Metal works

Session: 2024-2028

Submitted by:

Moeen Zubair 2024-DS-26

Supervised by:

Dr. Faiza Iqbal

Mr. Sharjeel Tariq

Course:

Database Systems

Institute of Data Science
University Of Engineering and Technology,
Lahore, Pakistan

Contents

1 Introduction

1.1 Project Overview

Metalworks is a web-based Safety and Sector Management System designed to help small and medium-sized factories manage their operations efficiently. The project features a fully functional website with separate dashboards for customers and administrators. Customers can browse products, update their profiles, and provide feedback, while administrators can add or edit products and manage employees. All operations are handled using a Flask backend, MySQL database, and a frontend built with HTML, CSS, JavaScript, and Tailwind. Additionally, the system connects MySQL data to Power BI for real-time business analysis, enabling factory owners to monitor inventory, employee activities, and customer feedback through visual dashboards. Metalworks provides a strong foundation for factories to shift from offline to digital workflows.

1.2 Scope and Objectives

The scope Metalworks focuses on digitalizing the core operations of small to medium-sized factories that are currently operating offline. The system provides functionalities such as product management, employee management, customer account management, and feedback collection through an easy-to-use website. It also includes backend data storage using MySQL and advanced data visualization through Power BI, ensuring that factory owners can monitor and analyze their operations in real-time. In the future, the system aims to expand by adding modules for employee attendance tracking, custom order placement, and order assignment to employees.

Objectives:

- To provide a digital platform for factories to manage products, employees, and customer data.
- To integrate real-time data storage and retrieval using Flask and MySQL.
- To enable customers to browse products, update profiles, and submit feedback easily.
- To allow administrators to add, edit, and manage products and employees efficiently.
- To connect the database to Power BI for professional-level data analysis and reporting.
- To prepare small and medium factories for the digital shift required in the era of AI and smart technologies.

1.3 Key Features

- **User-Friendly Website:**
  A clean, responsive website developed using HTML, CSS, JavaScript, and Tailwind for an enhanced user experience.
- **Customer Dashboard:**
  Customers can browse products, update their username and password, view services, and provide feedback after checkout.
- **Admin Dashboard:**
  Admins can add new products, edit existing ones, manage employee records, and oversee inventory management. Changes made by the admin instantly reflect on the home page.
- **Dynamic Product Management:**
  New products added by the admin are automatically saved to the MySQL database and displayed on the live website.
- **Feedback System:**
  Customer feedback collected at checkout is displayed on the home page for new visitors to view, improving transparency and trust.
- **Flask and MySQL Integration:**
  Backend built using Flask and Python OOP principles, ensuring secure data handling and smooth communication with the MySQL database.
- **Power BI Data Visualization:**
  Real-time data is extracted from MySQL Workbench and analyzed in Power BI dashboards for both business insights and safety management.
- **Future Expansion Ready:**
  The system is designed to be expandable with additional features like employee attendance tracking, custom order management, and employee order assignment.

2 Technology Stack

Frontend Technologies

- **HTML5:**
  Used to create the structure and content of the web pages. It ensures that the website is well-organized, semantic, and easily accessible across different devices.
- **CSS3:**
  Applied for designing and styling the web pages, giving a professional and clean appearance to the platform. CSS3 is also used to make the website responsive and visually appealing.
- **JavaScript:**
  JavaScript adds interactivity to the website, allowing dynamic changes without needing to reload pages. Features like updating user profiles or handling product views are managed using JavaScript.
- **Tailwind CSS:**
  A utility-first CSS framework that allowed rapid UI development with a modern, responsive, and mobile-friendly design. Tailwind made it easy to implement consistent styling across all components of the platform.

Backend Technologies

- **Python (Flask Framework):**
  Flask, a lightweight Python web framework, was used to handle server-side development. It manages all the backend operations such as user authentication, product management, employee data handling, and routing between different web pages.
- **OOP (Object-Oriented Programming):**
  The backend code follows OOP principles to organize the application into manageable classes and objects. This structure improves the reusability, maintainability, and scalability of the code.
- **Jinja2 Templating Engine:**
  Jinja2 is integrated with Flask to render dynamic HTML templates. It allows the server to pass data to the frontend easily and ensures a seamless user experience by embedding server-side variables inside the HTML.

Database Technologies

- **MySQL Workbench:**
  MySQL is used as the relational database to store all application data, including product details, customer accounts, employee records, and customer feedback.
  - Real-time operations like inserting, updating, and deleting records are performed using Flask-MySQL integration.
  - The database structure ensures data integrity and allows fast retrieval for the website and dashboards.

Business Intelligence and Reporting

- **Power BI:**

Power BI is integrated into the project to visualize operational data from MySQL.

  - It is used to create **dynamic dashboards** that provide insights into inventory management, customer feedback trends, employee records, and business performance.
  - Data from MySQL Workbench is connected to Power BI using direct connectors, allowing real-time data refresh.
  - Visualizations like pie charts, bar graphs, tables, and trend lines are used to help factory owners and administrators make **informed decisions** based on the latest data.
  - Power BI helps small to medium factories easily understand their business health without needing advanced technical knowledge.

Other Supporting Tools

- **Flask-MySQL Connector:**
  A Python package used to connect Flask applications to the MySQL database securely and efficiently.
- **VS Code:**
  Visual Studio Code was used as the main code editor for developing the entire project with extensions for Python, Flask, HTML, CSS, and MySQL support.

3 Users of the Project

The Metalworks Safety Management System is designed to cater to two main types of users:

1. Customers (Clients/Factory Buyers):

- Customers can browse and view the available products and services offered by the factory.
- They have access to a personal dashboard where they can:
  - Update their username and password.
  - Browse and select products.
  - Provide feedback after completing a purchase.
- Their feedback is displayed publicly on the home page, helping future customers build trust in the factory's services.

2. Admin (Factory Owner/Manager):

- Admins have a separate secure dashboard with extended functionalities:
  - Add new products and services to the system.
  - Edit or update existing products and manage inventory.
  - Add and manage employee records.
  - View customer feedback and manage customer data.
- Admin actions directly update the live website by interacting with the backend database, ensuring the system remains current and efficient.

In short:

- **Customers** focus on interacting with products and services.
- **Admins** focus on managing the overall business operations through the dashboard.

4 Database Overview

The Metalworks Safety Management System uses a structured relational database built with **MySQL Workbench** to efficiently store, manage, and retrieve data related to products, customers, employees, and feedback.

Key Database Tables:

- **Products Table:**
  - Stores all details about the products, such as product ID, name, description, price, and availability.
  - Every time a new product is added by the Admin, it is saved into this table and automatically displayed on the home page.
- **Customers Table:**
  - Contains customer account information, including username, password, and personal details.
  - Supports customer login, profile updates, and password changes.
- **Employees Table:**
  - Records employee information such as employee ID, name, role, and other relevant details.
  - Helps the Admin manage factory employees.
- **Feedback Table:**
  - Stores feedback submitted by customers after they complete a purchase.
  - Feedback entries are retrieved and displayed on the home page to improve transparency and customer trust.
- **Orders Table:** *(Optional/Future Expansion)*
  - Will be used to record customer orders and facilitate the upcoming feature of "Custom Orders" and "Order Assignments" to employees.

Database Functionality:

- **Data Insertion:**
  - New records for products, employees, and feedback are inserted dynamically through the Admin and Customer dashboards.
- **Data Retrieval:**
  - Data is retrieved from the database to display products on the website, customer profiles, and customer feedback on the home page.
- **Data Update and Management:**
  - Customers can update their profile details.
  - Admins can update product information and employee records.
- **Power BI Integration:**
  - The database is directly connected to **Power BI** to fetch real-time data for generating dynamic dashboards and reports.
  - This integration helps factory owners analyze their sales, customer satisfaction, employee management, and business performance.
  - **In short:**
    My database acts as the **central system** that keeps my website functional, my dashboards meaningful.

5 Entity-Relationship Diagram (ERD)

Below is the Entity-Relationship Diagram (ERD) illustrating the database structure of the CRUD database:

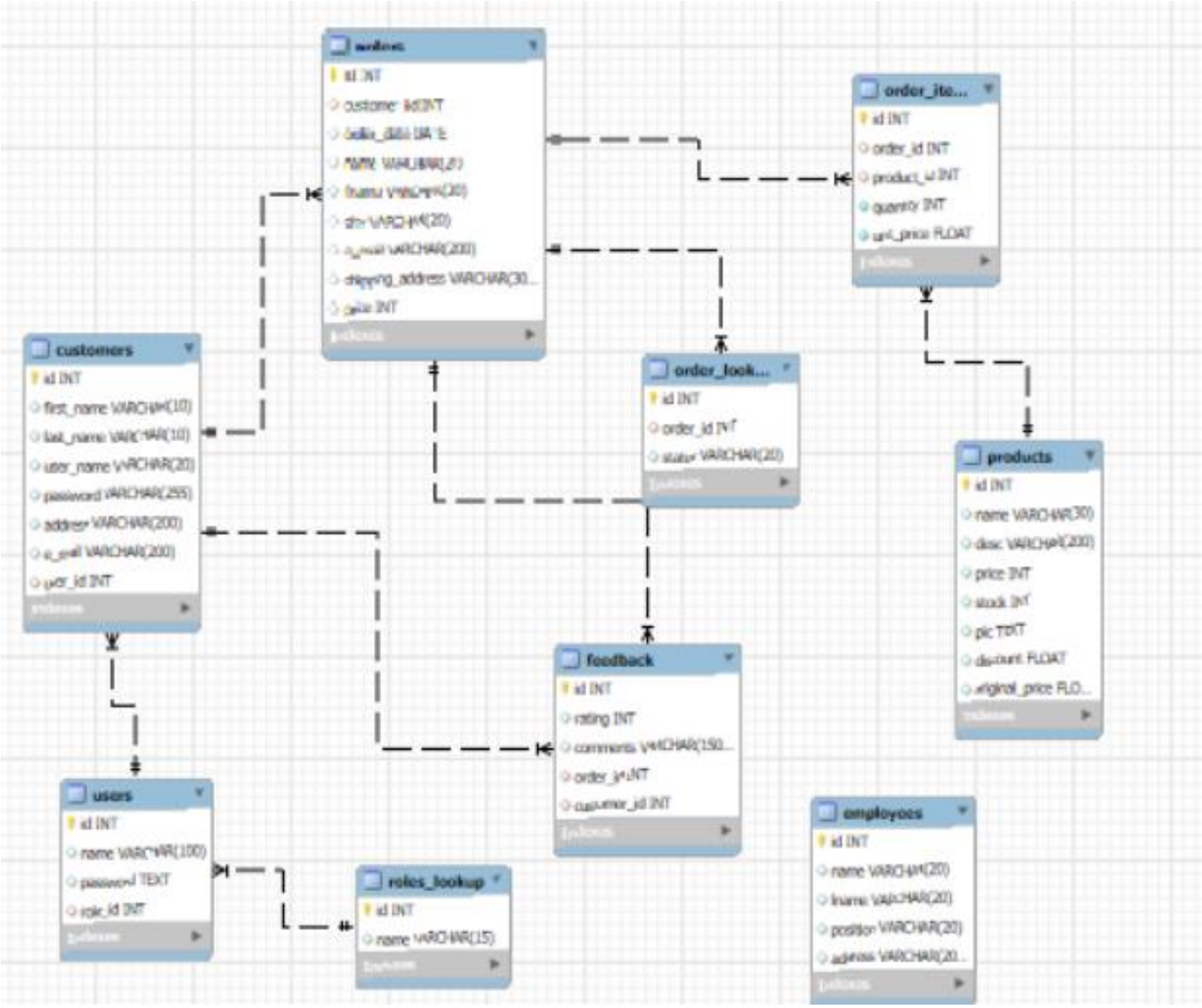


Figure 1: ER Diagram of CRUD database

6 Explanation of Entities

 Role (roles_lookup):

- Stores the types of users (e.g., Admin, Customer).
- Attributes: id, name.
- Connected with the **User** entity through role_id.

 User (users):

- Represents the login information for all system users.
- Attributes: id, name, password, role_id.
- Linked with **Role** and **Customer** entities.

 Customer (customers):

- Represents customers who browse, purchase products, and provide feedback.
- Attributes: id, first_name, last_name, user_name, password, address, e_mail, user_id.
- Connected with **User**, **Order**, and **Feedback** entities.

 Product (products):

- Represents the products available for sale.
- Attributes: id, name, desc, price, stock, pic, discount, original_price.
- Linked with **OrderItem** entity to track which products were ordered.

 Order (orders):

- Represents a customer's placed order.
- Attributes: id, customer_id, order_date, name, fname, city, e_mail, shipping_address, price.
- Linked with **OrderItem**, **OrderLookup**, and **Feedback**.

 OrderItem (order_items):

- Represents the products associated with a specific order.
- Attributes: id, order_id, product_id, quantity, unit_price.
- Linked with **Order** and **Product**

 OrderLookup (order_lookup):

- Stores the status of each order (e.g., Pending, Delivered).
- Attributes: id, order_id, status.
- Directly linked with the **Order** entity.

 Employee (employees):

- Represents factory employees.
- Attributes: id, name, lname, position, address.
- Currently managed manually by Admins.

 Feedback (feedback):

- Represents customer feedback on their orders.
- Attributes: id, rating, comments, order_id, customer_id.
- Linked with **Customer** and **Order** entities.

7. Relationship Participation

Role → User

- **One-to-Many Relationship**:
  - One **Role** can be assigned to many **Users**.
  - Each **User** must belong to exactly one **Role** (via `role_id` Foreign Key).

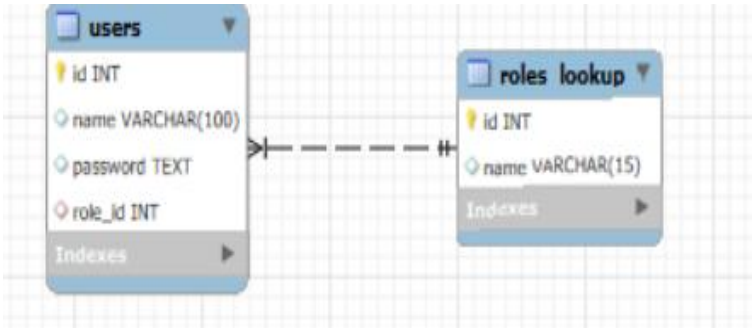


Figure 2: Relationship between Role and User

User → Customer

- **One-to-Many Relationship**:
  - A **User** (usually an Admin) can create/manage multiple **Customers**.
  - Each **Customer** is linked to one **User** (via user_id Foreign Key).

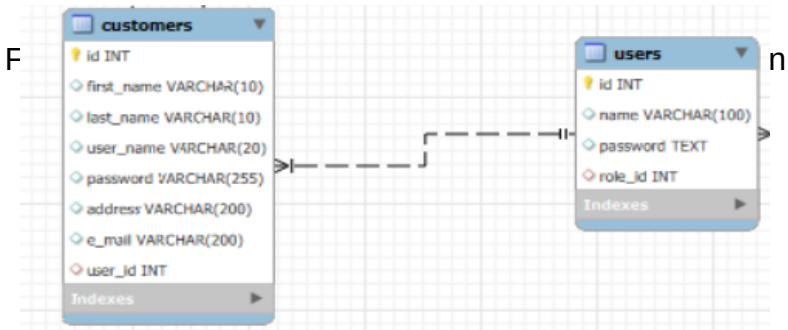


Figure 3: Relationship between User and Customer

Customer → Order

- **One-to-Many Relationship**:
  - One **Customer** can place multiple **Orders**.
  - Each **Order** belongs to one **Customer** (via customer_id Foreign Key).

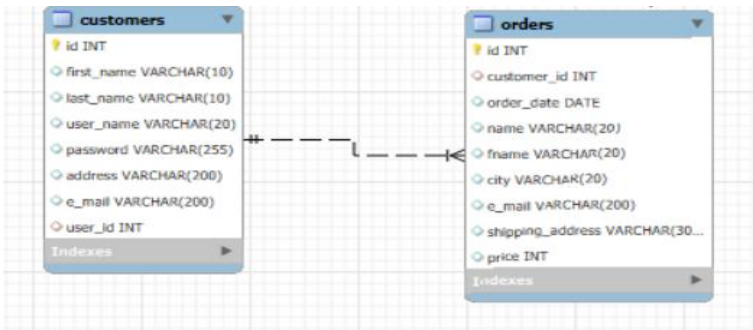


Figure 4: Relationship between Customer and Order

Customer → Feedback

- **One-to-Many Relationship**:
  - A **Customer** can give feedback multiple times.
  - Each **Feedback** belongs to one **Customer** (via customer_id Foreign Key).

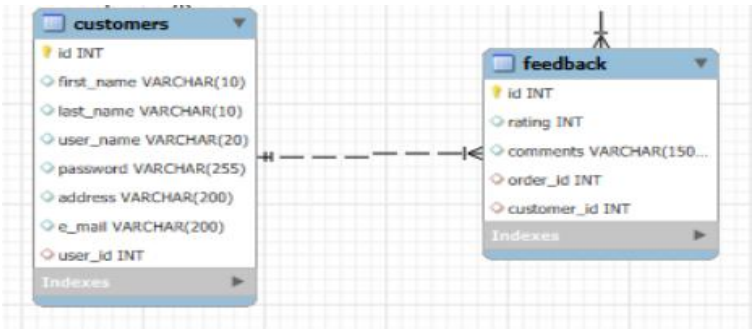


Figure 5: Relationship between Customer and Feedback

Order → OrderItem

- **One-to-Many Relationship**:
  - An **Order** can contain multiple **OrderItems** (i.e., multiple products within the same order).
  - Each **OrderItem** belongs to one **Order** (via order_id Foreign Key).

Figure 6: Relationship between Order and Order_Item

Order → Feedback

- **One-to-Many Relationship**:
  - One **Order** can have multiple **Feedback** entries (usually one, but the design allows multiple).
  - Each **Feedback** is linked to an **Order** (via order_id Foreign Key).

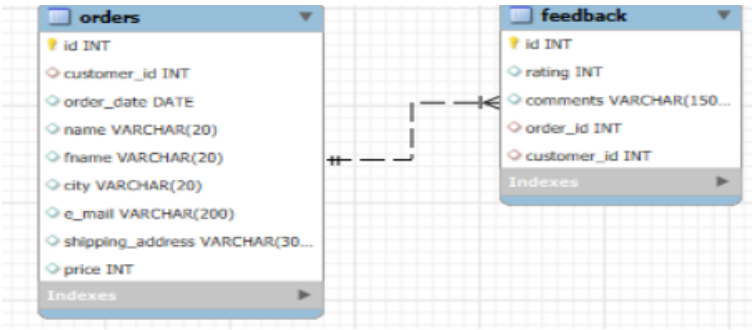


Figure 7: Relationship between Order and Feedback

Product → OrderItem

- **One-to-Many Relationship**:
  - One **Product** can appear in multiple **OrderItems** (i.e., multiple different orders can include the same product).
  - Each **OrderItem** references one **Product** (via product_id Foreign Key).

