Technische Hochschule Bingen
Fachbereich 2 – Technik, Informatik und Wirtschaft
Angewandte Bioinformatik (B. Sc.)

Implementation of a Datawarehouse Prototype

Scientific Report for DAWE
abgegeben am: 03.07.2025
von: Luka Faensen

Dozent: Prof. Dr. Asis Hallab

List of Figures

Contents

1 Introduction

Databases play a central part in the current digital infrastructure. Every student of bioinformatics should know the basics of web development and data warehouses. During this project in the module DAWE, our task was to set up a backend and frontend system that teaches us the basics of fullstack development [1]. The language chosen for this project was Python, as it has great documentation and support in online communities. The aim for this project is to build a functional prototype database that supports the CRUD functions inside a simple User Interface.

2 Materials

2.1 Backend Dependencies

The following table lists all Python package versions used in the backend:

Package Name	**Version**
alembic[7]	1.16.2
biopython	1.85
blinker	1.9.0
click	8.2.1
Flask	3.1.1
flask-cors	6.0.1
Flask-Migrate	4.1.0
Flask-SQLAlchemy	3.1.1
greenlet	3.2.3
itsdangerous	2.2.0
Jinja2	3.1.6
Mako	1.3.10
MarkupSafe	3.0.2
numpy	2.3.1
psycopg2-binary	2.9.10
SQLAlchemy	2.0.41
typing_extensions	4.14.0
Werkzeug	3.1.3

Note: This list includes direct dependencies and their underlying dependencies.

2.2 Frontend Dependencies

Frontend Technologies, Tools, Frameworks, and Libraries:

Package Name	Version
Node.js	22.17.0
npm	10.9.2
react	16.3.0
react-router-dom	7.6.3
axios	1.10.0
molstar	4.18.0
react-scripts	5.0.1
web-vitals	2.1.4

3 Methods

3.1 Navbar

A global Navigation-Bar is implemented in `Navbar.js`. This allows fast access to the protein list, the addition of new proteins, and the "Upload Fasta" functions where bulks of new proteins can be uploaded via a .fasta file.

3.2 Master-Detail View

The user interface begins with the master view when loaded the first time. Here the user sees a detailed list of the proteins currently in the database.

This view separates into two main areas - the master view presents all proteins on the current page with basic information about them, shown in Figure 1. The next logical element is the detail view, which shows all relevant information about the protein selected in the master view. Here are also the interaction buttons for the selected entry, like delete, edit and if a PDBId exists, 3DStructure as seen in Figure 2.

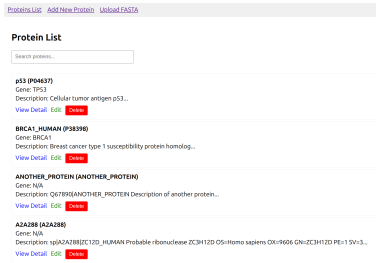

Figure 1: Master-View

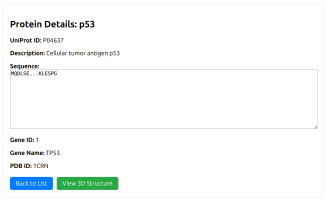

Figure 2: Detail-View

3.3 Implementation

The implementation for this master-detail view is managed by two dedicated React components: ProteinList.js for the master view and ProteinDetail.js for the detail view. The navigation between those two components is facilitated by react-router-dom[5](This implements dynamic routing inside this web app).

3.3.1 Master-Component

- **Data-view:** This component is responsible to display a paginated list of proteins.
- **Interaction:** Users can filter the protein list via the search(`<input type="text">`) bar. The two buttons "previous" and "next" allow for navigation through bigger datasets.
- **Detail-view:** Every protein entry in the list includes a "View Detail"-Link(`<Link to="/proteins/:id">`, that routes the user to the proteins entry. For fast access to the "Edit" and "Delete" functions, those two buttons have been located next to the detail-view for each protein in the master-view as well.
- **API-Interaction:** This component calls protein data via `proteinService.getProteins()` from the backend. Searchterms and pagination parameters are conserved.
- **State-Management:** `useState` Hooks are used to manage loading, errors, current page, pagination and searchterms. `useEffect` is used to reload data after searching and pagination

3.3.2 Detail-Component

- **Dataview:** This component gets the Protein-ID via the URL-Parameters (`useParams`). Based on this ID it calls all the details via `proteinService.getProtein()` from the backend.
- **3D-Structure:** For proteins that have an PDB-ID, a button "View 3D Structure" is shown. This navigates the user to the embedded 3D-Visualization tool.(Mol*[2])
- **Navigation:** A simple "back to list" link allows the return to the master view.

