i-Wish

JAVA Application

Contributors:

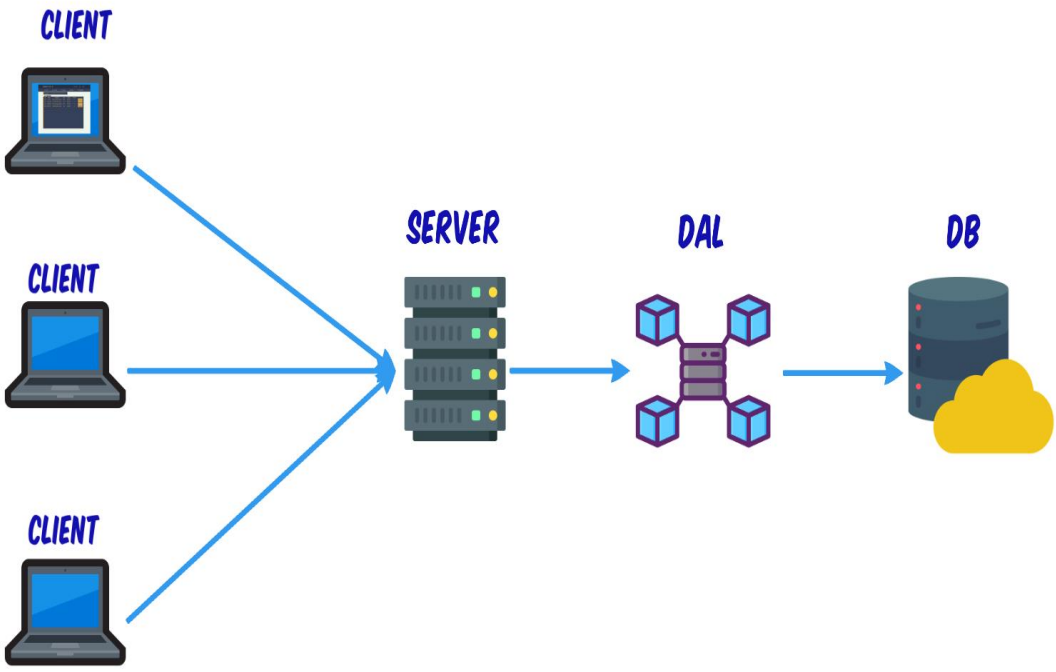
-Ahmed Jumaa
-Ahmed Sami
-Hussein Khaled
-Mariem Maged

i-Wish Application

Abstract:

i-Wish is a sophisticated desktop application designed to enhance user experience by facilitating the process of gifting items to friends. Users can add friends, create wish lists, view friends' wish lists, and contribute to buying items from friends' wish lists.

Application Architecture:



Client

- **Register/Sign-in:** Users can register for a new account or sign in to an existing account.
- **Add/Remove Friend:** Users can add or remove friends from their friends list.
- **Accept/Decline Friend Request:** Users can accept or decline friend requests from other users.
- **Create, Update, Delete My Wish List:** Users can manage their wish lists by creating, updating, or deleting items.
- **View My Friends List:** Users can view a list of their friends.
- **View My Friends Wish List:** Users can view the wish lists of their friends.
- **Contribute in Buying Items:** Users can contribute a specific amount of money towards buying items from a friend's wish list.
- **Receive Notification as Buyer:** Users receive a notification when a gift item's price is completed.
- **Receive Notification as Receiver:** Users receive a notification when an item from their wish list is bought by specific friends.
- **Friendly GUI:** The application provides a user-friendly interface to enhance user experience and enjoyment.

Server

- **Start/Stop:** The server can be started or stopped as needed.
- **Manipulate the Database:**
  - Establish a connection to the database.
  - Execute queries to retrieve or update data.
  - Add items to the database from where users can build their wish lists (via admin, database insertion, Amazon Web Services, or any RESTful web service or web crawling (Bonus)).
- **Handle Client Connections:** The server manages connections from clients.
- **Handle Client Requests:** The server processes requests from clients.
- **Handle Completion of Gift Item Shares:** The server informs participating clients when the shares of a gift item are completed and notifies the wish list owner that an item has been gifted to them by specific friends.

Notification Server:

- **Send Notifications:** The server sends notifications to users when specific events occur (e.g., completion of gift item shares, purchase of items from wish list).
- **Receive Notification Requests:** The server receives requests from other components of the system to send notifications to users.
- **Handle Notification Status:** The server tracks the status of notifications (e.g., sent, delivered, read) and updates them accordingly.

