Technische Hochschule Deggendorf

Faculty of Applied Computer Science

Master Life Science Informatics

Documentation

NBREATH_DB - Nasal Bacterial & REspiratory Atlas in Humans DataBase - Django based Web Application

The Landscape of the Nasal Microbiome and Its Relationship with Other Body Site Microbiomes in Humans

Author

Ridaa Fatima

Supervisors

Prof. Dr. Javier Valdes
Deggendorf Institute of Technology,
Faculty of Computer Science

Dr. Reihaneh Mostolizadeh
Justus Liebig University Giessen,
Bioinformatics and Systems Biology

Contents

1 Welcome to NBREATH_DB - Nasal Bacterial & REspiratory Atlas in Humans DataBase

NBREATH_DB is a Django (Python) based web application designed to manage and visualize Nasal Microbiomes and their migrations and interactions.
It integrates a relational PostgreSQL database for structured data storage with a Neo4j graph database for exploring nasal micorbiomes migrations and interactions.

1.1 Documentation Overview

This documentation provides information on how to install and run the application, via Docker Compose setup.
The documentation also explains how to use the web interface for data entry and Neo4j graph visualization. Additionally, it includes descriptions of the database models, Django forms, view functions, templates and front-end integration using D3.js for interactive graph visualization.

1.2 Quick Access to Main Features

The main functionalities available to users include:

- Manage and track biological entities: Species, Body Sites, Diseases, Products, Interactions, Migrations, and Product Events.
- Perform CRUD (Create, Read, Update, Delete) operations via user friendly forms and templates.
- Export data to Neo4j and explore interactive graphs of entity relationships.
- Access all lists and visualizations through the web interface.
- Run the application reliably using Docker Compose with all dependencies configured.

The following sections of this documentation guide the user through project folder structure, installation, environment setup, database models, forms, views, templates, and graph visualization.

2 Project Folder Structure

The project is organized as follows:

```
NasoBiomeKnowledgeBase/      # Repository root

 manage.py                   # Django command line utility
 NasoBiome/                  # Django app
 requirements.txt            # Python dependencies
 Dockerfile                  # Web service environment
 static/                     # CSS, JS, images
 templates/                  # HTML templates for rendering
    pages
 NasoBiomeKnowledgeBase/     # Inner Django project
     __init__.py
     settings.py             # Django Central configuration file
     urls.py                 # Global URL routing
     wsgi.py                 # WSGI entry point
     asgi.py                 # ASGI entry point

 docker-compose.yaml         # Docker Compose orchestration
 nginx/                      # Nginx configuration
 README.md                   # Documentation
```

Key project components include:

`manage.py` Command line utility for managing the Django project (running server, migrations, etc.).

`docker-compose.yaml` Defines services and dependencies for containerized execution.

`NasoBiome/` Main Django application containing models, views, forms, URLs, and core application logic.

`templates/` HTML templates used by views to render forms, lists, detail pages, and dashboards.

2.1 Django Project Structure

The Django project initialized using the command:

```
django-admin startproject config .
```

This created the core project structure as follows:

```
config/
 __init__.py
 settings.py
 urls.py
 asgi.py
 wsgi.py
```

Functionality of key files:

`settings.py`: Contains project settings including database configuration, installed applications, middleware, templates, and static files settings. Django loads the settings from settings.py.

`urls.py`: Defines global URL routing for the project. Routes incoming requests to the appropriate Django apps and views.

`wsgi.py`: Entry point for WSGI-compatible web servers, used in deployment.

`asgi.py`: Entry point for ASGI-compatible web servers, supporting asynchronous features and WebSockets.

`__init__.py`: Marks the directory as a Python package.

2.2 NBREATH_DB Application

Application contains:

```
NasoBiomeKnowledgeBase/NasoBiome/
 models.py                  # Defines database models
 forms.py                   # App specific routes
 views.py                   # Logic and request handling
 urls.py                    # App specific routes
 neo4j_integration.py       # Neo4j graph export
 admin.py                   # Administration setup
 templates
    index.html
 static
     style.css
     script.js
```

3 Installation

3.1 Docker Setup and Application Containerization

The application is fully containerized using **Docker** and orchestrated with **Docker Compose** for easy deployment. This approach ensures that all services including Django, PostgreSQL and Neo4j are configured correctly and can communicate seamlessly.

3.1.1 System Requirements

The following software must be installed on the host system:

- Docker (version ≥ 20.10)
- Docker Compose (version ≥ 2.0)

No local installation of Python, PostgreSQL, Neo4j is required.

3.2 Environment Setup – Docker Compose (Install Python Dependencies)

All Python package dependencies required to run the application are listed in the file named `requirements.txt`. This ensures that the environment has all necessary libraries for web functionality, database access and data processing.

Docker Compose File All service definitions, network configurations, and environment variables are contained in the `docker-compose.yml` file.
In the Docker Compose setup, the Python dependencies listed in `requirements.txt` are automatically installed when building the Django service image.

How It Works

- The Dockerfile for the Django service copies the `requirements.txt` file into the container:

```
COPY requirements.txt .
```

- It then runs `pip install` to install all dependencies inside the container:

```
RUN pip install --no-cache-dir -r requirements.txt
```

- When the Docker image is built via `docker-compose.yml`, all required Python packages are installed in the container.
- The Django service container starts with all dependencies ready, so the application runs without requiring any manual installation of Python packages on the host system.

3.3 Running the Application - Using Docker (Recommended)

Docker Compose ensures all services (Django, PostgreSQL, Neo4j) run consistently.

Clone the Repository

Clone the latest source code from GitHub:

```
git clone
   https://github.com/MSBIDynamics/Nasal_Community_database.git
cd Nasal_Community_database
```

Build and Start Containers

```
docker compose up --build
```

This builds the Docker images and starts all services. Dependencies from `requirements.txt` are installed automatically inside the container.

Run Database Migrations Inside Docker

```
docker exec -it NasoBiomeKnowledgeBase python manage.py migrate
```

Load Initial Data into the Database

Once the migrations have been applied, the database can be populated with initial data from a fixture file (data.json):

```
docker exec -it NasoBiomeKnowledgeBase python manage.py loaddata
   data.json
```

This command reads the fixture file and inserts predefined nasal microbiome samples, metadata, and related information into the PostgreSQL database inside the Django container. This step ensures the application has the same data needed to reproduce analyses and visualizations.

Start Django Server Inside Docker

```
docker exec -it NasoBiomeKnowledgeBase python manage.py
   runserver 0.0.0.0:8000
```

3.3.1 Access the Application

After successful startup, the following services will be available: Service URL
Web Application `http://localhost:8000`
Neo4j Browser `http://localhost:7474`

User Note

Always clone the repository first.
Use Docker Compose to build and run all services, then apply migrations.

4 Usage

4.1 Data Storage and Processing Workflow

The application manages biological metadata and relational information through a combination of PostgreSQL and Neo4j. The data processing workflow consists of the following steps:

4.1.1 Model Definitions `models.py`

Each biological entity is represented as a normalized database table defined as a Python class in the `models.py` file of the application using Django's Object-Relational Mapping (ORM).

All information related to an entity is stored in model fields, which define the structure, data type and constraints of the underlying database columns.

These fields are automatically translated by Django into PostgreSQL compatible schema definitions during database migration. Table 4.1 summarizes the main biological entities (Models) and their roles.

Table 4.1: Biological entities (Models) and their roles in the application

Model	Role
Species	Microbial species entity
BodySite	Anatomical body site entity
Disease	Disease entity
Product	Product entity
SpeciesInteraction	Species-to-species relationship
MigrationPattern	Species migration relationship
ProductEvent	Product–species relationship

Model Extensibility and Updates

Authorized users or administrators can modify the schema by updating model classes in the `models.py` file.

Any changes to model fields are synchronized with the PostgreSQL database using Django's built-in migration framework.

Command-Line Access

After modifying or adding model fields, database updates are applied via the command line:

```
python manage.py makemigrations
python manage.py migrate
```

These commands automatically generate and apply migration files, ensuring consistency between the application models and the underlying PostgreSQL schema.

4.1.2 Frontend Django Forms `forms.py`

The application provides Django forms defined in `forms.py` for all models in `models.py`. Forms enable structured data entry and editing through the web interface.

Usage via Web Interface

- Each form corresponds to a specific model (e.g., `SpeciesForm`, `BodySiteForm`).
- Forms are implemented using Django's `ModelForm` framework, automatically generating fields, validating input, and integrating with the database.
- The same form is reused for both creating new records and updating existing records.

Rendered Templates Forms are displayed in HTML templates under `templates`, allowing users to submit or edit data via the browser.

4.1.3 View Functions (`views.py`) - Access and Usage

The main application logic is implemented in Django view functions defined in `views.py`. Views act as the intermediary between URL routing, database models, and templates, handling incoming HTTP requests and coordinating interactions between the user, the database, and the frontend.

Each view function determines the action to perform based on the HTTP request type:

- **GET requests:** Retrieve and display data, listing records.
- **POST requests:** Submit form data for creating new records or updating existing records.
- **PUT requests:** Update existing resources.

- **DELETE requests:** Permanently remove records from the database.

Views handle CRUD operations for all entities, passing processed data to templates for rendering.

Selected views interact with Neo4j for graph-based data export and visualization.

Imports and Dependencies

At the top of `views.py`, views import all necessary models and forms:

```
from .models import Species, BodySite, Disease, Product,
                    SpeciesInteraction, MigrationPattern,
                       ProductEvent

from .forms import SpeciesForm, BodySiteForm, DiseaseForm,
   ProductForm,
                   SpeciesInteractionForm, MigrationPatternForm,
                      ProductEventForm
```

These imports enable views to perform database operations, process user input, render templates, and interact with Neo4j.

4.1.4 URL Routing `urls.py`

All view functions are connected to URL endpoints defined in `urls.py`.
For example, to add a new species:

```
/species/add
```

Similarly, list, view, update, delete, and export actions are accessible via their corresponding URLs.

4.1.5 Templates and CRUD Operations

Templates provide a user-friendly interface to manage all biological entities.

Data passed from views is rendered in templates using tables, forms and interactive elements.

Templates ensure a clean, organized presentation of information so users can add, view, update, or delete records without directly interacting with the database.

Templates Being Used

- **Add new records:** `*_add.html`
- **List records:** `*_list.html`
- **View and Update records:** `*_detail.html`. Update forms are embedded in the detail page.
- **Delete records:** Handled via buttons on the detail page; no separate delete template is used.

Step-by-Step Workflow

- **Adding Records:** Navigate to the **Add** page, fill out the form, and submit.
- **Viewing Records:** Go to the **view** page, browse entries, and click a record to see details.
- **Updating Records:** Open the **detail** page, edit the pre-filled form, and submit changes.
- **Deleting Records:** Click the **delete** button on the detail page; the system confirms and removes the record.

4.1.6 Neo4j Integration

The application integrates Neo4j for graph database. View functions in `views.py` interact with Neo4j using the official Python driver, enabling graph queries and visualization.

Exporting Data to Neo4j

- Individual models (e.g., `Species`, `BodySite`, `Disease`, `Product`) can be exported.
- All models can be exported simultaneously via the endpoint: `/export/all/`.
- Users can push updates to Neo4j using the **Export to Neo4j** button on the Home Page whenever new records are added or existing records are modified.
- This ensures the graph visualization always reflects the latest data.

Integration Functions All Neo4j operations are implemented in `neo4j_integration.py`

Access from Python Advanced users can directly interact with Neo4j via Python scripts using the official driver:

```
from neo4j import GraphDatabase

driver = GraphDatabase.driver(
"bolt://neo4j:7687",
auth=("neo4j", "neo4jpassword")
)

# Update when using a different host
```

Retrieve all nodes and relationships using the Cypher query:

```
MATCH (n)-[r]->(m)
RETURN n, r, m
```

This allows full visualization and analysis of entity relationships in the system.

4.2 User Interface and Pages

4.2.1 Home Page `home.html`

The Home Page serves as the main entry point and dashboard of the application, providing a consolidated view of all biological data.

4.2.2 Functionality

Browse all entities: Species, Body Sites, Diseases, Products, Species Interactions, Migration Patterns, and Product Events.

Navigate to specific lists (Interactions, Migrations, Product Events) to view, edit, or delete records.

4.3 Graph Visualization Template (`graph.html`)

Renders interactive graph visualizations of relationships between biological entities from Neo4j.

Interactive graph visualization of entity relationships from Neo4j.

Rendered using D3.js for dynamic, force-directed layouts.

Allows intuitive exploration of relationships without needing Cypher or graph knowledge.

4.3.1 User Interaction

- Click nodes to show details in a sidebar.
- Drag nodes to explore connections.
- Zoom and pan for detailed exploration.
- Highlight connected nodes and edges by clicking.