# Project Purpose and Context

The **MammysBread** repository implements a web application (a bakery’s online storefront) built with Python/Flask. It’s a public-facing site for a bakery (“Mammy’s Bread”) that lists products (breads) and has an administrative interface. The code uses Flask for the backend, Jinja2 templates for pages, and supports multiple languages (Armenian, English and Russian) via Flask-Babel. An administrative “sysadmin” module indicates functionality for staff (e.g. posting HR positions or managing products). Overall, the purpose is to present bakery products online (with pages like Home, Products, etc.) and allow administrators to manage content. This is a full-stack monolithic application rather than a microservices architecture.

# Technology Stack Overview

* **Backend:** Python 3 with the Flask web framework. Flask is a lightweight (“micro”) web framework that handles routing and templates[[2]](https://python-data-science.readthedocs.io/en/latest/flask.html#:~:text=16).
* **Internationalization:** Flask-Babel is used for i18n (the README explicitly mentions *“MULTILINGUAL SITE WITH FLASK BABEL”* and even shows pip install Flask-Babel). Translations are managed with Babel (a /translations folder with .po/.mo files).
* **Database:** MySQL is used as the relational database. The README includes instructions to restore a MySQL dump (Dump.sql) into a database using mysql -u root -p db < Dump.sql. A Python DB module (mmb\_db.py) wraps the MySQL connector.
* **Caching/Session Store:** Redis is installed and started (the README shows sudo apt install redis-server; sudo systemctl start redis-server), so Redis is used for caching storage.
* **Frontend:** HTML/CSS/JavaScript for the client. The repository languages breakdown shows about 51% HTML, 21% JS, 19% Python, 7% CSS[[5]](https://github.com/Misha818/mammysbread" \l ":~:text=Languages), indicating mostly server-rendered pages.
* **Other Tools:** The requirements.txt presumably lists Flask, Flask-Babel, PyMySQL or MySQL connector, etc. The code is Python-based, and standard tools (git, pip, etc.) are used.

# Backend Architecture

The backend is a monolithic Flask application. Key aspects include:

* **Application Setup (app.py):** This is the main entry point. It creates the Flask app, configures extensions (e.g. Babel for translations), and registers routes or blueprints.
* **Database Layer (mmb\_db.py):** This module encapsulates database access. It probably defines functions to get a database connection and perform queries (e.g. fetching product records). MySQL is used, so this file likely uses a MySQL connector library to connect(host, user, pass, database). Based on convention, it might have functions like get\_all\_products() or query\_db(...).
* **Product Logic (products.py):** This module probably functions related to displaying products.
* **Admin Logic (sysadmin.py):** This module handles administrative functions. It includes user authentication, product management (add/edit products), or even posting new HR positions (given a commit message about HR). It likely also uses the DB layer and renders special admin templates.
* **Core API/Routes:** While not a JSON API, the app’s “core routes” map URLs to functions. Examples include / (home), /products, /product/<id>, /contacts, plus admin routes like /stuff, /login etc. Such routes retrieve data and render it via Jinja2 templates.
* **Database Schema:** The MySQL dump defines tables for products, categories, users, etc.

Overall, the backend uses Flask routes to handle requests, uses mmb\_db.py for raw SQL queries to MySQL, and returns rendered HTML. Redis would be used under the hood to cache login attempts.

# Frontend Architecture

The frontend is server-rendered HTML with supporting CSS/JS:

* **Templates and Static Files:** Per Flask conventions, HTML templates are placed in the templates/ directory and static assets (CSS, JS, images) in static/[[8]](https://python-data-science.readthedocs.io/en/latest/flask.html#:~:text=16). For example, templates/base.html and templates/public.html define the common layout (header, footer, nav), and other pages (index.html, products.html, etc.) extend this base templates. The static folder contains an images/ subfolder and CSS/JS files.
* **Component Structure:** Given the repository languages, most UI components are static HTML pieces that include Jinja. A plausible hierarchy is that base.html and public.html define site-wide components (navigation bar, footer), and child templates inherit from it. Pages like products.html loop over product entries to display product cards. This might be organized with template blocks (e.g. {% block content %}{% endblock %} in base). A simplified component hierarchy might look like: Base layout → Home page template, Products list template, Product detail template, Admin templates.
* **State Management:** Most “state” is server-side. For example, the user’s language preference or login status is stored in the session. On the client side, there isn’t complex state; values are passed via requests or stored in hidden form fields.