Database:

This database is designed to manage wish lists and contributions for a wishing application. It consists of the following tables:

1. **CONTRIBUTIONS**
    - Description: Stores information about contributions made by users towards items in the wish list.
    - Columns:
        - **CONTRIBUTOR_NAME** (VARCHAR(50)): Name of the user making the contribution.
        - **ITEM_ID** (VARCHAR(20), NOT NULL): Identifier of the item.
        - **CONTRIBUTION_AMOUNT** (INTEGER, NOT NULL): Amount contributed.
        - **WISHING_USER** (VARCHAR(50)): Name of the user who created the wish list item.
2. **FRIENDS**
    - Description: Stores friendships between users.
    - Columns:
        - **USER_NAME1** (VARCHAR(50)): Name of the first user.
        - **USER_NAME2** (VARCHAR(50)): Name of the second user.
3. **ITEMS**
    - Description: Stores information about items in the wish list.
    - Columns:
        - **ITEM_ID** (VARCHAR(50), NOT NULL): Unique identifier of the item.
        - **ITEM_NAME** (VARCHAR(50), NOT NULL): Name of the item.
        - **DESCRIPTION** (VARCHAR(50)): Description of the item.
        - **PRICE** (INTEGER): Price of the item.
        - **CATEGORY** (VARCHAR(50)): Category of the item.
4. **NOTIFICATION**
    - Description: Stores notifications for users.
    - Columns:
        - **USER_NAME** (VARCHAR(50), NOT NULL): Name of the user receiving the notification.
        - **MESSAGE** (VARCHAR(100)): Notification message.
        - **STATUS** (VARCHAR(50), NOT NULL): Status of the notification.

5. **REQUESTS**
    - Description: Stores friendship requests between users.
    - Columns:
        - **SENT_FROM** (VARCHAR(50)): Name of the user sending the request.
        - **SENT_TO** (VARCHAR(50)): Name of the user receiving the request.
        - **STATUS** (VARCHAR(20)): Status of the request.

6. **USERS**
    - Description: Stores information about users.
    - Columns:
        - **USER_NAME** (VARCHAR(50), NOT NULL): Unique username of the user.
        - **FULL_NAME** (VARCHAR(50), NOT NULL): Full name of the user.
        - **PHONE** (VARCHAR(50), NOT NULL): Phone number of the user.
        - **EMAIL** (VARCHAR(50), NOT NULL): Email address of the user.
        - **CREDIT_CARD** (VARCHAR(50), NOT NULL): Credit card information of the user.
        - **BALANCE** (INTEGER, NOT NULL): Current balance of the user.
        - **PASSWORD** (VARCHAR(50), NOT NULL): Password of the user.
7. **WISH_LIST**
    - Description: Stores the wish list items of users.
    - Columns:
        - **USER_NAME** (VARCHAR(50), NOT NULL): Name of the user.
        - **ITEM_ID** (VARCHAR(50), NOT NULL): Identifier of the item in the wish list.
        - **COLLECTED_AMOUNT** (INTEGER): Amount collected for the item.

Multi-Client Server Application:

Data Access Layer:

Overview: The Data Access Layer (DAL) is a key component of the application responsible for interacting with the database. It provides an abstraction layer that separates the application's business logic from the database operations, improving maintainability and scalability. This document provides an overview of the DAL, including its purpose, architecture, and key components.

Purpose: The primary purpose of the DAL is to facilitate the interaction between the application and the underlying database. It abstracts the complexity of database operations, such as querying, updating, and deleting data, allowing the application to focus on its core functionality. The DAL also helps improve performance by optimizing database queries and transactions.

Architecture:

The DAL is designed using a layered architecture, consisting of the following components:

1. **Data Access Objects (DAOs):** DAOs are responsible for encapsulating the logic for interacting with a specific database table or entity. Each DAO provides methods for querying, updating, and deleting data related to its corresponding entity.
2. **Connection Management:** The DAL manages the database connections, ensuring that connections are opened and closed efficiently to avoid resource leaks and improve performance.
3. **Exception Handling:** The DAL includes mechanisms for handling database-related exceptions, such as connection errors, query failures, and transaction rollbacks, to ensure robustness and reliability.
4. **Query Building:** The DAL provides utilities for building SQL queries dynamically based on user inputs or predefined criteria, enabling flexible and efficient data retrieval.
5. **Transaction Management:** The DAL supports transaction management, allowing multiple database operations to be grouped into a single transaction for consistency and atomicity.

