Acme Food Bank Inventory Tracking System

Solution Approach

Acme Corporation



760bf1b0-aa9c-4c1e-acfb-06eb2cac77ca**Team 9**

Dante Meyer

Yaru Gao

May 24, 2024

**TABLE OF CONTENTS**

[I. Introduction 3](#_Toc82521031)

[II. System Overview 3](#_Toc82521032)

[III. Architecture Design 3](#_Toc82521033)

[III.1. Overview 3](#_Toc82521034)

[III.2. Subsystem Decomposition 3](#_Toc82521035)

[I.1.1. [Subsystem Name] 4](#_Toc82521036)

[a) Description 4](#_Toc82521037)

[b) Concepts and Algorithms Generated 4](#_Toc82521038)

[c) Interface Description 4](#_Toc82521039)

[I.1.2. [Include sections III.2, III.3, etc., for other subsystems] 4](#_Toc82521040)

[IV. Data design 4](#_Toc82521041)

[V. User Interface Design 4](#_Toc82521042)

[VI. Glossary 5](#_Toc82521043)

[VII. References 5](#_Toc82521044)

[VIII. Appendix A - UI Mockups 5](#_Toc82521045)

# Introduction

The Acme Food Bank faces significant challenges in managing frequent and varied donations of food and hygiene products. Manual data management and entry are difficult due to the volume of the product entries. This process is less efficient, not always timely, and prone to errors, requiring significant effort to ensure accurate tracking of quantities and expiration dates.

Our project aims to develop a reliable inventory tracking system to address these challenges. By utilizing a combination of Python, Flask, HTML, and SQLite, we will create a web application that allows users to easily interact with a centralized database. The system will facilitate real-time updates, provide secure user authentication, and support other features that can enhance the operational efficiency of the Acme Food Bank. This document provides an overview of the system of our project and its design of archetecture, data, and user interface.

# System Overview

The Acme Food Bank Inventory Tracking System is designed to provide a comprehensive solution for managing inventory operations. For architecture design, the system utilizes a client-server model with a Flask-based server handling HTTP requests and interactions with a SQLite database. For data design, the SQL database stores various types of data, including inventory and donation information, user account data, and transaction records. For UI design, the system features a web-based graphical interface that allows users to perform tasks such as updating inventory, managing donations, handling transactions, and generating reports. The subsequent sections will delve into the detailed architecture design, data storage, and user interface components of the system.

# Architecture Design

## Overview

Our program uses a client/server architectural pattern [1], in which the user’s browser communicates with a flask server via HTTP requests over the network, functioning as a client. The server pulls data from static folders, templates, and databases in order to package the data and return it via the network to the client’s browser, usually in the form of an HTML page or other file.

The client’s responsibility is to provide a graphical interface and display files for the user, in addition to forms and scripts that allow them to easily send data packaged as an HTML GET/POST request to the server. The server’s responsibility is to process said HTML requests via Flask routes, perform necessary database/file operations if asked to do so, and provide feedback to the user by routing them to the requested page, or commit the intended changes to the database.

The client-server architecture was chosen because it is the most simple and intuitive way to configure a web app, as that is the structure most HTTP communication revolves around. The project involves giving many users the ability to modify a single centralized data source, which a server is perfectly suited for handling, and browsers naturally function as great clients which also are capable of sending HTTP requests to servers over the network. Flask itself also is excellent at facilitating multiple users at the same time by being able to modify web page output via routing and templating, and an SQL database is capable of providing secure login data storage, cleanly fulfilling all the project requirements. Another benefit of the client-server architecture is that any modifications to the server require absolutely no modifications on the client side, as the program has nothing to do with it other than sending it information.

Below is a diagram of the system’s architecture:

## Subsystem Decomposition

### [ Database system ]

#### Description

Responsible for communicating with/generating the database, returning query results and making modifications.

#### Concepts and Algorithms Generated

Contained within models.py, the system can be initialized within the base application script to provide a connection under the application context via a software factory pattern, allowing for other submodules to interact with it afterwards.

Flask-SQLAlchemy also provides an ORM (Object-Relational-Mapping) to the database itself, allowing developers to interact with the database as a streamlined Python object.