In summary, the frontend consists of Jinja2 templates for each page type, static assets for styling and client scripts, and Flask-flows handle page navigation.

# Code Organization and Key Files

The code is organized into a few modules, mainly in the project root:

* **app.py**: Main application file that initializes the Flask app, configures Babel (and the database URI, session keys), and registers routes or blueprints. This is the entry point that runs the server.
* **mmb\_db.py**: Database helper module. Contains functions to connect to MySQL and execute queries (e.g. get\_all\_products(), get\_product(id), etc.). Abstracts SQL commands away from route functions.
* **products.py**: Contains business logic for products. Defines functions related to product listing, product details, search, etc. May be imported by app.py or used as a blueprint.
* **sysadmin.py**: Contains administrative logic. Define Flask routes for managing the site (login, add/edit products, etc.). This could be a separate blueprint or simply additional routes within app.py.
* **templates/**: Directory with HTML template files. Includes base.html, home.html, products.html, product\_detail.html, admin/ (for admin pages), etc. These define the UI structure.
* **static/**: Directory with static resources like images, CSS, and JavaScript. For example, CSS files for layout/theme, JS libraries, and product images (the README specifically deals with permissions for static/images).
* **translations/ and babel.cfg**: For i18n. Contains .po and .mo files (the README shows Babel extraction/compilation steps). babel.cfg configures how text is extracted from Python and templates.
* **Dump.sql**: A SQL dump file used to initialize the MySQL database schema and populate initial data[[3]](https://github.com/Misha818/mammysbread#:~:text=Restore%20the%20database%20from%20the,p%20db%20%3C%20Dump.sql). This would be run before launching the app.
* **Configuration & Env:** There is an env~ folder for defining key elements, and a .gitignore.

The code is modestly modularized: database access is in one module, application logic split between a couple of modules, and Flask handles routing. Key files to look at for understanding the app are app.py (entrypoint and overall routing) and mmb\_db.py (database layer).

# Data Flow (Frontend ↔ Backend)

The application follows a typical request/response flow:

1. **User Action:** The user’s browser sends an HTTP request (e.g. clicking on “Products” or submitting a form).
2. **Web Server / Flask:** The request is received by the web server which proxies it to the Flask application. The Flask router matches the URL to a view function.
3. **Backend Processing:** Inside that view, the code may perform business logic. Typically it will call the database module. For instance: items = mmb\_db.get\_all\_products().
4. **Database Query:** The mmb\_db module connects to MySQL and executes a SQL query like SELECT \* FROM products WHERE language=.... MySQL returns the result set.
5. **Template Rendering:** The Flask view takes the returned data and passes it into a template: render\_template('products.html', products=items). Jinja2 fills in the HTML with the data (looping over products to display each one).
6. **Response to Client:** The server sends back the rendered HTML page. The browser receives it and displays it. If there was static content (CSS/JS/images), those are fetched as separate static requests (served directly by Nginx or Flask).
7. **Static Assets:** Any static images (like product photos) or CSS/JS files are served from the static/ directory. This keeps assets fast and separate from dynamic routes.

In short, the flow is: **Browser → Flask route → Database (MySQL) → Flask → Browser**.

*Data Flow Diagram:* Conceptually, the data flow can be visualized as:

[Browser] --HTTP Request--> [Flask App]  
 Flask App --SQL Query--> [MySQL DB]  
 [MySQL DB] --Query Results--> Flask App  
 Flask App --Render Template--> [Browser]

*(See* *Overall System Architecture Diagram* *below for an overview.)*

# System Architecture Diagram

*Figure: Overall system architecture. The Flask app handles HTTP requests, queries the MySQL database for product data, then returns rendered HTML to the browser. Static assets are served from the static/ folder.*

# Data Flow Diagram

*Figure: Data flow between frontend and backend. A client request (e.g. “GET /products”) hits Flask, which calls the database module. MySQL returns product data, and Flask renders it into an HTML response for the browser.*

# Component/Template Hierarchy

*Figure: Component hierarchy (template inheritance). The base layout (base.html) provides common navigation and style. Individual pages like index.html, products.html, and admin\_panel.html extend this base template to fill in page-specific content.*