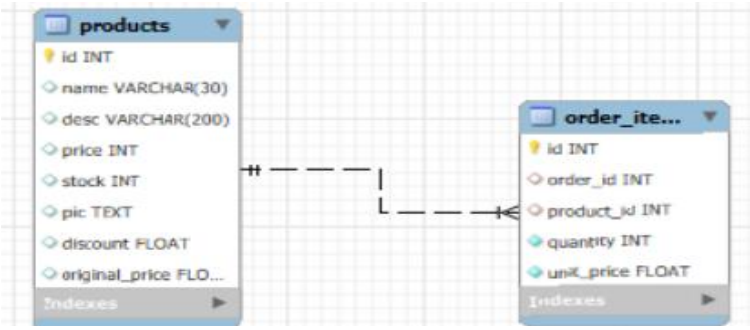


Figure 8: Relationship between Product and Order_item

Employee

- **Standalone Entity**:
  - Currently, the **Employee** table is independent.
  - It is not linked with any other table for now but is intended for future expansion (such as assigning orders to employees).

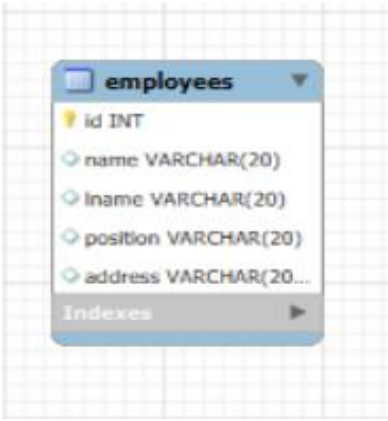


Figure 8: Employee Table

8. Business Rules

1. User and Role Management

- Each **User** must be assigned a **Role** (such as Admin, Staff, etc.) at the time of creation.
- A **Role** can be linked to multiple **Users**, but every **User** must have only one **Role**.

2. Customer Management

- Each **Customer** must be created and managed under a specific **User** (Admin or Staff).
- A **Customer** must have a **unique username** and **email address**.

3. Product Management

- Each **Product** must include important information such as **Name**, **Description**, **Price**, **Stock**, and **Image Path**.
- **Discounts** can be applied to products; if no discount is present, it defaults to 0%.

4. Order and Order Items

- An **Order** must always be associated with exactly **one Customer**.
- An **Order** can contain **multiple products** through the **OrderItems** table.
- Each **OrderItem** must record the **quantity** and **unit price** at the time of order placement.
- The **total price** of the order must reflect the sum of all order items (though calculated outside the database currently).

5. Order Status Tracking

- Each **Order** must have exactly **one corresponding status** record in **OrderLookup**.
- Default **Order status** when created is set to **"Pending"**.
- Status can be updated to reflect the order process (e.g., Shipped, Delivered, Cancelled).

6. Feedback System

- A **Customer** can submit feedback for **each Order** they place.
- Feedback includes a **rating (numerical)** and optional **comments** about the order experience.

7. Employee Management

- **Employee** details (Name, Last Name, Position, Address) must be stored for administrative purposes.
- **Employees** are independent currently but will be assigned orders manually in future upgrades (Planned feature).

8. Authentication and Security

- **User** and **Customer** passwords are securely stored (though encryption is suggested for production systems).
- Login authentication must validate both **Username/Email** and **Password**.

9. Data Integrity Rules

- **Foreign keys** ensure relational consistency between **Users**, **Customers**, **Orders**, **Products**, etc.
- **Cascade deletion** is set up for some tables (e.g., deleting an Order automatically deletes related OrderItems and Lookup entries).

9. Object-Oriented Programming (OOP) Architecture

To ensure clean separation of concerns, modular design, and scalable development, the backend logic of *Metalworks* was implemented using **Object-Oriented Programming (OOP)** principles in Python (Flask).

1. Class Structure Overview

Content:
The codebase is structured around multiple service classes:

- UserService
- CustomerService
- ProductService
- OrderService
- EmployeeService
- CartService

Each class encapsulates business logic related to a specific domain of the application, promoting the **Single Responsibility Principle (SRP)**.

2. UserService – Authentication and Role Management

Content:
This class handles user registration and login. Passwords are securely hashed using Werkzeug before storing them in the database. The register_user method allows creation of new users with an assigned role, and login verifies user credentials during login.

```python
from werkzeug.security import generate_password_hash,check_password_hash

class UserService:
    def __init__(self, db_session):
        self.db_session = db_session

    def register_user(self, uname, password, role):
            hashed_password = generate_password_hash(password)
            user = User(name=uname, password=hashed_password, role=role)
            self.db_session.add(user)
            self.db_session.commit()

    def login(self, uname, password):
            user = self.db_session.query(User).filter_by(name=uname).first()
            if user and check_password_hash(user.password, password):
                return user
            return None
```

3. CustomerService – Customer and Feedback Management

Content:
Responsible for:

- Registering new customers while linking them to a user record.

```
def add_customer(self, first_name, last_name, user_name, password, address, email):
    existing_customer = self.db_session.query(Customer).filter_by(user_name=user_name).first()
    if existing_customer:
        return None
    role = self.db_session.query(Role).filter_by(name="customer").first()
    if not role:
        raise Exception("Customer role not found in the database. Please seed roles_lookup table.")
    new_user = User(
        name=user_name,
        password=generate_password_hash(password),
        role_id=role.id
    )
    self.db_session.add(new_user)
    self.db_session.flush()
    new_customer = Customer(
        first_name=first_name,
        last_name=last_name,
        user_name=user_name,
        password=new_user.password,
        address=address,
        e_mail=email,
        user_id=new_user.id
    )
    self.db_session.add(new_customer)
    self.db_session.commit()
    return new_customer
```

- Allowing customers to give feedback on orders.

```
def add_feedback(self,order_id,customer_id,rating,comments):
    feedback=Feedback(
        order_id= order_id,
        customer_id= customer_id,
        rating=rating,
        comments=comments)
    if feedback:
        self.db_session.add(feedback)
        self.db_session.commit()
```

- Retrieving customer orders and feedback.

```
def get_feedback(self):
    return self.db_session.query(Feedback).order_by(Feedback.id.desc()).all()

def get_customer_orders(self, customer_id):
    customer = self.db_session.query(Customer).filter_by(id=customer_id).first()
    return customer.orders if customer else []
```

- Updating customer credentials.

```
def update_customer_username(self, customer_id, new_username):
    customer = self.db_session.query(Customer).get(customer_id)
    if customer:
        customer.first_name = new_username
        self.db_session.commit()

def update_customer_password(self, customer_id, new_password):
    customer = self.db_session.query(Customer).get(customer_id)
    if customer:
        customer.password = generate_password_hash(new_password)
        self.db_session.commit()
```

This class maintains customer-related business logic and ensures role-based user creation by linking to the Role table.

4. ProductService – Product Management and Discount Handling

Content:
The ProductService class manages product CRUD operations and supports:

- Sorting products by name or price.

```
class ProductService:
    def __init__(self, db_session):
        self.db_session = db_session

    def get_sorted_products(self, sort_by):
        sort_options = {
            'price_asc': Product.price.asc(),
            'price_desc': Product.price.desc(),
            'name_desc': Product.name.desc(),
            'name_asc': Product.name.asc()
        }
```

- Applying or removing discounts.

```
def give_discount(self, product_id, discount_percent):
    product = self.db_session.query(Product).get(product_id)
    if product:
        product.original_price = product.price
        product.price *= (1 - discount_percent / 100)
        self.db_session.commit()

def remove_discount(self, product_id):
    product = self.db_session.query(Product).get(product_id)
    if product and product.original_price is not None:
        product.price = product.original_price
        product.original_price = None
        self.db_session.commit()
```

- Fetching product lists or details.

It includes methods like add_product, delete_product, update_product, give_discount, and remove_discount.

5. OrderService – Order Placement and Status Updates

Content:
Handles all operations related to orders:

- Placing new orders and storing order items.

```
def place_order(self, address, e_mail, name, lname, city, price, customer_id):
    order = Order(
        order_date=datetime.now(),
        shipping_address=address,
        e_mail=e_mail,
        name=name,
        fname=lname,
        city=city,
        price=price,
        customer_id=customer_id
    )
    self.db_session.add(order)
    self.db_session.commit()
    return order
```

It supports both customer and admin views of orders.

6. EmployeeService – Managing Staff Records

Content:
Designed for administrative tasks related to employees. It provides functionality to:

- Add new employees.

```
class EmployeeService():
    def __init__(self, db_session):
        self.db_session = db_session

    def add_employee(self, name,lname, position,address):
        employee = Employee(name=name,lname=lname, position=position,address=address)
        self.db_session.add(employee)
        self.db_session.commit()
```

- View all current employees.

```
def view_employee(self):
    return self.db_session.query(Employee).all()
```

- Remove employees by ID (fire).

```
def fire_employee(self,emp_id):
    employee = self.db_session.query(Employee).filter_by(id=emp_id).first()
    if employee:
        self.db_session.delete(employee)
        self.db_session.commit()
        return True
    return False
```

This class makes employee management easy for admin users.

7. CartService – Session-Based Cart Functionality

Content:
CartService is designed to handle cart operations using Flask's session object:

- Add, update, and remove items in the cart.

```
class CartService:
    def __init__(self, session):
        self.session = session

    def add_to_cart(self, product_id, quantity):
        cart = self.session.get('cart', {})
        if product_id in cart:
            cart[product_id] += quantity
        else:
            cart[product_id] = quantity
        self.session['cart'] = cart
        self.session.modified = True

    def update_quantity(self, product_id, quantity):
        cart = self.session.get('cart', {})
        if product_id in cart:
            cart[product_id] = quantity
            self.session['cart'] = cart
            self.session.modified = True

    def remove_from_cart(self, product_id):
        cart = self.session.get('cart', {})
        if product_id in cart:
            del cart[product_id]
            self.session['cart'] = cart
            self.session.modified = True
```

- Clear the cart.

```
def clear_cart(self):
    self.session['cart'] = {}
    self.session.modified = True
```

- Calculate the total price using real-time product data.

```python
def get_total(self, productservice):
    cart = self.session.get('cart', {})
    total = 0

    for product_id, quantity in cart.items():
        product = productservice.get_product_by_id(int(product_id))
        if product:
            total += product.price * quantity

    return total
```

This class is crucial for managing the shopping experience before checkout.