#### Interface Description

Services Provided:

1. Service name: Data querying  
   Service provided to: **[ Main server scripts ]**  
   Description: Queries the database based on an SQL statement or general classifier and returns a list of elements for use by the server itself or another subsystem.

Services Required:

1. Service name: Flask app context  
   Provided by: **[ Main server ]**

### [ Templating system ]

#### Description

Using Jinja HTML templates, the templating system receives data from the Flask server and compiles full HTML files and returns them to the user. This prevents having to manually format and return HTML from within the Flask code itself, instead relying on an external mutable and readable file. The template compilation functionality is built into Flask itself via the *render\_template* module.

#### Concepts and Algorithms Generated

HTML templates provide an additional level of modularity, as Flask has the ability to make templates reuse each other in order to avoid reusing HTML code, or to procedurally generate elements via loops and conditionals embedded within templates.

#### Interface Description

Services Provided:

1. Service name: HTML Rendering  
   Service provided to: **[ Main server ]**  
   Description: Receives a template request and returns a fully compiled HTML file built using Jinja markup, filled with data provided by the server.

Services Required:

1. Service name: Templating requests, supplementary data  
   Provided by: **[ Main server ]**

### [ Main server ]

#### Description

The primary Flask server backend script, responsible for integrating all subsystems and starting/running the flask server. Provides “routes” which are supplied to the user via web URLs, that perform different tasks depending on the route, using other subsystems.

#### Concepts and Algorithms Generated

The script uses a “software factory” pattern, assembling and initializing submodules stored in different scripts like the database module to integrate them with the app and provide a global context. Every route is also “ingested” into the server, and connected to Flask itself as it runs on the device. The script is also capable of connecting blueprints which contain further routes and procedures.

#### Interface Description

Services Provided:

1. Service name: App context  
   Service provided to: **[ Database system ]**  
   Description: Initializes the database ORM, providing it a context to run in.
2. Service name: Routes  
   Service provided to: **[ Database system, Templating system]**Description: Creates routes that deliver data from HTTP requests to template generators or the SQLAlchemy ORM.

Services Required:

1. Service name: Flask, Python environment  
   Provided by: **[ Python, Flask, other packages ]**

# Data design

For data design, our system employs a SQLite database to store and manage various data types. The primary data structures include tables for inventory data, donation records, user accounts, and transaction histories. The inventory table maintains details such as item names, quantities, and expiration dates. The donation records table logs information about donations received, including donor details and donation dates. The user accounts table stores authentication and profile information for users, while the transaction history table tracks all inventory movements and updates. Temporary data structures are used during processing to handle intermediate data states before committing changes to the database.

# User Interface Design

For user interface design, our system comprises six key sections: the main dashboard, a section for donation and donor information, an inventory management area, a report generation module, a recipient order management section, and a user account management area. These pages play various functions, including updating donations and inventory, processing transactions, and generating data reports. The interface features an interactive web layout with a navigation bar that allows users to quickly jump to the specific page they need, a search bar enabling users to look up specific inventory and donation information, and an account information section to be added later. Users can log in to their accounts stored in the database and perform different roles based on their permissions, such as making transactions as recipients, adding donations as donors, or managing inventories as food bank staff. This design ensures efficient navigation and utilization of the system's features for maintaining and managing inventory effectively.

(Veiw Appendix A - UI mockups to learn more about our UI design.)

# Glossary

Client-Server Architecture: A network architecture where a client (user interface) communicates with a server to request services or resources.

Flask: A lightweight web framework in Python used to build web applications.

SQLite: A C-language library that provides a lightweight, disk-based database, which doesn't require a separate server process.

HTTP (HyperText Transfer Protocol): The protocol used for transmitting hypertext requests and information on the internet.

ORM (Object-Relational Mapping): A programming technique for converting data between incompatible type systems using object-oriented programming languages.

SQL (Structured Query Language): A standard programming language for managing and manipulating databases.

Jinja: A templating engine for Python, used to generate HTML dynamically by combining templates with data sources.

HTML (HyperText Markup Language): The standard markup language used for creating web pages.

CSS (Cascading Style Sheets): A style sheet language used for describing the presentation of a document written in HTML.

