Warehouse Management System (WMS)

Database Systems

Section 201

Marist College
School of Computer Science and Mathematics

Submitted to:
Dr. Reza Sadeghi

September 15th, 2023

Table of Contents

Project Report

Team Name (1)

Warehouse Workers

Team Members (1)

1. Joshua Chenoweth.....
   Joshua.chenoweth1@marist.edu (Team Head)
2. John Biolo……..
   John.Biolo1@marist.edu (Team Member)
3. Annie Lee……
   Annie.Lee1@marist.edu (Team Member)
4. Evan Brown……...
   Evan.Brown2@marist.edu (Team Member)
5. Nicholas DiPardo……..…
   Nicolas.DiPardo1@marist.edu (Team Member)

Description of Team Members (1)

Joshua Chenoweth: A sophomore at Marist College majoring in computer science with a focus in software development. Loves golf and playing basketball whenever possible.
Annie Lee: A senior at Marist College majoring in computer science with a focus in software development.
John Biolo: A Junior at Marist college majoring in data science and analytics. Likes to golf and very into baseball and football.
Evan Brown: A computer science major at Marist College with a concentration in software development. On the rowing team and loves hiking.
Nicolas DiPardo: A major in game design at Marist College. On the Esports team and really enjoys spending time with his friends.

WMS Description & Objective (2)

In the warehouse and supply chain industry, changes to inventory, as well as shipment information, can become overwhelming at times. This is why logistics

companies create and implement warehouse management systems. These systems are highly configurable and efficient, and they allow companies to take strong control of their product and manage every aspect of it. Whether it's storage, shipment, or distribution, the warehouse management systems are created to keep track of everything that occurs within the company. They also give company leaders the ability to manipulate and maneuver their product by using the system. Shipping and exporting companies prefer specific things when it comes to the capabilities of their WMS, first of which being the internet enabled. To provide accessibility and control of products, a WMS should be accessible through the internet. It is crucial that employees can access information through the web to keep proper maintenance of the inventory at hand. A good WMS will also need real-time capabilities since it is important to have real-time access to a constantly shifting WMS to plan and move inventory with real-time decisions. A good WMS will also need packaging options to determine the best way of processing and exporting goods through the shipment process. It will also need proper stock rotation methods to keep efficient tracking of the variety of stock constantly moving in and out of the warehouse. It will need an inventory transaction log to keep track of the history, shipment addressing, and movement of products coming through the warehouse. And finally, a good user interface is needed to promote user reporting, as well as timely refunds and returns.

Understanding the logistics of a proper WMS will assist in the creation of one, as well as the productivity of anyone attempting to utilize it. It is essential that our WMS promotes efficiency in all departments and should be a very satisfying project to complete.

Related Works (3)

ShipHero (1):

ShipHero is one of the nation's leading companies for warehouse management systems. It is simple in the sense that it makes picking, packaging, and shipping easier. It allows users to pack, pick, and ship more efficiently allowing for an increase in profits as well as employee training speed. Its company is integrated within big names such as Amazon, UPS, FedEx, Oracle, and many more.

Pros:

- 99% plus recording shipping accuracy
- 30% faster shipping
- 35% reduction in warehouse costs
- 3x increase in picking efficiency
- Amazing customer support

Cons:

- When inventory is lost it's a large scale
- Limited dashboard on the graphics side
- Labor costs cause ShipHero to end up paying for itself very quickly

Oracle NetSuite (2):

Oracle NetSuite is a program designed on cloud software; it happens to be the number one cloud business software. It is also a unified management suite, meaning it encompasses much more than just warehouse management systems. With its online flexibility, it attempts to cut out the middleman and really connect the seller to the consumer.

Pros:

- Trusted worldwide
- One system for all
- Software that grows along with the company
- Real-time visibility
- Flexible and dynamic

Cons:

- Customer support isn't nearly as easy

- The website can slow sometimes, reaching new locations on the system can take a while
- Roles and workflows can be made complex

3PL Warehouse Manager from Extensiv (3):

3PL prides itself on being one of the most up to date warehouse management systems software has to offer. They utilize a warehouse system that is built to scale, reduces labor costs, increases visibility, all while increasing efficiency. It automates billing and integrates with API, EDI, Shopping carts, retailers, and shipper connections. Not to mention prebuilt systems will work seamlessly with this software.

Pros:

- More tedious tasks are handled efficiently
- Adaptable service options
- More money is made in the long run
- Cloud services increase integration and efficiency for the purpose of growth
- Online billing and transaction APIs allow for a streamlined purchasing process

Cons:

- Due to the cloud and third-party nature, some control is lost
- Poor service will reflect very poorly on your company
- Larger startup cost

The Merits of the WMS (4)

- Our system would offer a significant increase in overall warehouse output.

- A proper system like ours would cut production error rates almost completely out of the picture.
- Proper inventory organization would be solidified because of our system.
- Shipping rates would greatly increase due to the organization introduced by our WMS.
- Our volume of warehouse shipments will increase because of the improved shipping speed.
- Cloud capabilities would offer more online system storage as well as an increase in flexibility amongst admins.
- The system's online capabilities would allow for clean and neat user tracking throughout the entirety of the shipping process.

A user would ideally select our product to prioritize efficiency in warehouse management. If we can stabilize consistency and efficiency, our WMS should excel in organization, flexibility, speed, and overall production output. This is why we believe any user would and should select our product as their future warehouse management system.

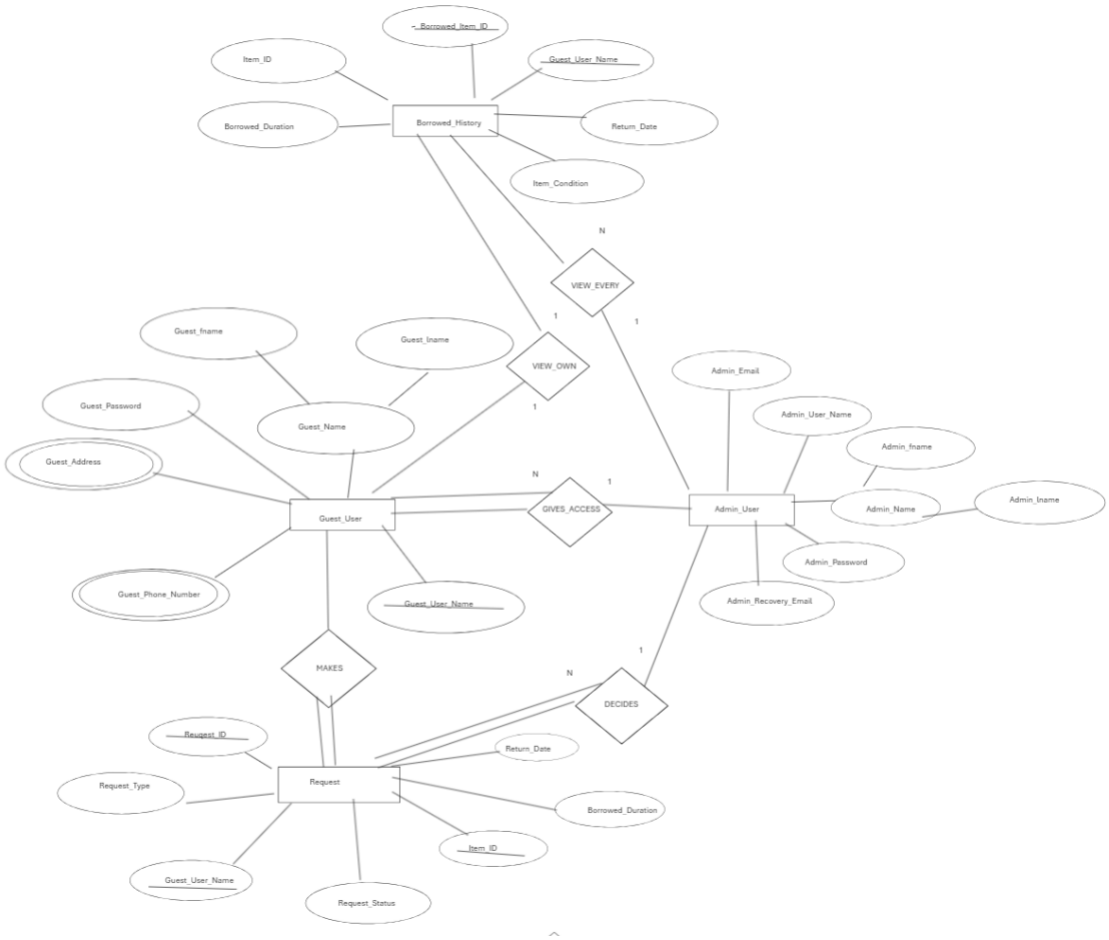
GitHub Repository Address (5)

https://github.com/JoshuaChenoweth/Warehouse_Workers

External Models (6)

Admin_User External Model:

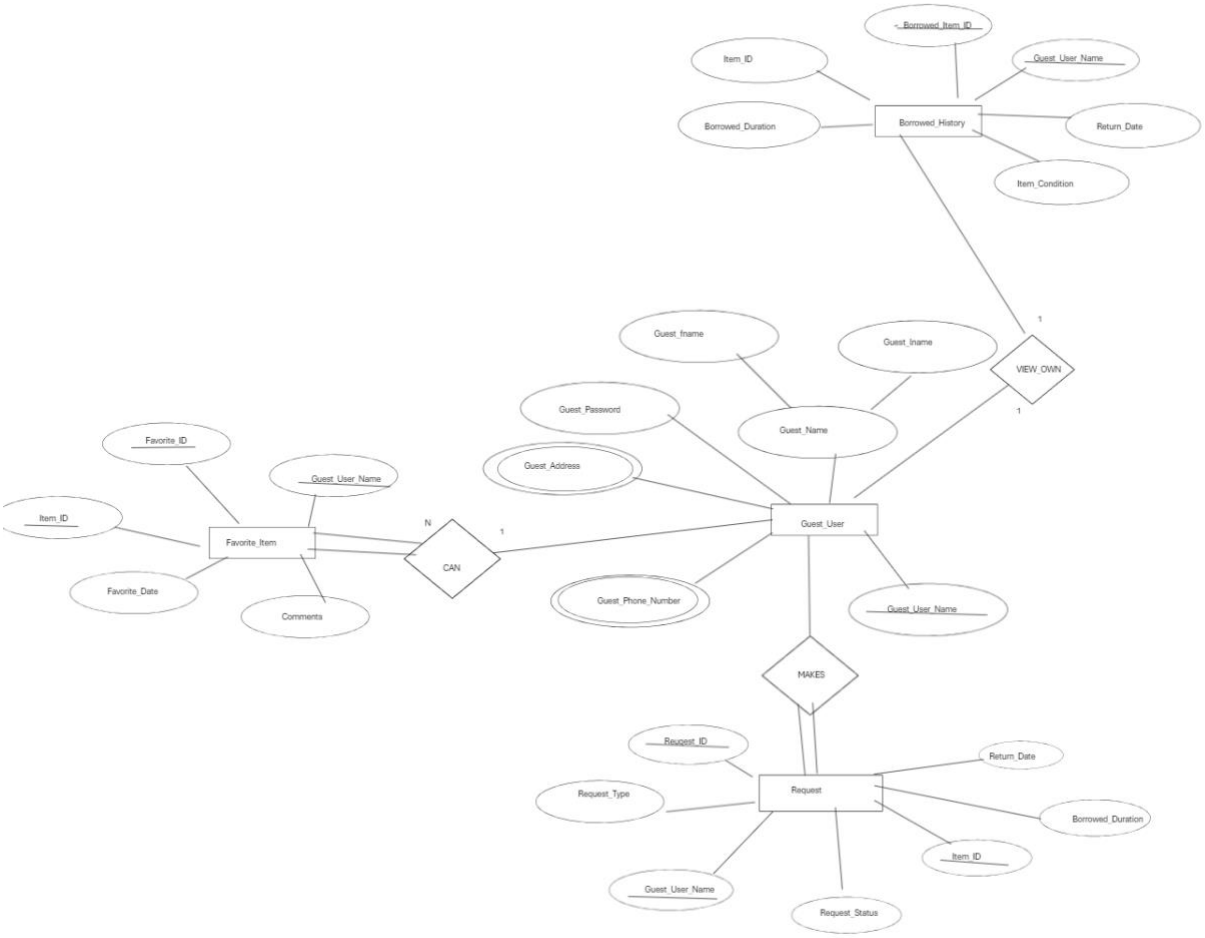
As shown above, our admin user is able to manage views of the guest user, decide incoming item requests, and view every guest user's borrowed item history. This shows an interface of what the admin user is able to see from their side of the database.



Guest_User External Model:

As shown above, our guest user is able to create their own borrowed item history, make an item request, and favorite their items. This shows an

interface of what the guest user is able to see from their side of the database.



Entity Relationship Model (ER) (7)

Our WMS is designed to efficiently manage, and track information related to a warehouse operation. Figure 1 shows how our entities and attributes are related, with a description of the entities and how they were chosen below:

Entities:

1. Admin_User: Represents employees/employers/managers who have access to manage the warehouse system. Each admin user has a unique username (“Admin_User_Name”) as the primary key.
2. Guest_User: Represents clients of the warehouse system. Each guest user has a unique username (“Guest_User_Name”) as the primary key.
3. Department: Represent individual departments in the warehouse. Each department has a unique identifier (“Department_ID”) as the primary.
4. Item: Represents the items stored in the warehouse. Each item has a unique identifier (“Item_ID”) as the primary key.
5. Order: Represents orders placed for items in the warehouse. Each order has a unique identifier (“Order_ID”) as the primary key.
6. Request: Represents requests made by guest users to borrow/buy items. Each request has a unique identifier (“Request_ID”) as the primary key.
7. Borrowed_History: Represents the history of items borrowed by users. Each borrowed item has a unique identifier (“Borrowed_Item_ID”) as the primary key.
8. Supplier: Represents the suppliers providing items to the warehouse. Each supplier has a unique identifier (“Supplier_ID”) as the primary key.
9. Favorited_Item: Represent items that guest users have marked as favorites. Each favorite item has a unique identifier (“Favorite_ID”) as the primary key.
10. Returns: Represents returns made by guest users for previously ordered items. Each return has a unique identifier (“Return_ID”) as the primary key.

The attributes within each entity are chosen to collect essential information about each warehouse entity:

Attributes:

1. Admin_User:
    1. Admin_User_Name: Uniquely identifies each admin user, allowing for efficient user management.
    2. Admin_Password: Stores password for authentication and security purposes.

3. Admin_Name (Admin_fname, Admin_lname): Separating the admin's first and last names allows for personalized communication and identification.
4. Admin_Email: Stores email address of the admin for communication and contact purposes.
5. Admin_Recovery_Email: Provides a secondary email address for account recovery and security.

2. Guest_User:
    1. Guest_User_Name: Uniquely identifies guest users within the system.
    2. Guest_Password: Stores password for guest user authentication.
    3. Guest_Name (Guest_fname, Guest_lname): Separates guest user's first and last names for personalized interactions.
    4. Guest_Address: Collects guest user's address for shipping and communication purposes.
    5. Guest_Email: Stores email address of the guest user for communication and notifications.
    6. Guest_Recovery_Email: Provides secondary email address for account recovery and security.
    7. Guest_Phone_Number: Stores phone number of the guest user for contact purposes.
3. Department:
    1. Department_ID: Serves as primary key to uniquely identify each department.
    2. Department_Name: Stores name/label of department for easy identification.
    3. Manager_Name: Collects name of department manager for organizational purposes.
    4. Quantity: Keeps track of quantity of items within department, aiding in inventory management.
    5. Department_Type: Describes type/category of department to provide additional context about department's purpose/function.

4. Item:
    1. Item_ID: Uniquely identifies each item in warehouse.
    2. Supplier_ID: Uniquely identifies each supplier.
    3. Item_Name: Stores name/description of item for easy identification.
    4. Item_Type: Specifies type/category of item.
    5. Weight: Provides weight of item (for physical information purposes).
    6. Height: Provides height of item (for physical information purposes).
    7. Storage_Condition: Describes the condition in which item is stored, which can be important for maintaining item quality.
    8. Stored_Time: Records time at which the item was stored in warehouse.
    9. Place: Specifies exact location within warehouse where item is stored.
5. Order:
    1. Order_ID: Uniquely identifies each order, facilitating order tracking and management.
    2. Item_ID: Uniquely identifies each item in warehouse.
    3. Order_Type: Describes type of order, providing information about its purpose.
    4. Order_Date: Records date of when order was placed.
    5. Order_Status: Indicates status of order, allowing order tracking.
    6. Total_Price: Stores total price of order.
    7. Payment_Method: Records method used for payment.
    8. Sales/Discounts: Collects information related to any sales/discounts applied to order.
6. Request:
    1. Request_ID: Uniquely identifies each request.
    2. Request_Type: Specifies whether request is to borrow/buy item.
    3. Guest_User_Name: Links request to guest user making request.
    4. Item_ID: Links request to item being requested.
    5. Request_Duration: Records duration of request.
    6. Request_Status: Provides status of request.