8. Authentication and Security

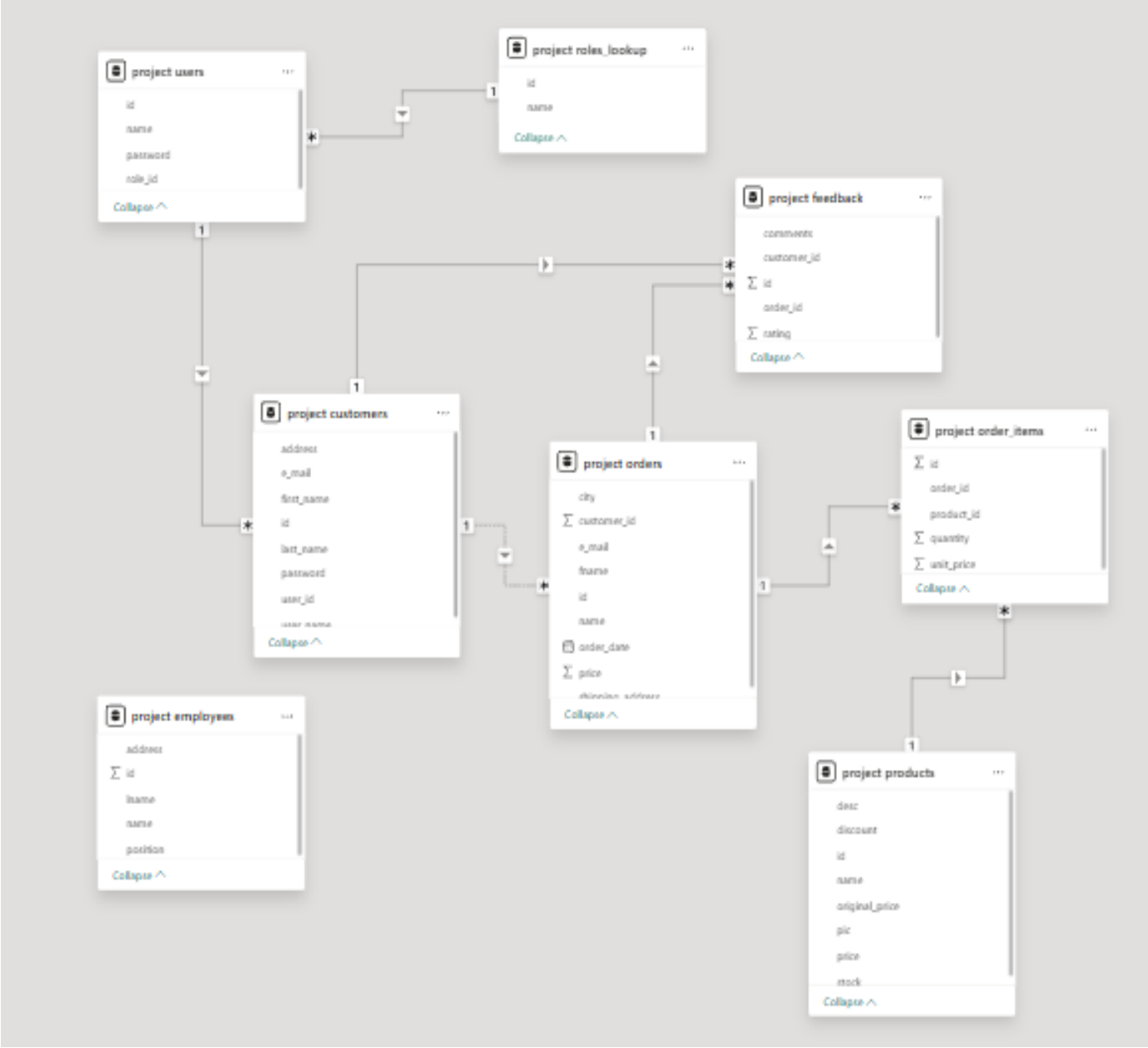
The **Metalworks** system incorporates multiple layers of security, especially in user authentication, role control, and data access:

- **Hashed Password Storage:** All passwords (admin and customer) are hashed using werkzeug.security.generate_password_hash, ensuring that raw passwords are never stored in the database.
- **Secure Login Authentication:** The system uses strict login checks. Both username and password are verified securely before granting access.
- **Role-Based Access Control:** Admins and Customers are assigned roles from the Role table, and their access to system features is governed accordingly.

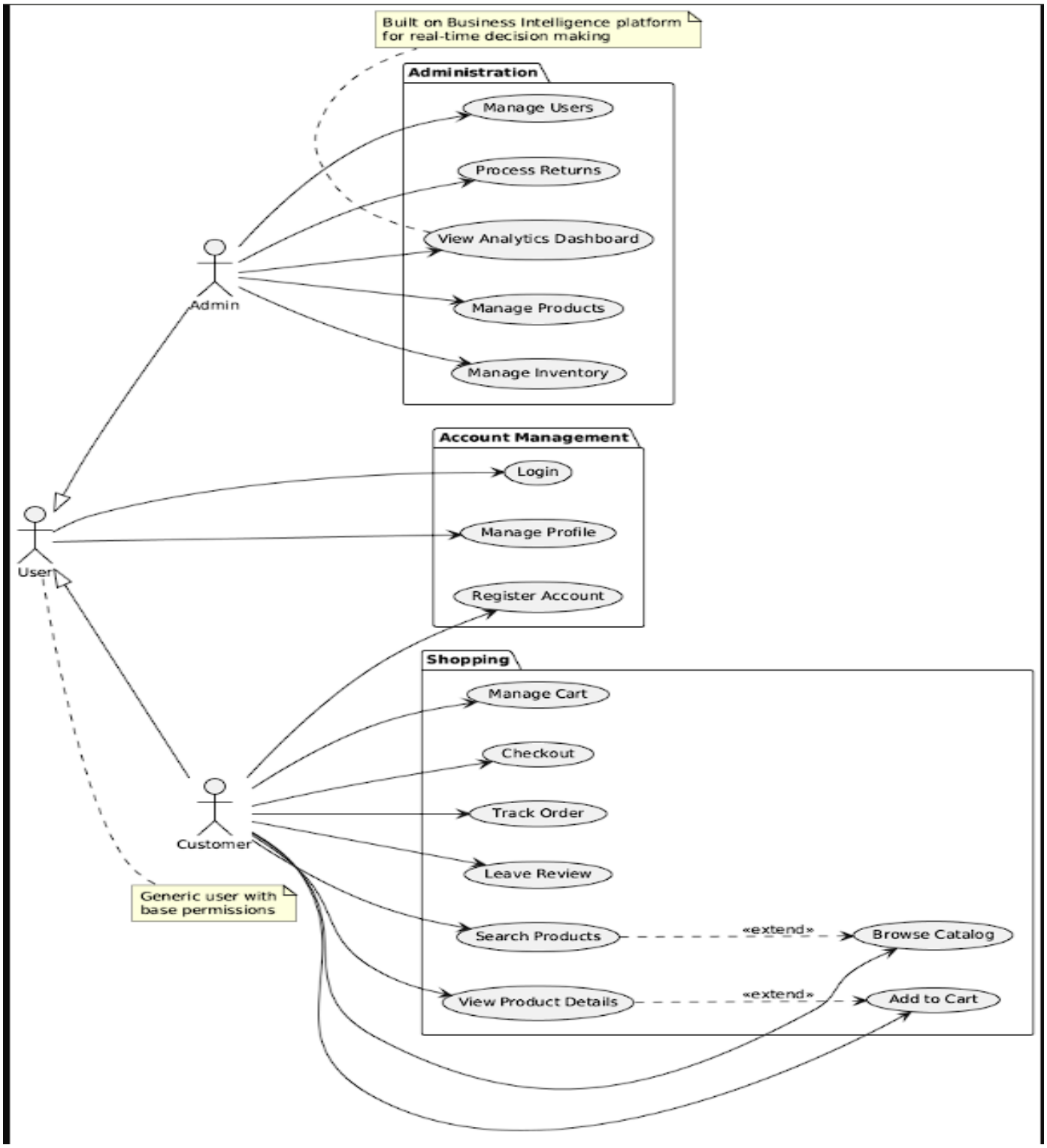
- **Admin Creation File:** A dedicated admin creation file/script is developed separately. This script securely inserts an Admin account into the system using OOP principles and hashed passwords. This ensures:
  - Only trusted people can initialize the system.
  - There is no manual SQL access needed to add an admin.
  - The creation is logged and handled like all other user operations.

```python
from db import db_session
from db import User, Role
from werkzeug.security import generate_password_hash,check_password_hash
admin_role = db_session.query(Role).filter_by(name='admin').first()

if not admin_role:
    admin_role = Role(name='admin')
    db_session.add(admin_role)
    db_session.commit()
    print("Admin role created.")
existing_admin = db_session.query(User).filter_by(name='admin').first()
if not existing_admin:
    admin_user = User(
        name='admin',
        password=generate_password_hash('admin123'),
        role_id=admin_role.id
    )
    db_session.add(admin_user)
    db_session.commit()
    print("Admin user created successfully!")
else:
    print("Admin user already exists.")
```