JavaScript: A programming language commonly used to create interactive effects within web browsers.

GUI (Graphical User Interface): A user interface that includes graphical elements, such as windows, icons, and buttons, for user interaction.

Database Schema: The structure of a database described in a formal language, representing the organization of data as a blueprint of how the database is constructed.

HTTP Requests (GET/POST): Methods used by the client to request data from the server (GET) or send data to the server (POST).

Templating System: A system used to generate HTML dynamically using templates and data inputs.

Routes: Defined URL patterns in a web application that are used to handle requests and return the appropriate response.

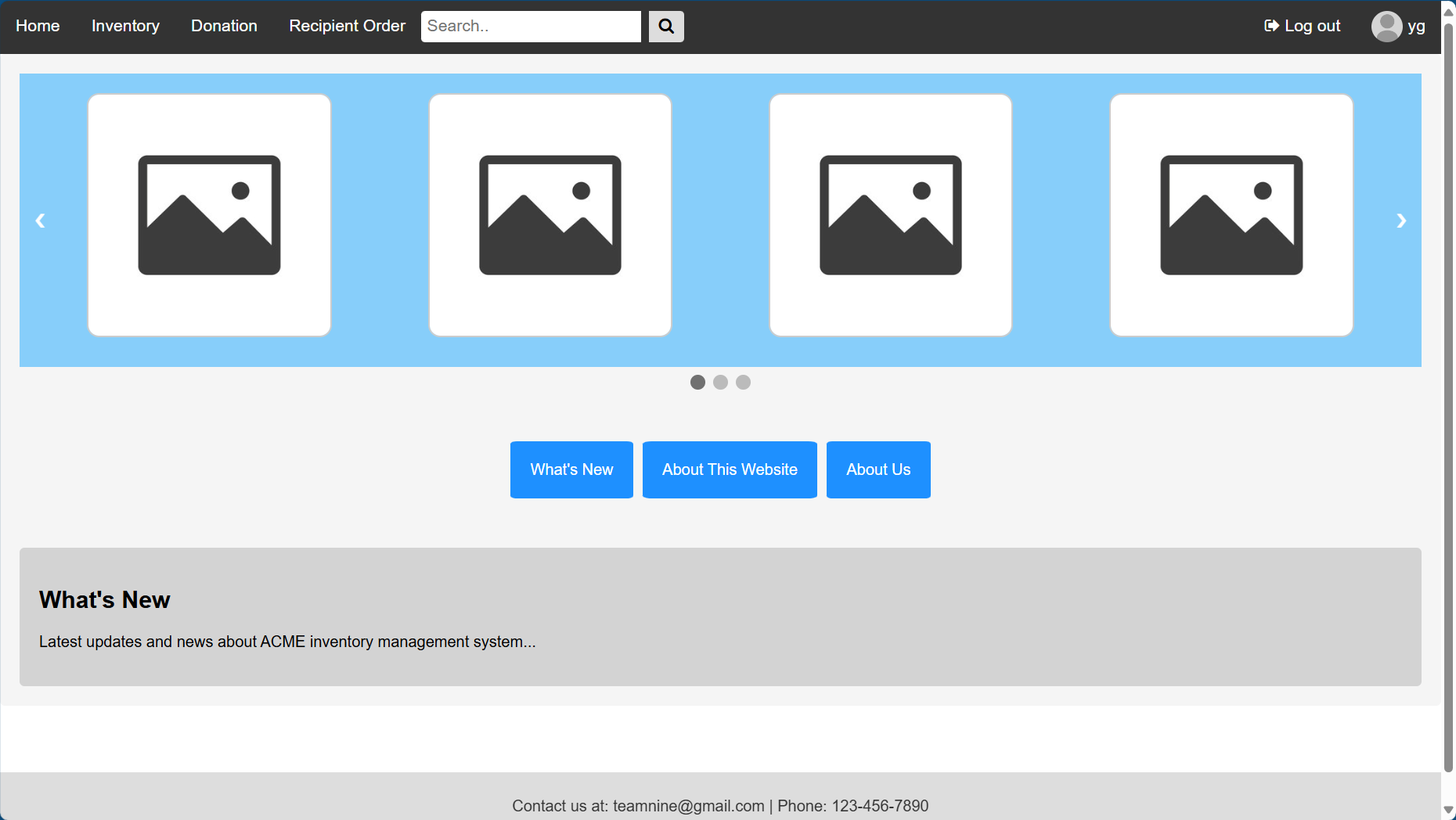
Static Files: Non-dynamic files such as CSS, JavaScript, and image files that are served directly to the user's browser.