Notification-Server Class**:**

- **Description:** This class represents a server that listens for incoming client connections and handles notifications to clients.
- **Attributes:**
  - **clients**: An **ArrayList** of **ClientHandler** instances representing connected clients.
- **Methods:**
  - **startServer()**: Starts the server by creating a **ServerSocket** and continuously accepting client connections. For each client, a new **ClientHandler** thread is created.
  - **notifyClients(String message)**: Sends a notification message to all connected clients by calling the **sendMessage** method of each **ClientHandler** instance in the **clients** list.

ClientHandler Class

- **Description:** This class represents a handler for individual client connections.
- **Attributes:**
  - **clientSocket**: The **Socket** representing the client connection.
  - **server**: The **NotificationServer** instance associated with this client handler.
  - **br**: A **BufferedReader** for reading input from the client.
  - **ps**: A **PrintStream** for sending output to the client.
- **Methods:**
  - **ClientHandler(Socket clientSocket, NotificationServer server)**: Constructor that initializes the **ClientHandler** with the client **Socket** and the associated **NotificationServer**.
  - **run()**: Implements the **Runnable** interface and defines the behavior of the client handler. It continuously listens for messages from the client and forwards them to the **NotificationServer** for broadcasting.
  - **sendMessage(String message)**: Sends a message to the client using the **PrintStream**.

Overall, the **NotificationServer** class manages the server socket and handles client connections, while the **ClientHandler** class manages individual client interactions, such as receiving messages from clients and sending messages to clients. Together, these classes enable the notification server to broadcast messages to all connected clients.

MultiClientServer class

The **MultiClientServer** class represents the server application of the i-Wish project. It is responsible for managing client connections, handling client requests, and coordinating communication between clients and the database. This class utilizes multithreading to handle multiple client connections simultaneously, ensuring efficient and responsive communication.
Main Features:

1. **Client Management:** The server maintains a list of connected clients (**clients**) and can add or remove clients as they connect or disconnect.
2. **Server Start/Stop:** The **startServer()** method starts the server by creating a **ServerSocket** and listening for client connections on port 5005. The server continues to run indefinitely, accepting new client connections.
3. **Client Handler:** The **ClientHandler** inner class implements the **Runnable** interface to handle individual client connections. Each **ClientHandler** instance manages a single client connection and is responsible for processing incoming requests from the client.
4. **Request Processing:** The **ClientHandler** class processes incoming requests from clients using a switch statement based on the request type. It deserializes JSON-formatted requests into appropriate DTO (Data Transfer Object) classes using the Gson library, performs the necessary database operations using the **DataAccessLayer** class, and sends responses back to clients.

```
switch (request_type){
    case "RegisterRequest":
        // Process registration request
        break;

    case "LoginRequest":
        // Process login request
        break;

    case "EditPassword":
        // Process edit password request
        break;

    // Other cases for handling different types of requests
}
```

5. **Database Interaction:** The server interacts with the database through the DataAccessLayer class, which encapsulates database operations such as user registration, login, profile management, friend management, wish list management, and notification management. These operations are performed using JDBC (Java Database Connectivity) to ensure secure and reliable database access.
6. **Notification Handling:** The server handles notifications by sending them to clients based on specific events, such as completion of gift item shares or purchase of items from wish lists. It uses the Gson library to serialize notification objects into JSON format and sends them to clients via the **PrintStream** associated with each client's socket.
7. **Thread Safety:** The server ensures thread safety by synchronizing access to shared resources, such as the list of connected clients (**clients**), to prevent concurrent modification issues.

Main Class

- **Description:** This class contains the main method to start the server components.
- **Attributes:**
  - **MCS**: An instance of **MultiClientServer** for handling client connections.
  - **NS**: An instance of **NotificationServer** for handling notifications to clients.
- **Methods:**
  - **main(String[] args)**: The entry point of the server application. It creates instances of **MultiClientServer** and **NotificationServer**, and starts them in separate threads using lambda expressions.

Overall, the **Main** class serves as the entry point for the server application, initializing and starting the server components necessary for handling client connections and notifications.

Client's Application:

Overview

The client application is designed to provide a user-friendly interface for interacting with the system. It consists of several themes, each represented by an FXML scene and controlled by a corresponding controller class. The application communicates with the server using socket communication, sending requests based on user interactions.

Themes

1. **AddFriend**: Allows users to add new friends to their friend list.
2. **Help**: Provides assistance and guidance on using the application.
3. **HomePage**: Displays the main page of the application with various options and features.
4. **Login**: Handles user authentication and login functionality.
5. **Market**: Displays products available for purchase.
6. **MyFriends**: Shows the user's list of friends and provides options to interact with them.
7. **MyProfile**: Allows users to view and update their profile information.
8. **Notification**: Displays notifications received from the server.
9. **Register**: Handles user registration and account creation.
10. **iWishQR**: Displays a QR code for the iWish feature.
11. **myWishList**: Shows the user's wishlist and allows them to manage it.

