VIJAY R

Superset ID: 5371616

Case Study on Digital Asset Management Application

Project Overview

The Digital Asset Management System is a comprehensive Python-based solution designed to streamline the management of physical and digital assets within an organization. It facilitates efficient asset tracking, maintenance, allocation, deallocation, and reservation. The system is developed using object-oriented principles in Python, incorporates MySQL for robust data storage, and follows a modular structure with packages for entities, data access operations, utilities, and exception handling. This ensures maintainability, scalability, and clear separation of concerns

Introduction

In any organization, managing physical assets efficiently is crucial for smooth operations and cost control. Assets such as computers, equipment, vehicles, and tools require constant monitoring, allocation, maintenance, and sometimes reservation. Without a proper system, tracking these activities manually can lead to errors, resource misuse, and poor accountability.

The Digital Asset Management System is designed to solve this challenge by offering a centralized platform to manage assets digitally. Developed using Python and MySQL, the system allows organizations to handle asset-related tasks such as adding new assets, allocating them to employees, recording maintenance activities, and managing reservations with ease. The software uses object-oriented programming principles and follows a modular structure for scalability and maintainability.

This project not only simplifies asset tracking but also reduces administrative burden, ensures timely maintenance, and improves resource utilization through digital automation.

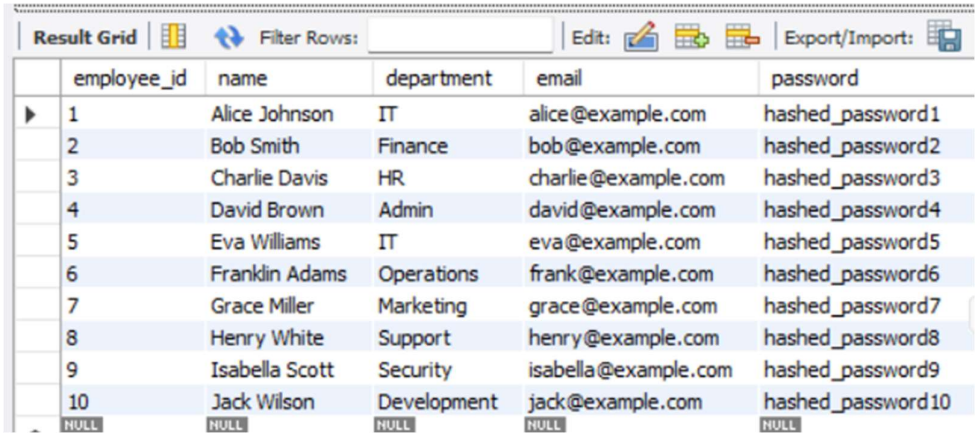
Schema Design:

1. employees table:

- employee_id (Primary Key)
- name
- department.
- email.
- password.

```
CREATE TABLE employees (
employee_id INT PRIMARY KEY AUTO_INCREMENT,
name VARCHAR(100) NOT NULL,
department VARCHAR(100),
email VARCHAR(100) UNIQUE NOT NULL,
password VARCHAR(255) NOT NULL
);
```

```
INSERT INTO employees (name, department, email, password) VALUES
('Alice Johnson', 'IT', 'alice@example.com', 'hashed_password1'),
('Bob Smith', 'Finance', 'bob@example.com', 'hashed_password2'),
('Charlie Davis', 'HR', 'charlie@example.com', 'hashed_password3'),
('David Brown', 'Admin', 'david@example.com', 'hashed_password4'),
('Eva Williams', 'IT', 'eva@example.com', 'hashed_password5'),
('Franklin Adams', 'Operations', 'frank@example.com', 'hashed_password6'),
('Grace Miller', 'Marketing', 'grace@example.com', 'hashed_password7'),
('Henry White', 'Support', 'henry@example.com', 'hashed_password8'),
('Isabella Scott', 'Security', 'isabella@example.com', 'hashed_password9'),
('Jack Wilson', 'Development', 'jack@example.com', 'hashed_password10');
```



Result Grid | Filter Rows: | Edit: | Export/Import:

employee_id	name	department	email	password
1	Alice Johnson	IT	alice@example.com	hashed_password1
2	Bob Smith	Finance	bob@example.com	hashed_password2
3	Charlie Davis	HR	charlie@example.com	hashed_password3
4	David Brown	Admin	david@example.com	hashed_password4
5	Eva Williams	IT	eva@example.com	hashed_password5
6	Franklin Adams	Operations	frank@example.com	hashed_password6
7	Grace Miller	Marketing	grace@example.com	hashed_password7
8	Henry White	Support	henry@example.com	hashed_password8
9	Isabella Scott	Security	isabella@example.com	hashed_password9
10	Jack Wilson	Development	jack@example.com	hashed_password10
NULL	NULL	NULL	NULL	NULL

2. assets table:

- asset_id (Primary Key): Unique identifier for each asset.
- name.
- type: Type of the asset (e.g., laptop, vehicle, equipment).
- serial_number: Serial number or unique identifier of the asset.
- purchase_date.
- location: Current location of the asset.
- status: Status of the asset (e.g., in use, decommissioned, under maintenance).
- owner_id: (Foreign Key): References the employee who owns the asset.

```
CREATE TABLE assets (
    asset_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    type VARCHAR(50) NOT NULL,
    serial_number VARCHAR(100) UNIQUE NOT NULL,
    purchase_date DATE NOT NULL,
    location VARCHAR(100),
    status ENUM('in use', 'decommissioned', 'under maintenance') NOT NULL,
    owner_id INT,
```

FOREIGN KEY (owner_id) REFERENCES employees(employee_id) ON DELETE SET NULL

);

INSERT INTO assets (name, type, serial_number, purchase_date, location, status, owner_id) VALUES

('Laptop', 'Electronics', 'SN123456', '2021-06-15', 'IT Office', 'in use', 1),

('Projector', 'Electronics', 'SN789012', '2022-08-20', 'Conference Room', 'under maintenance', 2),

('Office Chair', 'Furniture', 'SN345678', '2023-02-10', 'HR Cabin', 'in use', 3),

('Server', 'IT Equipment', 'SN901234', '2021-12-05', 'Data Center', 'in use', 4),

('Printer', 'Electronics', 'SN567890', '2024-04-22', 'Finance Dept', 'decommissioned', 5),

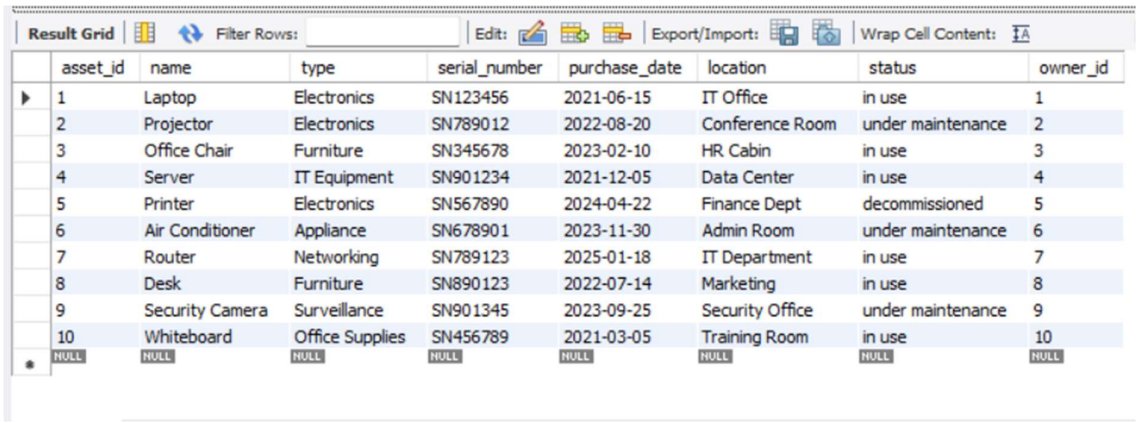
('Air Conditioner', 'Appliance', 'SN678901', '2023-11-30', 'Admin Room', 'under maintenance', 6),

('Router', 'Networking', 'SN789123', '2025-01-18', 'IT Department', 'in use', 7),

('Desk', 'Furniture', 'SN890123', '2022-07-14', 'Marketing', 'in use', 8),

('Security Camera', 'Surveillance', 'SN901345', '2023-09-25', 'Security Office', 'under maintenance', 9),

('Whiteboard', 'Office Supplies', 'SN456789', '2021-03-05', 'Training Room', 'in use', 10);



Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

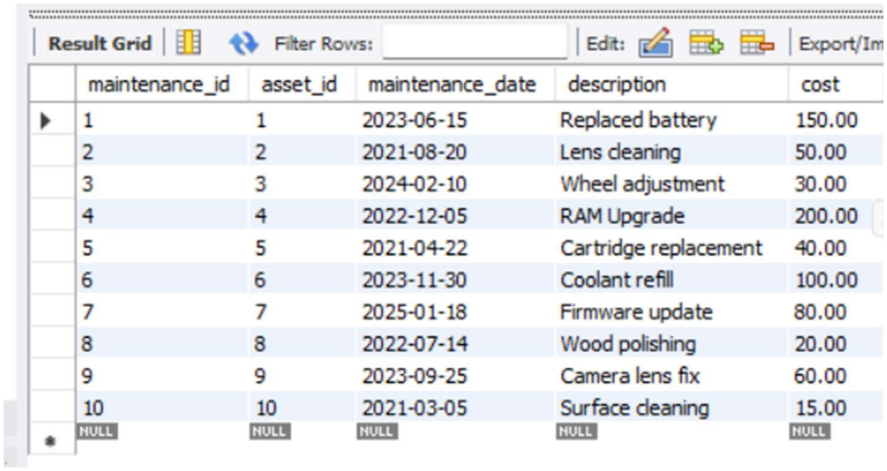
asset_id	name	type	serial_number	purchase_date	location	status	owner_id
1	Laptop	Electronics	SN123456	2021-06-15	IT Office	in use	1
2	Projector	Electronics	SN789012	2022-08-20	Conference Room	under maintenance	2
3	Office Chair	Furniture	SN345678	2023-02-10	HR Cabin	in use	3
4	Server	IT Equipment	SN901234	2021-12-05	Data Center	in use	4
5	Printer	Electronics	SN567890	2024-04-22	Finance Dept	decommissioned	5
6	Air Conditioner	Appliance	SN678901	2023-11-30	Admin Room	under maintenance	6
7	Router	Networking	SN789123	2025-01-18	IT Department	in use	7
8	Desk	Furniture	SN890123	2022-07-14	Marketing	in use	8
9	Security Camera	Surveillance	SN901345	2023-09-25	Security Office	under maintenance	9
10	Whiteboard	Office Supplies	SN456789	2021-03-05	Training Room	in use	10
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

3. maintenance_records table:

- maintenance_id (Primary Key): Unique identifier for each maintenance record.
- asset_id (Foreign Key): References the asset for which maintenance was performed.
- maintenance_date.

- description: Description of the maintenance activity.
- cost: Cost associated with the maintenance.

```
CREATE TABLE maintenance_records (
 maintenance_id INT PRIMARY KEY AUTO_INCREMENT,
asset_id INT NOT NULL,
maintenance_date DATE NOT NULL,
description TEXT,
cost DECIMAL(10,2),
  FOREIGN KEY (asset_id) REFERENCES assets(asset_id) ON DELETE CASCADE
);

INSERT INTO maintenance_records (asset_id, maintenance_date, description, cost)
VALUES
(1, '2023-06-15', 'Replaced battery', 150.00),
(2, '2021-08-20', 'Lens cleaning', 50.00),
(3, '2024-02-10', 'Wheel adjustment', 30.00),
(4, '2022-12-05', 'RAM Upgrade', 200.00),
(5, '2021-04-22', 'Cartridge replacement', 40.00),
(6, '2023-11-30', 'Coolant refill', 100.00),
(7, '2025-01-18', 'Firmware update', 80.00),
(8, '2022-07-14', 'Wood polishing', 20.00),
(9, '2023-09-25', 'Camera lens fix', 60.00),
(10, '2021-03-05', 'Surface cleaning', 15.00);
```



Result Grid | Filter Rows: | Edit: | Export/Im

maintenance_id	asset_id	maintenance_date	description	cost
1	1	2023-06-15	Replaced battery	150.00
2	2	2021-08-20	Lens cleaning	50.00
3	3	2024-02-10	Wheel adjustment	30.00
4	4	2022-12-05	RAM Upgrade	200.00
5	5	2021-04-22	Cartridge replacement	40.00
6	6	2023-11-30	Coolant refill	100.00
7	7	2025-01-18	Firmware update	80.00
8	8	2022-07-14	Wood polishing	20.00
9	9	2023-09-25	Camera lens fix	60.00
10	10	2021-03-05	Surface cleaning	15.00
NULL	NULL	NULL	NULL	NULL

4. asset_allocations table:

- allocation_id (Primary Key): Unique identifier for each asset allocation.
- asset_id (Foreign Key): References the asset that is allocated.
- employee_id (Foreign Key): References the employee to whom the asset is allocated.
- allocation_date: Date when the asset was allocated.
- return_date: Date when the asset was returned (if applicable).

```
CREATE TABLE asset_allocations (
    allocation_id INT PRIMARY KEY AUTO_INCREMENT,
    asset_id INT NOT NULL,
    employee_id INT NOT NULL,
    allocation_date DATE NOT NULL,
    return_date DATE,
    FOREIGN KEY (asset_id) REFERENCES assets(asset_id) ON DELETE CASCADE,
    FOREIGN KEY (employee_id) REFERENCES employees(employee_id) ON DELETE CASCADE
);
```

INSERT INTO asset_allocations (asset_id, employee_id, allocation_date, return_date)
VALUES

(1, 1, '2024-03-10', NULL),

(2, 2, '2022-09-15', '2023-09-15'),

(3, 3, '2023-01-20', NULL),

(4, 4, '2025-02-28', NULL),

(5, 5, '2024-07-05', '2024-08-05'),

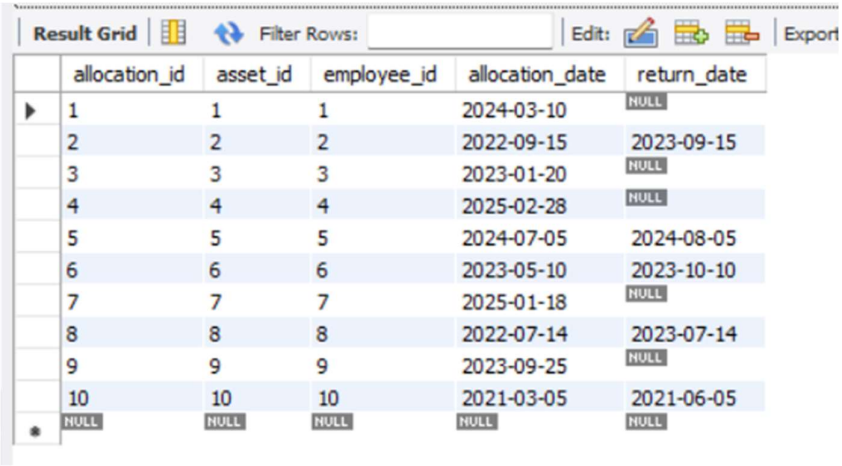
(6, 6, '2023-05-10', '2023-10-10'),

(7, 7, '2025-01-18', NULL),

(8, 8, '2022-07-14', '2023-07-14'),

(9, 9, '2023-09-25', NULL),

(10, 10, '2021-03-05', '2021-06-05');

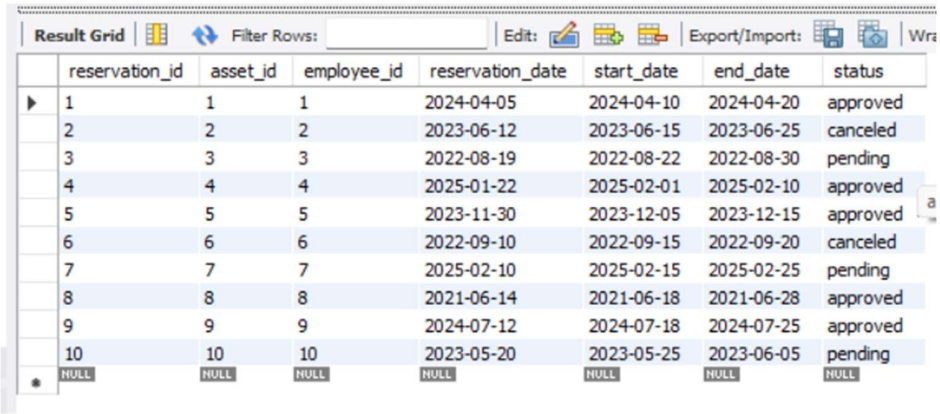


allocation_id	asset_id	employee_id	allocation_date	return_date
1	1	1	2024-03-10	NULL
2	2	2	2022-09-15	2023-09-15
3	3	3	2023-01-20	NULL
4	4	4	2025-02-28	NULL
5	5	5	2024-07-05	2024-08-05
6	6	6	2023-05-10	2023-10-10
7	7	7	2025-01-18	NULL
8	8	8	2022-07-14	2023-07-14
9	9	9	2023-09-25	NULL
10	10	10	2021-03-05	2021-06-05
NULL	NULL	NULL	NULL	NULL

5. reservations table (to store order details):

- reservation_id (Primary Key): Unique identifier for each reservation.
- asset_id (Foreign Key): References the asset that is being reserved.
- employee_id (Foreign Key): References the employee who made the reservation.
- reservation_date: Date when the reservation was made.
- start_date: Date when the reserved asset is needed.
- end_date: Date when the reservation ends.
- status: Status of the reservation (e.g., pending, approved, canceled).