# References

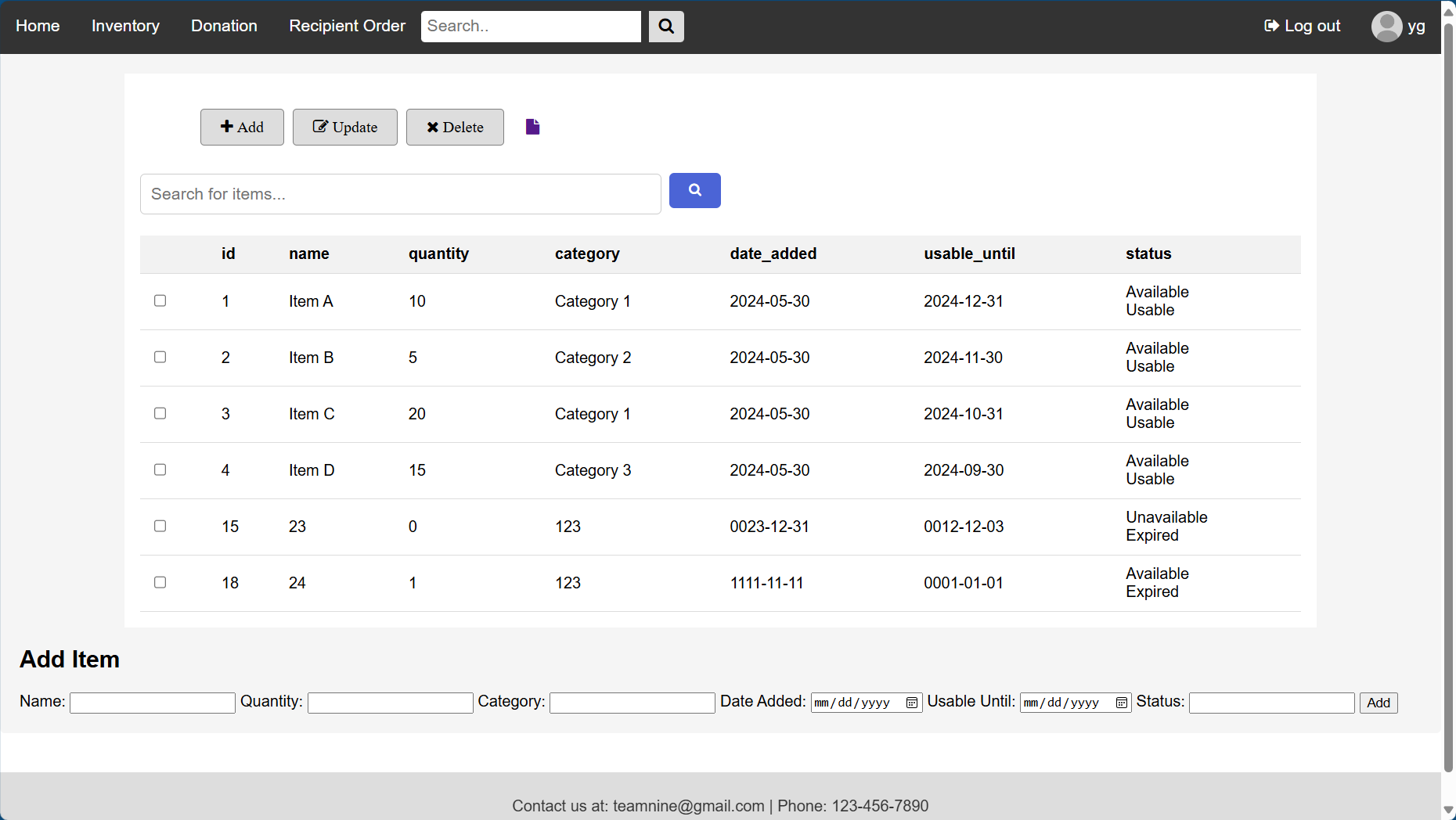
|  |  |
| --- | --- |
| [1] | A. S. Ritvik Gupta, "Software Architecture Patterns: What Are the Types and Which Is the Best One for Your Project," Turing Enterprises Inc., 15 October 2023. [Online]. Available: https://www.turing.com/blog/software-architecture-patterns-types/#Client-Server\_Pattern. [Accessed 29 May 2024]. |