3.3.3 Edit-Component

- This component manages the editing and creation of new entries. Its called seperatly via `/create` and `/edit/:id`
- In editing mode the block "Uni-Prot ID" is shown as readOnly to keep the primary key, while all other parameters are editable.

- Inputs are updated via `handleChange` and via `handleSubmit` sent to the backend, where the functions `proteinService.createProtein()` or `proteinService.updateProtein()` are called respetivly.

3.4 API & Server

This section outlines the backend implementation, focusing on the API design, CRUD operations, search functionality with pagination, and server management.

3.4.1 Frontend-Backend Interaction and API Type

The frontend and backend communicate via a **RESTful API**[8]. This was chosen because it's the first option in the assignment. REST uses standard HTTP methods (GET, POST, PUT, DELETE) to operate on resources. This approach works great with the resource-oriented kind of work of protein data management.

3.4.2 Implementation of CRUD Operations

The backend, implemented using Flask, provides a set of RESTful endpoints for managing protein records. All operations are handled within the `app.py` file.

- **Create (`POST /proteins`):**
  - Receives JSON data via an HTTP `POST` request.
  - Basic validation checks for required fields (`id`, `name`) and uniqueness of the `id`.
  - A new `Protein` object is instantiated with the provided data, including `id`, `name`, `description`, `sequence`, `gene_id`, and `pdb_id`.
  - The object is added to the database session and committed.
  - Returns the serialized new protein object with a `Created` status.
- **Read (`GET /proteins` and `GET /proteins/<protein_id>`):**
  - **List Retrieval (`GET /proteins`):** Fetches a paginated list of proteins. This supports search queries.
  - **Detail Retrieval (`GET /proteins/<protein_id>`):** Retrieves a single protein record by its unique `protein_id`. Gives a `404 Not Found` status if the protein does not exist.
  - Protein objects are serialized into JSON format before being sent as responses.
- **Update (`PUT /proteins/<protein_id>`):**
  - Receives JSON data via an HTTP `PUT` request for a specific `protein_id`.

- Retrieves the existing protein record from the database. Returns `404 Not Found` if not found.
- Updates the protein's attributes with the provided data.
- The changes are committed to the database session.
- Returns the serialized updated protein object.

- **Delete (`DELETE /proteins/<protein_id>`):**
  - Receives an HTTP `DELETE` request for a specific `protein_id`.
  - Retrieves the existing protein record. Returns `404 Not Found` if not found.
  - The object is removed from the database session and committed.
  - Returns a `204 No Content` status upon successful deletion.

3.4.3 Search Functionality with Pagination

Efficient data navigation is provided through search and pagination features on the `GET /proteins` endpoint.

- **Search:**
  - The endpoint accepts a `search` query parameter (e.g., `/proteins?search=p53`).
  - The backend performs a case-insensitive search (`ilike`), including `Protein.name`, `Protein.description`, `Gene.name`, and `Organism.name`. This is done by performing `outerjoin` operations on the `Protein`, `Gene`, and `Organism` models to allow comprehensive filtering.

- **Pagination:**
  - The endpoint accepts `page` and `per_page` query parameters (e.g., `/proteins?page=2&per_page=20`).
  - Flask-SQLAlchemy's `paginate()` method is used to retrieve a specific subset of records, improving performance.
  - The API response includes metadata about the pagination, such as `current_page`, `total_pages`, and `total_items`, enabling the frontend to build dynamic pagination controls.

3.4.4 FASTA Upload Implementation

A dedicated endpoint `POST /upload_fasta` handles the upload and processing of FASTA files.

- The endpoint expects a file upload via `multipart/form-data`.
- It validates the presence of a file and its extension (`.fasta`, `.fa`, `.fna`, `.fas`).

- `BioPython`'s `SeqIO.parse()` function is utilized to efficiently parse the FASTA file content.
- For each record in the FASTA file, the `protein_id`, `name`, `description`, and `sequence` are extracted. To extract the UniProt Accession ID (e.g., `P12345` from `>sp|P12345|MY_PROTEIN_1`)
- Before insertion, the database is checked for existing proteins with the same ID to prevent duplicates. Existing entries are skipped and reported.
- New protein records are added to the database.
- The endpoint returns a list including the count of added and skipped proteins.

3.4.5 Server Configuration and Execution

The backend server is configured via `config.py` and managed using Flask's[4] command-line interface.