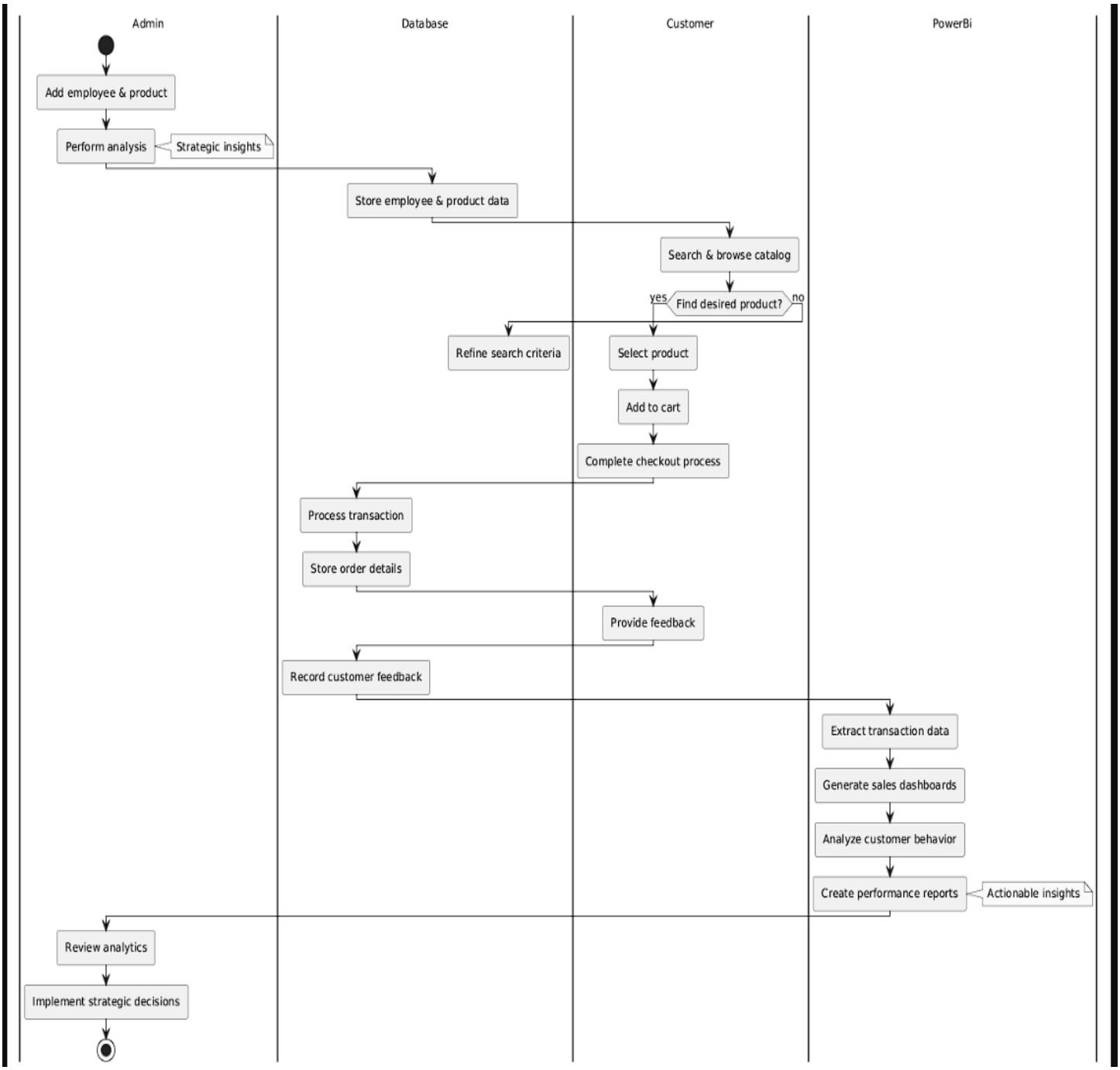
10. Model View Diagram



11. Use Case diagram



12. Data Flow diagram

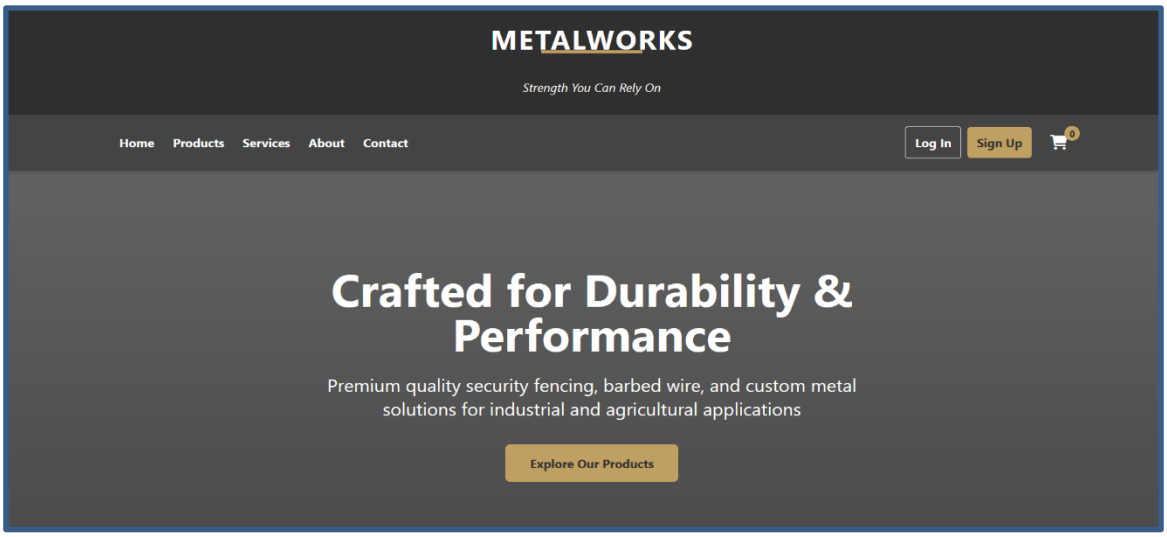


13. User Interface Design

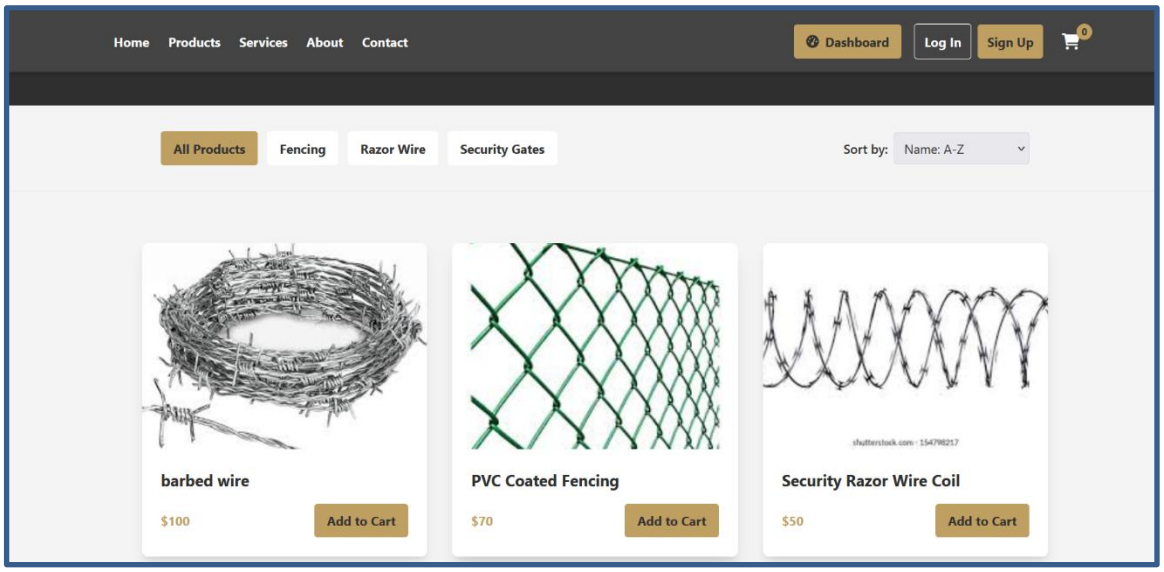
Web Pages

Metalworks will feature user-friendly web pages accessible via standard web browsers. These pages will provide interfaces for users to inter- act with various system functionalities, such as submitting performance evaluations, viewing reports, and managing user accounts.