7. Borrowed_History:
    1. Borrowed_Item_ID: Uniquely identifies each borrowed item.
    2. Guest_User_Name: Links borrowed item to guest user who borrowed it.
    3. Item_ID: Links borrowed item to item that was borrowed.
    4. Borrowed_Duration: Records duration for which item was borrowed.
    5. Return_Date: Records date on which item was returned.
    6. Return_Condition: Describes condition of returned item.
8. Supplier:
    1. Supplier_ID: Uniquely identifies each supplier.
    2. Supplier_Name: Stores name of supplier for identification.
    3. Supplier_Address: Collects address of supplier.
    4. Supplier_Email: Stores email address of the supplier for communication.
    5. Supplier_Contact_Number: Records contact number of suppliers for communication purposes.
9. Favorited_Item:
    1. Favorite_ID: Uniquely identifies each favorite item, allowing tracking of user preferences.
    2. Guest_User_Name: Links favorited item to guest user who marked it as a favorite.
    3. Item_ID: Links favorited item to item that was marked as a favorite.
    4. Favorite_Date: Records date when item was marked as favorite.
    5. Comments: Allows guest user to add note/comment to specific favorite item.
10. Returns:
    1. Return_ID: Uniquely identifies each return.
    2. Order_ID: Links return to order associated with returned item.
    3. Guest_User_Name: Links return to guest user who initiated return.
    4. Item_ID: Links return to item being returned.
    5. Guest_Address: Records address provided by guest user for return.

6. Payment_Return: Records which payment method was used to return remaining money to guest user.

The relationships, along with the types of participation and cardinalities, linked with each entity were decided based on:

- **Admin_User gives access to Guest_User (1 to N, partial on admin, total on guest):** This relationship represents the association between guest users and admin users, indicating that multiple guest users can be associated with a single admin user (N to 1). It implies that an admin user has control or access rights over multiple guest users. There is partial participation on the admin, meaning that not all admin users are associated with a guest user. However, a gues user must be associated with an admin user to gain access to the system.
- **Guest_User able to view own Borrowed_History (1 to 1, partial on guest, total on history):** This relationship signifies that each guest user has a one-to-one relationship with their own borrowed history. It means that each guest user can view their specific borrowing history, and their borrowing history is unique to them. Partial participation on guest users mean that they are not required to have a borrowing history. However, every borrowing history is associated with at least one guest user.
- **Admin_User able to view every Borrowed_History (N to 1, partial on both sides):** This relationship indicates that each admin user can view the borrowing history of multiple users (N to 1). It means that admin users have the authority to access and view the borrowing histories of guest users. Partial participation on both sides implies that not all admin users need to be associated with a borrowing history, and not all borrowing histories need to be associated with admin users.
- **Admin_User decides Request (1 to N, partial on admin, total on request):** This relationship represents that each admin user can decide on multiple requests made by guest users (1 to N). It means that admin users have the authority to approve or reject requests. The partial participation on

the admin side and total on the request side indicates that every request must be associated with an admin user, but not all admin users need to be associated with a request.

- **Guest_User makes Request (1 to N, partial on guest, total on request):** This relationship signifies that each guest user can make multiple requests (1 to N), such as requests to borrow or buy items. Total participation on the request side means that every request must be associated with a guest user, while partial participation on the guest side implies that not all guest users need to make requests.
- **Guest_User can Favorite_Item (1 to N, partial on guest, total on favorite item):** This relationship indicates that each guest user can mark multiple items as favorites (1 to N). Total participation on the favorite item side means that every favorite item must be associated with a guest user, while partial participation on the guest side implies that not all guest users need to mark items as favorites.
- **Request turns into Order (1 to 1, partial on request, total on order):** This relationship indicates that some requests can turn into orders (e.g., approved purchase requests). Total participation on the order side means that every order must be associated with a request, but not all requests necessarily become orders.
- **Order has Item (M to N, total on both sides):** This relationship signifies that each order can have multiple items (M to N), and each item can be part of multiple orders. Total participation on both sides means that every order must be associated with one or more items, and every item must be associated with one or more orders.
- **Item in Department (N to 1, total on both sides):** This relationship represents that multiple items can be stored in a single department (N to 1), and each item belongs to a specific department. Total participation on both sides means that every item must be associated with a department, and every department must have one or more items.

- **Item from Supplier (N to 1, total on item, partial on supplier):** This relationship indicates that each item is supplied by one supplier (N to 1), but not all items need to be associated with a supplier. Total participation on the item side means that every item must be associated with a supplier, while partial participation on the supplier side implies that not all suppliers need to be associated with items.
- **Order can have Return (1 to 1, partial on order, total on return):** This relationship signifies that each order can have a return associated with it (1 to 1). Partial participation on the order side means that not all orders need to have returns, while total participation on the return side means that every return must be associated with an order.

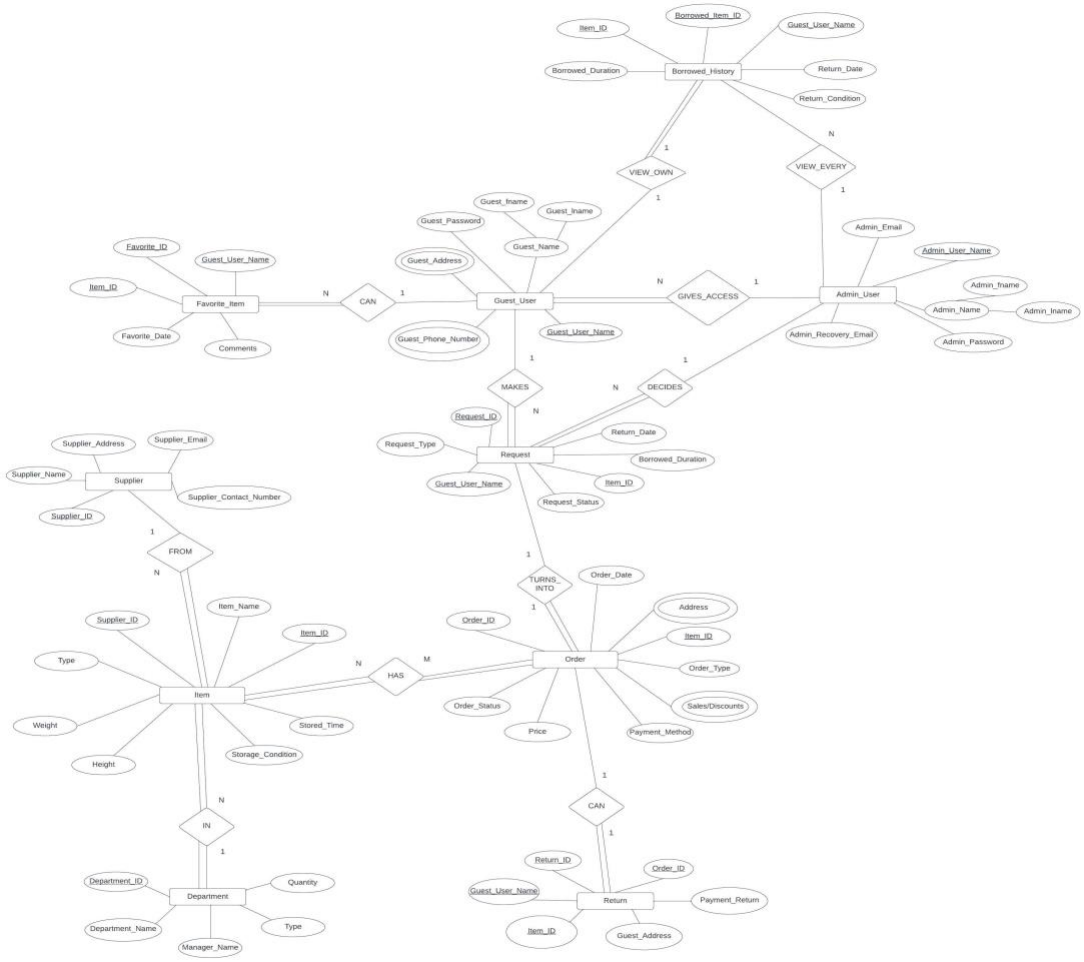


Figure 1. ER Diagram

Enhanced Entity Relationship Model (EER) (8)

To deeply go into the specifics required for each entity, the EER (Enhanced Entity Relationship) Diagram (Figure 2) shows the attributes we chose and how our primary keys are used in multiple instances (foreign keys). Below is a list of all the primary keys in each entity:

1. Admin_User: Admin_User_Name
2. Guest_User: Guest_User_Name
3. Department: Department_ID
4. Item: Item_ID
5. Order: Order_ID
6. Request: Request_ID
7. Borrowed_History: Borrowed_Item_ID
8. Supplier: Supplier_ID
9. Favorited_Item: Favorite_ID
10. Returns: Return_ID

Below is a description of each relationship between two entities that share keys:

- Order has Item: The keys between Order, Item, Return, and Supplier are shared to Order because of all of the relationships connected with Item.
- Item has Department: The keys between Item, Department, and Supplier are shared to Department because of all of the relationships connected with them.

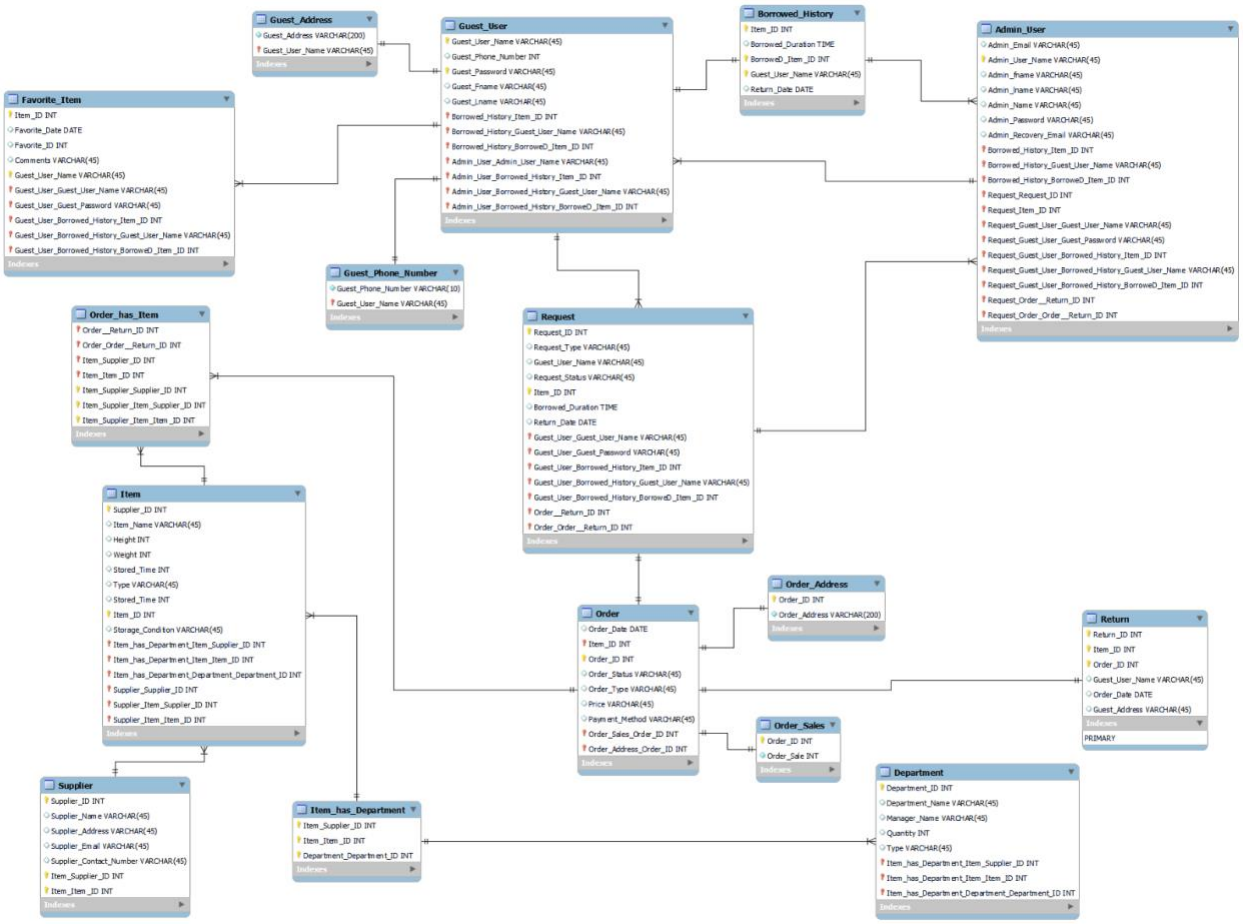


Figure 2. EER Diagram

Database Development (9)

1. Database and Table Creation

   Database Name: Warehouse

```
create database if not exists warehouse;
```

2. Guest_User Table

```
CREATE TABLE Guest_User (
    Guest_User_Name varchar(45) primary key,
    Guest_fname varchar(40),
    Guest_lname varchar(40),
    Guest_password char(20),
    Guest_address varchar(255),
    Guest_Phone_Number int
);
```

Attribute	Data Type	Description
Guest_User_Name	varchar(45)	Primary key, Guest user's username
Guest_fname	varchar(40)	Guest user's first name
Guest_lname	varchar(40)	Guest user's last name
Guest_password	char(20)	Guest user's password
Guest_address	varchar(255)	Guest user's address
Guest_Phone_Number	int	Guest user's phone number

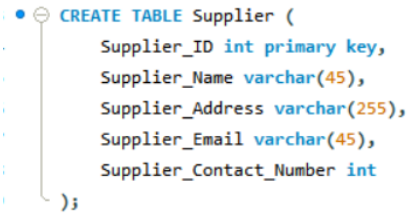
3. Admin_User Table

```
CREATE TABLE Admin_User (
    Admin_User_Name varchar(45) primary key,
    Admin_Email varchar(40),
    Admin_fname varchar(40),
    Admin_lname varchar(40),
    Admin_Password char(20),
    Admin_Recovery_Email varchar(40)
);
```

Attribute	Data Type	Description

Admin_User_Name	varchar(45)	Primary key, Admin user's username
Admin_fname	varchar(40)	Admin user's first name
Admin_lname	varchar(40)	Admin user's last name
Admin_password	char(20)	Admin user's password
Admin_Recovery_Email	varchar(40)	Admin user's recovery email

4. Supplier Table



```
CREATE TABLE Supplier (
    Supplier_ID int primary key,
    Supplier_Name varchar(45),
    Supplier_Address varchar(255),
    Supplier_Email varchar(45),
    Supplier_Contact_Number int
);
```

Attribute	**Data Type**	**Description**
Supplier_ID	int	Primary key, Supplier's ID
Supplier_Name	varchar(45)	Supplier's name
Supplier_Address	varchar(255)	Supplier's address
Supplier_Email	varchar(45)	Supplier's email
Supplier_Contact_Number	int	Supplier's contact number

5. Item Table

```
CREATE TABLE Item (
    Item_ID int primary key,
    Item_Name varchar(40),
    Supplier_ID int,
    Height decimal,
    Weight decimal,
    Stored_Time datetime,
    Type_of_item varchar(45),
    Storage_Condition varchar(45),
    foreign key (Supplier_ID) references Supplier(Supplier_ID)
);
```

Attribute	Data Type	Description
Item_ID	int	Primary key, unique item identifier
Item_Name	varchar(40)	Name of item
Supplier_ID	int	Identifier for the supplier of the item
Height	decimal	Height of the item
Weight	decimal	Weight of the item
Stored_Time	datetime	Timestamp of when the item was stored
Type_of_Item	varchar(45)	Type/category of the item
Storage_Condition	varchar(45)	Condition in which the item is stored

6. Borrowed_History Table

```
CREATE TABLE Borrowed_History (
    Borrowed_Item_ID int primary key,
    Item_ID int,
    Borrowed_Duration double,
    Return_Date date,
    Return_Condition varchar(45),
    Guest_User_Name varchar(45),
    foreign key (Guest_User_Name) references Guest_User(Guest_User_Name),
    foreign key (Item_ID) references Item(Item_ID)
);
```

Attribute	Data Type	Description
Borrowed_Item_ID	int	Primary key, unique borrowed item ID

Item_ID	int (foreign key)	Identifier for the borrowed item
Borrowed_Duration	double	Duration for which the item is requested
Return_Date	date	Date when the item was returned
Return_Condition	varchar(45)	Condition of the returned item
Guest_User_Name	varchar(45) (foreign key)	Name of the guest user who borrowed the item

7. Favorite_Item Table

```
CREATE TABLE Favorite_item (
    Favorite_ID int primary key,
    Item_ID int,
    Favorite_date date,
    comments text,
    Guest_User_Name varchar(45),
    foreign key (Guest_User_Name) references Guest_User(Guest_User_Name),
    foreign key (Item_ID) references Item(Item_ID)
);
```

Attribute	**Data Type**	**Description**
Favorite_ID	int	Primary key, unique favorite item ID
Item_ID	int (foreign key)	Identifier for the favorite item
Favorite_date	date	Date when the item was marked as a favorite
Comments	text	Comments or notes related to the favorite item
Guest_User_Name	varchar(45) (foreign key)	Name of the guest user who borrowed the item

8. Request Table

```
CREATE TABLE Request (
    Request_Id int primary key,
    Guest_User_Name varchar(45),
    Item_Id int,
    Borrowed_Duration double,
    Request_Type varchar(45),
    Request_Status varchar(45),
    Return_Date date,
    foreign key (Guest_User_Name) references Guest_User(Guest_User_Name),
    foreign key (Item_ID) references Item(Item_ID)
);
```