- **Configuration (`config.py`):**
  - The `Config` class defines application-wide settings, primarily the `SQLALCHEMY_DATABASE_URI`[6]. This URI specifies the connection string to the PostgreSQL database[3], including credentials and host.
  - `SQLALCHEMY_TRACK_MODIFICATIONS` is set to `False` to conserve resources.
- **Execution Commands:**
  1. Clone into the Git-Repository:

     ```
     git clone https://github.com/studfaensen/scaling-octo-spork
     ```

  2. Navigate to the backend directory:

     ```
     cd data_warehouse_app/backend
     ```

  3. Activate the Python virtual environment:

     ```
     source venv/bin/activate
     ```

  4. Set the `FLASK_APP` environment variable to point to the main application file:

     ```
     export FLASK_APP=app.py
     ```

  5. Run database migrations (if schema changes occurred or for initial setup):

     ```
     flask db init
     flask db migrate -m "Initial database setup"
     flask db upgrade
     ```

6. Start the Flask development server:

   ```
   flask run
   ```

   The server will typically be accessible at `http://127.0.0.1:5000/`.

7. To interact with the backend the frontend can be started with: (this automatically opens a new tab in the standard browser)

   ```
   cd ../frontend/
   npm start
   ```

4 Database Integration

The application's data is built upon a PostgreSQL database, managed through an **Object-Relational Mapping (ORM)**. Specifically, **Flask-SQLAlchemy** was chosen as the integration tool using the SQLAlchemy library.

4.1 Database Schema

The database schema is designed to represent biological data, particularly focusing on proteins. The schema comprises three main tables: `organisms`, `genes`, and `proteins`.

- `organisms` **Table:**
  - `id`: Integer, Primary Key. Auto-incrementing identifier for each organism.
  - `name`: String (255 characters), Unique, Not Null. The common name of the organism (e.g., "Homo sapiens").
  - `taxonomy_id`: String (50 characters), Unique. A unique identifier from a taxonomic database (e.g., NCBI Taxonomy ID "9606").
  - **Relationships:** Establishes a one-to-many relationship with the `genes` table, indicating that one organism can be associated with multiple genes.
- `genes` **Table:**
  - `id`: Integer, Primary Key. Auto-incrementing identifier for each gene.
  - `name`: String (255 characters), Unique, Not Null. The name of the gene (e.g., "TP53").
  - `description`: Text. A longer description of the gene's function or characteristics.
  - `organism_id`: Integer, Foreign Key referencing `organisms.id`. This links a gene to its originating organism.

  - **Relationships:** Forms a many-to-one relationship with the `organisms` table and a one-to-many relationship with the `proteins` table, meaning one gene can be associated with multiple proteins.

- `proteins` **Table:**
  - `id`: String (20 characters), Primary Key. This field stores the UniProt Accession ID, serving as the unique identifier for each protein (e.g., "P04637"). The length constraint is applied to adhere to common UniProt ID formats.
  - `name`: String (255 characters), Not Null. The common name or short identifier of the protein.
  - `description`: Text. A detailed description of the protein.
  - `sequence`: Text. The amino acid sequence of the protein.
  - `gene_id`: Integer, Foreign Key referencing `genes.id`. Links a protein to its associated gene. This field is nullable to accommodate proteins for which gene information may not be immediately available or parsed from the FASTA input.
  - `pdb_id`: String (10 characters), Nullable. This field stores the Protein Data Bank (PDB) ID, which is a unique identifier for experimentally determined 3D structures of biomolecules (e.g., "1CRN"). It allows for proteins without known structures.
  - **Relationships:** Establishes a many-to-one relationship with the `genes` table.

The schema is defined in `backend/models.py` using Flask-SQLAlchemy. This maps Python classes to database tables.

4.2 Why this Integration (ORM)

The decision to utilize an Object-Relational Mapping (ORM) approach, specifically Flask-SQLAlchemy, was made by a random number generator after looking up different approaches in the lectures. In hindsight, this proved to be a good decision as Flask was wonderful to work with.

- The ORM removes the neccessity to write raw SQL queries, so interaction with the database works with familiar python objects and mehtods.
- While PostgreSQL is currently used, using SQLAlchemy provides a quick and easy switch to another relational database system.(e.g., MySQL, SQLite) with minimal changes
- Using Object-Oriented Modeling the ORM gives an object-oriented representation of the database schema, mapping tables to Python classes and individual rows to instances of these classes. This is really intuitive.

4.3 Working with data

Data is handled via Flask-SQLAlchemy's session management.