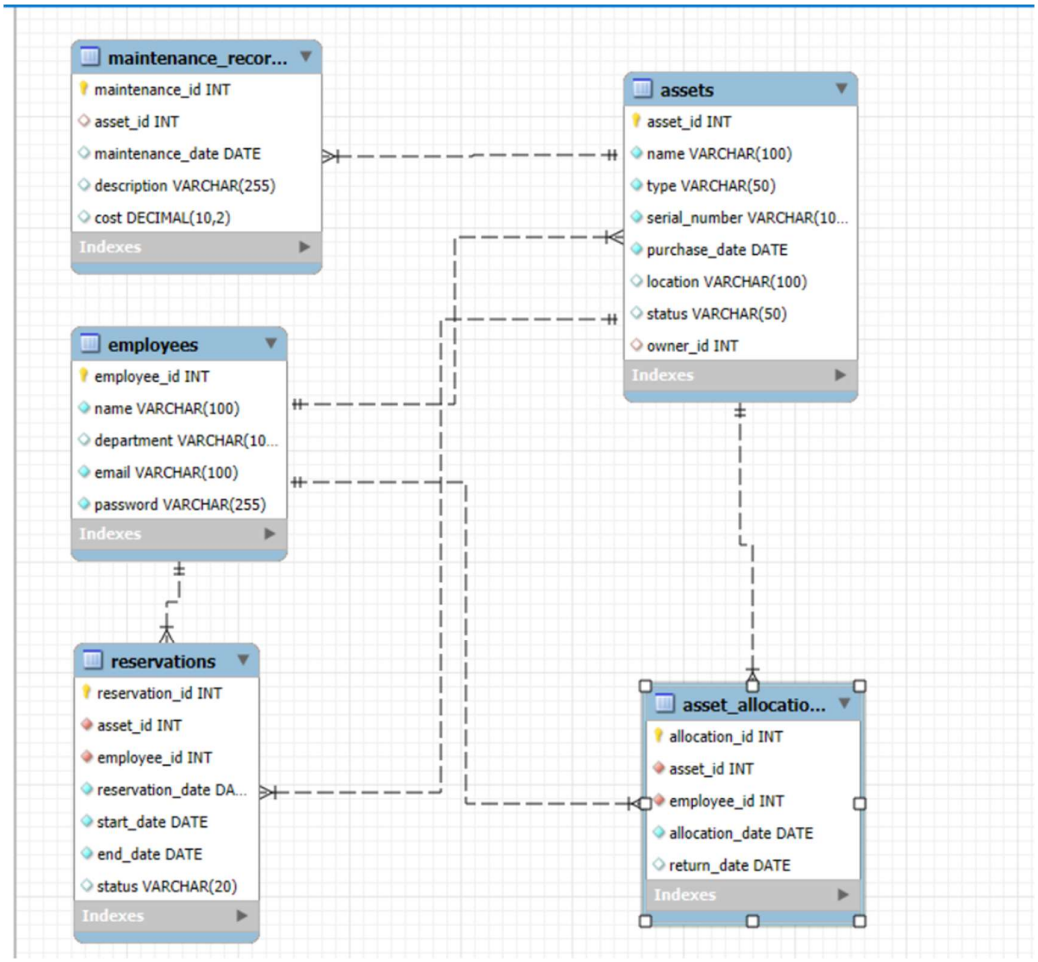
```
CREATE TABLE reservations (
    reservation_id INT PRIMARY KEY AUTO_INCREMENT,
    asset_id INT NOT NULL,
    employee_id INT NOT NULL,
    reservation_date DATE NOT NULL,
    start_date DATE NOT NULL,
    end_date DATE NOT NULL,
    status ENUM('pending', 'approved', 'canceled') NOT NULL,
    FOREIGN KEY (asset_id) REFERENCES assets(asset_id) ON DELETE CASCADE,
    FOREIGN KEY (employee_id) REFERENCES employees(employee_id) ON DELETE CASCADE
);
INSERT INTO reservations (asset_id, employee_id, reservation_date, start_date, end_date, status) VALUES
(1, 1, '2024-04-05', '2024-04-10', '2024-04-20', 'approved'),
(2, 2, '2023-06-12', '2023-06-15', '2023-06-25', 'canceled'),
(3, 3, '2022-08-19', '2022-08-22', '2022-08-30', 'pending'),
(4, 4, '2025-01-22', '2025-02-01', '2025-02-10', 'approved'),
(5, 5, '2023-11-30', '2023-12-05', '2023-12-15', 'approved'),
(6, 6, '2022-09-10', '2022-09-15', '2022-09-20', 'canceled'),
(7, 7, '2025-02-10', '2025-02-15', '2025-02-25', 'pending'),
(8, 8, '2021-06-14', '2021-06-18', '2021-06-28', 'approved'),
(9, 9, '2024-07-12', '2024-07-18', '2024-07-25', 'approved'),
(10, 10, '2023-05-20', '2023-05-25', '2023-06-05', 'pending');
```



Result Grid | Filter Rows: | Edit: | Export/Import:

reservation_id	asset_id	employee_id	reservation_date	start_date	end_date	status
1	1	1	2024-04-05	2024-04-10	2024-04-20	approved
2	2	2	2023-06-12	2023-06-15	2023-06-25	canceled
3	3	3	2022-08-19	2022-08-22	2022-08-30	pending
4	4	4	2025-01-22	2025-02-01	2025-02-10	approved
5	5	5	2023-11-30	2023-12-05	2023-12-15	approved
6	6	6	2022-09-10	2022-09-15	2022-09-20	canceled
7	7	7	2025-02-10	2025-02-15	2025-02-25	pending
8	8	8	2021-06-14	2021-06-18	2021-06-28	approved
9	9	9	2024-07-12	2024-07-18	2024-07-25	approved
10	10	10	2023-05-20	2023-05-25	2023-06-05	pending
NULL	NULL	NULL	NULL	NULL	NULL	NULL

ER Diagram:



Entity Descriptions:

1. Asset

- **Description: Represents the physical or digital asset owned by the organization.**
- **Attributes:**
    - asset_id (Primary Key): Unique identifier for each asset.
    - name: Name of the asset.
    - asset_type: Type/category of the asset (e.g., Laptop, Printer).
    - serial_number: Manufacturer-assigned serial number.
    - purchase_date: Date on which the asset was purchased.
    - location: Physical location of the asset.

- status: Indicates if the asset is available, allocated, under maintenance, or reserved.
- owner_id: Foreign key linking to the employee responsible for the asset.

2. Employee

- **Description:** Stores information about employees who are responsible for or assigned to assets.
- **Attributes:**
  - employee_id (Primary Key): Unique identifier for each employee.
  - name: Name of the employee.
  - department: Department the employee belongs to.
  - email: Contact email of the employee.

3. Maintenance_Record

- **Description:** Tracks maintenance activities performed on assets.
- **Attributes:**
  - maintenance_id (Primary Key): Unique ID for each maintenance entry.
  - asset_id (Foreign Key): Asset on which maintenance was performed.
  - maintenance_date: Date of the maintenance.
  - description: Details of the maintenance activity.
  - cost: Cost incurred during maintenance.

4. Asset_Allocation

- **Description:** Stores details of asset allocations to employees.
- **Attributes:**
  - allocation_id (Primary Key): Unique identifier for each allocation record.
  - asset_id (Foreign Key): Asset being allocated.
  - employee_id (Foreign Key): Employee receiving the asset.
  - allocation_date: Date on which the asset was assigned.
  - return_date: Date when the asset was returned (nullable for ongoing allocations).

5. Reservation

- **Description:** Handles reservations made by employees for using an asset during a specific period.
- **Attributes:**
    - reservation_id (Primary Key): Unique reservation identifier.
    - asset_id (Foreign Key): Asset being reserved.
    - employee_id (Foreign Key): Employee making the reservation.
    - reservation_date: Date when the reservation was made.
    - start_date: Starting date of asset usage.
    - end_date: Ending date of asset usage.
    - status: Status of reservation (e.g., reserved, cancelled).

- These entities are connected through primary and foreign key relationships as shown in the ER diagram.
- The database design ensures data integrity and supports efficient querying and reporting for asset management tasks.