Functionality

- Each FXML scene is associated with a controller class that controls the behavior of the scene.
- Controllers use listeners to capture user interactions and send requests to the server.
- The server processes requests using a switch case, performing database operations as necessary.
- Responses from the server are displayed to the user or used to update the application state.

Communication

- The client application communicates with the server using socket communication.
- Requests are sent to the server, which processes them and returns a response.
- Responses are handled by the client to update the UI or display relevant information to the user.

Technologies Used

- JavaFX for the user interface.
- Socket communication for client-server interaction.
- FXML for defining the UI layout.
- Controller classes for handling user interactions.
- MySQL database for storing application data.

DTOs:

ItemDTO
Represents an item in the market.

- **Attributes:**
  - **itemId** (String): The unique identifier of the item.
  - **itemName** (String): The name of the item.
  - **description** (String): A brief description of the item.
  - **price** (int): The price of the item.
  - **collectedAmount** (int): The amount of money collected for the item.
  - **category** (String): The category of the item.
- **Constructors:**
  - **ItemDTO()**: Default constructor.
  - **ItemDTO(String itemId, String itemName, String description, int price, int collectedAmount, String category)**: Constructor to set all attributes.
- **Getters and Setters:**
  - Getter and setter for each attribute.

ItemListDTO
Represents a list of items.

- **Attributes:**
  - **requestType** (String): The type of request associated with the item list.
  - **items** (ArrayList<ItemDTO>): The list of items.
- **Constructors:**
  - **ItemListDTO()**: Default constructor.
  - **ItemListDTO(String requestType)**: Constructor to set the request type.
- **Getters and Setters:**
  - **getRequestType()**, **setRequestType(String requestType)**: Getter and setter for the request type.
  - **getItems()**, **setItems(ArrayList\<ItemDTO\> items)**: Getter and setter for the list of items.

NotificationDTO
Attributes:

- **requestType** (String): The type of request.
- **username** (String): The username of the user.
- **message** (String): The message of the notification.
- **status** (String): The status of the notification.

Constructors:

1. **NotificationDTO()**: Default constructor.
2. **NotificationDTO(message: String)**: Parameterized constructor to initialize **message**.
3. **NotificationDTO(requestType: String, username: String)**: Parameterized constructor to initialize **requestType** and **username**.

Methods:

- Getters and setters for all attributes.

MyProfileDTO
Represents the user's profile information.

- **Attributes:**
  - **requestType** (String): The type of request associated with the profile.
  - **name** (String): The user's name.
  - **username** (String): The user's username.
  - **email** (String): The user's email address.
  - **phone** (String): The user's phone number.
  - **credit** (String): The user's credit card information.
  - **balance** (int): The user's account balance.
  - **password** (String): The user's password.
  - **wishlist** (int): The number of items in the user's wishlist.
  - **friends** (int): The number of friends the user has.
- **Constructors:**
  - **MyProfileDTO(String requestType, String username, String password)**: Constructor to set the request type, username, and password.
  - **MyProfileDTO(String requestType, String username)**: Constructor to set the request type and username.
- **Getters and Setters:**
  - Getter and setter for each attribute.

MyWishlistDTO
Attributes:

- **requestType** (String): The type of request.
- **username** (String): The username of the user.
- **itemid** (String): The ID of the item.
- **itemname** (String): The name of the item.
- **description** (String): The description of the item.
- **category** (String): The category of the item.
- **price** (int): The price of the item.
- **collectedAmount** (int): The amount of the item collected.

Constructors:

1. **MyWishlistDTO(requestType: String, username: String)**: Parameterized constructor to initialize **requestType** and **username**.
2. **MyWishlistDTO(itemid: String, itemname: String, description: String, price: int, category: String, collectedAmount: int)**: Parameterized constructor to initialize all attributes.
3. **MyWishlistDTO(requestType: String, username: String, itemid: String)**: Parameterized constructor to initialize **requestType**, **username**, and **itemid**.

Methods:

- Getters and setters for all attributes.

FriendDTO
Attributes:

- **name** (String): The name of the friend.
- **wishlist** (ArrayList<ItemDTO>): The wishlist of the friend.
- **sentRequest** (boolean): Indicates if a friend request has been sent to this friend.

Constructors:

1. **FriendDTO(name: String, wishlist: ArrayList<ItemDTO>)**: Parameterized constructor to initialize **name** and **wishlist**.

Methods:

- Getters and setters for all attributes.