Attribute	Data Type	Description
Request_ID	int	Primary key, unique request identifier
Guest_User_Name	varchar(45) (foreign key)	Name of the guest user making the request
Item_ID	Int (foreign key)	Identifier for the requested item
Borrowed_Duration	double	Duration for which the item is requested
Request_Type	varchar(45)	Type of request (e.g., borrow, buy)
Request_Status	varchar(45)	Status of the request (e.g., pending, approved)
Return_Date	date	Date when the item is expected to be returned

9. Orders Table

```
CREATE TABLE Returns (
    Returns_ID int primary key,
    Item_ID int,
    Orders_ID int,
    Guest_User_Name varchar(45),
    Orders_Date date,
    Guest_address varchar(255),
    foreign key (Guest_User_Name) references Guest_User(Guest_User_Name),
    foreign key (Item_ID) references Item(Item_ID),
    foreign key (Orders_ID) references Orders(Orders_ID)
);
```

Attribute	Data Type	Description
Orders_ID	int	Primary key, unique order identifier
Orders_Date	date	Date of the order
Orders_Status	varchar(45)	Status of the order (e.g., pending)
Orders_Type	varchar(45)	Type of order (e.g., purchase, rental)
Guest_address	varchar(255)	Shipping address
Price	decimal	Price of the order
Payment_Method	varchar(45)	Payment method (e.g., credit card)
Sales_Discounts	varchar(45)	Discounts applied to the order
Item_ID	Int (foreign key)	Identifier for the item associated with the order

10. Returns Table

```
CREATE TABLE Orders(
    Orders_ID int primary key,
    Orders_Date date,
    Orders_Status varchar(45),
    Orders_Type varchar(45),
    Guest_address varchar(255),
    Price decimal,
    Payment_Method varchar(45),
    Sales_Discounts varchar(45),
    Item_ID int,
    foreign key (Item_ID) references Item(Item_ID)
);
```

Attribute	Data Type	Description
Returns_ID	int	Primary key, unique return identifier
Item_ID	int (foreign key)	Identifier for the returned item

Orders_ID	int (foreign key)	Identifier for the associated order
Guest_User_Name	varchar(45) (foreign key)	Name of the guest user who returned the item
Orders_Date	date	Date of the associated order
Guest_address	varchar(255)	Address of the guest user returning the item

11. Department Table

```
CREATE TABLE Department(
    Department_ID int primary key,
    Department_Name varchar(45),
    Manager_Name varchar(45),
    Quantity int,
    Department_Type varchar(45)
);
```

Attribute	**Data Type**	**Description**
Department_ID	int	Primary key, unique department identifier
Department_Name	varchar(45)	Name of the department
Manager_Name	varchar(45)	Name of the department manager
Quantity	int	Quantity of items in the department
Department_Type	varchar(45)	Type/category of the department

String & Spatial Data Types (10)

*(See fig. 3-9 as reference below)

(4) In our further research of data types, we have chosen to focus on the following data types: string and spatial. With this further knowledge of data types, we can understand more of their purposes included in database management systems, such as ours.

(5) In MySQL, string data types are versatile for storing various kinds of data in tables. These data types include CHAR, VARCHAR, BINARY, VARBINARY, BLOB, TEXT, ENUM, and SET. CHAR stores a fixed number of characters, right-padded with spaces, while VARCHAR stores variable-length data without padding. BINARY and VARBINARY are similar but store binary data, with VARBINARY being variable in length. COLLATE is used for proper data storage. BLOB and TEXT are for large data, with BLOB for binary and TEXT for non-binary data. They have no padding, can't have default values, and must specify an index prefix length. ENUM is for selecting from a predefined list of values and offers compact storage. It's important to avoid mixing literal values with index values. SET is a string object allowing multiple values from a specified list, with a limit of 64 distinct characters. Trailing spaces are removed, and values are stored numerically. Find values in a SET using functions like find_in_set(). Overall, MySQL provides a range of string data types to suit different data storage needs.

(6) Spatial data types are used to represent physical information about specific locations, which can include point locations or geometric objects like countries, regions, and roads. Common spatial data types include GEOMETRY, POINT, LINESTRING, and POLYGON, while collections of these types include MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, and GEOMETRYCOLLECTION. Spatial objects can be represented in queries using two standard formats: Well-known text (WKT) for ASCII format and Well-known binary (WKB) for binary data exchange.

Spatial data often requires a spatial reference system (SRS), which is a coordinate-based system for geographic locations. Types of SRS include projected (for flat surfaces) and geographic (representing longitude and latitude). MySQL uses SRID (Spatial Reference ID) to denote SRS, with SRID 0 representing an infinite flat Cartesian plane with no assigned units.

MySQL provides ways to create spatial columns using the CREATE TABLE or ALTER TABLE statements, supported by various storage engines. These spatial

columns can have an SRID attribute to specify the spatial reference system which can be populated with spatial data and using an internal format, they can be converted to WKT or WKB formats using INSERT statements. Geometric values stored in tables can be fetched in internal format or converted to WKT or WKB formats using functions like ST_asText() and ST_asBinary(). MySQL uses R-trees for spatial indexing on spatial columns, allowing for efficient multidimensional object indexing. Spatial indexes can be created using the SPATIAL INDEX keyword, which creates an R-tree index or, for storage engines that support it, a B-tree index for exact-value lookups.

In summary, MySQL provides robust support for string and spatial data types, allowing users to represent and manipulate the types of data and geographic information they want.

```
use elmasri_company;

select * from dept_locations;

-- creates table of department locations in elmasri_company
CREATE TABLE dept_coordinates(location_name VARCHAR(255), coordinates POINT);

-- inserts three locations' coordinates into dept_coordinates
INSERT INTO dept_coordinates(location_name, coordinates)
VALUES ('Houston', POINT(29.7604, 95.3698));

INSERT INTO dept_coordinates(location_name, coordinates)
VALUES ('Stafford', POINT(38.4221, 77.4083));

INSERT INTO dept_coordinates(location_name, coordinates)
VALUES ('Bellaire', POINT(29.7058, 95.4588));


-- displays all data residing in dept_coordinates
select * from dept_coordinates;
```

Figure 3. Example of using POINT

Figure 4. Output of code for POINT

```sql
use world;

-- creates table using polygon data type
CREATE TABLE shapes (Sname VARCHAR(255), Spoints POLYGON);

-- inserts a square into table "shapes"
INSERT INTO shapes (Sname, Spoints)
VALUES ('square', ST_GeomFromText('POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))'));

-- inserts a triangle into table "shapes"
INSERT INTO shapes (Sname, Spoints)
VALUES ('triangle', ST_GeomFromText('POLYGON((0 0, .5 1, 1 0, 0 0))'));

-- displays all data in table "shapes"
select * from shapes;
```

Figure 5. Example of using POLYGON (6)

Figure 6. Output of code for POLYGON

```sql
-- Create a table with different string data types
DROP TABLE IF EXISTS sakila.stringexample;
CREATE TABLE StringExample (
    char_example CHAR(10),
    varchar_example VARCHAR(255),
    blob_example BLOB,
    nchar_example NCHAR(10),
    nvarchar_example NVARCHAR(255),
    binary_example BINARY(10),
    varbinary_example VARBINARY(255),
    text_example TEXT
);

-- Sample data
INSERT INTO StringExample
VALUES
    ('CharValue', 'VarcharValue', 'BlobData', N'NCharValue', N'NVarCharValue', 0x010203, 0x040506, 'TextData');
```

Figure 7. Example of using STRING

```
-- Length and data of each data type displyed in the table
SELECT
    'char_example' AS data_type, CHAR_LENGTH(char_example) AS length, char_example AS data FROM StringExample
UNION ALL
SELECT
    'varchar_example' AS data_type, CHAR_LENGTH(varchar_example) AS length, varchar_example AS data FROM StringExample
UNION ALL
SELECT
    'blob_example' AS data_type, CHAR_LENGTH(blob_example) AS length, HEX(blob_example) AS data FROM StringExample
UNION ALL
SELECT
    'nchar_example' AS data_type, CHAR_LENGTH(nchar_example) AS length, nchar_example AS data FROM StringExample
UNION ALL
SELECT
    'nvarchar_example' AS data_type, CHAR_LENGTH(nvarchar_example) AS length, nvarchar_example AS data FROM StringExample
UNION ALL
SELECT
    'binary_example' AS data_type, CHAR_LENGTH(binary_example) AS length, HEX(binary_example) AS data FROM StringExample
UNION ALL
SELECT
    'varbinary_example' AS data_type, CHAR_LENGTH(varbinary_example) AS length, HEX(varbinary_example) AS data FROM StringExample
UNION ALL
SELECT
    'text_example' AS data_type, CHAR_LENGTH(text_example) AS length, text_example AS data FROM StringExample;
```

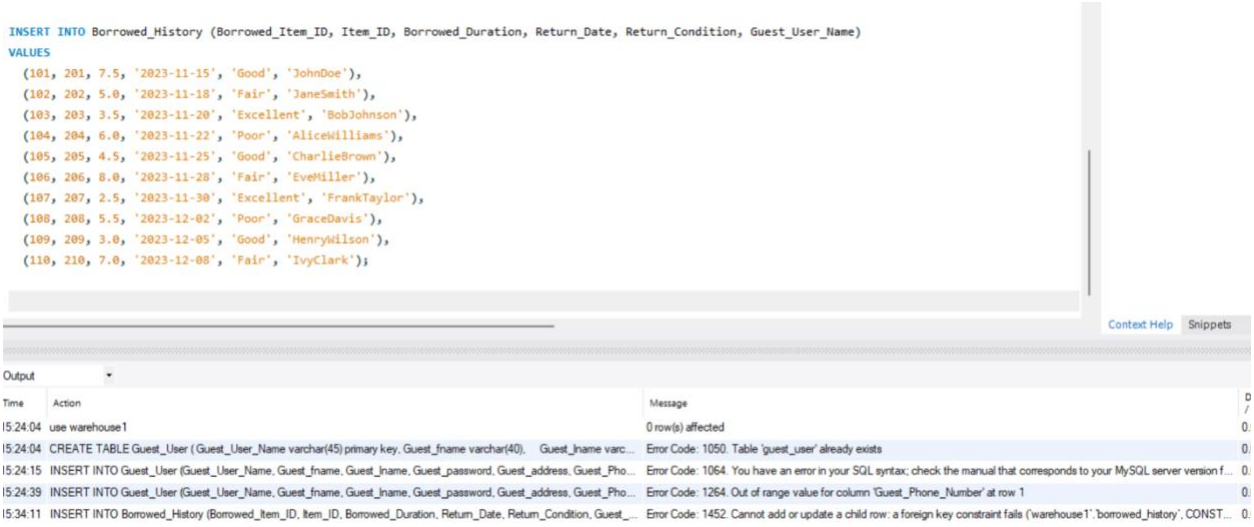
Figure 8. Example of using STRING (cont.)

data_type	length	data
char_example	9	CharValue
varchar_example	12	VarcharValue
blob_example	8	426C6F6244617461
nchar_example	10	NCharValue
nvarchar_example	13	NVarCharValue
binary_example	10	01020300000000000000
varbinary_example	3	040506
text_example	8	TextData

Figure 9. Output of code for STRING

Loading Data and Performance Enhancements (11)

Foreign Key Constraint Error:



```
INSERT INTO Borrowed_History (Borrowed_Item_ID, Item_ID, Borrowed_Duration, Return_Date, Return_Condition, Guest_User_Name)
VALUES
  (101, 201, 7.5, '2023-11-15', 'Good', 'JohnDoe'),
  (102, 202, 5.0, '2023-11-18', 'Fair', 'JaneSmith'),
  (103, 203, 3.5, '2023-11-20', 'Excellent', 'BobJohnson'),
  (104, 204, 6.0, '2023-11-22', 'Poor', 'AliceWilliams'),
  (105, 205, 4.5, '2023-11-25', 'Good', 'CharlieBrown'),
  (106, 206, 8.0, '2023-11-28', 'Fair', 'EveMiller'),
  (107, 207, 2.5, '2023-11-30', 'Excellent', 'FrankTaylor'),
  (108, 208, 5.5, '2023-12-02', 'Poor', 'GraceDavis'),
  (109, 209, 3.0, '2023-12-05', 'Good', 'HenryWilson'),
  (110, 210, 7.0, '2023-12-08', 'Fair', 'IvyClark');
```

Context Help | Snippets

Output

Time	Action	Message
15:24:04	use warehouse1	0 row(s) affected
15:24:04	CREATE TABLE Guest_User (Guest_User_Name varchar(45) primary key, Guest_fname varchar(40), Guest_lname varc...	Error Code: 1050. Table 'guest_user' already exists
15:24:15	INSERT INTO Guest_User (Guest_User_Name, Guest_fname, Guest_lname, Guest_password, Guest_address, Guest_Pho...	Error Code: 1064. You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version f...
15:24:39	INSERT INTO Guest_User (Guest_User_Name, Guest_fname, Guest_lname, Guest_password, Guest_address, Guest_Pho...	Error Code: 1264. Out of range value for column 'Guest_Phone_Number' at row 1
15:34:11	INSERT INTO Borrowed_History (Borrowed_Item_ID, Item_ID, Borrowed_Duration, Return_Date, Return_Condition, Guest_...	Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails (`warehouse1`.`borrowed_history`, CONST...

The Borrowed_History table contains two foreign keys, Guest_User_Name and Item_ID. When we tried to add data into those columns within the table, we got the error because the foreign keys were not added correctly. In order to avoid this error, we had to create and run the tables where the foreign key comes from, and then the corresponding table with the foreign key. The parent table needed to be created so that the tables with foreign keys can run properly.

Handling Foreign Key Constraints:

```
-- primary key constraints
INSERT INTO Guest_User (Guest_User_Name, Guest_fname, Guest_lname, Guest_password, Guest_address, Guest_Phone_Number)
VALUES ('guest1', 'John', 'Doe', 'password1', '123 Main St', '1234567890');
-- foreign key constraint
INSERT INTO Orders (Orders_ID, Orders_Date, Orders_Status, Orders_Type, Address, Price, Payment_Method, Sales_Discounts, Item_ID)
VALUES (11, '2023-12-10', 'Processing', 'Online', '123 Main St', 100.00, 'Credit Card', '5% off', 99);
-- Unique Key Constraint
INSERT INTO Supplier (Supplier_ID, Supplier_Name, Supplier_Address, Supplier_Email, Supplier_Contact_Number)
VALUES (11, 'Supplier11', '123 Main St', 'supplier1@example.com', '1234567890');
-- Data type jey constraint
INSERT INTO Guest_User (Guest_User_Name, Guest_fname, Guest_lname, Guest_password, Guest_address, Guest_Phone_Number)
VALUES ('guest11', 'John', 'Doe', 'password11', '123 Main St', 'invalid_phone');
```

Primary Key Constraint: Our primary key constraint prevents the insertion of a duplicate primary key in any given table. In our scenario above, if a another guest 1 is queried to be inserted into the table, an error will be thrown since an identical primary key already exists.

Foreign Key Constraint: In the example above, we attempt to insert an order with a non-existent item ID into the table. An error is then thrown as the foreign key constraint in the orders table ensures that the ID being inputted also exists in our item table.

Unique Key Constraint: A unique key constraint ensures non-duplicacy across the extent of a table. In our example we attempt to insert an email that is identical to one already in the table. An error is than thrown to prevent the addition of duplicate data in the slot of a unique key constraint limited column.

Data Type Constraint: Data type constraints limit the insertion of an improper data type into specific columns. In this example we entered a string of characters into a column thats limited to integers (Phone number). An error was thrown due to out data type constraint.

Importing Data:

```
CREATE TABLE Guest_User (
    Guest_User_Name varchar(45) primary key,
    Guest_fname varchar(40),
    Guest_lname varchar(40),
    Guest_password char(20),
    Guest_address varchar(255),
    Guest_Phone_Number varchar(15)
);

INSERT INTO Guest_User (Guest_User_Name, Guest_fname, Guest_lname, Guest_password, Guest_address, Guest_Phone_Number)
VALUES
('guest1', 'John', 'Doe', 'password1', '123 Main St', 1234567890),
('guest2', 'Jane', 'Smith', 'password2', '456 Oak St', 9876543210),
('guest3', 'Mike', 'Johnson', 'password3', '789 Pine St', 5551234567),
('guest4', 'Emily', 'Williams', 'password4', '101 Elm St', 1112223333),
('guest5', 'David', 'Miller', 'password5', '202 Maple St', 9998887777),
('guest6', 'Sarah', 'Jones', 'password6', '303 Birch St', 4445556666),
('guest7', 'Chris', 'Davis', 'password7', '404 Cedar St', 7776665555),
('guest8', 'Lisa', 'Anderson', 'password8', '505 Walnut St', 2223334444),
('guest9', 'Tom', 'Moore', 'password9', '606 Pineapple St', 6667778888),
('guest10', 'Amy', 'Wilson', 'password10', '707 Orange St', 3334445555);
```

Above is the process we took to insert 10 instances of data into our Guest_User_Table. After making data type constraints we inputted our specified instances into our table using the insert into command. For example, our first instance was for a man named John. We inserted his guest user name (guest1), his first guest_fname (John), his guest_lname (Doe), his guest_password (password1), his guest_address (123 main st), and his guest_phone_number (1234567890).

