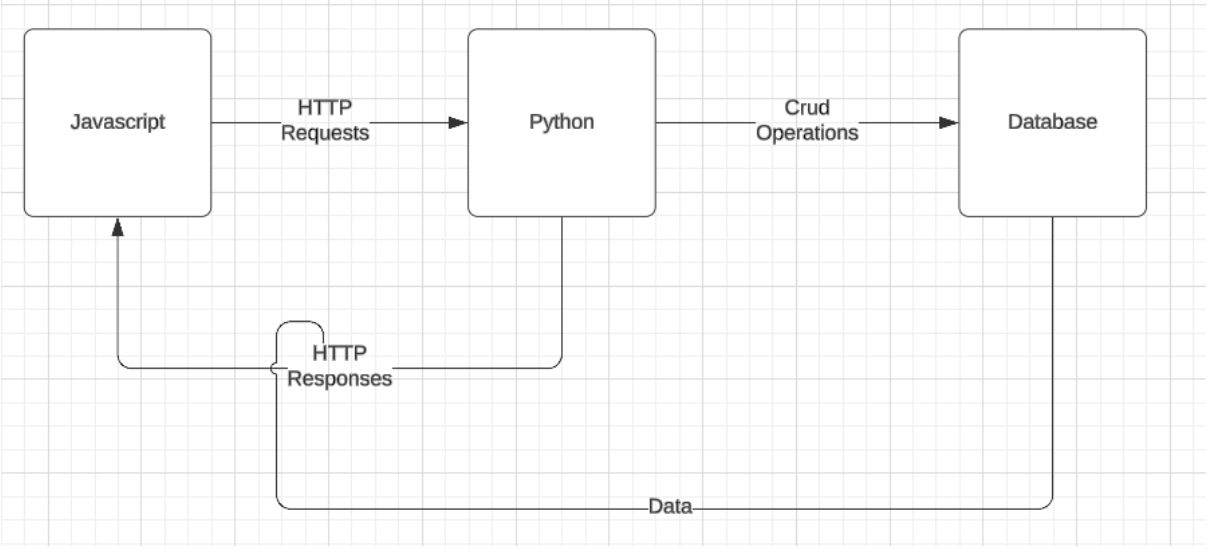
Tech Stack

- Backend:
    - Framework: Flask or Django (Python)
        - Flask: A lightweight and flexible micro-framework for building web applications in Python. It's easy to get started with and is suitable for small to medium-sized projects.
        - Django: A high-level Python web framework that encourages rapid development and clean, pragmatic design. It includes built-in features like authentication, ORM (Object-Relational Mapping), and admin panel.
- Frontend:
    - Framework/Libraries: Javascript
        - React.js: A JavaScript library for building user interfaces, developed by Facebook. It's component-based, efficient, and widely used for building single-page applications (SPAs).
        - Vue.js: A progressive JavaScript framework for building UIs, known for its simplicity and flexibility. It's great for creating interactive web interfaces and SPAs.
- Database:
    - SQL Database: PostgreSQL or SQLite
        - PostgreSQL: A powerful, open-source relational database system known for its reliability, robustness, and support for advanced features like JSONB data type and full-text search.
        - SQLite: A lightweight, embedded SQL database engine that's easy to set up and suitable for development and small-scale deployments.

Software Architecture

- Frontend:
    - The frontend of the application will be developed using JavaScript frameworks like React.js or Vue.js.
    - It will consist of various components responsible for rendering the user interface, handling user interactions, and making requests to the backend APIs.
    - The frontend will communicate with the backend through HTTP requests, typically using AJAX or fetch API to send requests and receive responses asynchronously.
    - Components such as forms, buttons, and input fields will handle user input and display data retrieved from the backend.
- Backend:
    - The backend of the application will be built using Python frameworks like Flask or Django.

    - It will handle incoming requests from the frontend, process data, interact with the database, and return responses to the frontend.
    - The backend will expose RESTful APIs that the frontend can consume to perform CRUD (Create, Read, Update, Delete) operations on data.
    - Authentication and authorization mechanisms will be implemented to secure access to protected resources and endpoints.
    - Business logic and application-specific functionality will be implemented in the backend to ensure data consistency and enforce business rules.
- Database:
    - The database will store and manage the application's data.
    - A relational database management system (RDBMS) like PostgreSQL or SQLite will be used to create and manage the database schema.
    - The backend will interact with the database using ORM (Object-Relational Mapping) libraries provided by the chosen Python framework (e.g., Flask-SQLAlchemy for Flask or Django ORM for Django).
    - Tables and relationships will be defined in the database schema to organize and store data efficiently.
    - The database will support CRUD operations as well as complex queries required by the application.



Use-Cases

Certainly! Here are some example use cases for the web application described in the architecture:

1. User Registration:
   - Actor: New User
   - Description: A new user wants to register an account on the platform.
   - Steps:

1. The user navigates to the registration page on the frontend.
2. The user fills out the registration form, providing their username, email, and password.
3. The frontend sends a POST request to the backend's registration endpoint.
4. The backend validates the user's input, creates a new user account, and stores it in the database.
5. The backend returns a success message to the frontend, indicating that the registration was successful.
6. The user is redirected to the login page to access their newly created account.

2. User Login:
-Actor: Registered User
- Description: A registered user wants to log in to their account.
- Steps:
1. The user navigates to the login page on the frontend.
2. The user enters their username/email and password.
3. The frontend sends a POST request to the backend's login endpoint with the user's credentials.
4. The backend verifies the credentials, generates a JWT (JSON Web Token), and returns it to the frontend.
5. The frontend stores the JWT in local storage or session storage for future authenticated requests.
6. The user is redirected to the home page or dashboard, indicating a successful login.

3. Scanning Process:
- Actor: Authenticated User
- Description: A user wants to scan an item using the application's scanning feature.
- Steps:
1. The user navigates to the scanning screen on the frontend.
2. The user clicks on the "Scan" button, triggering the scanning process.
3. The frontend accesses the device's camera and captures an image of the item.
4. The frontend sends the captured image to the backend's scanning endpoint via a POST request.
5. The backend receives the image, processes it using image recognition algorithms, and identifies the item.
6. The backend retrieves relevant information about the item from the database.
7. The backend sends the item information back to the frontend.
8. The frontend displays the results of the scanning process to the user.

4. Updating User Profile:
- Actor: Authenticated User
- Description: A user wants to update their profile information.
- Steps:
1. The user navigates to the profile settings page on the frontend.
2. The user modifies their profile information (e.g., name, email, password).
3. The frontend sends a PUT request to the backend's profile update endpoint with the updated information.
4. The backend verifies and updates the user's profile information in the database.
5. The backend returns a success message to the frontend.

6. The frontend updates the user's profile information displayed on the UI.