FriendListDTO
Attributes:
- **requestType** (String): The type of request.
- **userName** (String): The username of the user.
- **friends** (ArrayList<FriendDTO>): The list of friends.

Constructors:
1. **FriendListDTO(requestType: String, userName: String)**: Parameterized constructor to initialize **requestType** and **userName**.

Methods:
- **setFriends(friendNames: ArrayList<FriendDTO>)**: Sets the list of friends.
- **getFriends(): ArrayList<FriendDTO>**: Returns the list of friends.

These DTOs are used for data transfer between the client and the server in your application, representing various entities and their attributes.

FriendRequestDTO
Attributes:
- **sentFrom** (String): The username of the user who sent the friend request.
- **sentTo** (String): The username of the user to whom the friend request is sent.
- **status** (String): The status of the friend request.

Constructors:
1. **FriendRequestDTO(sentFrom: String, sentTo: String, status: String)**: Parameterized constructor to initialize all attributes.

Methods:
- Getters and setters for all attributes.

FriendRequestListDTO
Attributes:
- **UserName** (String): The username of the user.
- **requestType** (String): The type of request.
- **requests** (ArrayList<FriendRequestDTO>): The list of friend requests.

Constructors:
1. **FriendRequestListDTO(UserName: String, requestType: String, requests: ArrayList<FriendRequestDTO>)**: Parameterized constructor to initialize all attributes.

Methods:
- Getters and setters for all attributes.

LoginDTO
Attributes:

- **requestType** (String): The type of request.
- **username** (String): The username of the user.
- **password** (String): The password of the user.
- **status** (String): The status of the login attempt.
- **balance** (int): The balance of the user.
- **notiCount** (int): The number of notifications for the user.

Constructors:

1. **LoginDTO(requestType: String, username: String, password: String)**: Parameterized constructor to initialize **requestType**, **username**, and **password**.

Methods:

- Getters and setters for all attributes.

RegisterDTO
Attributes:

- **requestType** (String): The type of request.
- **username** (String): The username of the user.
- **fullname** (String): The full name of the user.
- **phone** (String): The phone number of the user.
- **email** (String): The email of the user.
- **creditcard** (String): The credit card information of the user.
- **balance** (int): The balance of the user.
- **password** (String): The password of the user.

Constructors:

1. **RegisterDTO(requestType: String, username: String, fullname: String, phone: String, email: String, creditcard: String, balance: int, password: String)**: Parameterized constructor to initialize all attributes.

Methods:

- Getters and setters for all attributes.

SearchFriendDTO
Attributes:

- **requestType** (String): The type of request.
- **userName** (String): The username of the user.
- **myname** (String): The username of the user performing the search.
- **usersList** (ArrayList<FriendDTO>): The list of users found in the search.

Constructors:

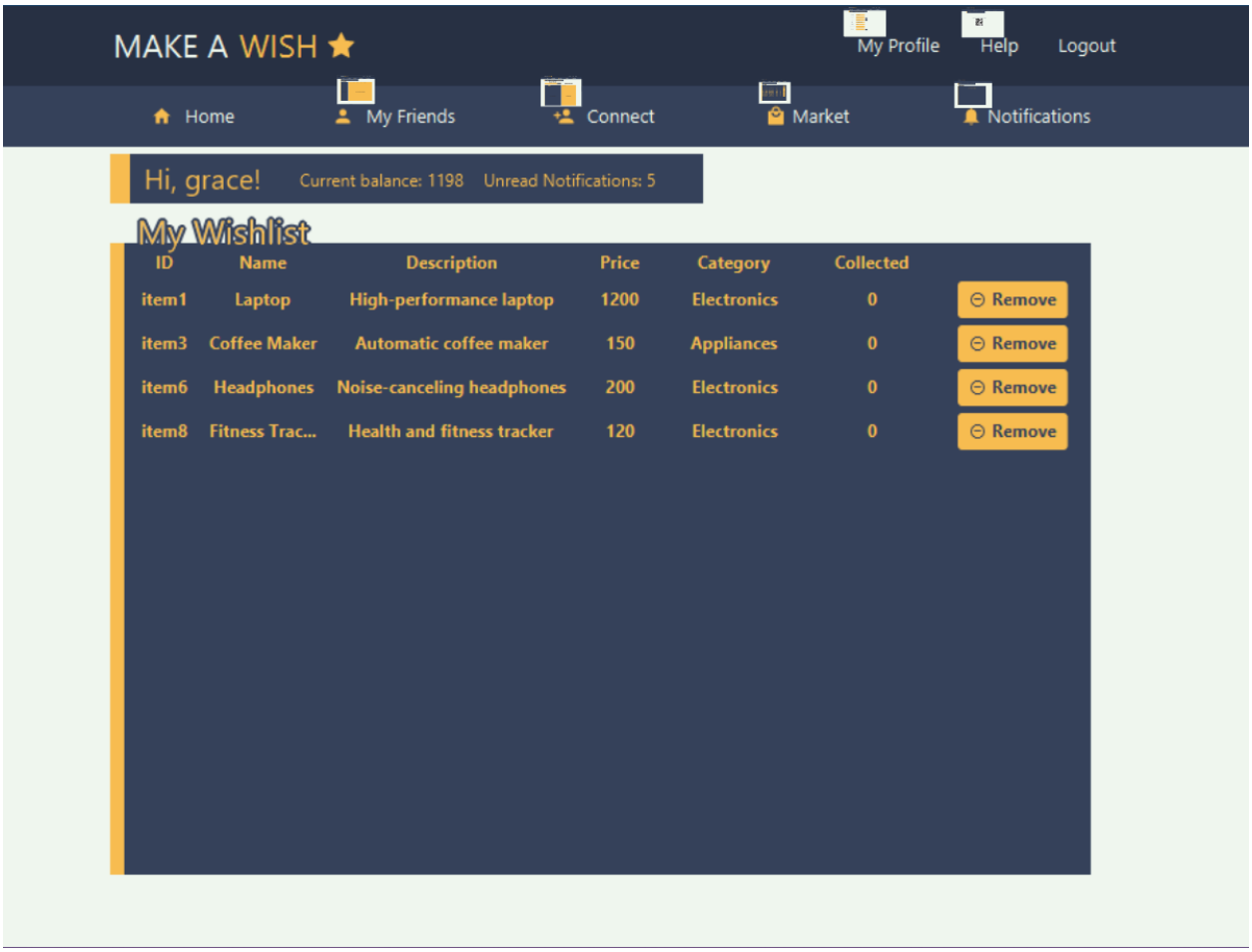
1. **SearchFriendDTO(requestType: String, userName: String, myname: String, usersList: ArrayList<FriendDTO>)**: Parameterized constructor to initialize all attributes.

Methods:

- Getters and setters for all attributes.

GUI:

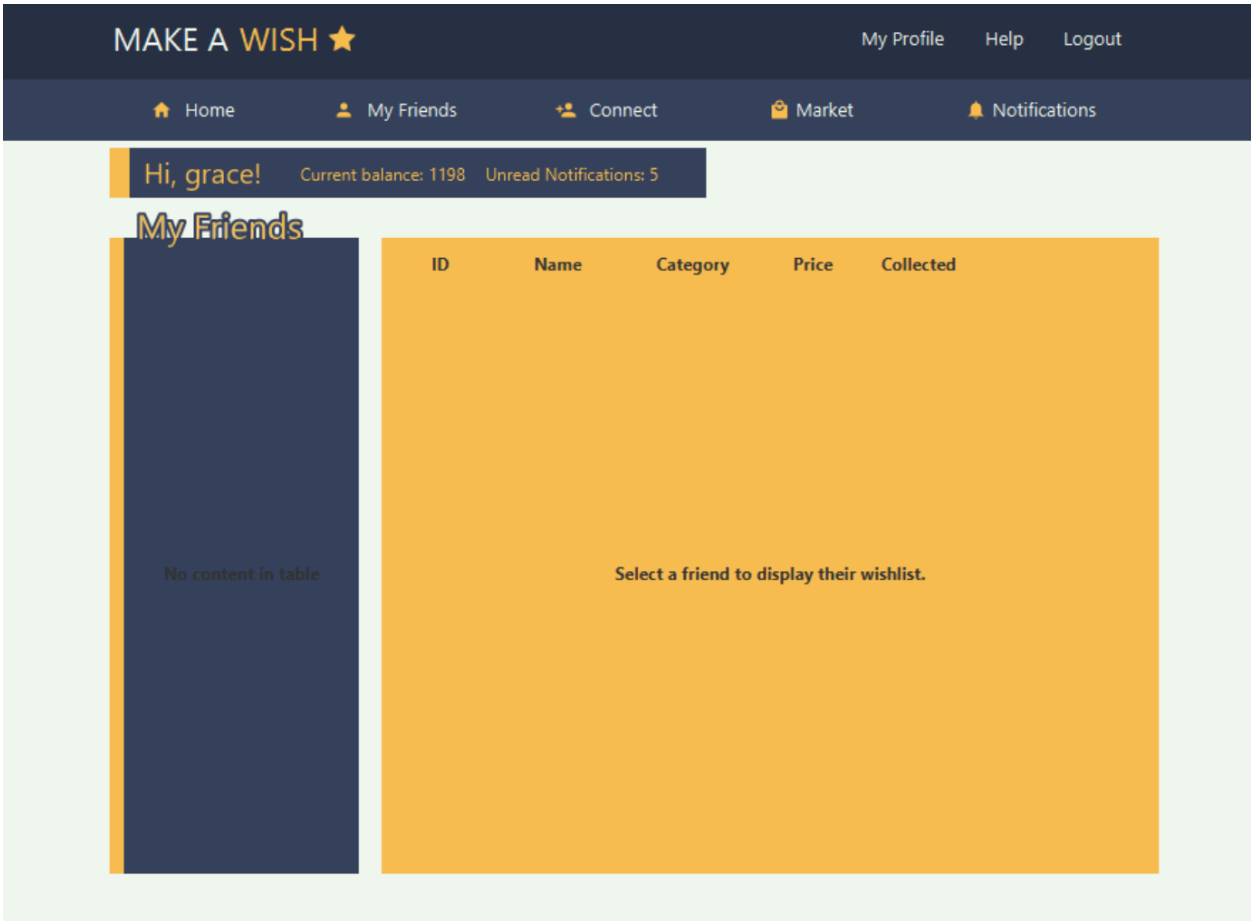
Home



Home Page
The Home Page is the first page the user sees after logging in. It features the user's wish list and navigation buttons for accessing other parts of the application.
Components:

- User's wish list
- Navigation buttons for Market, Friends, Connect, Notification, Logout, and Profile

Friends:



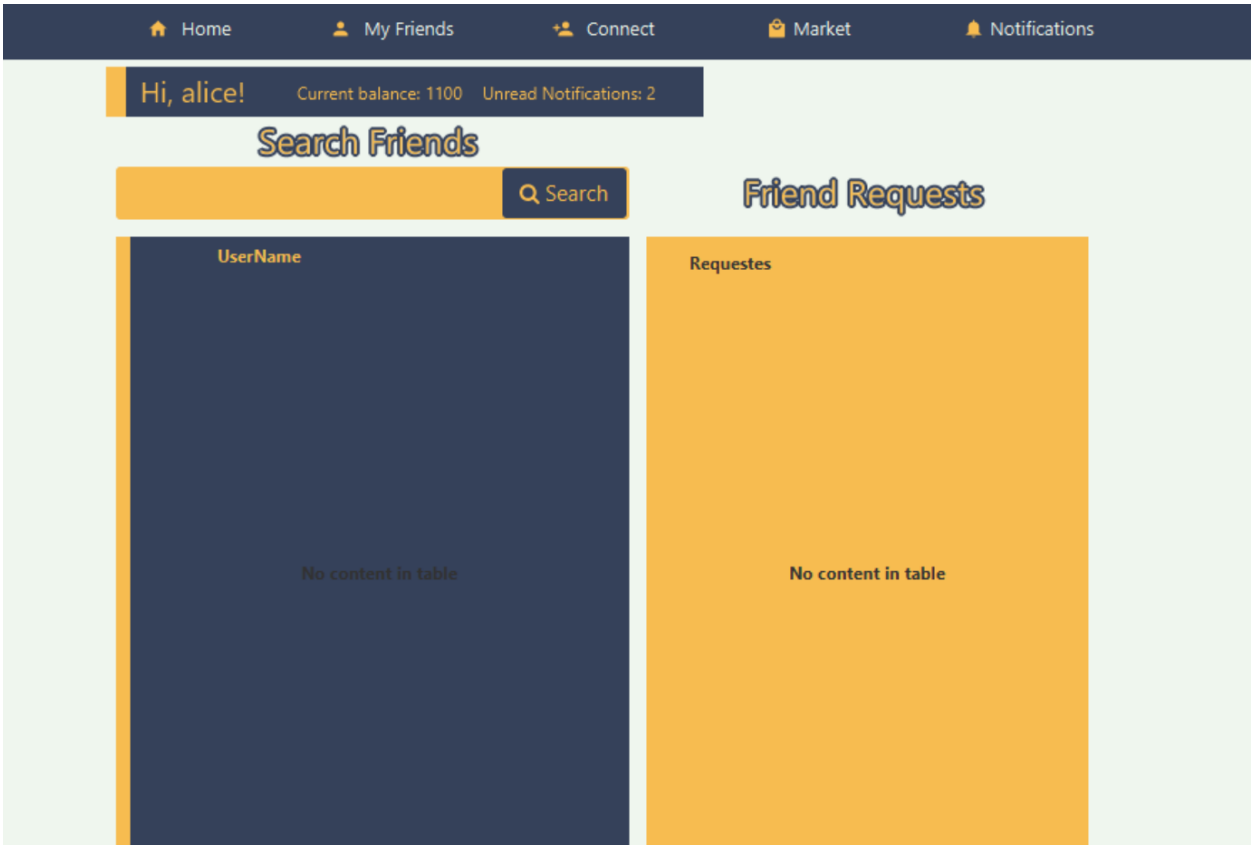
Friends

The Friends section displays the user's list of friends and allows the user to remove friends.