Optimizing Data Insertion:

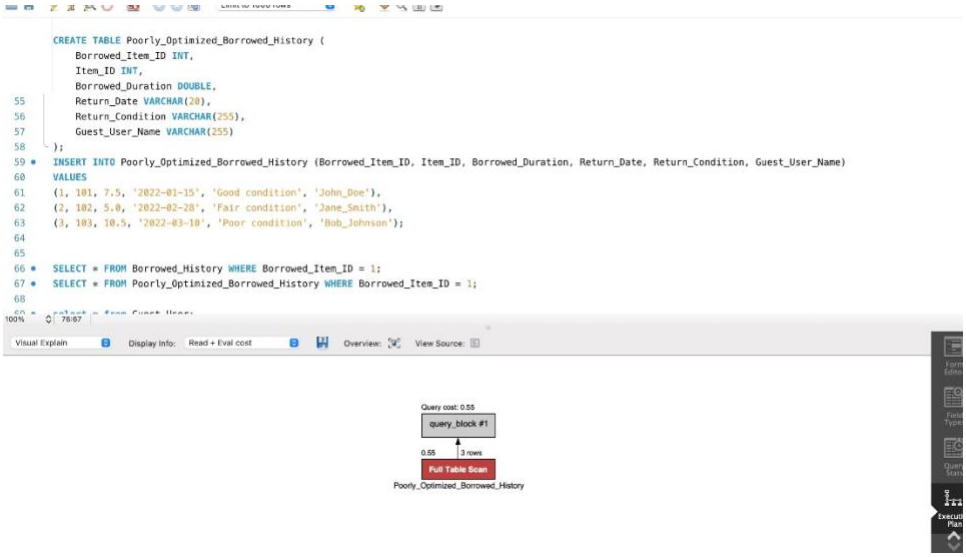
```
INSERT INTO Request (Request_Id, Guest_User_Name, Item_Id, Borrowed_Duration, Request_Type, Request_Status, Return_Date)
VALUES
(1, 'guest1', 1, 7.5, 'Borrow', 'Pending', NULL),
(2, 'guest2', 2, 5.0, 'Borrow', 'Approved', '2023-11-18'),
(3, 'guest3', 3, 3.5, 'Borrow', 'Rejected', NULL),
(4, 'guest4', 4, 2.0, 'Return', 'Pending', NULL),
(5, 'guest5', 5, 10.0, 'Return', 'Approved', '2023-11-25'),
(6, 'guest6', 6, 8.0, 'Borrow', 'Pending', NULL),
(7, 'guest7', 7, 6.5, 'Return', 'Rejected', NULL),
(8, 'guest8', 8, 4.0, 'Borrow', 'Approved', '2023-12-03'),
(9, 'guest9', 9, 9.5, 'Return', 'Pending', NULL),
(10, 'guest10', 10, 7.0, 'Borrow', 'Approved', '2023-12-08');
```

Shown above is the way we optimized our data insertion for maximum efficiency. There are several reasons as to why the methods we took were extremely efficient. For starters, we used a bulk method of insertion. We didn't bother to waste memory or computing power with multiple insertion statements for each instance, we only used one. Secondly, our previously specified data parameters for the table allowed for specific designation of memory amongst each of our instances. Our use of primary and foreign key constraints also prevented the existence of redundant data as well as the lack of consistent data. And finally, our effiecient transaction handling. Which remains crucial for the overall maintenence for our entire database.

<u>Quantatative evidence of proper table normalization:</u>

```
CREATE TABLE Borrowed_History (
    Borrowed_Item_ID int primary key,
    Item_ID int,
    Borrowed_Duration double,
    Return_Date date,
    Return_Condition varchar(45),
    Guest_User_Name varchar(45)
);
INSERT INTO Borrowed_History (Borrowed_Item_ID, Item_ID, Borrowed_Duration, Return_Date, Return_Condition, Guest_User_Name)
VALUES
(1, 1, 7.5, '2023-11-15', 'Good', 'guest1'),
(2, 2, 5.0, '2023-11-18', 'Excellent', 'guest2'),
(3, 3, 3.5, '2023-11-20', 'Fair', 'guest3');
```

<u>Quantatative evidence of poor table normalization:</u>



Normalization Check:

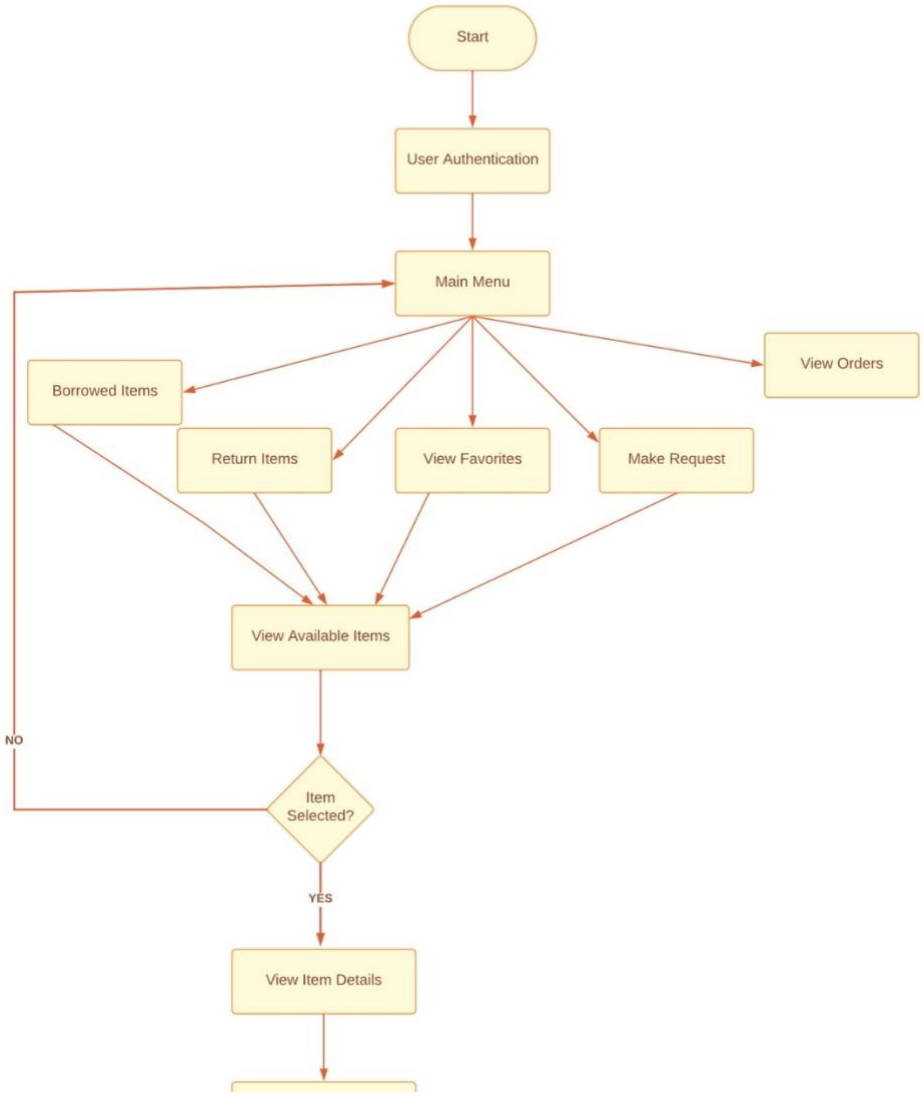
1NF: The database is already in its first normal form. In this normal form, each table ensures that data is stored atomically. The Guest_User, Admin_User, Supplier, Item, Borrowed_History, Favorite_Item, Request, Orders, Returns, and Department tables all fulfill the requirements of the first normal form.

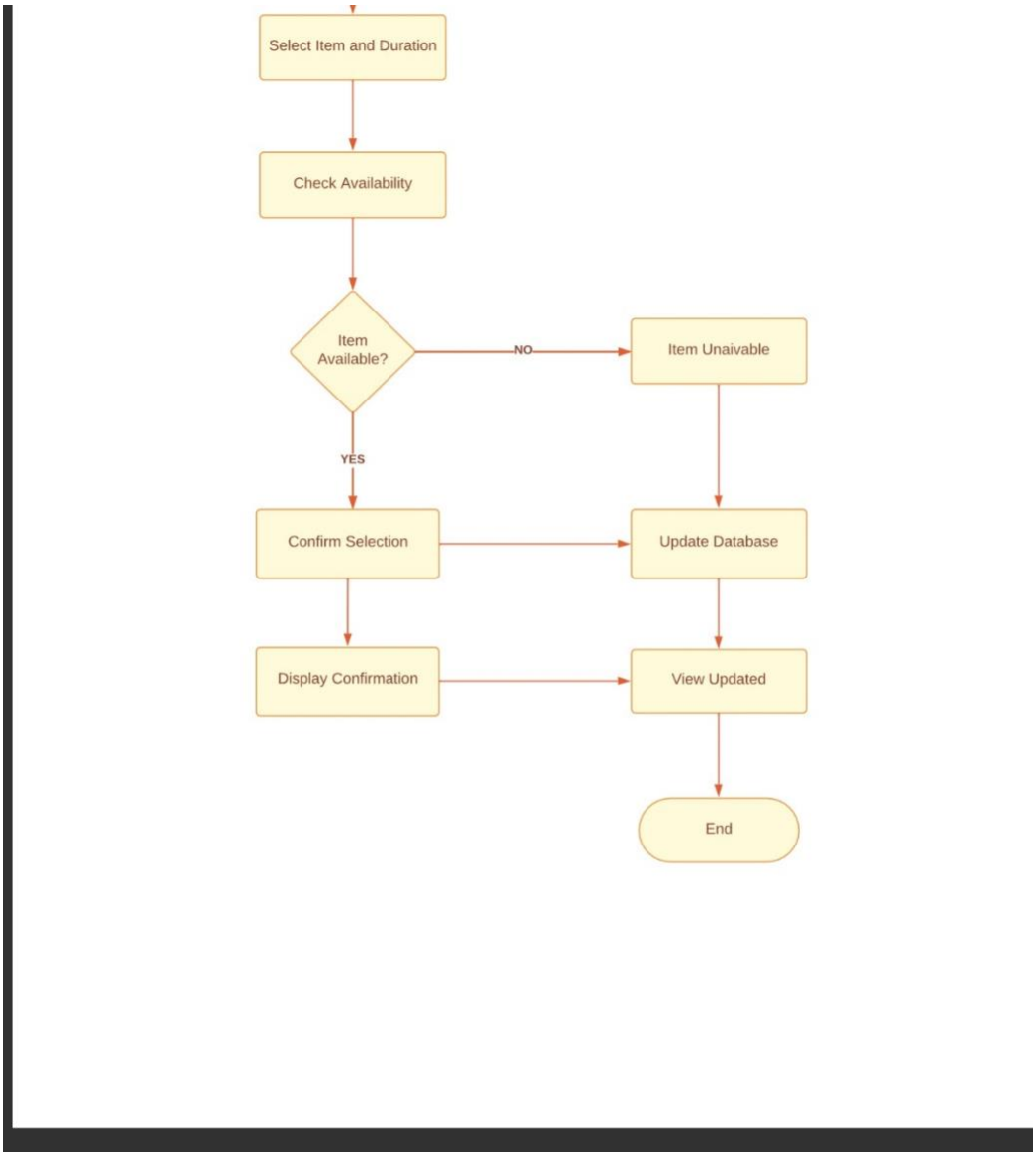
2NF: There are no partial dependencies in second normal form, therefore, the tables are already in their second form.

3NF: Each table in the schema has been designed to ensure that no column is transitively dependent on the primary key. For example, in the Department table, there are no transitive dependencies, and the data is organized in a way that supports data integrity and consistency. The overall design of the database aligns with the principles of 3NF, providing a robust and normalized structure that facilitates efficient data storage and retrieval while minimizing redundancy and dependency issues.

Application Development: (12)

UX Design Flowchart:





Views' Implementation of Pages:

Start/User authentication:

```
-- Start/User Authentication page
CREATE VIEW Start_Page AS
SELECT 'Welcome' AS Message;
```

Start and user authentication go hand and hand with one another. The start page links only to user authentication, or in this case, human verification. The start page will contain a simple message like “Welcome” that has a button that links the user to

another page that utilizes a method of user authentication. This view may be updatable as its message to the user is one-to-one.

Main Menu:

```
-- Main Menu page
CREATE VIEW Main_Menu_Page AS
SELECT 'View Orders' AS Menu_Option
UNION
SELECT 'Make Requests'
UNION
SELECT 'View Favorites'
UNION
SELECT 'Return Items'
UNION
SELECT 'Borrow Items';
```

The main menu will most likely be our largest page. Considering it's the main menu, it's going to contain a lot of links to other pages with all other pages also having a link to the main menu. Making this a one to many relationship. The main menu will allow you to choose between five options: view orders, make requests, view favorites, return items, and borrow items. This view's one-to-many relationship does not allow for updatabilty in our database.

View available items:

```
-- View available items page
CREATE VIEW Items_Details AS
SELECT
    Item_ID,
    Item_Name,
    Supplier_ID,
    Height,
    Weight,
    Stored_Time,
    Type_of_item,
    Storage_Condition
FROM Item;
```

This page is reached regardless of whatever option the user chooses, making it a many to one relationship. In this page, a list of available items can be filtered and presented

based upon what option the user selected on main menu. It will also have an automatic functionality that takes the user back to main menu if they decide not to select an available item. This view cannot be updated because of its many-to-one cardinality.

View item details:

```
-- View Items_Details page
CREATE VIEW Items_Details AS
SELECT
    Item_ID,
    Item_Name,
    Supplier_ID,
    Height,
    Weight,
    Stored_Time,
    Type_of_item,
    Storage_Condition
FROM Item;
```

This page will contain database information of the item the user selected on the view items page. So the item's price, information, identification and anything else a buyer would want to know. The user can reroute back to the page prior to this one if they wanted to. This view cannot be updated because of its many-to-one cardinality.

Select item and duration:

```
-- View items and duration page
CREATE VIEW Item_Selection AS
SELECT
    Item_ID,
    Item_Name,
    Supplier_ID,
    Height,
    Weight,
    Stored_Time,
    Type_of_item,
    Storage_Condition
FROM Item;
```

This page locks the user into this item and the purchasing process for it. Purchase confirmations as well as the duration of time the user may wish to utilize the item for will be available on this page. This view cannot be updated because of its many-to-one cardinality.

Check Availability:

```
-- Check availability page
CREATE VIEW Check_Availability AS
SELECT
    Item_ID,
    CASE
        WHEN CURRENT_DATE <= Stored_Time THEN 'Item Available'
        ELSE 'Item Unavailable'
    END AS Availability_Status
FROM Item;
```

This page will automatically check for the availability of the item the user selected. Depending on the status of the item the user will be routed to a specific page/conclusion, making this a one to many relationship; not allowing for further updatability.

Item Available:

```
-- Item available page
CREATE VIEW Item_Available AS
SELECT
    i.Item_ID,
    i.Item_Name,
    i.Type_of_item,
    i.Stored_Time,
    o.Order_ID,
    o.Order_Date,
    o.Address
FROM Item i
JOIN Orders o ON i.Item_ID = o.Item_ID;
```

If the item is available the user will be brought to a confirmation page that one, lists the items information, two, lists the users purchasing info and addressing info, and three, the shipping information. There will also be a page that displays the confirmation of said order immediately after it is complete making this a 1 to 1 relationship; allowing further updatability to create a more efficient database.

Item Unavailable:

```
-- Item unavilable page
CREATE VIEW Item_Unavailable AS
SELECT
    Item_ID,
    'Item Unavailable' AS Availability_Status
FROM Item
WHERE CURRENT_DATE > Stored_Time;
```

If the listed item is unavailable, then the user will be routed to a page listing the items current unavailability status. The user will then be routed back to the listed items page where they can search for another potential item. This view is one-to-one, as each user would be directly related to one item's availability; thus, allowing further updatability.

End Page:

```
-- End page
CREATE VIEW End_Page AS
SELECT
    o.Order_ID,
    o.Order_Status,
    o.Order_Type,
    o.Address,
    o.Price,
    o.Payment_Method,
    o.Sales_Discounts,
    i.Item_Name,
    i.Type_of_item
FROM Orders o
JOIN Item i ON o.Item_ID = i.Item_ID;
```

After this process the database will be updated as a result of user interaction with the item's in our warehouse. This will be a many to one relationship as it is the result of a multitude of outcomes that come from the purchasing process. Many-to-one relationship do not support further updatability.

Graphical User Interface Design: (12 continued)

Connection to Database:

Description:

The provided Python script develops a graphical user interface (GUI) application using the Tkinter library. This application relates to our warehouse management database, as indicated by the MySQL connection parameters. The

script includes the necessary imports, such as Tkinter for GUI components, messagebox for displaying messages, PIL for image processing, and mysql.connector for connecting to a MySQL database. The MySQL connection is configured with parameters host, user, password, and database name.

Outline:

Libraries:

- tkinter: Used for GUI development
- messagebox: Tkinter submodule for displaying message boxes
- PIL (Python Imaging Library): Utilized for image processing
- mysql.connector: Facilitates connections and interactions with a MySQL database

MySQL Connection Parameters:

- host: "localhost"
- user: "root"
- password: "Joshua092003"
- database: "warehouse1"

MySQL Connection:

- db: Establishes a connection to a MySQL database using the specified parameters

Source Code:

```
import tkinter as tk
from tkinter import messagebox
from PIL import Image, ImageTk
import mysql.connector

# MYSQL parameters
db = mysql.connector.connect (
        host="localhost",
        user="root",
        password="Joshua092003",
        database="warehouse1"
)
```

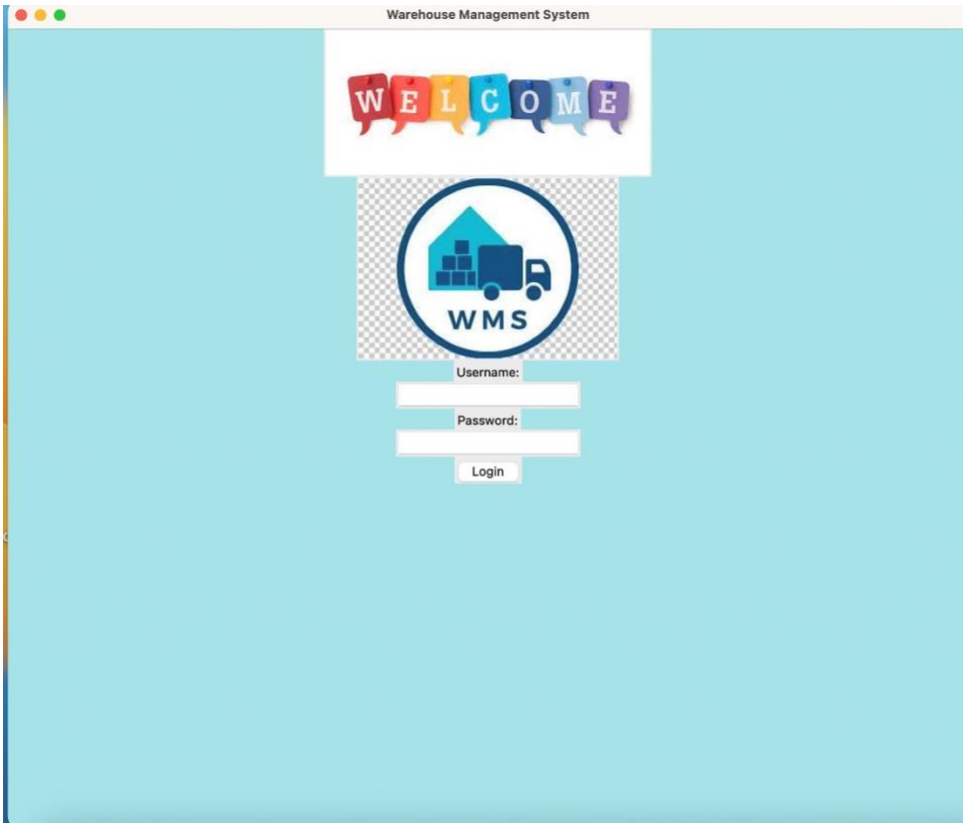
Login Page:

Description:

The authenticate_user function takes a username and password as parameters and attempts to authenticate the user by querying a MySQL database. It first checks if the database connection (db) exists, and if so, it creates a cursor to execute SQL queries. The function attempts to find a matching user in both the "Guest_User" and "Admin_User" tables. If a match is found in either table, it returns the corresponding user type ("guest" or "admin"). Error handling is implemented to catch any MySQL-related errors, and the cursor is closed in the finally block. If no match is found, the function returns None.

The login function is responsible for retrieving the entered username and password from the GUI entry widgets. It performs basic validation to ensure that both fields are filled in. If the fields are not filled, a message box is displayed to inform the user. Otherwise, it calls the authenticate_user function with the provided credentials. Depending on the user type returned by authenticate_user, it displays a success message and triggers the display of the main menu, tailored for either a guest or an admin. If the authentication fails, an appropriate error message is shown.

The GUI part of the code creates the main window using Tkinter, sets its title to "Warehouse Management System," and defines its size as 1000x1000 pixels. Two images, "welcome.jpg" and "wms.png," are loaded using Pillow, converted to Tkinter PhotoImage objects, and displayed in labels (label_welcome and label_wms). The GUI also includes labels and entry widgets for the username and password, along with a login button (button_login). The login button is associated with the login function, which initiates the user authentication process.



Outline:

Main Window Setup:

- Main window with welcome and WMS images
- Username and Password entry fields
- Login button

Authentication:

- Function `authenticate_user(username, password)` to validate user credentials against the database

Login Handling:

- Function `login()` to handle the login process:
  - Retrieve username and password
  - Validate the user through `authenticate_user`
  - Display appropriate messages for successful and unsuccessful logins

Source code:

```
def authenticate_user(username, password):
if db:
try:
cursor = db.cursor()

# Check if the user is in Guest_User table
query_guest = "SELECT * FROM Guest_User WHERE Guest_User_Name = %s AND
Guest_password = %s"
cursor.execute(query_guest, (username, password))
result_guest = cursor.fetchone()

if result_guest:
return "guest"

# Check if the user is in Admin_User table
query_admin = "SELECT * FROM Admin_User WHERE Admin_User_Name = %s AND
Admin_password = %s"
cursor.execute(query_admin, (username, password))
result_admin = cursor.fetchone()

if result_admin:
return "admin"
#error handling
except mysql.connector.Error as err:
print(f"Error: {err}")

finally:
cursor.close()

return None
# defines the login process function
def login():
#gets the username and password from the database
username = entry_username.get()
password = entry_password.get()
# Throws an error if the user tries to submit a login when one or more of the text
boxes are blank
if username == "" or password == "":
messagebox.showinfo("", "Username and password cannot be blank.")
else:
# Authenticate the user against the database
user_type = authenticate_user(username, password)
#if the user is a guest they are logged into the traditional main menu
if user_type == "guest":
messagebox.showinfo("", "Guest Login successful")
show_main_menu()
```

```
#if the user is an admin then they are logged into the admin version of the main
menu
elif user_type == "admin":
messagebox.showinfo("", "Admin Login successful")
show_main_menu(admin=True)

else:
messagebox.showinfo("", "Incorrect username or password")
```

Buttons and picture placement information:

```
# Create the main window
window = tk.Tk()
window.title("Warehouse Management System")
window.geometry('1000x1000')

# Load images with Pillow
welcome_image = Image.open("welcome.jpg")
wms_image = Image.open("wms.png")

# Convert images to Tkinter PhotoImage objects
welcome_tk_image = ImageTk.PhotoImage(welcome_image)
wms_tk_image = ImageTk.PhotoImage(wms_image)

# Create labels to display images
label_welcome = tk.Label(window, image=welcome_tk_image)
label_welcome.pack()

label_wms = tk.Label(window, image=wms_tk_image)
label_wms.pack()

# Username label and entry
label_username = tk.Label(window, text="Username:")
label_username.pack()

entry_username = tk.Entry(window)
entry_username.pack()

# Password label and entry
label_password = tk.Label(window, text="Password:")
label_password.pack()

entry_password = tk.Entry(window, show="*")
entry_password.pack()

# creates Login button
button_login = tk.Button(window, text="Login", command=login)
button_login.pack()
```

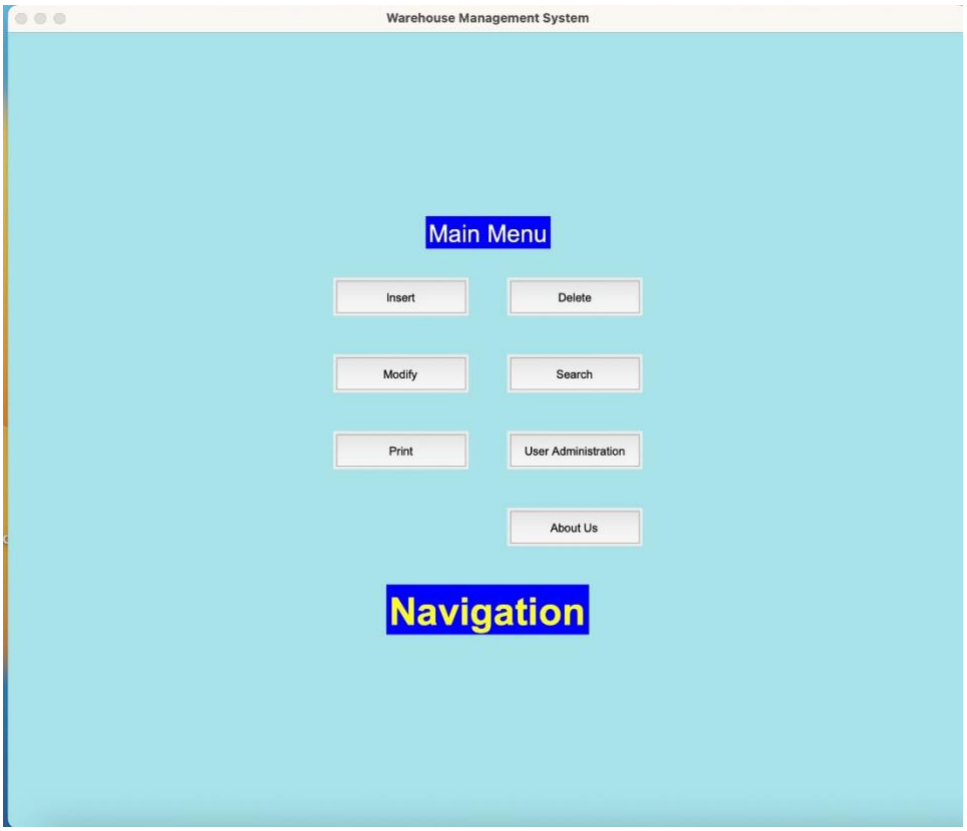
Main Menu Page:

Description:

The show_main_menu function begins by clearing all existing widgets in the main window to prepare for the main menu display. It creates a frame (main_menu_frame) with specified padding and a blue background, which serves as the main container for the menu elements. The title "Main Menu" is displayed at the top of the frame in a larger font with blue background and white text.

Following the title, the function adds buttons for various actions that the user can perform. These buttons include "Insert," "Delete," "Modify," "Search," and "Print," each associated with specific functions (insert_function, remove_function, modify_function, search_function, and print_item_details, respectively). The buttons are styled with a specific width, height, and font. Additionally, if the user is not logged in as an admin, an "Update Password" button is provided, linking to the show_update_password_window function.

For admin users, an additional "User Administration" button is included, allowing access to user administration functions through the user_admin_function. The "Navigation" label is displayed in a larger, bold font, and finally, the main menu frame is centered on the page.



Outline:

- Main Menu Display:
    - Function `show_main_menu(admin=False)`:
        - Clear the window
        - Create a frame for the main menu
        - Create a label for the menu title
        - Create buttons for insertion, deletion, modification, searching, printing, and user administration (if admin)
        - Display additional options for admin users
- Main Menu Buttons:
    - Button functions for each main menu option:
        - `insert_function()`: Create a window for item insertion
        - `remove_function()`: Create a window for item removal
        - `modify_function()`: Create a window for item modification
        - `search_function()`: Placeholder for search functionality

- `print_item_details()`: Create a window for printing item details
- `user_admin_function()`: Create a window for guest user functions (admin only)

Source Code:

```
def show_main_menu(admin=False):
#creates the main menu
for widget in window.winfo_children():
widget.destroy()

# segment creates the framework for the main menu's design
main_menu_frame = tk.Frame(window, padx=20, pady=20, bg='blue') # Set the
background color
main_menu_frame.pack(fill=tk.BOTH, expand=True)

# Creates the main menu title
label_menu = tk.Label(main_menu_frame, text="Main Menu", font=("Arial", 25),
bg='blue', fg='white') # Set background and text color
label_menu.grid(row=0, column=0, columnspan=2, pady=10)

# below are the buttons for the action pages provided for the user
btn_insert = tk.Button(main_menu_frame, text="Insert", command=insert_function,
width=15, height=2, font=("Arial", 12))
btn_insert.grid(row=1, column=0, pady=20, padx=(0, 20))

btn_delete = tk.Button(main_menu_frame, text="Delete",
command=remove_function, width=15, height=2, font=("Arial", 12))
btn_delete.grid(row=1, column=1, pady=20, padx=(20, 0))

btn_modify = tk.Button(main_menu_frame, text="Modify",
command=modify_function, width=15, height=2, font=("Arial", 12))
btn_modify.grid(row=2, column=0, pady=20, padx=(0, 20))

btn_search = tk.Button(main_menu_frame, text="Search",
command=search_function, width=15, height=2, font=("Arial", 12))
btn_search.grid(row=2, column=1, pady=20, padx=(20, 0))

btn_print = tk.Button(main_menu_frame, text="Print", command=print_item_details,
width=15, height=2, font=("Arial", 12))
btn_print.grid(row=3, column=0, pady=20, padx=(0, 20))
#If the user is a guest they will have access to a button that will allow them to
change their password
if not admin:
```

```
btn_update_password = tk.Button(main_menu_frame, text="Update Password",
command=show_update_password_window, width=15, height=2, font=("Arial", 12))
btn_update_password.grid(row=3, column=1, pady=20, padx=(20, 0))

# if the user is logged in as an admin then they will have access to the user
administration framework to manipulate the guest users
if admin:
btn_user_admin = tk.Button(main_menu_frame, text="User Administration",
command=user_admin_function, width=15, height=2, font=("Arial", 12))
btn_user_admin.grid(row=3, column=1, pady=20, padx=(20, 0))

# Create a label for the "navigation" text
label_warehouse = tk.Label(main_menu_frame, text="Navigation", font=("Arial", 40,
"bold"), bg='blue', fg='yellow') # Set background and text color
label_warehouse.grid(row=4, column=0, columnspan=2, pady=20)

# takes the main menu frame and centers it on the page
main_menu_frame.pack_propagate(False)
main_menu_frame.place(relx=0.5, rely=0.5, anchor=tk.CENTER)
```

Actions Pages:

Search Page:

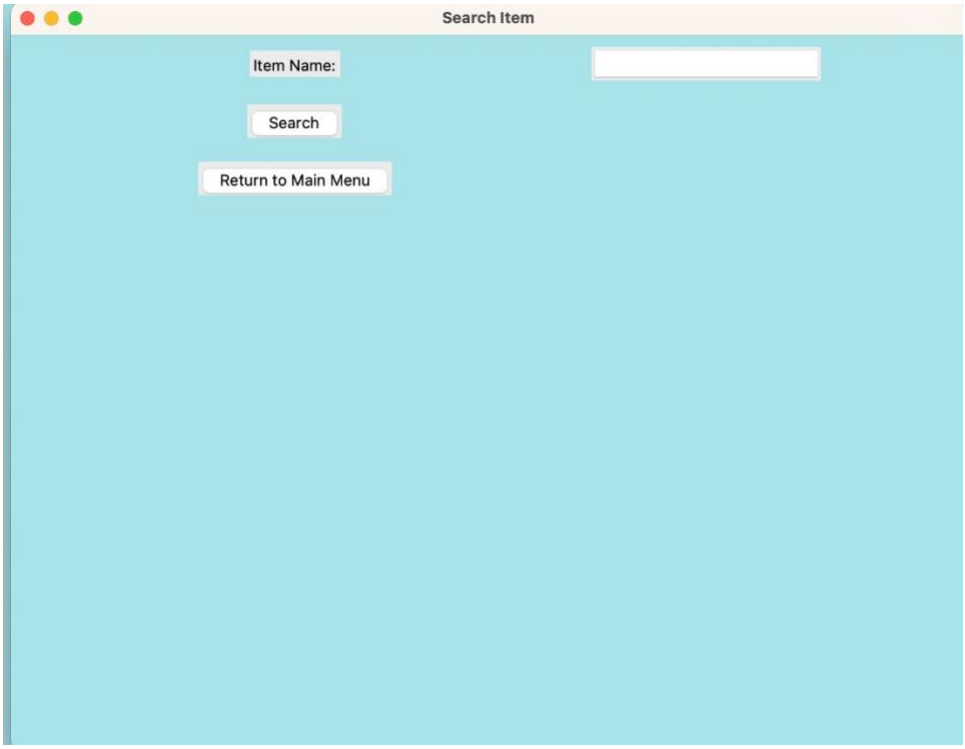
Description:

The search_function begins by creating a new top-level window (search_window) within the existing Tkinter application to facilitate the search process. This window is titled "Search Item" and has a predefined geometry. It includes labels and entry widgets for specifying the search criteria, such as the item name. The user can input the search criteria, and a "Search" button (btn_submit) is provided to initiate the search based on the entered criteria. Additionally, there is a "Return to Main Menu" button (btn_return) that allows users to close the search window and return to the main menu. The search button is associated with the search_item function, passing the entered search criteria as a parameter.

The search_item function, when invoked, attempts to execute a SQL query to retrieve item details from the MySQL database based on the provided search criteria. If the database connection (db) is established, it creates a cursor and executes the query. If the query returns results, the item details are displayed in a messagebox. If no results are found, a

message informs the user that the item was not found. In the event of a database error, an error message is displayed. Regardless of the outcome, the cursor is closed in the finally block, ensuring proper cleanup.

The else block at the end of the search_item function handles the scenario where there is a database connection error, showing an error message.



Outline:

- Item Searching:
    - Function `search_function()`: Placeholder for search functionality

Source Code:

```
def search_function():
# creates the window for searching
search_window = tk.Toplevel(window)
search_window.title("Search Item")
search_window.geometry('400x200')

# creates the textboxes and their labels
```

```
label_search_criteria = tk.Label(search_window, text="Item Name:")
label_search_criteria.grid(row=0, column=0, padx=10, pady=10)
entry_search_criteria = tk.Entry(search_window)
entry_search_criteria.grid(row=0, column=1, padx=10, pady=10)

# button for commiting search
btn_submit = tk.Button(search_window, text="Search", command=lambda:
search_item(entry_search_criteria.get()))
btn_submit.grid(row=1, column=0, columnspan=2, pady=10)

# Button to return to the main menu
btn_return = tk.Button(search_window, text="Return to Main Menu",
command=search_window.destroy)
btn_return.grid(row=2, column=0, columnspan=2, pady=10)

def search_item(search_criteria):
if db:
try:
cursor = db.cursor()
query = "SELECT * FROM Item WHERE Item_Name = %s"
cursor.execute(query, (search_criteria,))
item_details = cursor.fetchone()
# performs the query that retireves the search information from our
database

if item_details:
# Display item details in a messagebox
messagebox.showinfo("Item Details", f"Item ID: {item_details[0]}\n"
f"Item Name: {item_details[1]}\n"
f"Supplier ID: {item_details[2]}\n"
f"Height: {item_details[3]}\n"
f"Weight: {item_details[4]}\n"
f"Stored Time: {item_details[5]}\n"
f"Type of Item: {item_details[6]}\n"
f"Storage Condition: {item_details[7]}")
else:
messagebox.showinfo("", "Item not found.")
# Error handling
except mysql.connector.Error as err:
print(f"Error: {err}")
messagebox.showinfo("", "Failed to retrieve item details.")

finally:
cursor.close()
else:
messagebox.showinfo("", "Database connection error.")
```

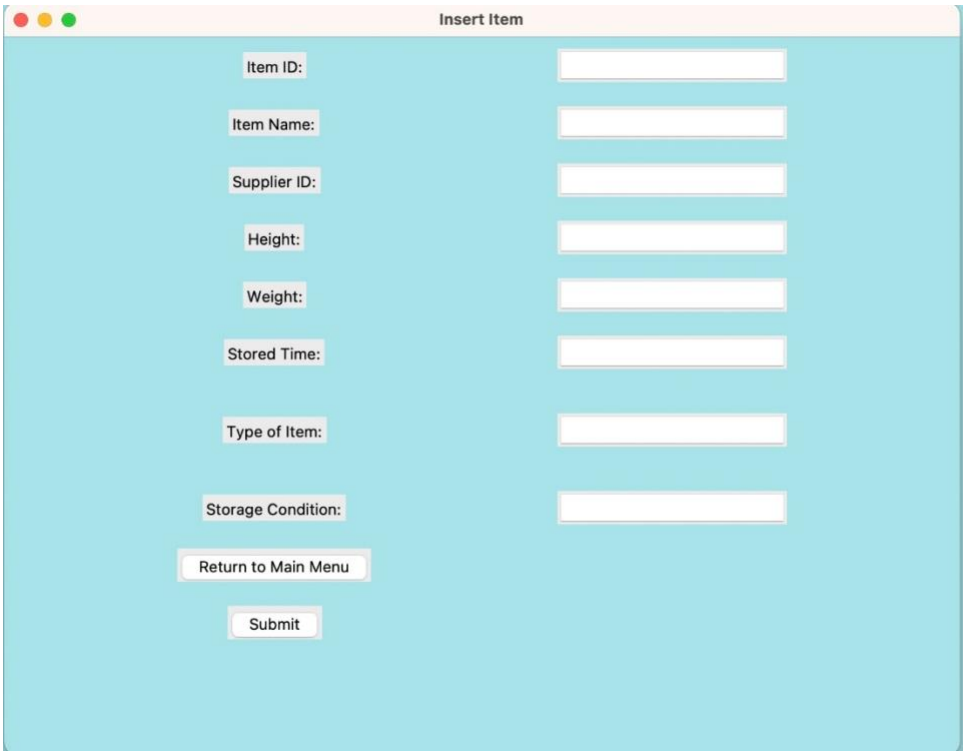
Insertion Page:

Description:

The insert_function creates a new top-level window (insert_window) within the Tkinter application specifically designed for inserting item details. The window is titled "Insert Item" and has a predefined geometry. It includes labels and entry widgets for various attributes of an item such as item ID, item name, supplier ID, height, weight, stored time, type of item, and storage condition. Buttons for "Return to Main Menu" and "Submit" are also provided at the bottom of the window. The "Return to Main Menu" button is associated with closing the insertion window, and the "Submit" button is associated with the submit_insert function, passing the entered values from the entry widgets as parameters.

The submit_insert function, when invoked by the "Submit" button, attempts to insert the provided item details into the MySQL database. It first checks if the database connection (db) is established. If so, it creates a cursor, prepares an SQL query for inserting the data into the "Item" table, and executes the query with the provided values. The changes are committed to the database, and a success message is displayed using messagebox.showinfo. In the event of a MySQL-related error, an error message is printed to the console, and a corresponding message box is displayed. The cursor is closed in the finally block to ensure proper cleanup.

The else block at the end of the submit_insert function handles the scenario where there is a database connection error, showing an error message.



Outline:

- Item Insertion:
    - Function `insert_function()`:
        - Create a window for item insertion with labels and entry fields
        - Button for submitting the insertion
        - Function `submit_insert()` to handle the database insertion

Source code:

```
def insert_function():
# Create a new window for item insertion
insert_window = tk.Toplevel(window)
insert_window.title("Insert Item")
insert_window.geometry('400x600')

# Label and Entry for item ID
label_item_id = tk.Label(insert_window, text="Item ID:")
label_item_id.grid(row=0, column=0, padx=10, pady=10)
entry_item_id = tk.Entry(insert_window)
```

```
entry_item_id.grid(row=0, column=1, padx=10, pady=10)

# Label and Entry for item name
label_item_name = tk.Label(insert_window, text="Item Name:")
label_item_name.grid(row=1, column=0, padx=10, pady=10)
entry_item_name = tk.Entry(insert_window)
entry_item_name.grid(row=1, column=1, padx=10, pady=10)

# Label and Entry for supplier ID
label_supplier_id = tk.Label(insert_window, text="Supplier ID:")
label_supplier_id.grid(row=2, column=0, padx=10, pady=10)
entry_supplier_id = tk.Entry(insert_window)
entry_supplier_id.grid(row=2, column=1, padx=10, pady=10)

# Label and Entry for height
label_height = tk.Label(insert_window, text="Height:")
label_height.grid(row=3, column=0, padx=10, pady=10)
entry_height = tk.Entry(insert_window)
entry_height.grid(row=3, column=1, padx=10, pady=10)


# Label and Entry for weight
label_weight = tk.Label(insert_window, text="Weight:")
label_weight.grid(row=4, column=0, padx=10, pady=10)
entry_weight = tk.Entry(insert_window)
entry_weight.grid(row=4, column=1, padx=10, pady=10)

# Label and Entry for stored time
label_stored_time = tk.Label(insert_window, text="Stored Time:")
label_stored_time.grid(row=5, column=0, padx=10, pady=10)
entry_stored_time = tk.Entry(insert_window)
entry_stored_time.grid(row=5, column=1, padx=10, pady=10)

# Label and Entry for type of item
label_type_of_item = tk.Label(insert_window, text="Type of Item:")
label_type_of_item.grid(row=6, column=0, padx=10, pady=10)
entry_type_of_item = tk.Entry(insert_window)
entry_type_of_item.grid(row=6, column=1, padx=10, pady=10)

# Label and Entry for storage condition
label_storage_condition = tk.Label(insert_window, text="Storage
Condition:")
label_storage_condition.grid(row=7, column=0, padx=10, pady=10)
entry_storage_condition = tk.Entry(insert_window)
entry_storage_condition.grid(row=7, column=1, padx=10, pady=10)

# Buttons for return to main menu and submit
```

```
btn_return = tk.Button(insert_window, text="Return to Main Menu",
command=insert_window.destroy)
btn_return.grid(row=8, column=0, columnspan=2, pady=10)

btn_submit = tk.Button(insert_window, text="Submit", command=lambda:
submit_insert(entry_item_id.get(), entry_item_name.get(),
entry_supplier_id.get(), entry_height.get(), entry_weight.get(),
entry_stored_time.get(), entry_type_of_item.get(),
entry_storage_condition.get()))
btn_submit.grid(row=9, column=0, columnspan=2, pady=10)

def submit_insert(item_id, item_name, supplier_id, height, weight,
stored_time, type_of_item, storage_condition):
# function is defined for actually inserting information into the database
if db:
try:
cursor = db.cursor()
query = "INSERT INTO Item (Item_ID, Item_Name, Supplier_ID, Height,
Weight, Stored_Time, type_of_item, Storage_Condition) VALUES (%s, %s, %s,
%s, %s, %s, %s, %s)"
values = (item_id, item_name, supplier_id, height, weight, stored_time,
type_of_item, storage_condition)
cursor.execute(query, values)
db.commit()
messagebox.showinfo("", "Item inserted successfully.")
#accesses the data provided from the user and creates an insert into query
for the database
#error handling
except mysql.connector.Error as err:
print(f"Error: {err}")
messagebox.showinfo("", "Failed to insert item.")

finally:
cursor.close()
else:
messagebox.showinfo("", "Database connection error.")
```

Removal Page:

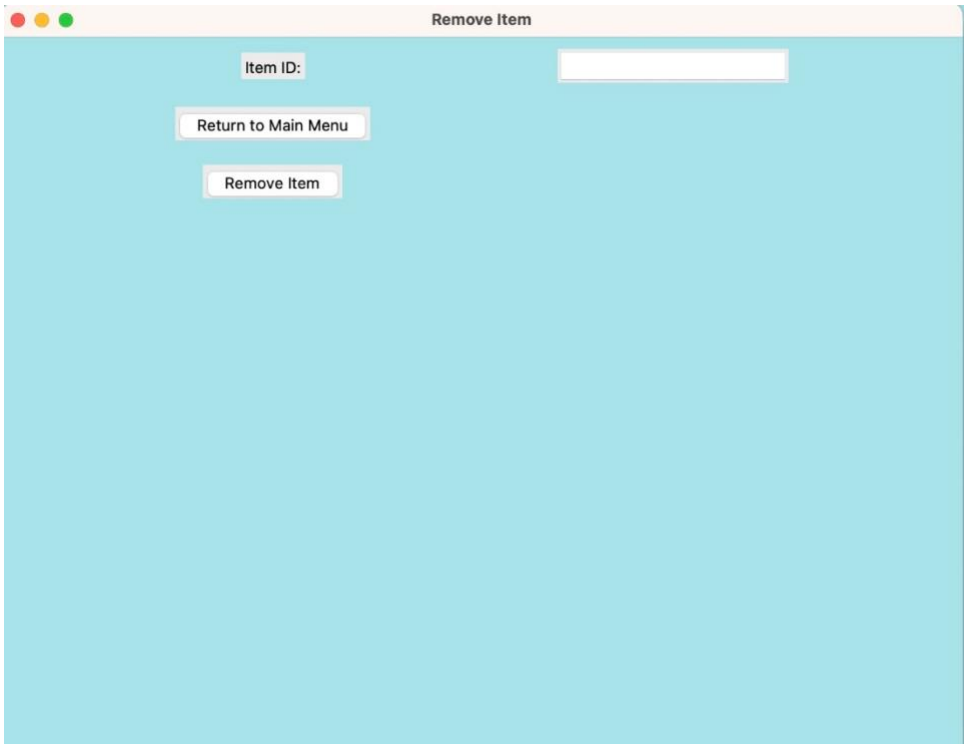
Description:

The remove_function creates a new top-level window (remove_window) within the Tkinter application specifically designed for removing items. The window is titled "Remove Item" and has a predefined geometry. It includes a label and an entry widget for specifying the item ID of the item to be removed. Buttons for "Return to Main Menu" and

"Remove Item" are also provided at the bottom of the window. The "Return to Main Menu" button is associated with closing the removal window, and the "Remove Item" button is associated with the submit_remove function, passing the entered item ID as a parameter.

The submit_remove function, when invoked by the "Remove Item" button, attempts to remove the specified item from the MySQL database. It first checks if the database connection (db) is established. If so, it creates a cursor, prepares an SQL query for deleting the item from the "Item" table, and executes the query with the provided item ID. The changes are committed to the database, and a success message is displayed using messagebox.showinfo. In the event of a MySQL-related error, an error message is printed to the console, and a corresponding message box is displayed. The cursor is closed in the finally block to ensure proper cleanup.

The else block at the end of the submit_remove function handles the scenario where there is a database connection error, showing an error message.



Outline:

- Item Removal:
    - Function `remove_function()`:
        - Create a window for item removal with labels and entry fields
        - Button for submitting the removal
        - Function `submit_remove()` to handle the database removal

Source code:

```
def remove_function():
# Create a new window for item removal
remove_window = tk.Toplevel(window)
remove_window.title("Remove Item")
remove_window.geometry('300x200')

# Uses item_ID to locate specific item for removal
label_item_id = tk.Label(remove_window, text="Item ID:")
label_item_id.grid(row=0, column=0, padx=10, pady=10)
entry_item_id = tk.Entry(remove_window)
entry_item_id.grid(row=0, column=1, padx=10, pady=10)

# Buttons for return to main menu and remove
btn_return = tk.Button(remove_window, text="Return to Main Menu",
command=remove_window.destroy)
btn_return.grid(row=1, column=0, columnspan=2, pady=10)

btn_remove = tk.Button(remove_window, text="Remove Item",
command=lambda: submit_remove(entry_item_id.get()))
btn_remove.grid(row=2, column=0, columnspan=2, pady=10)

def submit_remove(item_id):
# Function is defined and created for the submit removal button
if db:
try:
cursor = db.cursor()
query = "DELETE FROM Item WHERE Item_ID = %s"
cursor.execute(query, (item_id,))
db.commit()
messagebox.showinfo("", "Item removed successfully.")
```

```
#Uses the item ID provided from the user to locate the item it coorelates
with and remove it
except mysql.connector.Error as err:
print(f"Error: {err}")
messagebox.showinfo("", "Failed to remove item.")
#error handling
finally:
cursor.close()
else:
messagebox.showinfo("", "Database connection error.")
```

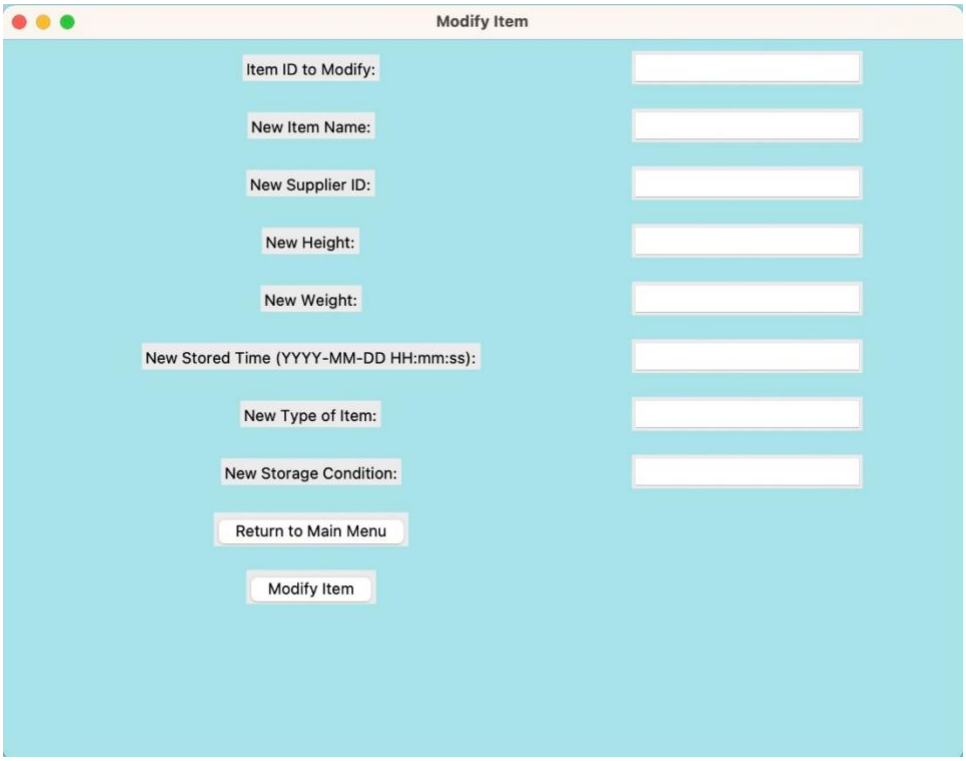
Modification Page:

Description:

The modify_function creates a new top-level window (modify_item_window) within the Tkinter application specifically designed for modifying items. The window is titled "Modify Item" and has a predefined geometry. It includes labels and entry widgets for specifying the item ID to be modified and new values for various attributes such as item name, supplier ID, height, weight, stored time, type of item, and storage condition. Buttons for "Return to Main Menu" and "Modify Item" are also provided at the bottom of the window. The "Return to Main Menu" button is associated with closing the modification window, and the "Modify Item" button is associated with the submit_modify_item function, passing the entered values from the entry widgets as parameters.

The submit_modify_item function, when invoked by the "Modify Item" button, attempts to modify the specified item in the MySQL database. It first checks if the database connection (db) is established. If so, it creates a cursor, prepares an SQL query for updating the data in the "Item" table based on the provided item ID, and executes the query with the new values. The changes are committed to the database, and a success message is displayed using messagebox.showinfo. In the event of a MySQL-related error, an error message is printed to the console, and a corresponding message box is displayed. The cursor is closed in the finally block to ensure proper cleanup.

The else block at the end of the submit_modify_item function handles the scenario where there is a database connection error, showing an error message.



Outline:

- Item Modification:
  - Function `modify_function()`:
    - Create a window for item modification with labels and entry fields
    - Button for submitting the modification
    - Function `submit_modify_item()` to handle the database modification

Source code:

```
def modify_function():
# Create a new window for item modification
modify_item_window = tk.Toplevel(window)
modify_item_window.title("Modify Item")
modify_item_window.geometry('600x600')
```

```
# Label and Entry for item ID to be modified
label_item_id_modify = tk.Label(modify_item_window, text="Item ID to
Modify:")
label_item_id_modify.grid(row=0, column=0, padx=10, pady=10)
entry_item_id_modify = tk.Entry(modify_item_window)
entry_item_id_modify.grid(row=0, column=1, padx=10, pady=10)

# Label and Entry for new item name
label_new_item_name = tk.Label(modify_item_window, text="New Item
Name:")
label_new_item_name.grid(row=1, column=0, padx=10, pady=10)
entry_new_item_name = tk.Entry(modify_item_window)
entry_new_item_name.grid(row=1, column=1, padx=10, pady=10)

# Label and Entry for new supplier ID
label_new_supplier_id = tk.Label(modify_item_window, text="New Supplier
ID:")
label_new_supplier_id.grid(row=2, column=0, padx=10, pady=10)
entry_new_supplier_id = tk.Entry(modify_item_window)
entry_new_supplier_id.grid(row=2, column=1, padx=10, pady=10)

# Label and Entry for new height
label_new_height = tk.Label(modify_item_window, text="New Height:")
label_new_height.grid(row=3, column=0, padx=10, pady=10)
entry_new_height = tk.Entry(modify_item_window)
entry_new_height.grid(row=3, column=1, padx=10, pady=10)

# Label and Entry for new weight
label_new_weight = tk.Label(modify_item_window, text="New Weight:")
label_new_weight.grid(row=4, column=0, padx=10, pady=10)
entry_new_weight = tk.Entry(modify_item_window)
entry_new_weight.grid(row=4, column=1, padx=10, pady=10)

# Label and Entry for new stored time
label_new_stored_time = tk.Label(modify_item_window, text="New Stored
Time (YYYY-MM-DD HH:mm:ss):")
label_new_stored_time.grid(row=5, column=0, padx=10, pady=10)
entry_new_stored_time = tk.Entry(modify_item_window)
entry_new_stored_time.grid(row=5, column=1, padx=10, pady=10)

# Label and Entry for new type of item
label_new_type_of_item = tk.Label(modify_item_window, text="New Type of
Item:")
label_new_type_of_item.grid(row=6, column=0, padx=10, pady=10)
entry_new_type_of_item = tk.Entry(modify_item_window)
entry_new_type_of_item.grid(row=6, column=1, padx=10, pady=10)
```

```
# Label and Entry for new storage condition
label_new_storage_condition = tk.Label(modify_item_window, text="New
Storage Condition:")
label_new_storage_condition.grid(row=7, column=0, padx=10, pady=10)
entry_new_storage_condition = tk.Entry(modify_item_window)
entry_new_storage_condition.grid(row=7, column=1, padx=10, pady=10)

# Buttons for return to main menu and modify
btn_return_modify = tk.Button(modify_item_window, text="Return to Main
Menu", command=modify_item_window.destroy)
btn_return_modify.grid(row=8, column=0, columnspan=2, pady=10)

btn_modify = tk.Button(modify_item_window, text="Modify Item",
command=lambda: submit_modify_item(entry_item_id_modify.get(),
entry_new_item_name.get(), entry_new_supplier_id.get(),
entry_new_height.get(), entry_new_weight.get(),
entry_new_stored_time.get(), entry_new_type_of_item.get(),
entry_new_storage_condition.get()))
btn_modify.grid(row=9, column=0, columnspan=2, pady=10)

def submit_modify_item(item_id_modify, new_item_name, new_supplier_id,
new_height, new_weight, new_stored_time, new_type_of_item,
new_storage_condition):
# creates a function for the modfication submission, and utilizes the info
provided from the user as constructors
if db:
try:
cursor = db.cursor()
query = "UPDATE Item SET Item_Name = %s, Supplier_ID = %s, Height = %s,
Weight = %s, Stored_Time = %s, type_of_item = %s, Storage_Condition = %s
WHERE Item_ID = %s"
values = (new_item_name, new_supplier_id, new_height, new_weight,
new_stored_time, new_type_of_item, new_storage_condition,
item_id_modify)
cursor.execute(query, values)
db.commit()
messagebox.showinfo("", "Item modified successfully.")
# Takes the info provided by the user and modfies the item and its qualities
based off the user input
except mysql.connector.Error as err:
print(f"Error: {err}")
messagebox.showinfo("", "Failed to modify item.")
#error handling
finally:
cursor.close()
else:
messagebox.showinfo("", "Database connection error.")
```

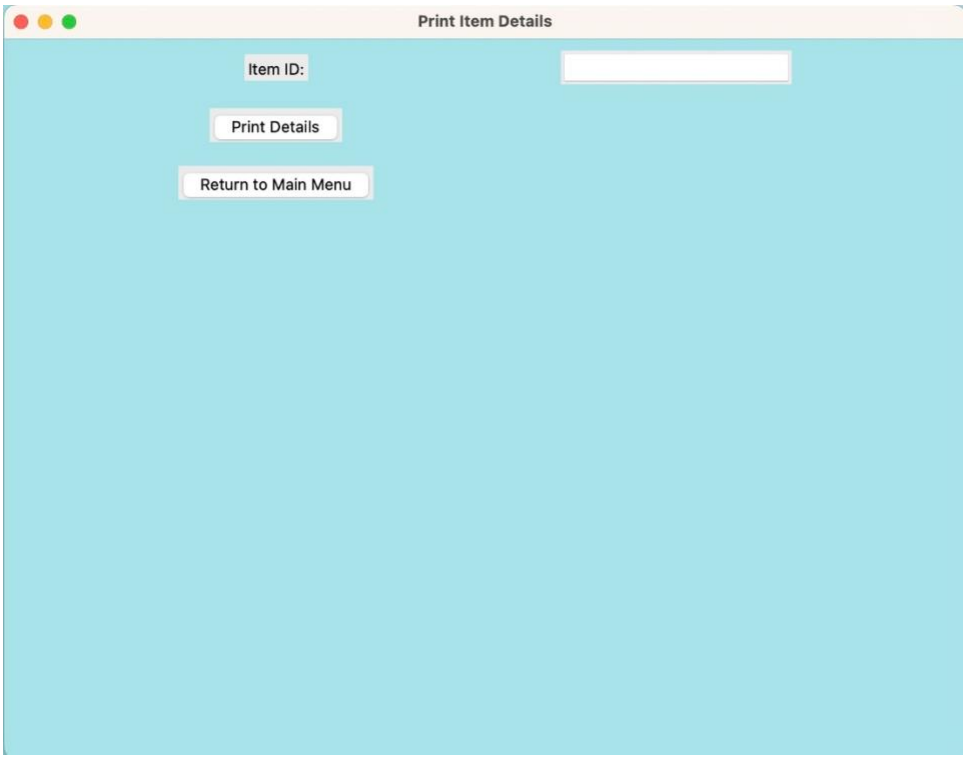
Print All Data:

Description:

The print_item_details function creates a new top-level window (print_window) within the Tkinter application specifically designed for printing item details. The window is titled "Print Item Details" and has a predefined geometry. It includes a label and an entry widget for specifying the item ID of the item whose details should be printed. Buttons for "Print Details" and "Return to Main Menu" are also provided at the bottom of the window. The "Print Details" button is associated with the print_details function, passing the entered item ID as a parameter, and the "Return to Main Menu" button is associated with closing the print window.

The print_details function, when invoked by the "Print Details" button, attempts to retrieve and display the details of the specified item from the MySQL database. It first checks if the database connection (db) is established. If so, it creates a cursor, prepares an SQL query for selecting item details based on the provided item ID, and executes the query. If the query returns results, the item details are displayed in a messagebox using messagebox.showinfo. If no results are found, a message informs the user that the item was not found. In the event of a MySQL-related error, an error message is printed to the console, and a corresponding message box is displayed. The cursor is closed in the finally block to ensure proper cleanup.

The else block at the end of the print_details function handles the scenario where there is a database connection error, showing an error message.



Outline:

- Item Printing:
    - Function `print_item_details()`:
        - Create a window for printing item details with labels and entry fields
        - Button for submitting and printing item details
        - Function `print_details()` to retrieve item details from the database

Source Code:

```
def print_item_details():

# Create a new window for printing item details
print_window = tk.Toplevel(window)
print_window.title("Print Item Details")
print_window.geometry('400x300')

# Label and Entry for item ID
label_item_id = tk.Label(print_window, text="Item ID:")
```

```
label_item_id.grid(row=0, column=0, padx=10, pady=10)
entry_item_id = tk.Entry(print_window)
entry_item_id.grid(row=0, column=1, padx=10, pady=10)

# Button for submitting and printing item details
btn_submit = tk.Button(print_window, text="Print Details",
command=lambda: print_details(entry_item_id.get()))
btn_submit.grid(row=1, column=0, columnspan=2, pady=10)

# Button to return to the main menu
btn_return = tk.Button(print_window, text="Return to Main Menu",
command=print_window.destroy)
btn_return.grid(row=2, column=0, columnspan=2, pady=10)

def print_details(item_id):\
# creates a function that outlines the functionality for the print button
if db:
try:
cursor = db.cursor()
query = "SELECT * FROM Item WHERE Item_ID = %s"
cursor.execute(query, (item_id,))
item_details = cursor.fetchone()
# Uses the ID provided by the user to fetch item details then print them
if item_details:
# Display item details in a messagebox
messagebox.showinfo("Item Details", f"Item ID: {item_details[0]}\n"
f"Item Name: {item_details[1]}\n"
f"Supplier ID: {item_details[2]}\n"
f"Height: {item_details[3]}\n"
f"Weight: {item_details[4]}\n"
f"Stored Time: {item_details[5]}\n"
f"Type of Item: {item_details[6]}\n"
f"Storage Condition: {item_details[7]}")
else:
messagebox.showinfo("", "Item not found.")

except mysql.connector.Error as err:
print(f"Error: {err}")
messagebox.showinfo("", "Failed to retrieve item details.")
#Error handling
finally:
cursor.close()
else:
messagebox.showinfo("", "Database connection error.")
```

About Us Page:

Description:

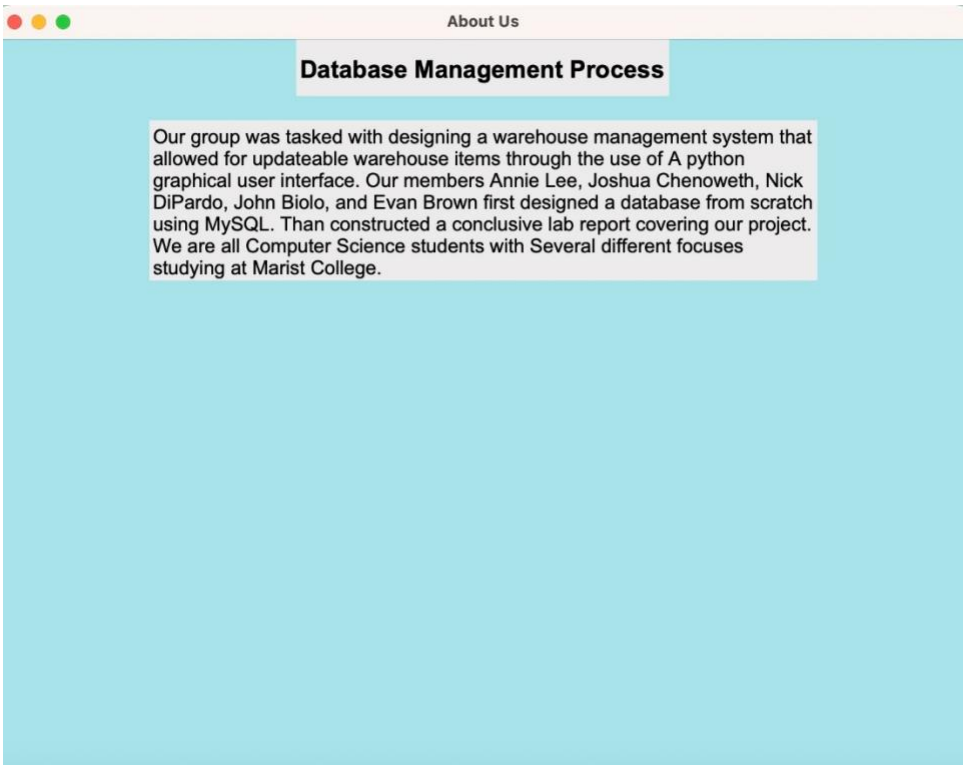
A new top-level window (about_window) is created using tk.Toplevel(window) to serve as the container for the “About Us”

information. The function adds a Label widget (label_title) to the about_window. This label serves as a large title for the page, displaying "Database Management Process." The title is styled with a larger font size (18) and bold text.

Another Label widget (label_about) is added to the window, displaying a paragraph of text (about_text) that provides information about the project. This paragraph includes details about the purpose of the system, its functionalities, and its user-friendly design. The text is styled with a slightly smaller font size (14) and is set to wrap within the specified width (550 pixels) with left justification.

The about_us function can be triggered by a button in the main

menu (btn_about_us). This button is created in the main menu frame and is associated with the about_us function. Clicking the button opens the "About Us" window, displaying the project details.



Outline:

- Function about_us():
    - Creates a window for the functions capabilities.
    - Uses labels to create a title and styling for the paragraph and text.
    - Contains a pragraph and displays the information regarding our group and the creation of our project.

Source Code:

```
def about_us():
    about_window = tk.Toplevel(window)
    about_window.title("About Us")
    about_window.geometry('800x600')
    about_window.configure(bg='#a6e3e9')

    # Large title
    label_title = tk.Label(about_window, text="Database Management
    Process", font=("Arial", 20, "bold"), pady=10)
    label_title.pack()

    about_text = (
```

```
"Our group was tasked with designing a warehouse management system
that allowed for updateable warehouse items through the use of "
"A python graphical user interface. Our members Annie Lee, Joshua
Chenoweth, Nick DiPardo, John Biolo, and Evan Brown first designed
a database"
" from scratch using MySQL. Than constructed a conclusive lab          report
covering our project. We are all Computer Science students    with"
" Several different focuses studying at Marist College."
# Add more details about your project as needed
                                                                    )

# Larger text
label_about = tk.Label(about_window, text=about_text, font=
("Arial", 16), wraplength=550, justify=tk.LEFT)
label_about.pack(padx=20, pady=20)
```

User Admin Handling + Guest Password Editing:

Admin Handling:

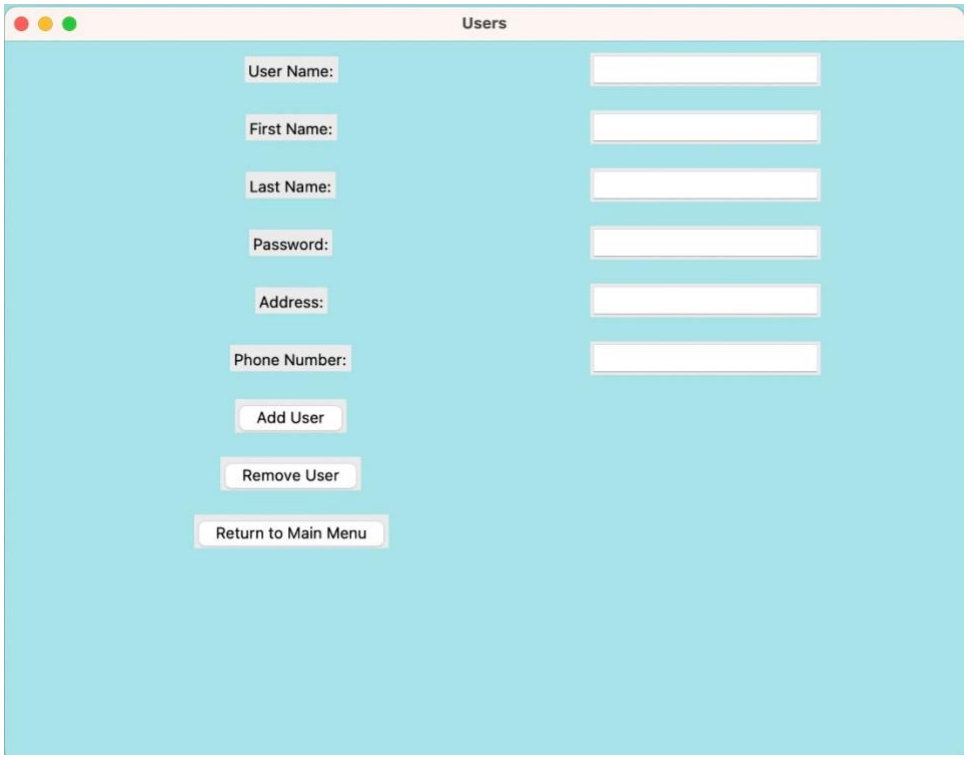
Description:

The user_admin_function function creates a new top-level window (guest_window) within the Tkinter application for guest user functions. This window is titled "Users" and has a specific geometry. It includes labels and entry widgets for user details such as user name, first name, last name, password, address, and phone number. Buttons for adding and removing users are provided, along with a button to return to the main menu. The functions add_guest_user and remove_guest_user handle the addition and removal of guest users, respectively. These functions interact with the MySQL database, executing SQL queries to insert or delete user information.

Similarly, the admin_user_function function creates a new top-level window (admin_window) for admin user functions. This window is titled "Admin User Functions" and has a specific geometry. It includes labels and entry widgets for admin user details such as user name, email, first name, last name, password, and recovery email. Buttons for adding and removing admin users are provided, along with a button to return to the main menu. The functions add_admin_user and remove_admin_user

handle the addition and removal of admin users, respectively. These functions interact with the MySQL database, executing SQL queries to insert or delete admin user information.

The else block at the end of the print_details function handles the scenario where there is a database connection error, showing an error message.



Outline:

User Administration (Guest):

- Function `user_admin_function()`:
  - Create a window for guest user functions with labels and entry fields
  - Buttons for adding and removing guest users
  - Functions `add_guest_user()` and `remove_guest_user()` to handle guest user operations

Admin User Functions:

- Placeholder functions for admin user functions:

- `admin_user_function()`:
    - Create a window for admin user functions
    - Buttons for adding and removing admin users
    - Functions `add_admin_user()` and `remove_admin_user()` to handle admin user operations

Source Code:

```
def user_admin_function():
# Create a new window for guest user functions
guest_window = tk.Toplevel(window)
guest_window.title("Users")
guest_window.geometry('400x600')

# Label and Entry for user details
label_user_name = tk.Label(guest_window, text="User Name:")
label_user_name.grid(row=0, column=0, padx=10, pady=10)
entry_user_name = tk.Entry(guest_window)
entry_user_name.grid(row=0, column=1, padx=10, pady=10)

label_fname = tk.Label(guest_window, text="First Name:")
label_fname.grid(row=1, column=0, padx=10, pady=10)
entry_fname = tk.Entry(guest_window)
entry_fname.grid(row=1, column=1, padx=10, pady=10)

label_lname = tk.Label(guest_window, text="Last Name:")
label_lname.grid(row=2, column=0, padx=10, pady=10)
entry_lname = tk.Entry(guest_window)
entry_lname.grid(row=2, column=1, padx=10, pady=10)

label_password = tk.Label(guest_window, text="Password:")
label_password.grid(row=3, column=0, padx=10, pady=10)
entry_password = tk.Entry(guest_window, show="*")
entry_password.grid(row=3, column=1, padx=10, pady=10)

label_address = tk.Label(guest_window, text="Address:")
label_address.grid(row=4, column=0, padx=10, pady=10)
entry_address = tk.Entry(guest_window)
entry_address.grid(row=4, column=1, padx=10, pady=10)

label_phone_number = tk.Label(guest_window, text="Phone Number:")
label_phone_number.grid(row=5, column=0, padx=10, pady=10)
entry_phone_number = tk.Entry(guest_window)
```

```
entry_phone_number.grid(row=5, column=1, padx=10, pady=10)

# Buttons for adding and removing users
btn_add_user = tk.Button(guest_window, text="Add User",
command=lambda: add_guest_user(
entry_user_name.get(), entry_fname.get(), entry_lname.get(),
entry_password.get(), entry_address.get(), entry_phone_number.get())
)
btn_add_user.grid(row=6, column=0, columnspan=2, pady=10)

btn_remove_user = tk.Button(guest_window, text="Remove User",
command=lambda: remove_guest_user(entry_user_name.get()))
btn_remove_user.grid(row=7, column=0, columnspan=2, pady=10)

# Button to return to the main menu
btn_return = tk.Button(guest_window, text="Return to Main Menu",
command=guest_window.destroy)
btn_return.grid(row=8, column=0, columnspan=2, pady=10)

def add_guest_user(user_name, fname, lname, password, address,
phone_number):
# Provides the functionality for the add guest user function
if db:
try:
cursor = db.cursor()
query = "INSERT INTO Guest_User (Guest_User_Name, Guest_fname,
Guest_lname, Guest_password, Guest_address, Guest_Phone_Number)
VALUES (%s, %s, %s, %s, %s, %s)"
cursor.execute(query, (user_name, fname, lname, password, address,
phone_number))
db.commit()
messagebox.showinfo("", "Guest User added successfully.")
# Gathers the guest user information given by the user and inserts a new
user into the guest_User table
except mysql.connector.Error as err:
print(f"Error: {err}")
messagebox.showinfo("", "Failed to add Guest User.")
#Error handling
finally:
cursor.close()
else:
messagebox.showinfo("", "Database connection error.")

def remove_guest_user(user_name):
#Provides the functionality for the removal of a guest user
if db:
try:
cursor = db.cursor()
```

```
query = "DELETE FROM Guest_User WHERE Guest_User_Name = %s"
cursor.execute(query, (user_name,))
db.commit()
messagebox.showinfo("", "Guest User removed successfully.")
# Does the same thing with the user information, just uses delete instead of
insert into
except mysql.connector.Error as err:
print(f"Error: {err}")
messagebox.showinfo("", "Failed to remove Guest User.")
# Error Handling
finally:
cursor.close()
else:
messagebox.showinfo("", "Database connection error.")

def admin_user_function():
# Creates a new window for admin user manipulaiton
admin_window = tk.Toplevel(window)
admin_window.title("Admin User Functions")
admin_window.geometry('400x300')

# Label and Entry for user details
label_user_name = tk.Label(admin_window, text="Admin User Name:")
label_user_name.grid(row=0, column=0, padx=10, pady=10)
entry_user_name = tk.Entry(admin_window)
entry_user_name.grid(row=0, column=1, padx=10, pady=10)

label_email = tk.Label(admin_window, text="Admin Email:")
label_email.grid(row=1, column=0, padx=10, pady=10)
entry_email = tk.Entry(admin_window)
entry_email.grid(row=1, column=1, padx=10, pady=10)

label_fname = tk.Label(admin_window, text="First Name:")
label_fname.grid(row=2, column=0, padx=10, pady=10)
entry_fname = tk.Entry(admin_window)
entry_fname.grid(row=2, column=1, padx=10, pady=10)

label_lname = tk.Label(admin_window, text="Last Name:")
label_lname.grid(row=3, column=0, padx=10, pady=10)
entry_lname = tk.Entry(admin_window)
entry_lname.grid(row=3, column=1, padx=10, pady=10)

label_password = tk.Label(admin_window, text="Password:")
label_password.grid(row=4, column=0, padx=10, pady=10)
entry_password = tk.Entry(admin_window, show="*")
entry_password.grid(row=4, column=1, padx=10, pady=10)

label_recovery_email = tk.Label(admin_window, text="Recovery Email:")
```

```
label_recovery_email.grid(row=5, column=0, padx=10, pady=10)
entry_recovery_email = tk.Entry(admin_window)
entry_recovery_email.grid(row=5, column=1, padx=10, pady=10)

# Buttons for adding and removing users
btn_add_user = tk.Button(admin_window, text="Add User",
command=lambda: add_admin_user(
entry_user_name.get(), entry_email.get(), entry_fname.get(),
entry_lname.get(), entry_password.get(), entry_recovery_email.get())
)
btn_add_user.grid(row=6, column=0, columnspan=2, pady=10)

btn_remove_user = tk.Button(admin_window, text="Remove User",
command=lambda: remove_admin_user(entry_user_name.get()))
btn_remove_user.grid(row=7, column=0, columnspan=2, pady=10)

# Button to return to the main menu
btn_return = tk.Button(admin_window, text="Return to Main Menu",
command=admin_window.destroy)
btn_return.grid(row=8, column=0, columnspan=2, pady=10)

def add_admin_user(user_name, email, fname, lname, password,
recovery_email):
#Does the same thing as the guest user functions just manipultes the admin
user table instead of the guest user table
if db:
try:
cursor = db.cursor()
query = "INSERT INTO Admin_User (Admin_User_Name, Admin_Email,
Admin_fname, Admin_lname, Admin_Password, Admin_Recovery_Email)
VALUES (%s, %s, %s, %s, %s, %s)"
cursor.execute(query, (user_name, email, fname, lname, password,
recovery_email))
db.commit()
messagebox.showinfo("", "Admin User added successfully.")

except mysql.connector.Error as err:
print(f"Error: {err}")
messagebox.showinfo("", "Failed to add Admin User.")

finally:
cursor.close()
else:
messagebox.showinfo("", "Database connection error.")

def remove_admin_user(user_name):
#Does the same thing as the guest user functions just manipultes the admin
user table instead of the guest user table
```

```
if db:
try:
cursor = db.cursor()
query = "DELETE FROM Admin_User WHERE Admin_User_Name = %s"
cursor.execute(query, (user_name,))
db.commit()
messagebox.showinfo("", "Admin User removed successfully.")

except mysql.connector.Error as err:
print(f"Error: {err}")
messagebox.showinfo("", "Failed to remove Admin User.")

finally:
cursor.close()
else:
messagebox.showinfo("", "Database connection error.")
```

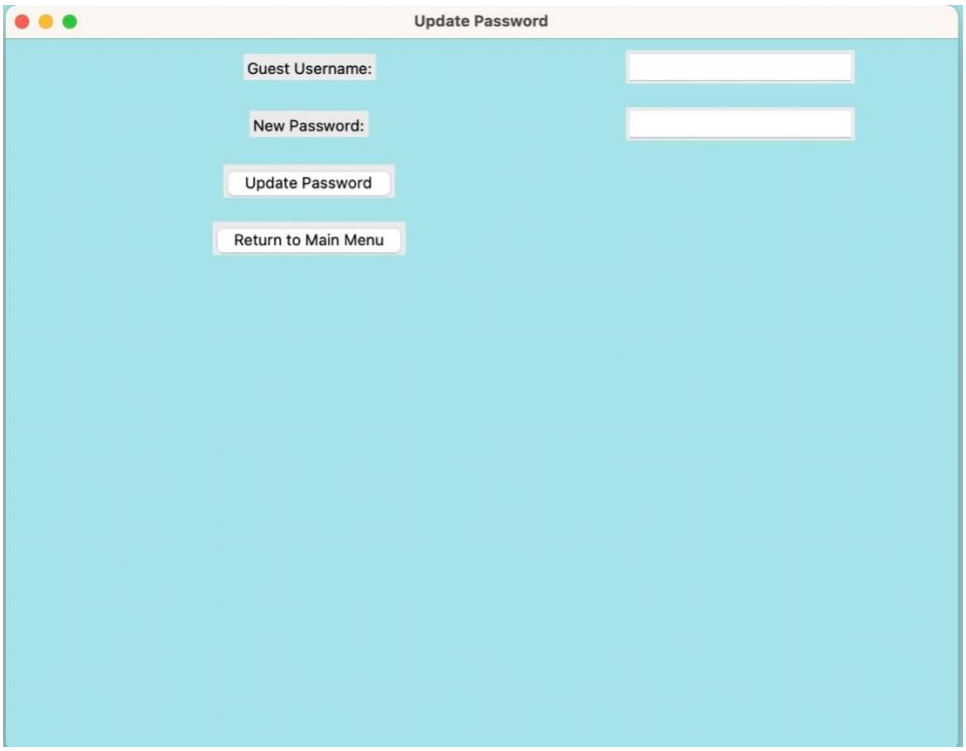
Guest Password Updates:

Description:

The update_password_function function creates a new top-level window (update_window) within the Tkinter application titled "Update Password" with a specific geometry. It includes labels and entry widgets for the guest username and the new password. A button (btn_update_password) is provided to trigger the password update. The command parameter of the button is set to a lambda function that calls the update_guest_password function with the values from the entry widgets.

The update_guest_password function is responsible for updating the password of a guest user in the MySQL database. It checks if the guest user exists by executing a SELECT query on the Guest_User table with the provided username. If the user exists, it proceeds to update the password using an UPDATE query. The db.commit() statement ensures that the changes are committed to the database.

The else block at the end of the print_details function handles the scenario where there is a database connection error, showing an error message.



Outline:

- Function `update_password_function`:
    - Create Tkinter window (`update_window`) for updating guest user passwords
    - Add labels and entry widgets for guest username and new password
    - Create button (`btn_update_password`) to trigger password update
    - Set the `command` parameter of button to lambda function calling `update_guest_password`
- Function `update_guest_password`:
    - Check if database connection (`db`) exists
    - - Create database cursor
    - - Execute SELECT query to check if guest user exists
    - - User exists:
        - - Execute UPDATE query to update the guest user's password

- - Commit the changes to the database
- - Show success message using `messagebox.showinfo’
-

Source Code:

```
def update_password_function():
##his function creates the entry buttons and labels for the guest users password manipulation
update_window = tk.Toplevel(window)
update_window.title("Update Password")
update_window.geometry('300x150')

label_username = tk.Label(update_window, text="Username:")
label_username.grid(row=0, column=0, padx=10, pady=10)

entry_username = tk.Entry(update_window)
entry_username.grid(row=0, column=1, padx=10, pady=10)

label_password = tk.Label(update_window, text="New Password:")
label_password.grid(row=1, column=0, padx=10, pady=10)

entry_password = tk.Entry(update_window, show="*")
entry_password.grid(row=1, column=1, padx=10, pady=10)

btn_update_password = tk.Button(update_window, text="Update Password",
command=lambda: update_guest_password(entry_username.get(),
entry_password.get()))
btn_update_password.grid(row=2, column=0, columnspan=2, pady=10)

def update_guest_password(guest_user_name, new_password):
# This funciton provides the actual functionality for the guest user password manipulation
if db:
try:
cursor = db.cursor()

# Check if the guest user exists
cursor.execute("SELECT * FROM Guest_User WHERE Guest_User_Name =
%s", (guest_user_name,))
user = cursor.fetchone()

if user:
# Update the password for the guest user
cursor.execute("UPDATE Guest_User SET Guest_Password = %s WHERE
Guest_User_Name = %s", (new_password, guest_user_name))
```

```
db.commit()
messagebox.showinfo("Success", "Password updated successfully!")
else:
messagebox.showerror("Error", "Guest user not found.")

except mysql.connector.Error as err:
print(f"Error: {err}")
messagebox.showerror("Error", f"Database error: {err}")

finally:
cursor.close()
else:
messagebox.showerror("Error", "Database connection error.")
```

Conclusion & Future Work: (13)

Taking everything we have done into consideration, we have learned how to create and manage a warehouse type database and apply our coding skills into Python to make a functional GUI to store certain information needed for our project. To fulfill a warehouse DBMS, models to create an outline of the database will help what kind of information to collect and store. And by researching on database data types and creating models to build the structure of our database, this has helped us decide what entities and attributes we needed in order to organize our warehouse. These two major areas will then help build the actual database, where we have stored the code in MySQL. Using this database, we can check for the different stages of normalization to reduce error checks in our tables. Having our database now complete, this help us create our Python GUI. Originally, this project was supposed to project how our database will be organized to store our desired information. But to be able to implement our interface on a GUI gives a sense of how our system of managing a warehouse would work in reality. Understanding the code building for our warehouse system helped us create a GUI where we can insert, edit, and delete information contained in our database

Some improvements we could have worked on throughout this project was to create a professional design for our GUI. In terms of time management and the broad topic of GUI building, we feel that the functionality of the GUI was not created to its fullest, therefore, not as complete in terms of looks. In short, the interface could have been made more user-friendly. But despite the simplicity, we built a system that was all according to our previous work and research. Another area where we could have improved on was our ER models. There could have been a better organization of our attributes and entities in our system. The names we chose for our entities and attributes could have been more concise (maybe abbreviated).

References: (14)

- https://www.brwilliams.com/blog/best-warehouse-management-system-features/ (7)
- https://www.thefulfillmentlab.com/blog/benefits-of-a-warehouse-management-system-choosing-wms-software#:~:text=Warehouse%20Management%20Systems%20%26%20Your%20Supply%20Chain&text=A%20WMS%20keeps%20the%20whole,before%20they%20become%20major%20issues (6)
- https://shiphero.com/ (1)
- https://www.netsuite.com/portal/home.shtml (2)
- Extensiv.com/3PL-demo (3)
- https://dev.mysql.com/doc/refman/8.0/en/data-types.html (4)
- https://press.rebus.community/programmingfundamentals/chapter/string-data-type/ (5)
- https://www.cockroachlabs.com/blog/spatial-data-types/#:~:text=The%20two%20primary%20spatial%20data,of%20an%20object%20on%20earth (6)