# VIII. Appendx A - UI Mockups

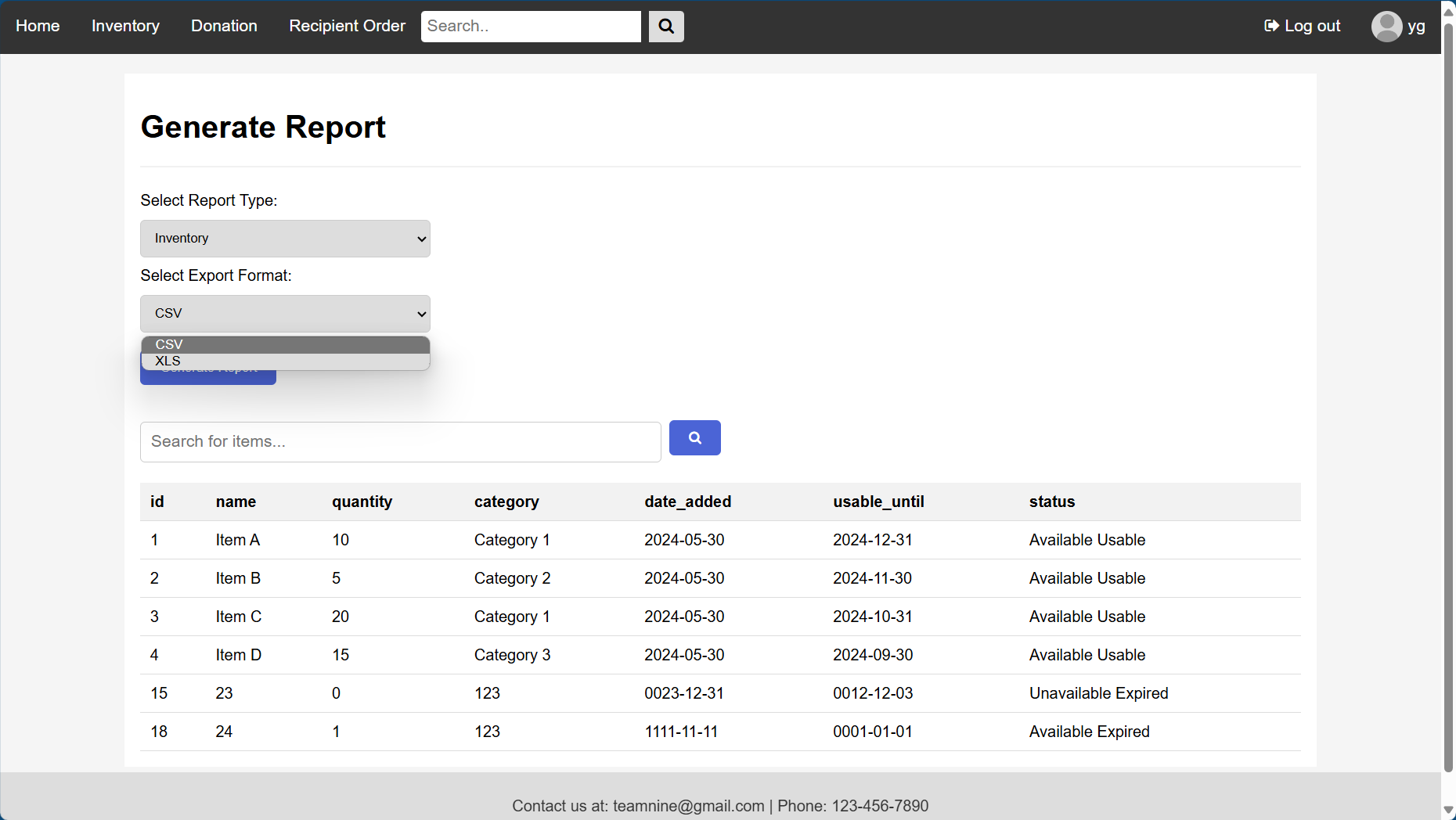
(Image 1: homepage)