Components:

- List of friends
- Remove friend button for each friend

Search



Connect

The Connect section allows the user to search for friends, add new friends, accept friend requests, or deny friend requests.

Components:

- Search bar for finding friends
- Add friend button
- Accept and Deny buttons for friend requests

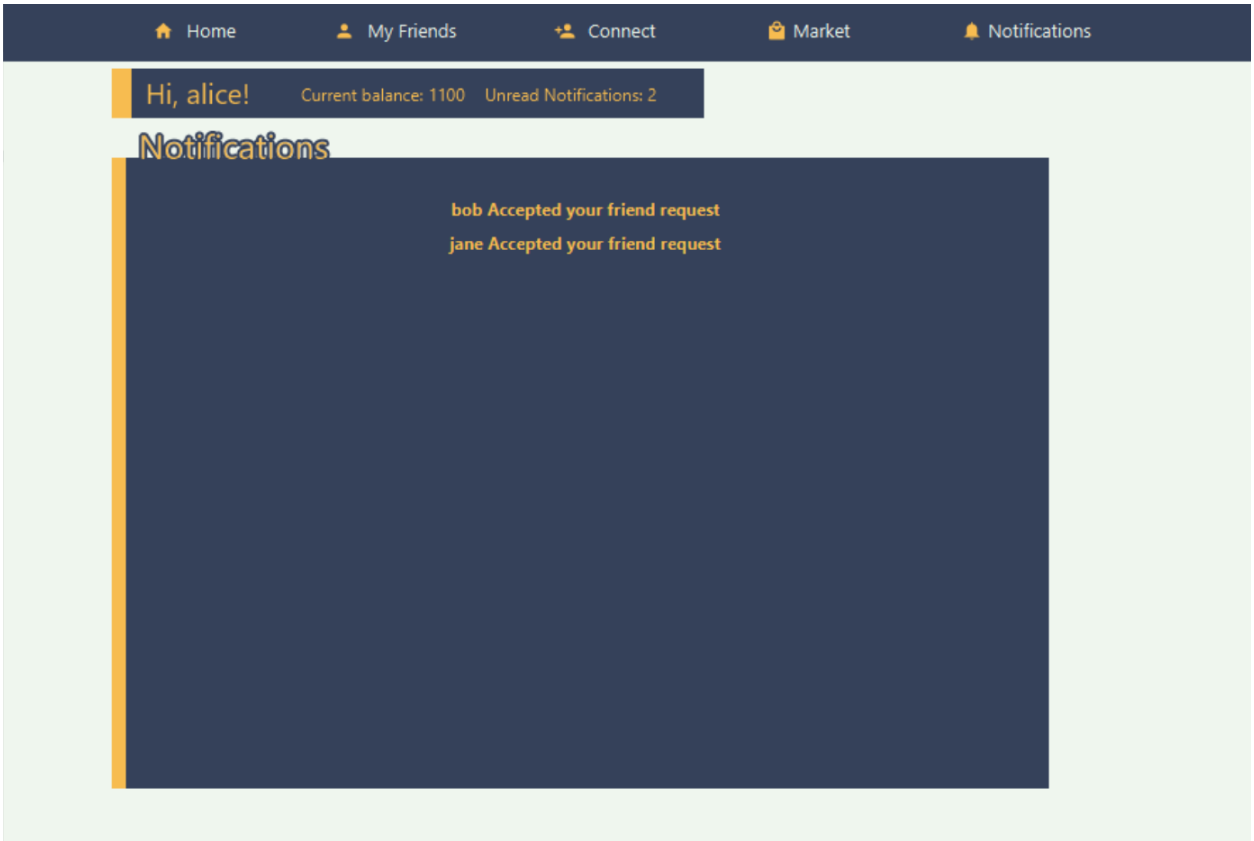
Market

The **Market** section contains items that the user can add to their wish list.

Components:

- List of items available for selection
- Add to wish list button for each item

Notifications:



Notification

The Notification section displays notifications that arise, such as friend requests or other important messages.

Components:

- List of notifications
- Read notification button for each notification