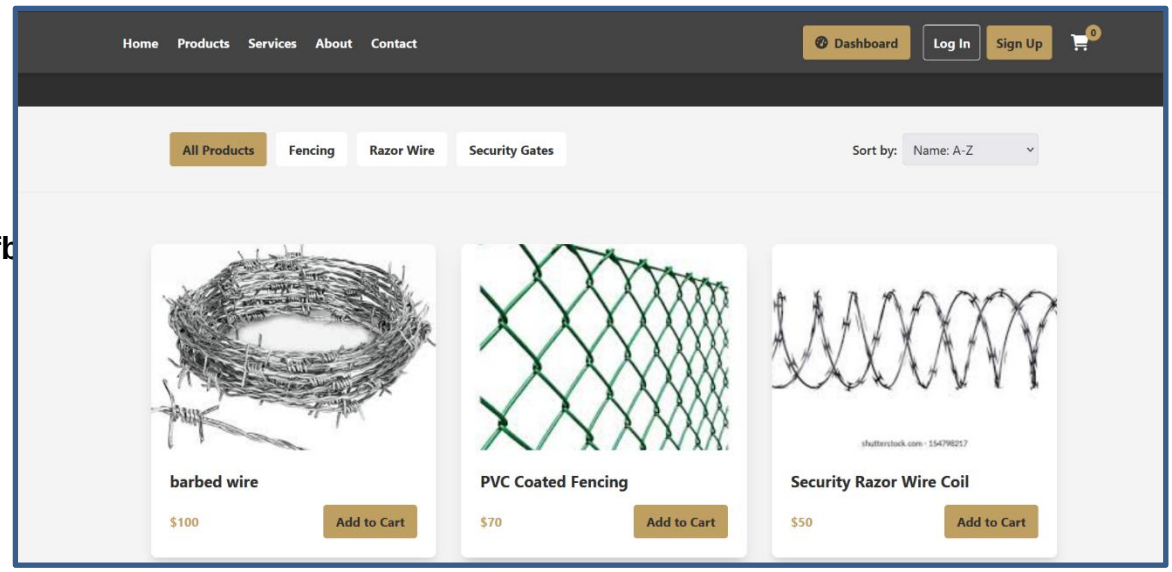
Main Page



Product Page

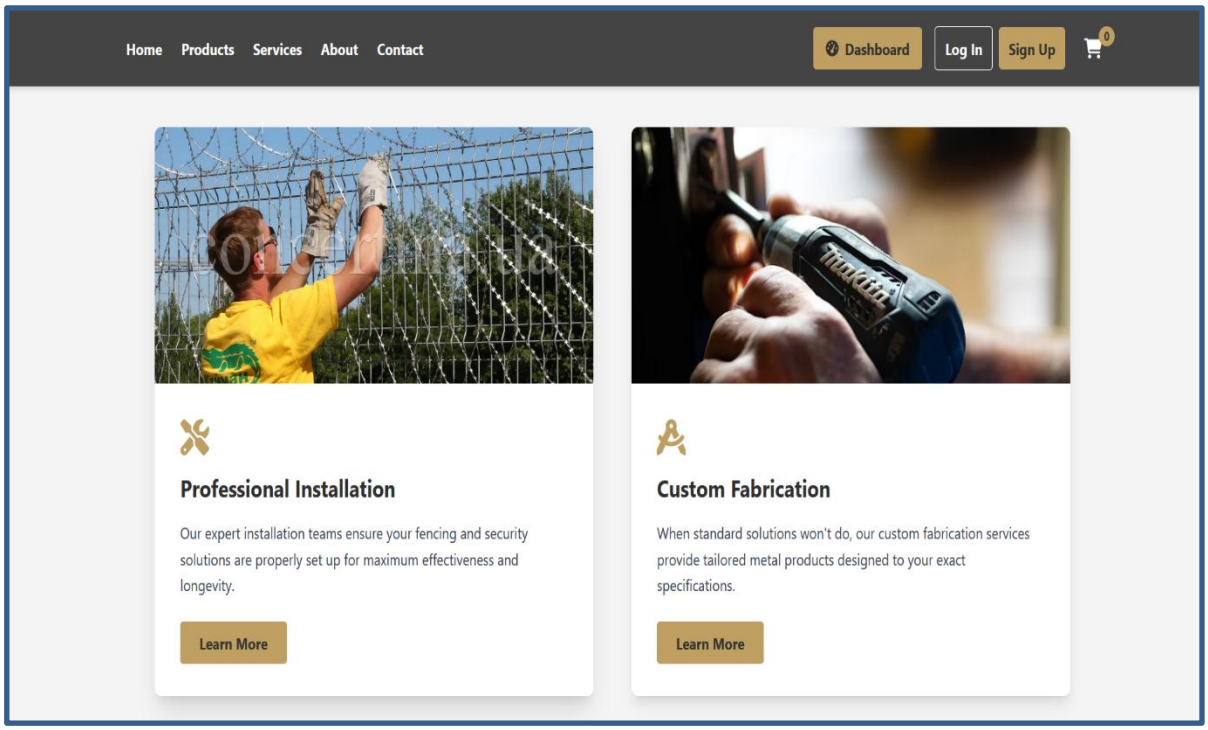


Services Page



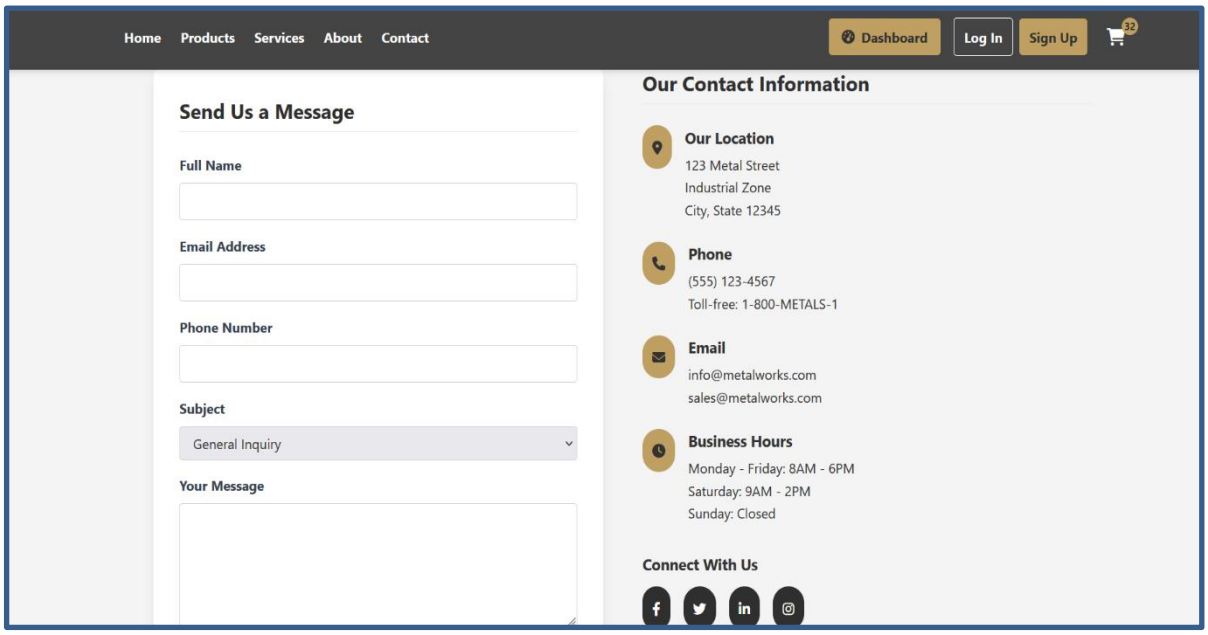
fb

About Page



Contact Page

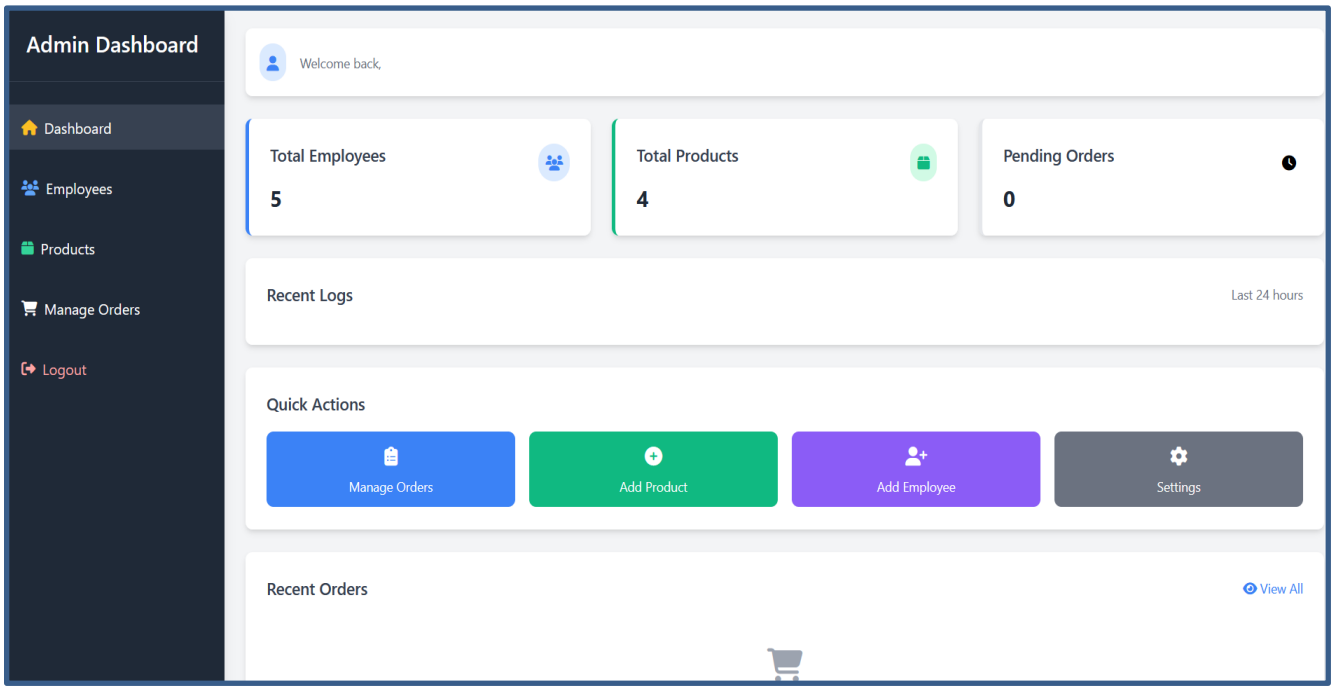
.