Create the model/entity classes corresponding to the schema within package entity with variables declared private, constructors(default and parametrized) and getters,setters methods:

Description:

Model/entity classes were implemented in the entity package based on the database schema. Each class represents a table, including Asset, Employee, Reservation, AssetAllocation, and MaintenanceRecord. Variables were declared private to ensure encapsulation. Both default and parameterized constructors were provided for flexible object creation. Getter and setter methods were defined for each variable to enable controlled access and modification of the data.

entity/asset.py:

```
class Asset:
   def __init__(self, asset_id=None, name=None, asset_type=None, serial_number=None,
purchase_date=None,
                location=None, status=None, owner_id=None):
```

```
        self.__asset_id = asset_id
        self.__name = name
        self.__asset_type = asset_type
        self.__serial_number = serial_number
        self.__purchase_date = purchase_date
        self.__location = location
        self.__status = status
        self.__owner_id = owner_id
```

entity/asset_allocation.py:

```
class AssetAllocation:
    def __init__(self, allocation_id=None, asset_id=None, employee_id=None,
allocation_date=None, return_date=None):
        self.__allocation_id = allocation_id
        self.__asset_id = asset_id
        self.__employee_id = employee_id
        self.__allocation_date = allocation_date
        self.__return_date = return_date
```

entity/employee.py:

```
class Employee:
    def __init__(self, employee_id=None, name=None, department=None, email=None,
password=None):
        self.__employee_id = employee_id
        self.__name = name
        self.__department = department
        self.__email = email
        self.__password = password

    # Getters
    def get_employee_id(self):
        return self.__employee_id

    def get_name(self):
        return self.__name

    def get_department(self):
        return self.__department

    def get_email(self):
```

```
        return self.__email

    def get_password(self):
        return self.__password

    # Setters
    def set_name(self, name):
        self.__name = name

    def set_department(self, department):
        self.__department = department

    def set_email(self, email):
        self.__email = email

    def set_password(self, password):
        self.__password = password
```

entity/maintenance_record.py:

```
class MaintenanceRecord:
    def __init__(self, maintenance_id=None, asset_id=None, maintenance_date=None,
description=None, cost=None):
        self.__maintenance_id = maintenance_id
        self.__asset_id = asset_id
        self.__maintenance_date = maintenance_date
        self.__description = description
        self.__cost = cost
```

entity/reservation.py:

```
class Reservation:
    def __init__(self, reservation_id=None, asset_id=None, employee_id=None,
reservation_date=None,
                 start_date=None, end_date=None, status=None):
        self.__reservation_id = reservation_id
        self.__asset_id = asset_id
        self.__employee_id = employee_id
        self.__reservation_date = reservation_date
        self.__start_date = start_date
        self.__end_date = end_date
        self.__status = status
```

6. Service Provider Interface/Abstract class:

Keep the interfaces and implementation classes in package dao

• Define an AssetManagementService interface/abstract class with methods for adding/removing asset and its management. The following methods will interact with database.

a. Add Asset:

i. Method: boolean addAsset(Asset asset)

ii. Description: Adds a new asset to the system.

b. Update Asset:

i. Method: boolean updateAsset(Asset asset)

ii. Description: Updates information about an existing asset.

c. Delete Asset:

i. Method: boolean deleteAsset(int assetId)

ii. Description: Deletes an asset from the system based on its ID.

d. Allocate Asset:

i. Method: boolean allocateAsset(int assetId, int employeeId, String allocationDate)

ii. Description: Allocates an asset to an employee on a specified allocation date.

e. Deallocate Asset:

i. Method: boolean deallocateAsset(int assetId, int employeeId, String returnDate)

ii. Description: Deallocates an asset from an employee on a specified return date.

f. Perform Maintenance:

i. Method: boolean performMaintenance(int assetId, String maintenanceDate, String description, double cost)

ii. Description: Records maintenance activity for an asset, including the date, description, and cost.

g. Reserve Asset:

i. Method: boolean reserveAsset(int assetId, int employeeId, String reservationDate, String startDate, String endDate)

ii. Description: Reserves an asset for a specified employee for a specific period, starting from the start date to the end date. The reservation is made on the reservation date.

h. Withdraw Reservation:

i. Method: boolean withdrawReservation(int reservationId)

ii. Description: Withdraws a reservation for an asset identified by the reservation ID. The reserved asset becomes available for allocation again.

Description:

The AssetManagementService interface in the dao package defines the contract for managing assets and their lifecycle. It includes methods to add, update, delete, allocate, deallocate, maintain, reserve, and withdraw reservations for assets. Each method is designed to interact with the MySQL database and handle core functionalities like inserting, updating, or deleting records in corresponding tables. The AssetManagementServiceImpl class implements all these methods and provides the actual logic using SQL queries for each operation. This separation ensures clean architecture and makes the system modular and testable.

dao/asset_management_service.py:

```
from abc import ABC, abstractmethod

class AssetManagementService(ABC):

    @abstractmethod
    def add_asset(self, name, asset_type, serial_number, purchase_date, location, status,
owner_id):
        pass

    @abstractmethod
    def update_asset(self, asset_id, name=None, location=None, status=None):
        pass

    @abstractmethod
    def delete_asset(self, asset_id):
        pass

    @abstractmethod
    def allocate_asset(self, asset_id, employee_id, allocation_date):
        pass

    @abstractmethod
    def deallocate_asset(self, asset_id):
        pass

    @abstractmethod
    def perform_maintenance(self, asset_id, maintenance_date, description, cost):
        pass

    @abstractmethod
```

```
    def reserve_asset(self, asset_id, employee_id, reservation_date, start_date, end_date,
status):
        pass
```

7. Implement the above interface in a class called AssetManagementServiceImpl in package dao:

dao/asset_management_service_impl.py:

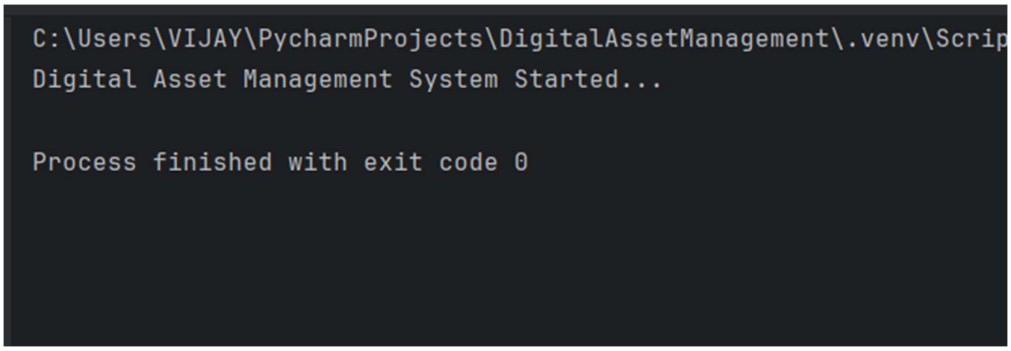
```
from dao.asset_management_service import AssetManagementService
import mysql.connector
from myexceptions.asset_not_found_exception import AssetNotFoundException

class AssetManagementServiceImpl(AssetManagementService):

    def __init__(self):
        self.conn = mysql.connector.connect(
            host='localhost',
            user='root',
            password='#vijaysql**',
            database='digital_asset'
        )
        self.cursor = self.conn.cursor()

    def add_asset(self, name, asset_type, serial_number, purchase_date, location, status,
owner_id):
        self.cursor.execute("""
            INSERT INTO assets (name, type, serial_number, purchase_date, location, status,
owner_id)
            VALUES (%s, %s, %s, %s, %s, %s, %s)
        """, (name, asset_type, serial_number, purchase_date, location, status, owner_id))
        self.conn.commit()

    def update_asset(self, asset_id, name=None, location=None, status=None):
        updates = []
        values = []
        if name:
            updates.append("name=%s")
            values.append(name)
        if location:
            updates.append("location=%s")
            values.append(location)
        if status:
            updates.append("status=%s")
```

```
            values.append(status)
        if updates:
            query = f"UPDATE assets SET {', '.join(updates)} WHERE asset_id=%s"
            values.append(asset_id)
            self.cursor.execute(query, tuple(values))
            self.conn.commit()

    def delete_asset(self, asset_id):
        self.cursor.execute("DELETE FROM assets WHERE asset_id=%s", (asset_id,))
        self.conn.commit()

    def allocate_asset(self, asset_id, employee_id, allocation_date):
        self.cursor.execute("SELECT * FROM assets WHERE asset_id=%s", (asset_id,))
        asset = self.cursor.fetchone()
        if not asset:
            raise AssetNotFoundException("Asset not found.")
        if asset[6] != "available":
            print("Asset is not available for allocation.")
            return None
        self.cursor.execute("""
            INSERT INTO asset_allocations (asset_id, employee_id, allocation_date)
            VALUES (%s, %s, %s)
        """, (asset_id, employee_id, allocation_date))
        self.cursor.execute("UPDATE assets SET status='allocated' WHERE asset_id=%s",
(asset_id,))
        self.conn.commit()
        return True

    def deallocate_asset(self, asset_id):
        self.cursor.execute("DELETE FROM asset_allocations WHERE asset_id=%s",
(asset_id,))
        self.cursor.execute("UPDATE assets SET status='available' WHERE asset_id=%s",
(asset_id,))
        self.conn.commit()

    def perform_maintenance(self, asset_id, maintenance_date, description, cost):
        self.cursor.execute("SELECT * FROM assets WHERE asset_id=%s", (asset_id,))
        asset = self.cursor.fetchone()
        if not asset:
            raise AssetNotFoundException("Asset not found.")
        self.cursor.execute("""
            INSERT INTO maintenance_records (asset_id, maintenance_date, description, cost)
            VALUES (%s, %s, %s, %s)
        """, (asset_id, maintenance_date, description, cost))
        self.conn.commit()
```

```
    def reserve_asset(self, asset_id, employee_id, reservation_date, start_date, end_date,
status):
        self.cursor.execute("""
            INSERT INTO reservations (asset_id, employee_id, reservation_date, start_date,
end_date, status)
            VALUES (%s, %s, %s, %s, %s, %s)
        """, (asset_id, employee_id, reservation_date, start_date, end_date, status))
        self.conn.commit()

    def withdraw_reservation(self, reservation_id):
        conn = None
        cursor = None
        try:
            conn = self.db.get_connection()
            cursor = conn.cursor()
            get_query = """select r.asset_id, a.status
                        from reservations r
                        join assets a on r.asset_id = a.asset_id
                        where r.reservation_id = %s"""
            cursor.execute(get_query, (reservation_id,))
            reservation = cursor.fetchone()
            if not reservation:
                raise AssetNotFoundException("reservation id not found.")
            asset_id, current_status = reservation
            if current_status.lower() != 'reserved':
                print(f"cannot withdraw reservation - asset is not reserved (current status:
{current_status})")
                return False
            update_asset_query = "update assets set status = 'available' where asset_id = %s"
            cursor.execute(update_asset_query, (asset_id,))
            update_reservation_query = "update reservations set status = 'withdrawn' where
reservation_id = %s"
            cursor.execute(update_reservation_query, (reservation_id,))
            conn.commit()
            print("reservation withdrawn successfully! asset is now available.")
            return True
        except mysql.connector.Error as e:
            print(f"database error: {e}")
            return False
        finally:
            if cursor is not None:
                cursor.close()
            if conn is not None:
                conn.close()
```

main.py:

```
from dao.asset_management_service_impl import AssetManagementServiceImpl

if __name__ == "__main__":
    service = AssetManagementServiceImpl()
    print("Digital Asset Management System Started...")
```



Connect your application to the SQL database:

8. Write code to establish a connection to your SQL database.

- Create a utility class DBConnection in a package util with a static variable connection of Type Connection and a static method getConnection() which returns connection.
- Connection properties supplied in the connection string should be read from a property file.

Description:

The DBConnection class in the util package establishes and manages the connection to the MySQL database using Python's mysql.connector module. It defines a static method get_connection() that returns a connection object. The database credentials such as host, user, password, and database name are stored in a db.properties file and loaded using Python's configparser module. This setup ensures secure, centralized, and reusable database connectivity, enabling consistent access for all operations across the system.

util/DBConnection.py:

```
import mysql.connector
from util.DBPropertyUtil import get_db_properties

class DBConnection:
    @staticmethod
    def get_connection():
```

```
    db_props = get_db_properties()
    return mysql.connector.connect(
        host=db_props["host"],
        port=db_props["port"],
        user=db_props["user"],
        password=db_props["password"],
        database=db_props["database"]
    )
```

util/DBPropertyUtil.py:

```
import configparser
import os

def get_db_properties():
    config = configparser.ConfigParser()
    properties_file = os.path.join(os.path.dirname(__file__), "db.properties")

    if not os.path.exists(properties_file):
        raise FileNotFoundError(f"Database properties file not found: {properties_file}")

    config.read(properties_file)

    try:
        return {
            "host": config.get("DEFAULT", "db.host"),
            "port": config.get("DEFAULT", "db.port"),
            "user": config.get("DEFAULT", "db.user"),
            "password": config.get("DEFAULT", "db.password"),
            "database": config.get("DEFAULT", "db.name"),
        }
    except Exception as e:
        raise Exception(f"Error reading database properties: {e}")
```

util/db.properties:

```
[DEFAULT]
db.host=localhost
db.port=3306
db.user=root
db.password=your password
db.name=database_name
```

Main.py:

```
from util.DBConnection import DBConnection

if __name__ == "__main__":
    conn = DBConnection.get_connection()
    if conn:
        print("Connected to the Database successfully!")
```

```
C:\Users\VIJAY\PycharmProjects\DigitalAssetManagement\.venv\Scripts\pyt
Connected to the Database successfully!

Process finished with exit code 0
```

9. Create the exceptions in package myexceptions and create the following custom exceptions and throw them in methods whenever needed. Handle all the exceptions in main method

- **AssetNotFoundException:** throw this exception when employee enters an invalid asset id which doesn't exist in db
- **AssetNotMaintainException:** throw this exception when employee need the asset which is not maintained for 2 years.

Description:

To ensure robust error management and improve system reliability, custom exceptions were implemented within a dedicated package named myexceptions. This task involved creating and integrating two domain-specific exception classes that handle asset-related error scenarios in a user-friendly and controlled manner.

1. **AssetNotFoundException**
   This exception is thrown when a user attempts to access or manipulate an asset using an invalid asset ID that does not exist in the database. It ensures that operations like allocation, maintenance, or reservation are not performed on nonexistent records. This improves system consistency by validating input against existing data.
2. **AssetNotMaintainException**
   This exception is thrown when a user tries to allocate or reserve an asset that has not been maintained for over two years. It enforces the business rule that outdated or

unmaintained assets should not be issued to employees, ensuring operational safety and compliance with maintenance policies.

Both exceptions were triggered programmatically from service layer methods based on validation logic. Exception messages provide informative feedback to the user, and all raised exceptions are caught and handled in the main driver class (AssetManagementApp) using appropriate try-except blocks. This separation of exception logic promotes better modularity, maintainability, and error traceability in the application.

myexceptions/asset_not_found_exception.py:

```
class AssetNotFoundException(Exception):
    def __init__(self, message="Asset not found."):
        super().__init__(message)
```

myexceptions/asset_not_maintain_exception.py:

```
class AssetNotMaintainException(Exception):
    def __init__(self, message="Asset has not been maintained for more than 2 years."):
        super().__init__(message)
```

Main.py:

```
from dao.asset_management_service_impl import AssetManagementServiceImpl
from myexceptions.asset_not_found_exception import AssetNotFoundException
from myexceptions.asset_not_maintain_exception import AssetNotMaintainException

if __name__ == "__main__":
    service = AssetManagementServiceImpl()

    try:
        asset_id = int(input("Enter Asset ID: "))
        asset = service.get_asset_by_id(asset_id)
        print("Asset Details:", asset)

        service.check_asset_maintenance(asset_id)
        print("Asset is properly maintained.")

    except AssetNotFoundException as e:
        print("Error:", e)

    except AssetNotMaintainException as e:
        print("Warning:", e)
```

```
except Exception as e:
    print("Unexpected error:", e)
```

10. Create class named AssetManagementApp with main method in app Trigger all the methods in service implementation class by user choose operation from the following menu.

- Add Asset:
- Update Asset:
- Delete Asset: • Allocate Asset:
- Deallocate Asset:
- Perform Maintenance:
- Reserve Asset:

Description:

A main class named AssetManagementApp was created within the app package to serve as the central interface for user interaction with the system. This class provides a menu-driven console application that allows users to perform all major operations supported by the Digital Asset Management System.

Upon execution, the application displays a numbered list of options, allowing users to select operations such as adding, updating, deleting, allocating, deallocating assets, performing

maintenance, and reserving assets. Based on the user's choice, the corresponding method from the AssetManagementServiceImpl class is triggered.

Each operation prompts the user to input required information (such as asset details or employee IDs), which is then validated and passed to the service layer. The service layer in turn interacts with the MySQL database to perform the requested action, and informative success or failure messages are printed to the console.

This main class also includes exception handling to manage scenarios where the user inputs invalid asset IDs or attempts operations on assets that haven't been maintained as per the defined business rules. Custom exceptions such as AssetNotFoundException and AssetNotMaintainException are caught and handled gracefully to provide clear feedback.

By providing an intuitive, text-based menu and routing logic, AssetManagementApp bridges the gap between users and backend services, allowing for complete system control in a user-friendly manner.

app/asset_management_app.py:

```
from dao.asset_management_service_impl import AssetManagementServiceImpl
from myexceptions.asset_not_found_exception import AssetNotFoundException
from myexceptions.asset_not_maintain_exception import AssetNotMaintainException

class AssetManagementApp:
    def __init__(self):
        self.service = AssetManagementServiceImpl()

    def display_menu(self):
        print("\n=== Asset Management System ===")
        print("1. Add Asset")
        print("2. Update Asset")
        print("3. Delete Asset")
        print("4. Allocate Asset")
        print("5. Deallocate Asset")
        print("6. Perform Maintenance")
        print("7. Reserve Asset")
        print("8. Exit")

    def run(self):
        while True:
            self.display_menu()
            choice = input("Enter your choice: ")

            try:
                if choice == "1":
                    name = input("Enter asset name: ")
                    asset_type = input("Enter asset type: ")
```

```
            serial_number = input("Enter serial number: ")
            purchase_date = input("Enter purchase date (YYYY-MM-DD): ")
            location = input("Enter location: ")
            status = input("Enter status (in use / decommissioned / under maintenance): ")
            owner_id = input("Enter owner ID: ")
            self.service.add_asset(name, asset_type, serial_number, purchase_date, location,
status, owner_id)

        elif choice == "2":
            asset_id = int(input("Enter asset ID to update: "))
            new_location = input("Enter new location: ")
            new_status = input("Enter new status: ")
            self.service.update_asset(asset_id, new_location, new_status)

        elif choice == "3":
            asset_id = int(input("Enter asset ID to delete: "))
            self.service.delete_asset(asset_id)

        elif choice == "4":
            asset_id = int(input("Enter asset ID to allocate: "))
            employee_id = int(input("Enter employee ID: "))
            allocation_date = input("Enter allocation date (YYYY-MM-DD): ")
            self.service.allocate_asset(asset_id, employee_id, allocation_date)

        elif choice == "5":
            asset_id = int(input("Enter asset ID to deallocate: "))
            self.service.deallocate_asset(asset_id)

        elif choice == "6":
            asset_id = int(input("Enter asset ID for maintenance: "))
            maintenance_date = input("Enter maintenance date (YYYY-MM-DD): ")
            description = input("Enter maintenance description: ")
            cost = float(input("Enter maintenance cost: "))
            self.service.perform_maintenance(asset_id, maintenance_date, description, cost)

        elif choice == "7":
            asset_id = int(input("Enter asset ID to reserve: "))
            employee_id = int(input("Enter employee ID: "))
            reservation_date = input("Enter reservation date (YYYY-MM-DD): ")
            start_date = input("Enter start date (YYYY-MM-DD): ")
            end_date = input("Enter end date (YYYY-MM-DD): ")
            self.service.reserve_asset(asset_id, employee_id, reservation_date, start_date,
end_date)

        elif choice == "8":
            print("Exiting the application...")
```

```
                break

            else:
                print("Invalid choice! Please try again.")

        except AssetNotFoundException as e:
            print(f"Error: {e}")

        except AssetNotMaintainException as e:
            print(f"Error: {e}")

        except Exception as e:
            print(f"An unexpected error occurred: {e}")

if __name__ == "__main__":
    app = AssetManagementApp()
    app.run()
```

OUTPUT

Add Asset:

```
=== Asset Management System ===
1. Add Asset
2. Update Asset
3. Delete Asset
4. Allocate Asset
5. Deallocate Asset
6. Perform Maintenance
7. Reserve Asset
8. Exit
Enter your choice: 1
Enter asset name: Mobile
Enter asset type: Electronics
Enter serial number: SN654321
Enter purchase date (YYYY-MM-DD): 2025-03-30
Enter location: HR Cabin
Enter status (in use / decommissioned / under maintenance): in use
Enter owner ID: 2
✅ Asset added successfully!
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

asset_id	name	type	serial_number	purchase_date	location	status	owner_id
1	Laptop	Electronics	SN123456	2021-06-15	IT Office	in use	1
2	Projector	Electronics	SN789012	2022-08-20	Conference Room	under maintenance	2
3	Office Chair	Furniture	SN345678	2023-02-10	HR Cabin	in use	3
4	Server	IT Equipment	SN901234	2021-12-05	Data Center	in use	4
5	Printer	Electronics	SN567890	2024-04-22	Finance Dept	decommissioned	5
6	Air Conditioner	Appliance	SN678901	2023-11-30	Admin Room	under maintenance	6
7	Router	Networking	SN789123	2025-01-18	IT Department	in use	7
8	Desk	Furniture	SN890123	2022-07-14	Marketing	in use	8
9	Security Camera	Surveillance	SN901345	2023-09-25	Security Office	under maintenance	9
10	Whiteboard	Office Supplies	SN456789	2021-03-05	Training Room	in use	10
12	Mobile	Electronics	SN654321	2025-03-30	HR Cabin	in use	2
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Update Asset:

```
=== Asset Management System ===
1. Add Asset
2. Update Asset
3. Delete Asset
4. Allocate Asset
5. Deallocate Asset
6. Perform Maintenance
7. Reserve Asset
8. Exit
Enter your choice: 2
Enter asset ID to update: 3
Enter new location: Office 2
Enter new status: under maintenance
✅ Asset updated successfully!
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

asset_id	name	type	serial_number	purchase_date	location	status	owner_id
1	Laptop	Electronics	SN123456	2021-06-15	IT Office	in use	1
2	Projector	Electronics	SN789012	2022-08-20	Conference Room	under maintenance	2
3	Office Chair	Furniture	SN345678	2023-02-10	Office 2	under maintenance	3
4	Server	IT Equipment	SN901234	2021-12-05	Data Center	in use	4
5	Printer	Electronics	SN567890	2024-04-22	Finance Dept	decommissioned	5
6	Air Conditioner	Appliance	SN678901	2023-11-30	Admin Room	under maintenance	6
7	Router	Networking	SN789123	2025-01-18	IT Department	in use	7
8	Desk	Furniture	SN890123	2022-07-14	Marketing	in use	8
9	Security Camera	Surveillance	SN901345	2023-09-25	Security Office	under maintenance	9
10	Whiteboard	Office Supplies	SN456789	2021-03-05	Training Room	in use	10
12	Mobile	Electronics	SN654321	2025-03-30	HR Cabin	in use	2
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Delete Asset:

```
=== Asset Management System ===
1. Add Asset
2. Update Asset
3. Delete Asset
4. Allocate Asset
5. Deallocate Asset
6. Perform Maintenance
7. Reserve Asset
8. Exit
Enter your choice: 3
Enter asset ID to delete: 3
✅ Asset deleted successfully!
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

asset_id	name	type	serial_number	purchase_date	location	status	owner_id
1	Laptop	Electronics	SN123456	2021-06-15	IT Office	in use	1
2	Projector	Electronics	SN789012	2022-08-20	Conference Room	under maintenance	2
4	Server	IT Equipment	SN901234	2021-12-05	Data Center	in use	4
5	Printer	Electronics	SN567890	2024-04-22	Finance Dept	decommissioned	5
6	Air Conditioner	Appliance	SN678901	2023-11-30	Admin Room	under maintenance	6
7	Router	Networking	SN789123	2025-01-18	IT Department	in use	7
8	Desk	Furniture	SN890123	2022-07-14	Marketing	in use	8
9	Security Camera	Surveillance	SN901345	2023-09-25	Security Office	under maintenance	9
10	Whiteboard	Office Supplies	SN456789	2021-03-05	Training Room	in use	10
13	Mobile	Electronics	SN654321	2025-03-30	HR Cabin	in use	2
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Allocate Asset:

```
=== Asset Management System ===
1. Add Asset
2. Update Asset
3. Delete Asset
4. Allocate Asset
5. Deallocate Asset
6. Perform Maintenance
7. Reserve Asset
8. Exit
Enter your choice: 4
Enter asset ID to allocate: 1
Enter employee ID: 1
Enter allocation date (YYYY-MM-DD): 2025-02-20
✅ Asset allocated successfully!
```

Result Grid | Filter Rows: | Edit: | Export/Im

allocation_id	asset_id	employee_id	allocation_date	return_date
1	1	1	2024-03-10	NULL
2	2	2	2022-09-15	2023-09-15
3	3	3	2023-01-20	NULL
4	4	4	2025-02-28	NULL
5	5	5	2024-07-05	2024-08-05
6	6	6	2023-05-10	2023-10-10
7	7	7	2025-01-18	NULL
8	8	8	2022-07-14	2023-07-14
9	9	9	2023-09-25	NULL
10	10	10	2021-03-05	2021-06-05
11	1	1	2025-02-20	NULL
NULL	NULL	NULL	NULL	NULL

Deallocate Asset:

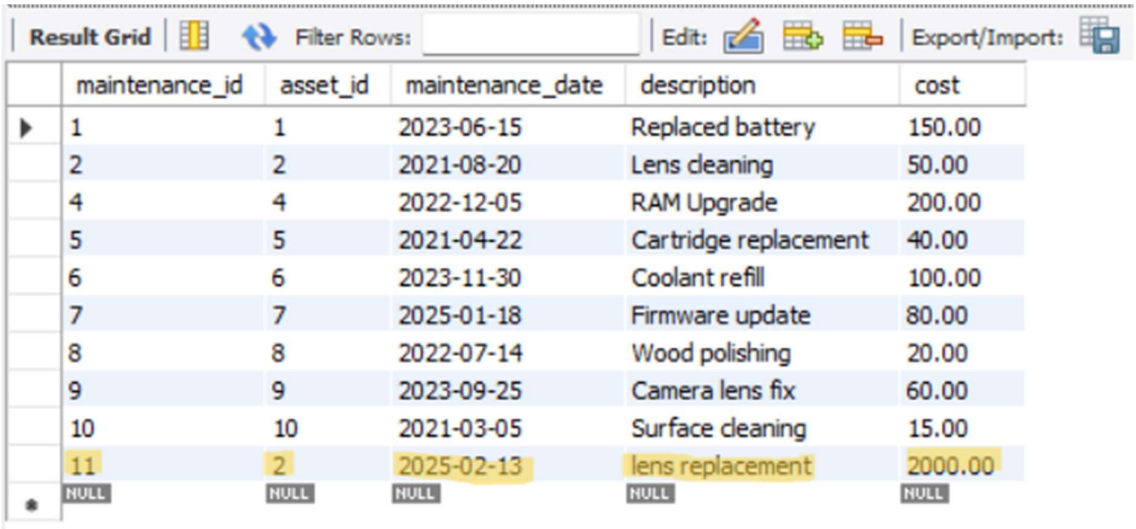
```
=== Asset Management System ===
1. Add Asset
2. Update Asset
3. Delete Asset
4. Allocate Asset
5. Deallocate Asset
6. Perform Maintenance
7. Reserve Asset
8. Exit
Enter your choice: 5
Enter asset ID to deallocate: 1
✅ Asset deallocated successfully!
```

Result Grid | Filter Rows: | Edit: | Export/

allocation_id	asset_id	employee_id	allocation_date	return_date
1	1	1	2024-03-10	2025-03-31
2	2	2	2022-09-15	2023-09-15
4	4	4	2025-02-28	NULL
5	5	5	2024-07-05	2024-08-05
6	6	6	2023-05-10	2023-10-10
7	7	7	2025-01-18	NULL
8	8	8	2022-07-14	2023-07-14
9	9	9	2023-09-25	NULL
10	10	10	2021-03-05	2021-06-05
11	1	1	2025-02-20	2025-03-31
NULL	NULL	NULL	NULL	NULL

Perform Maintenance:

```
=== Asset Management System ===
1. Add Asset
2. Update Asset
3. Delete Asset
4. Allocate Asset
5. Deallocate Asset
6. Perform Maintenance
7. Reserve Asset
8. Exit
Enter your choice: 6
Enter asset ID for maintenance: 2
Enter maintenance date (YYYY-MM-DD): 2025-02-13
Enter maintenance description: lens replacement
Enter maintenance cost: 2000
✅ Maintenance record added successfully!
```

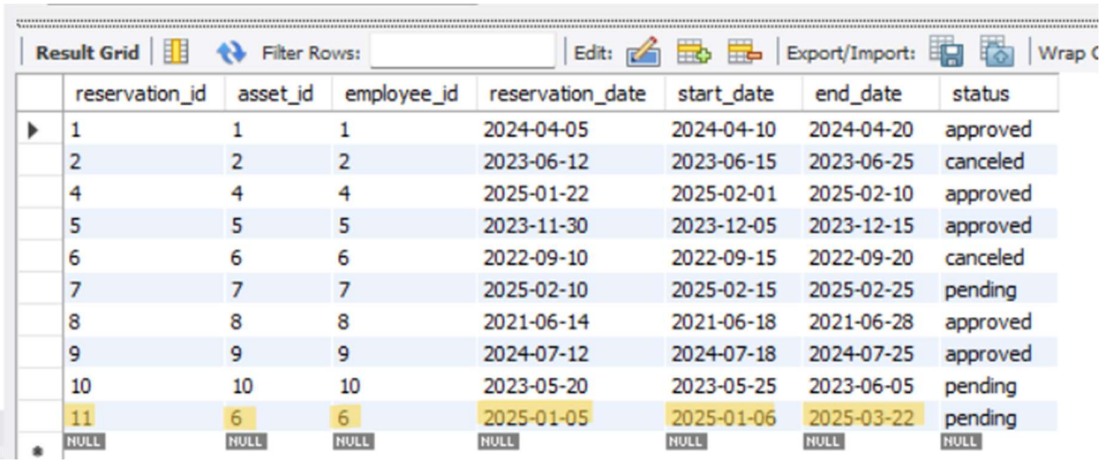


Result Grid | Filter Rows: | Edit: | Export/Import:

maintenance_id	asset_id	maintenance_date	description	cost
1	1	2023-06-15	Replaced battery	150.00
2	2	2021-08-20	Lens cleaning	50.00
4	4	2022-12-05	RAM Upgrade	200.00
5	5	2021-04-22	Cartridge replacement	40.00
6	6	2023-11-30	Coolant refill	100.00
7	7	2025-01-18	Firmware update	80.00
8	8	2022-07-14	Wood polishing	20.00
9	9	2023-09-25	Camera lens fix	60.00
10	10	2021-03-05	Surface cleaning	15.00
11	2	2025-02-13	lens replacement	2000.00
NULL	NULL	NULL	NULL	NULL

Reserve Asset:

```
=== Asset Management System ===
1. Add Asset
2. Update Asset
3. Delete Asset
4. Allocate Asset
5. Deallocate Asset
6. Perform Maintenance
7. Reserve Asset
8. Exit
Enter your choice: 7
Enter asset ID to reserve: 6
Enter employee ID: 6
Enter reservation date (YYYY-MM-DD): 2025-01-05
Enter start date (YYYY-MM-DD): 2025-01-06
Enter end date (YYYY-MM-DD): 2025-03-22
✅ Asset reserved successfully!
```



Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap

reservation_id	asset_id	employee_id	reservation_date	start_date	end_date	status
1	1	1	2024-04-05	2024-04-10	2024-04-20	approved
2	2	2	2023-06-12	2023-06-15	2023-06-25	canceled
4	4	4	2025-01-22	2025-02-01	2025-02-10	approved
5	5	5	2023-11-30	2023-12-05	2023-12-15	approved
6	6	6	2022-09-10	2022-09-15	2022-09-20	canceled
7	7	7	2025-02-10	2025-02-15	2025-02-25	pending
8	8	8	2021-06-14	2021-06-18	2021-06-28	approved
9	9	9	2024-07-12	2024-07-18	2024-07-25	approved
10	10	10	2023-05-20	2023-05-25	2023-06-05	pending
11	6	6	2025-01-05	2025-01-06	2025-03-22	pending
NULL	NULL	NULL	NULL	NULL	NULL	NULL

Unit Testing

11. Create Unit test cases for Digital Asset Management System are essential to ensure the correctness and reliability of your system. Following questions to guide the creation of Unit test cases:

- Write test case to test asset created successfully or not.
- Write test case to test asset is added to maintenance successfully or not.
- Write test case to test asset is reserved successfully or not.
- write test case to test exception is thrown correctly or not when employee id or asset id not found in database.

Description:

Unit testing was implemented as a crucial step in ensuring the correctness, reliability, and robustness of the Digital Asset Management System. A test module named test_asset_management.py was created under the test directory using Python's built-in unittest framework. These test cases systematically verify that each major functionality of the system works as intended.

The unit tests are structured to validate the following operations:

- Asset Creation Test: This test verifies whether a new asset is successfully added to the database with all expected attributes (such as name, type, and purchase date). A positive assertion confirms that the method returns True when the asset is created without errors.
- Maintenance Record Test: This test checks if an asset can be linked to a valid maintenance record with correct details including date, description, and cost. It ensures that the system can track the service history of assets as expected.
- Asset Reservation Test: This test confirms whether the system allows an employee to reserve an asset within a specified date range. It validates the reservation insertion logic and ensures proper foreign key references are handled.
- Exception Handling Test: Custom exceptions like AssetNotFoundException and AssetNotMaintainException are tested to confirm they are raised when the user tries to perform operations on nonexistent or non-maintained assets. These negative test cases validate the robustness of error handling in real-world scenarios.

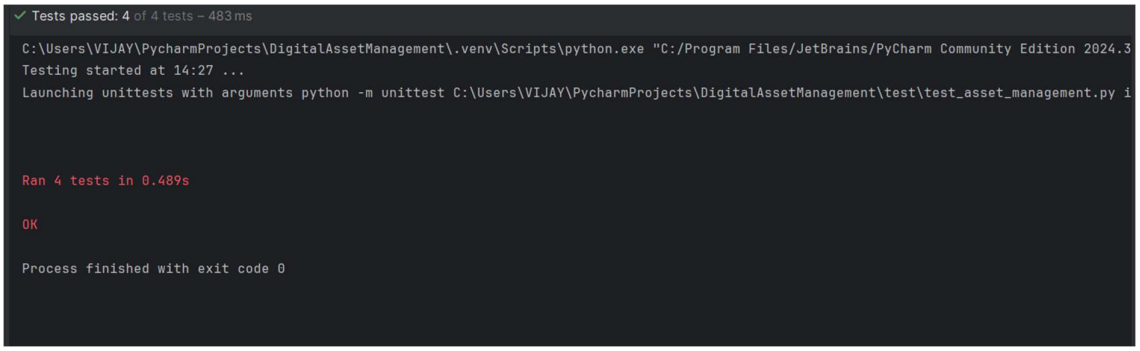
These unit tests interact with the MySQL database using test data and are automatically executed to catch functional errors early in the development lifecycle. Any database-related exceptions (such as missing fields or invalid references) are logged and reviewed for fixing.

Overall, the unit testing strategy ensures that the system behaves consistently under both normal and erroneous inputs, and supports smooth future enhancements by providing a reliable testing foundation.

test/test_asset_management.py:

```
import unittest
import mysql.connector
from dao.asset_management_service_impl import AssetManagementServiceImpl
from myexceptions.asset_not_found_exception import AssetNotFoundException

class TestAssetManagement(unittest.TestCase):

    def setUp(self):
        self.service = AssetManagementServiceImpl()
        self.conn = mysql.connector.connect(
            host='localhost',
            user='root',
            password='#vijaysql**',
            database='digital_asset'
        )
        self.cursor = self.conn.cursor()

        self.cursor.execute("DELETE FROM maintenance_records")
        self.cursor.execute("DELETE FROM asset_allocations")
        self.cursor.execute("DELETE FROM reservations")
        self.cursor.execute("DELETE FROM assets")
        self.cursor.execute("DELETE FROM employees")

        self.cursor.execute("""
            INSERT INTO employees (employee_id, name, department, email, password)
            VALUES (1, 'Test User', 'IT', 'test@example.com', 'testpass')
        """)
        self.cursor.execute("""
            INSERT INTO assets (asset_id, name, type, serial_number, purchase_date, location,
status, owner_id)
            VALUES (100, 'Projector', 'Electronics', 'SN99999', '2023-01-01', 'Room 303',
'available', 1)
        """)
        self.conn.commit()

    def tearDown(self):
        self.cursor.close()
        self.conn.close()

    def test_add_maintenance(self):
        self.service.perform_maintenance(100, '2025-03-31', "Routine check", 1500.00)
        self.cursor.execute("SELECT * FROM maintenance_records WHERE asset_id = 100")
        result = self.cursor.fetchone()
        self.assertIsNotNone(result)
        self.assertEqual(result[3], "Routine check")
```

```
    def test_asset_not_found_exception(self):
        with self.assertRaises(AssetNotFoundException):
            self.service.perform_maintenance(999, '2025-03-31', "Invalid", 500.00)

    def test_asset_allocation(self):
        result = self.service.allocate_asset(100, 1, '2025-03-31')
        self.assertTrue(result)
        self.cursor.execute("SELECT * FROM asset_allocations WHERE asset_id = 100")
        self.assertIsNotNone(self.cursor.fetchone())

    def test_asset_reservation(self):
        self.service.reserve_asset(100, 1, '2025-03-31', '2025-04-01', '2025-04-05', 'reserved')
        self.cursor.execute("SELECT * FROM reservations WHERE asset_id = 100")
        result = self.cursor.fetchone()
        self.assertIsNotNone(result)
        self.assertEqual(result[6], 'reserved')

if __name__ == '__main__':
    unittest.main()
```



Future Enhancements

While the current implementation of the Digital Asset Management System provides essential functionalities like asset tracking, allocation, maintenance, and reservation, there are several opportunities for future development to make the system more scalable, intelligent, and user-friendly:

1. **Role-Based Access Control (RBAC):**
   Introduce a secure authentication and authorization system that restricts features based on user roles such as Admin, Employee, and Maintenance Staff. This will enhance data security and accountability.

2. **Web-Based GUI Interface:**
   Build a responsive web interface using frameworks like Flask or Django (for Python) to allow users to interact with the system via a browser instead of a command-line interface, making it more accessible and intuitive.

3. **Asset Usage Analytics:**
   Integrate analytics dashboards to visualize asset usage trends, maintenance frequency, allocation patterns, and reservation statistics. This would help management make informed decisions regarding procurement and retirement of assets.

4. **Email & SMS Notifications:**
   Implement automated alerts and notifications for upcoming maintenance, reservation confirmations, overdue returns, and low inventory status. This would improve user engagement and proactive asset management.

5. **QR Code/Barcode Integration:**
   Enable each asset to be tagged with a unique QR code or barcode. Scanning the code can fetch real-time asset information, speeding up processes like allocation, return, or maintenance logging.

6. **Cloud Integration:**
   Migrate the database to a cloud-based platform like AWS RDS or Google Cloud SQL to support multi-location access, scalability, and real-time backups.

7. **Predictive Maintenance using Machine Learning:**
   Analyze past maintenance records using machine learning models to predict future failures or service needs. This would reduce downtime and improve asset lifespan.

8. **Mobile App Support:**
   Develop a companion mobile app for asset tracking, on-the-go updates, and quick check-ins/check-outs, making the system even more accessible.

9. **Multi-language Support:**
   Add localization options to support multiple languages based on the user's preference, increasing usability across different regions or departments.

10. **Audit and Compliance Module:**
    Introduce an auditing system that logs all transactions, modifications, and accesses for compliance tracking, useful in sectors with strict regulatory requirements.

Business Logic

The **Digital Asset Management System** is designed to manage the lifecycle of organizational assets efficiently by incorporating structured business rules and logic. The system ensures assets are added, maintained, allocated, reserved, and monitored in a systematic and traceable manner. Below is the breakdown of the core business logic implemented:

1. Asset Addition and Management

- When a new asset is added using the addAsset() method, the system validates all mandatory fields such as asset name, type, and purchase date before inserting it into the database.
- The updateAsset() function ensures that any existing asset can be updated only if the asset ID exists. It prevents modification of non-existent or deleted assets.
- The deleteAsset() method allows removal of assets that are no longer in use or need to be decommissioned.

2. Asset Allocation and Deallocation

- Allocation is done using allocateAsset(assetId, employeeId, allocationDate), which ensures:
    - The asset is not already allocated to another employee.
    - Both asset and employee IDs exist in the system.
- Deallocation using deallocateAsset(assetId, employeeId, returnDate) ensures:
    - The asset was previously allocated to the specified employee.
    - The return date is logged to update the asset's availability status.

3. Maintenance Management

- The performMaintenance() function records each maintenance event with date, cost, and description.
- It checks if the asset ID exists before inserting a new maintenance record.
- Historical maintenance data is used later in the reservation logic to validate asset health.

4. Asset Reservation

- Assets are reserved using reserveAsset(assetId, employeeId, reservationDate, startDate, endDate).
- Before confirming the reservation:
    - The asset's availability within the requested time frame is verified.
    - The system checks that the asset has undergone maintenance within the past two years.
    - If the asset is found to be unmaintained for over two years, AssetNotMaintainException is thrown.

5. Withdraw Reservation

- Using withdrawReservation(reservationId), any upcoming reservation can be canceled.
- This logic reverts the asset's status back to available, making it ready for allocation or new reservations.

6. Exception Handling

- AssetNotFoundException: Triggered if a user tries to perform any operation with an asset ID that doesn't exist in the database.
- AssetNotMaintainException: Raised when attempting to reserve an asset that hasn't been maintained for two years, ensuring only reliable assets are in use.

7. Data Validation and Integrity

- All methods include input validation checks and maintain referential integrity using SQL constraints and foreign key checks between assets, employees, and their allocations/reservations.

This business logic ensures robust management of assets, reduces human error, and maintains traceability and accountability within the system. It's designed to be modular and extendable for future enhancements like analytics, mobile app support, and cloud deployment.

Conclusion

The Digital Asset Management System successfully streamlines the tracking, allocation, reservation, and maintenance of assets within an organization. By incorporating core object-oriented programming concepts in Python and establishing a reliable MySQL database connection, the system ensures secure and efficient asset handling. Each module—from asset creation to exception handling—has been designed to enforce data integrity, improve operational transparency, and reduce manual errors.

Through the implementation of structured business logic, custom exception handling, and unit testing, the system has been made robust and scalable for real-time deployment. The user-friendly menu-driven application allows seamless interaction for asset managers, while ensuring backend consistency via dynamic SQL operations.

This project lays a strong foundation for future enhancements such as real-time dashboards, asset depreciation tracking, and role-based access. Overall, the system contributes significantly to digitalizing asset workflows and improving resource utilization within an organization.