(Image 2: inventory page)

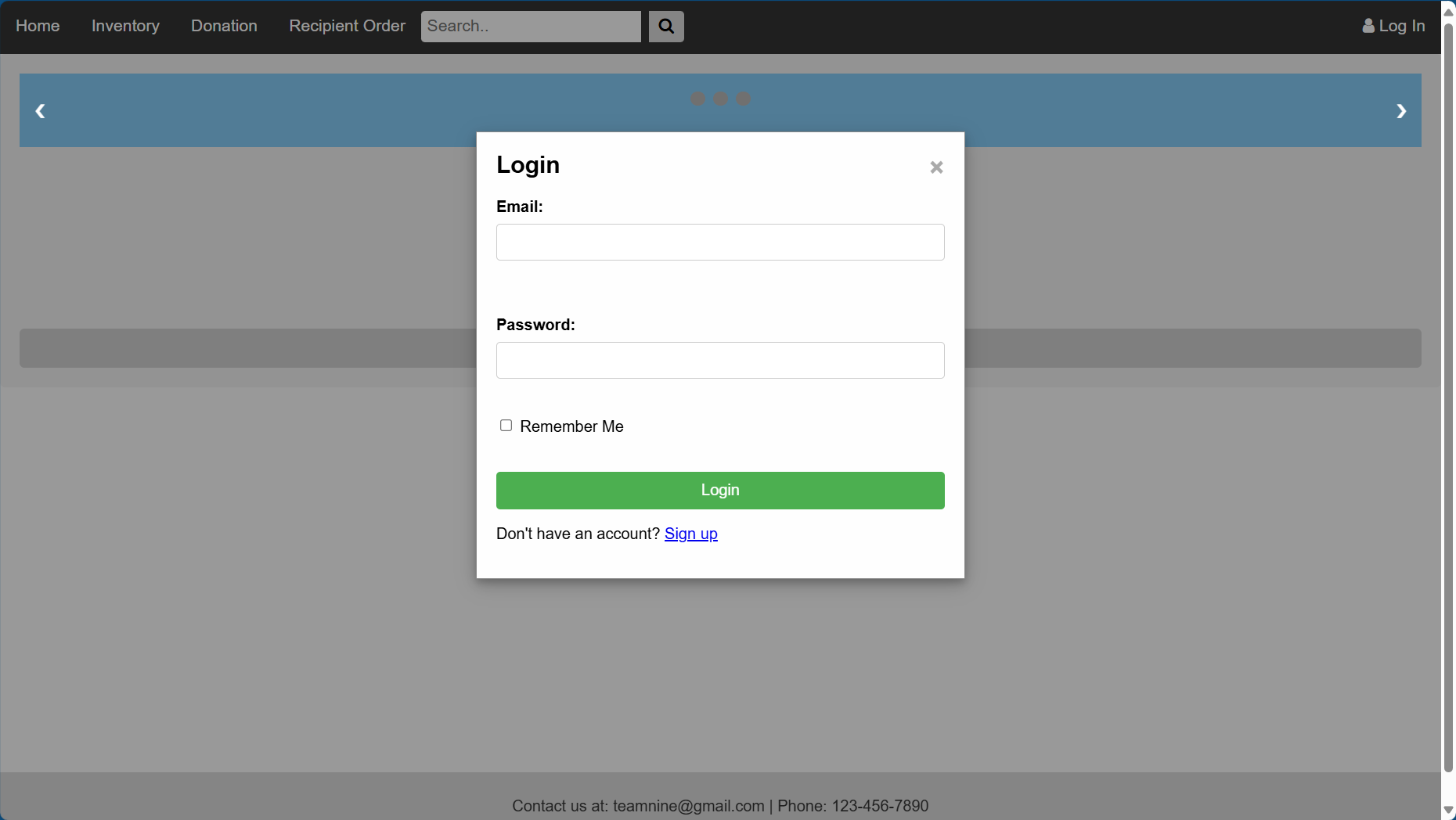


(Image 3: report generating page)