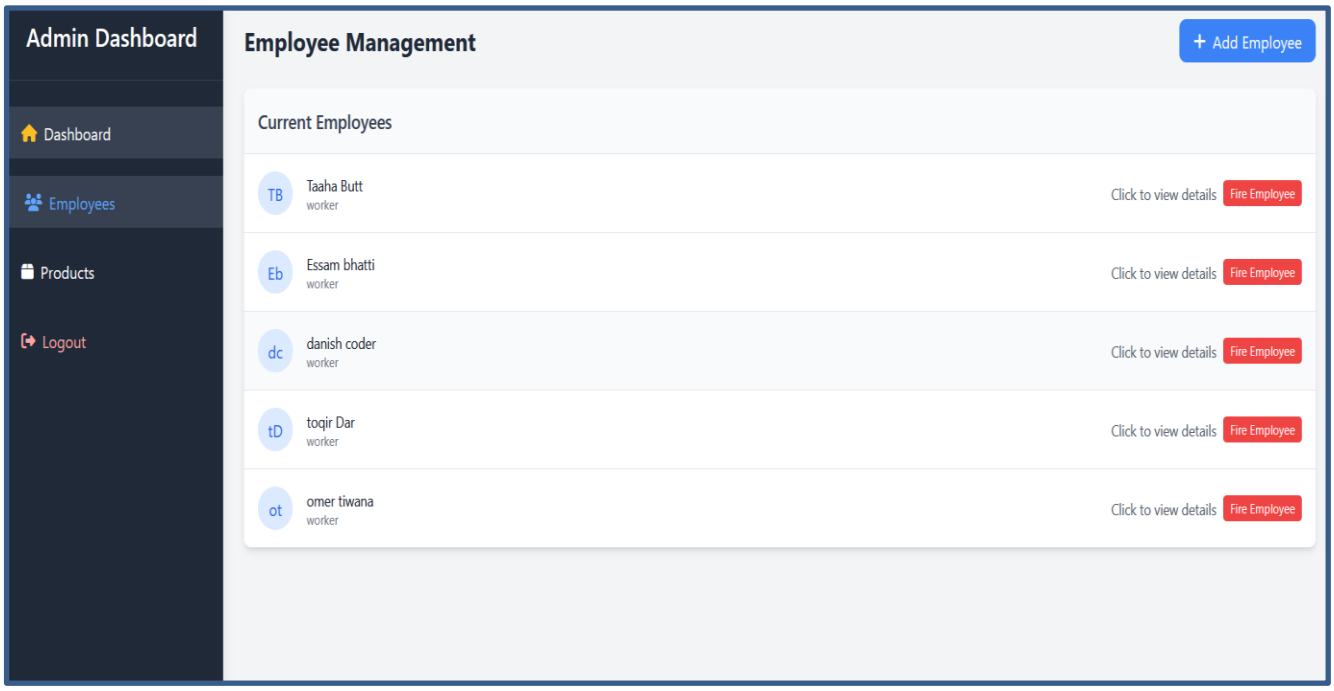
Create Admin

```
create_admin.py > ...
from db import db_session
from db import User, Role
from werkzeug.security import generate_password_hash,check_password_hash

admin_role = db_session.query(Role).filter_by(name='admin').first()
if not admin_role:
    admin_role = Role(name='admin')
    db_session.add(admin_role)
    db_session.commit()
    print("Admin role created.")

existing_admin = db_session.query(User).filter_by(name='admin').first()
if not existing_admin:
    admin_user = User(
        name='admin',
        password=generate_password_hash('admin123'),
        role_id=admin_role.id
    )
    db_session.add(admin_user)
    db_session.commit()
    print("Admin user created successfully!")
else:
    print("Admin user already exists.")
```

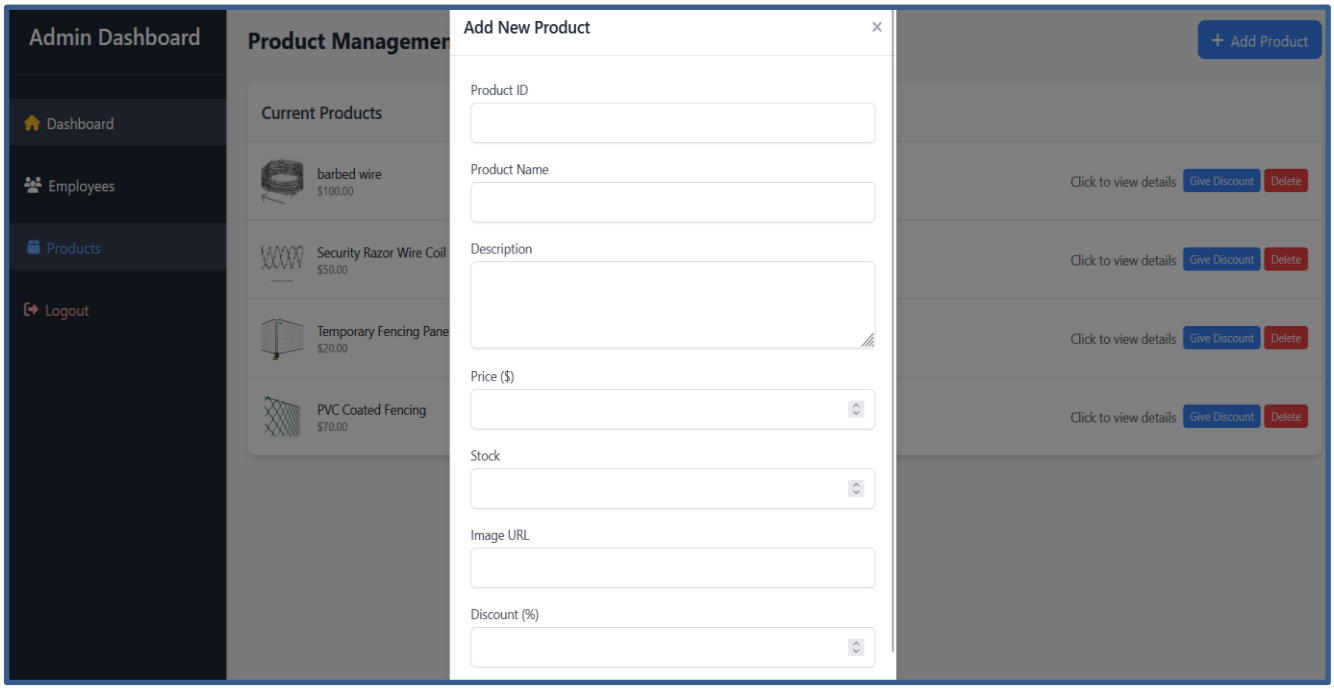
Admin dashboard


Admin Dashboard
Dashboard
Employees
Products
Manage Orders
Logout
Welcome back,
Total Employees
5
Total Products
4
Pending Orders
0
Recent Logs
Last 24 hours
Quick Actions
Manage Orders
Add Product
Add Employee
Settings
Recent Orders
View All