- **Writing Data:**
  - To create new records (e.g., adding a new protein), an instance of the corresponding model class (`Protein`, `Gene`, `Organism`) is created with the desired attributes.
  - This object is then added to the database session using `db.session.add(object_instance)`.
  - Changes are made permanent in the database by committing the session: `db.session.commit()`.
  - In case of errors during the commit process (e.g., constraint violations), `db.session.rollback()` is used to revert any pending changes, ensuring data integrity.
- **Data Retrieval (Reading Data):**
  - Retrieving single records by primary key is performed using `Model.query.get(primary_key_value)`.
  - For fetching collections of records or applying filters, the `Model.query` object is used, allowing for method chaining (e.g., `Model.query.filter_by(...)`, `Model.query.order_by(...)`).
  - Complex queries involving relationships (e.g., searching proteins by gene or organism name) utilize `join()` or `outerjoin()` methods to combine data from multiple tables.
  - Pagination is implemented using the `paginate()` method on the query object. This just fetches subsets of data for display in the frontend.

5 Discussion and Conclusion

During the progress of writing the code I often spent a lot of time working on little unimportant issues that added nothing to the final result. Learning to circumvent those distractions and staying focused on the current task was an important step in this journey. Using the Flask library was a great add-on that abstracted a lot of the menial work away. Something to note is that my current implementation proves to be at least minorly scalable as it managed .fasta files with protein counts into the thousands, although there has been no rigorous testing.

5.1 Encountered Challenges

During the development process, several technical challenges were encountered, which provided valuable learning experiences.

(This is only a short overview, as the exhausting list of my troubles with full-stack development would be outside the scope of this report):

- **FASTA Parsing and Data Truncation:** The initial FASTA upload implementation resulted in a `StringDataRightTruncation` error. This was caused by attempting to insert overly long parsed IDs from the FASTA header into the database's `id` column, which had a character limit of 20. The solution involved refining the parsing logic in `app.py` to extract only the relevant UniProt Accession ID and truncate it if necessary, ensuring compliance with the schema's constraints.
- **Molstar 3D Viewer Integration and CDN Issues:** Integrating the Molstar 3D viewer proved challenging due to persistent "Molstar object not found" errors. This was initially attributed to incorrect CDN paths or timing issues with dynamic script loading. After several iterations of CDN path adjustments, I just used a workaround that creates an iframe that links to the current Mol* implementation.

A huge problem is that the current prototype does not include user authentication or authorization. This is a grave security issue for any production-grade application, as it allows unrestricted access to CRUD operations.

5.2 Improvements to be made

- Add proper authentication
- Actually implement the Mol* visualizer
- Refactor code as during the development process a lot of approaches were abandoned, which resulted in a messy codebase

Eigenständigkeitserklärung

Ich bestätige, dass die eingereichte Arbeit eine Originalarbeit ist und von mir ohne weitere Hilfe verfasst wurde. Die Arbeit wurde nicht geprüft, noch wurde sie widerrechtlich veröffentlicht. Die eingereichte elektronische Version ist die einzige eingereichte Version.

Unterschrift	Ort und Datum

Erklärung zu Eigentum und Urheberrecht

Ich erkläre hiermit mein Einverständnis, dass die Technische Hochschule Bingen diese Arbeit Studierenden und interessierten Dritten zur Einsichtnahme zur Verfügung stellen und unter Nennung meines Namens (Luka Faensen) veröffentlichen darf.

Unterschrift	Ort und Datum

References

[1] Asis Hallab. *DAWE - Data Warehouse SS25 - Exam Assignment*. https://olat.vcrp.de/auth/RepositoryEnt Interactive Resource. (Visited on 07/03/2025).

[2] *Mol* Viewer Documentation*. https://molstar.org/viewer-docs/. (Visited on 07/03/2025).

[3] *PostgreSQL 17.5 Documentation*. https://www.postgresql.org/docs/17/index.html. May 2025. (Visited on 07/03/2025).

[4] *Quickstart — Flask Documentation (3.1.x)*. https://flask.palletsprojects.com/en/stable/quickstart/. (Visited on 07/03/2025).

[5] *React Router Home*. https://reactrouter.com/home. (Visited on 07/03/2025).

[6] *SQLAlchemy Documentation — SQLAlchemy 2.0 Documentation*. https://docs.sqlalchemy.org/en/20/. (Visited on 07/03/2025).

[7] *Welcome to Alembic's Documentation! — Alembic 1.16.2 Documentation*. https://alembic.sqlalchemy.org/en/latest/. (Visited on 07/03/2025).

[8] *What Is REST?* https://restfulapi.net/. Apr. 2025. (Visited on 07/03/2025).