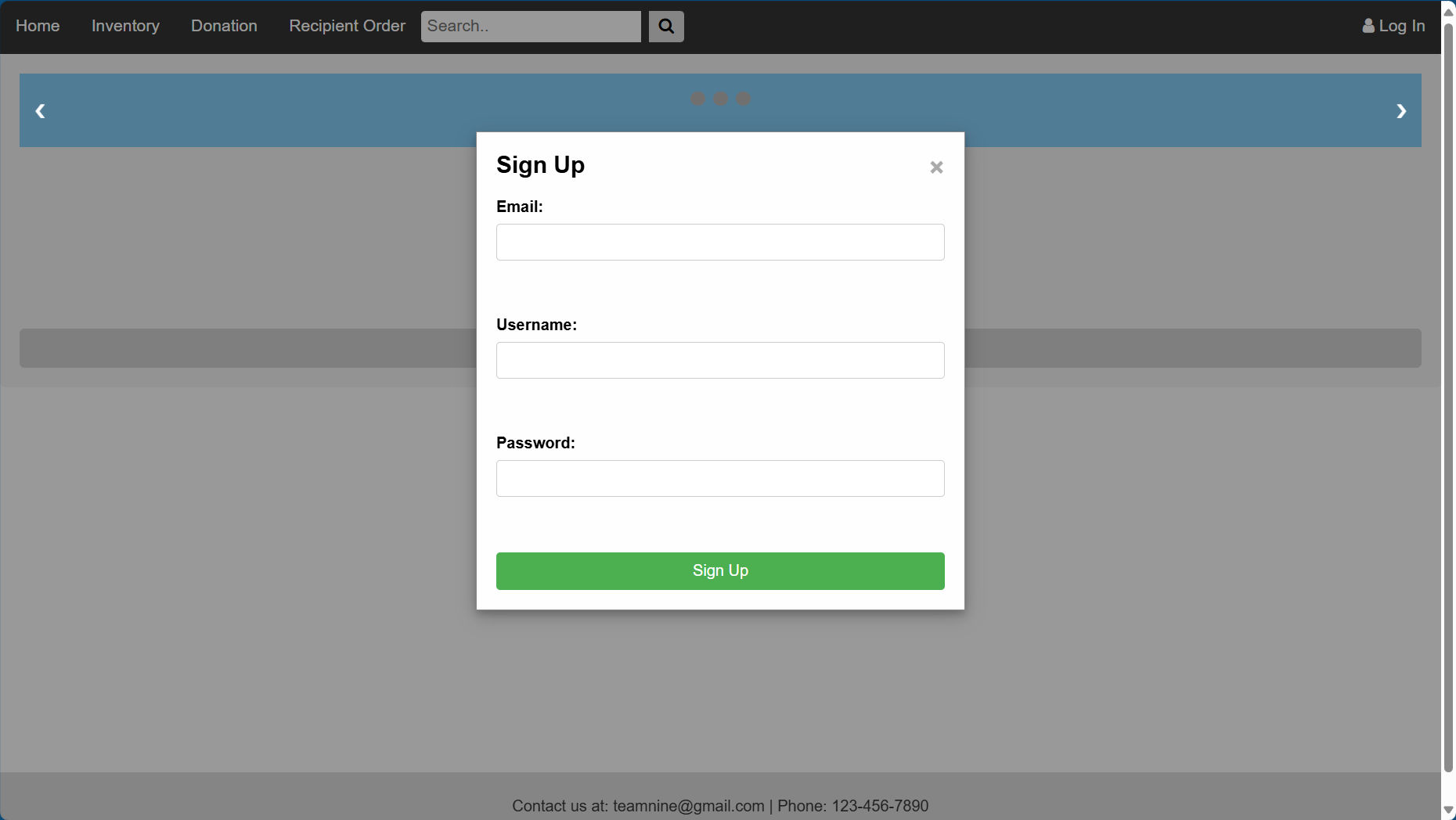


(Image 4: donation page)

(Image 5: log-in window)



(Image 6: Sign-up window)



(Image 7: User center)