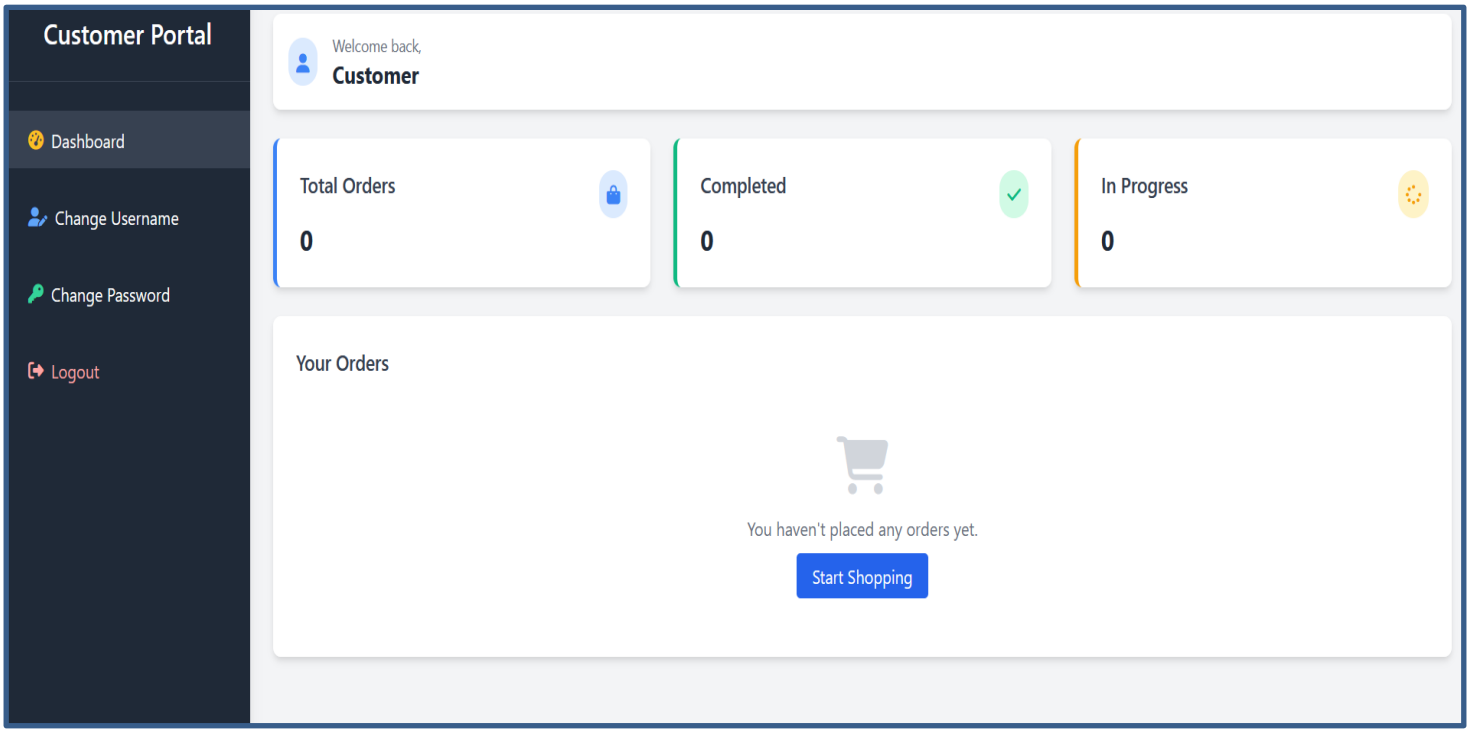
Add Employee



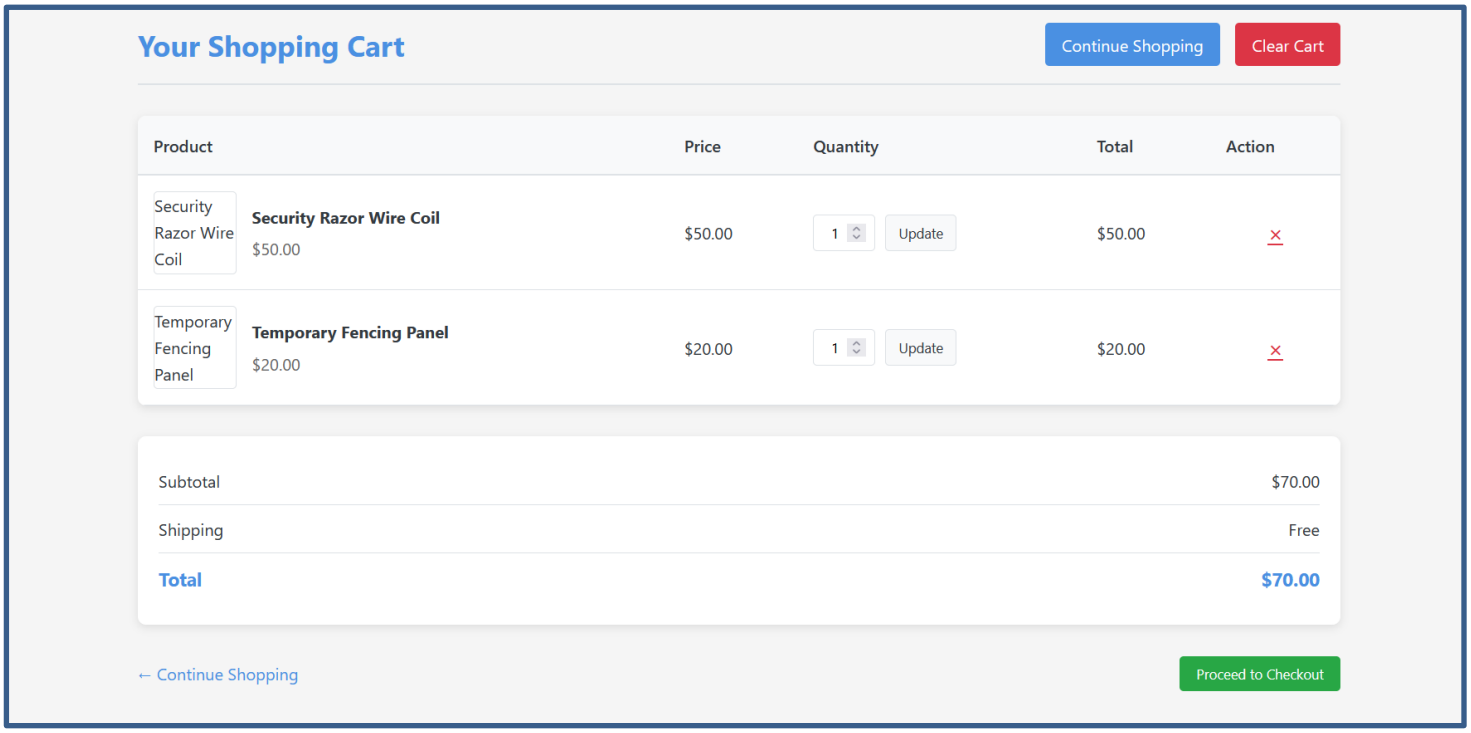
Add Product



Customer Dashboard

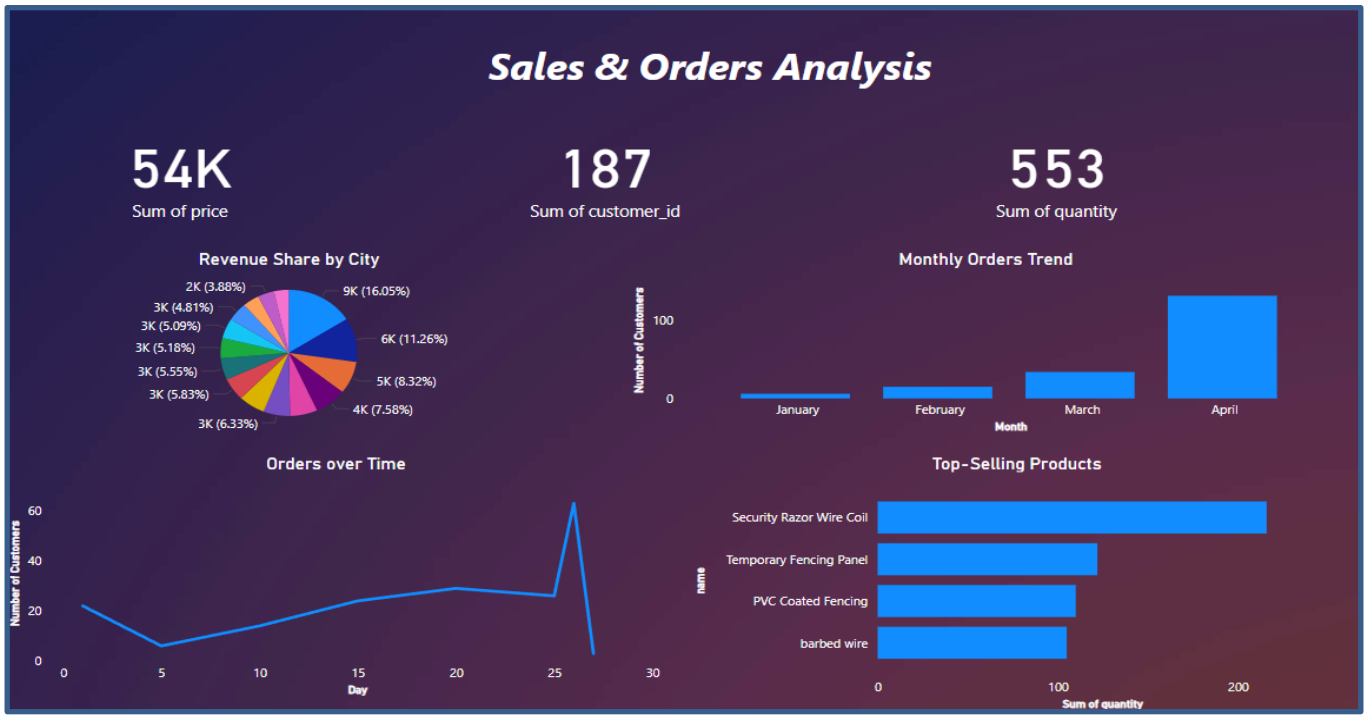


Cart page



14. Power BI

Integration with Power BI will allow users to access comprehensive data visualizations and analytics.



15. Dependencies in Relational Databases

In relational databases, **dependencies** describe the relationships between tables, ensuring data integrity and consistency. These dependencies can be categorized into **strong** and **weak** relationships.

Strong Relationships

A **strong relationship** occurs when the **dependent (child) entity** cannot exist without the **parent entity**. The child entity relies heavily on the parent entity for its existence.

Examples of Strong Relationships in my Schema:

1. **User → Role**:
    - A User cannot exist without being assigned a Role.
    - **Foreign Key**: role_id in User table → id in Role table.
2. **Customer → User**:
    - A Customer cannot exist without being linked to a User.
    - **Foreign Key**: user_id in Customer table → id in User table.
3. **Order → Customer**:
    - An Order must belong to a Customer.
    - **Foreign Key**: customer_id in Order table → id in Customer table.
4. **OrderItem → Order**:
    - An OrderItem cannot exist without an associated Order.
    - **Foreign Key**: order_id in OrderItem table → id in Order table.
5. **OrderItem → Product**:
    - An OrderItem depends on a Product.
    - **Foreign Key**: product_id in OrderItem table → id in Product table.
6. **Feedback → Order & Feedback → Customer**:
    - Feedback depends on both Order and Customer.
    - **Foreign Keys**: order_id in Feedback table → id in Order table; customer_id in Feedback table → id in Customer table.

Weak Relationships

A **weak relationship** occurs when the **dependent (child) entity** can exist independently of the **parent entity**. The child table may reference the parent but does not necessarily depend on it for its own existence.

Examples of Weak Relationships in my Schema:

1. **Employee**:
    - An Employee does not necessarily depend on other entities. It can exist independently without requiring a User or Order.
    - No foreign key relationship to other tables.
2. **Product**:
    - A Product can exist without being tied to a specific Order. However, once an order is placed, Products are included through the OrderItem table.
    - Products can exist independently of orders.

16. Future Goals

While the current system covers essential safety and inventory management features, several enhancements are planned to make the system even more powerful and user-centric:

- **Employee Attendance Management**
  A feature will be added to monitor employee attendance, allowing the admin to track check-ins, check-outs, and working hours. This will improve workforce management and accountability.
- **Custom Order Placement**
  Customers will be given the ability to place custom orders, specifying product quantity, customization details, and required delivery timelines. This will make the system more flexible and better aligned with customer needs.
- **Order Assignment to Employees**
  The system will be upgraded to allow admins to assign specific orders to particular employees, improving operational efficiency and tracking order responsibilities.
- **Enhanced Data Analysis in Power BI**
  More advanced analytical dashboards will be developed in Power BI, providing deeper insights into sales trends, employee performance, and customer satisfaction.

17. Conclusion

The **Metalworks Safety Management System** was developed to support small to medium-sized factories in moving towards digital solutions.The system provides features like inventory management, customer feedback, employee management, and online order handling, helping businesses operate more efficiently.

Using technologies such as **HTML**, **CSS**, **JavaScript**, **TailwindCSS**, **Python (Flask)**, **MySQL**, and **Power BI**, a complete platform was created.
Although there were time limitations due to learning new technologies, a strong and working system was successfully built.

In the future, the project will be expanded with features like employee attendance tracking, custom order placement, and more advanced analytics to better meet business needs.

18. Github Project Link

The project code and related materials can be found on GitHub at the following link: