
- name: Load environment variables
  run: |
    set -a
    source .env
    set +a
```

This ensures all configuration (such as DB_ENGINE, DB_NAME, DB_HOST) is centralized and consistent across local and CI runs.

Dependency and Linter Installation

The linters are explicitly installed as follows:

```
pip install black==25.1.0 flake8==7.3.0 isort==6.0.1
```

Each linter is configured via dedicated configuration files in `.github/configurations/python_linters/`.

Linting steps:

```
- name: Black linter
  run: black --config .github/configurations/python_linters/.black .
- name: Flake8 linter
  run: flake8 . --config .github/configurations/python_linters/.flake8
- name: Isort linter
  run: isort --sp .github/configurations/python_linters/.isort.cfg
--profile black .
```

Docker Image Building

The workflow builds containers needed for end-to-end app lifecycle testing:

```
- name: Build all required Docker images
  run: docker compose --env-file .env -f docker-compose.database.yml -f
docker-compose.override.yml build postgres upgrade-runner app-installer
```

Services Initialization and App Testing

PostgreSQL is started in the background:

```
- name: Start dependent services (postgres)
  run: docker compose --env-file .env -f docker-compose.database.yml -f
docker-compose.override.yml up -d postgres
```

Then, each of the two custom containers is run:

- **upgrade-runner**: applies app migrations, upgrades fixtures
- **app-installer**: tests fresh installability

```
- name: Run upgrade-runner container
  run: docker compose --env-file .env -f docker-compose.database.yml -f
docker-compose.override.yml run --rm upgrade-runner
```

```
- name: Run app-installer container
  run: docker compose --env-file .env -f docker-compose.database.yml -f
docker-compose.override.yml run --rm app-installer
```

Django Unit Tests

After integration-level validation, unit tests are executed using `pytest`:

```
- name: Python tests
  run: |
    source venv/bin/activate
    pytest tests/
```

Automated Release Deployment Workflow Overview

In addition to the pull request checks, a second workflow `deploy.yml` handles automatic version tagging and release once a pull request from `develop` is merged into `main`.

The Deploy New Version workflow is triggered only when:

- A pull request targeting `main` is merged
- The source branch is `develop`
- The PR title is a valid SemVer version (e.g. 1.0.0)

If conditions are met, the workflow:

- Parses the pull request title as a version
- Creates a Git tag (Version v1.0.0, etc.)
- Pushes the tag to the remote repository
- Uses softprops/action-gh-release to create a GitHub Release and auto-generate release notes

Key step of the workflow:

```
- name: Create GitHub Release
  uses: softprops/action-gh-release@v2
  with:
    tag_name: "v${{ env.VERSION }}"
    name: "Version v${{ env.VERSION }}"
    generate_release_notes: true
```

Additional DevOps Features

Branch Protection Rules

Pull requests are required to successfully pass the continuous integration (CI) workflow before being eligible for merging.

If the project involves multiple contributors, it is considered a best practice to enforce mandatory code review approvals prior to merging changes into protected branches.

Dynamic DB Configuration

The `settings.py` file reads the database engine from `.env`, supporting both PostgreSQL and SQLite, based on the .env-driven modern Django structure that allows scalability.

```
DATABASES = {
   "default": {
       "ENGINE": os.getenv("DB_ENGINE", "django.db.backends.sqlite3"),
       "NAME": os.getenv("DB_NAME") or str(BASE_DIR / "db.sqlite3"),
       "USER": os.getenv("DB_USER", ""),
       "PASSWORD": os.getenv("DB_PASSWORD", ""),
       "HOST": os.getenv("DB_HOST", ""),
       "PORT": os.getenv("DB_PORT", ""),
   }
}
```

Future Improvements and Extensions

Several enhancements could be introduced to further expand the capabilities and robustness of the DjangifyLab environment. First, **implementing monitoring and observability** tools, such as **Prometheus** for metrics collection and **Grafana** for data visualization, would allow developers to track performance indicators, container resource usage, and detect anomalies during the app installation or upgrade processes.

Furthermore, a valuable enhancement to the testing workflow would be the **inclusion of integration testing stages that involve widely-used Django extensions and middleware**. For instance, testing the integration of apps with libraries like:

- `django-allauth` for authentication and user registration,
- `django-rest-framework (DRF)` for building RESTful APIs,
- `django-cors-headers` for handling cross-origin requests.

would enable developers to identify potential integration conflicts early and improve the robustness and interoperability of their applications. These integration stages could be executed automatically within the CI/CD pipeline, where the selected third-party packages are installed and configured within the isolated testing environment before the app is installed and tested.

Support for **multiple database engines** (e.g., MySQL, Neo4j) through modularized .env and settings.py configurations could also improve portability.

Finally, deploying the sandbox in a **Kubernetes-based environment** could simulate even more realistic production scenarios, enabling DjangifyLab to evolve into a truly cloud-native testing framework.