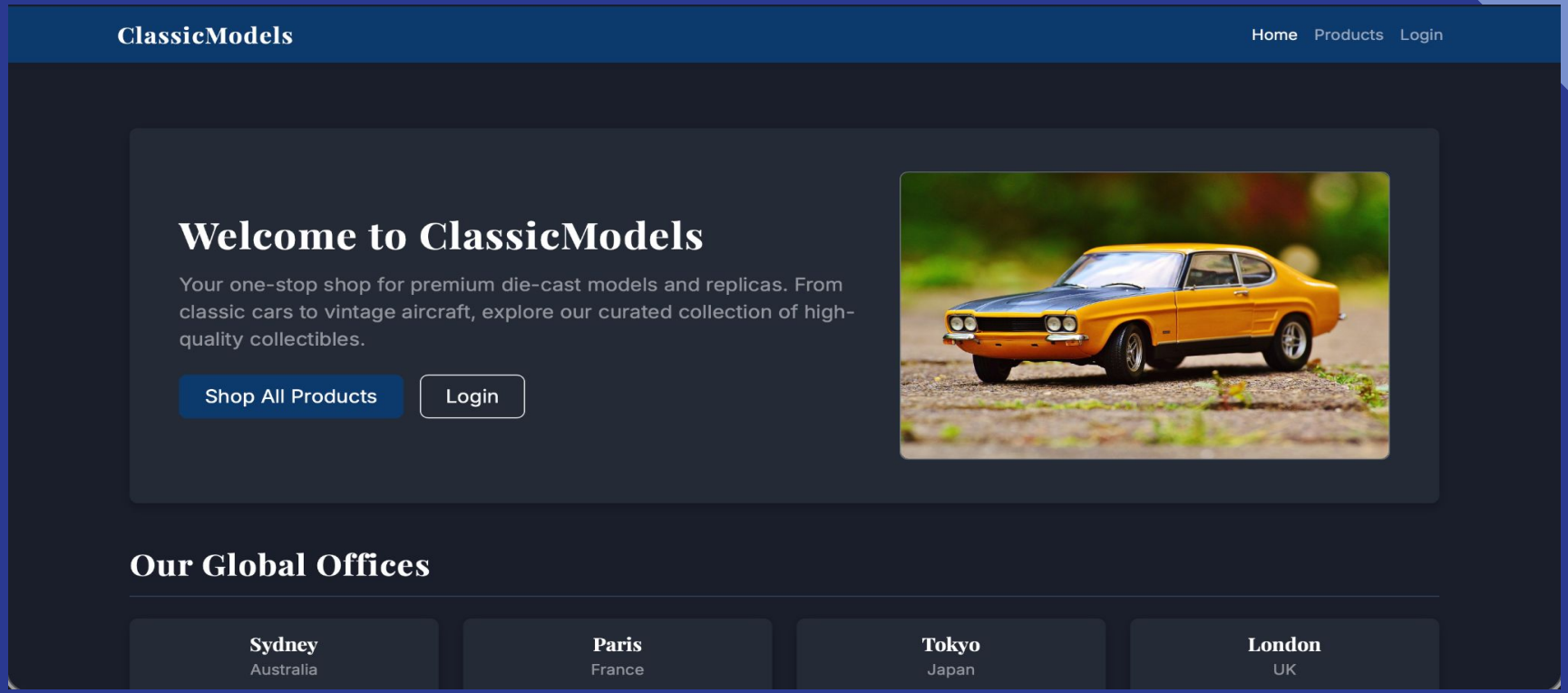
Lords of Data Project Presentation: ClassicModels

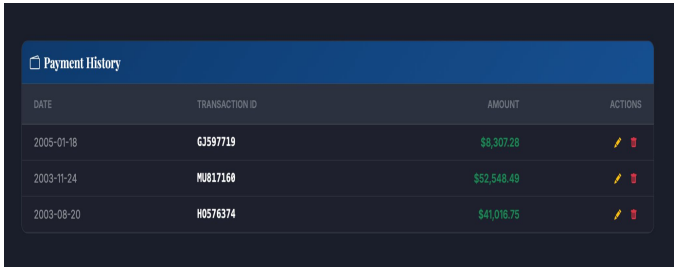


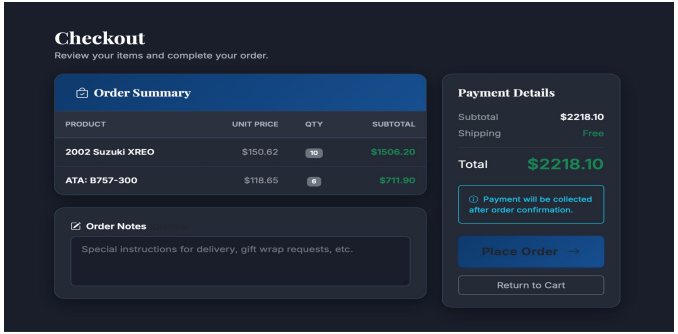
For the BLG317E Database Systems course

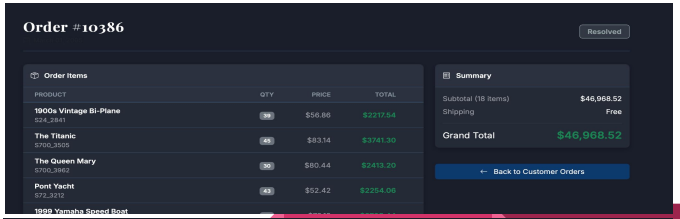
Introduction

Lords of Data consists of 5 members:

- **Muhammed Kağan TEMUR (150230093)**
  - Mainly responsible for the `orderdetails` table, as well as the `payments` table.
  - Handled order processing logic & finance analytics.
  - Helped design main website layout.
  - Helped set up Flask routes for the website's general flow.
  - Built the ER diagram.



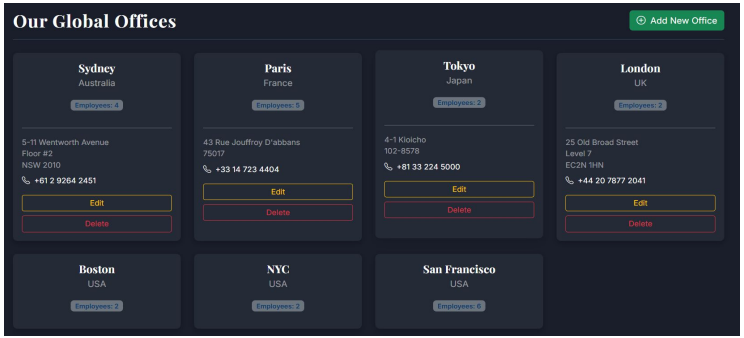


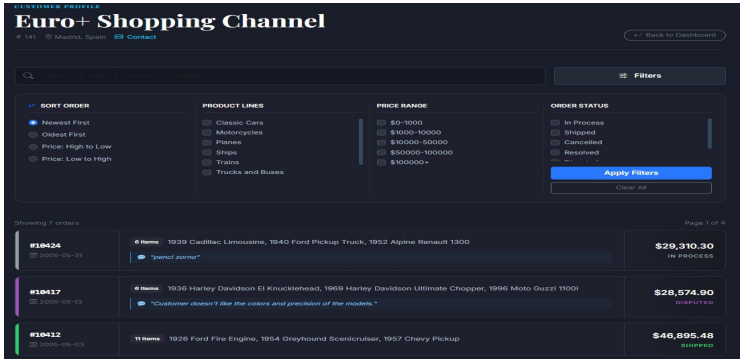


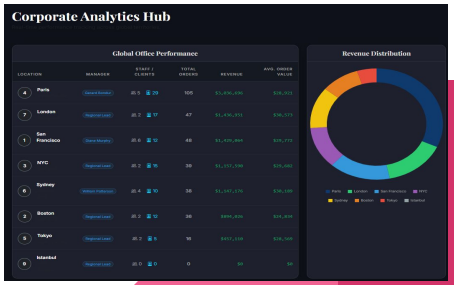
Introduction

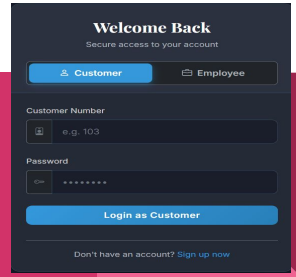
Lords of Data consists of 5 members:

- **Celal GÜNDOĞDU (150230007)**
  - Mainly responsible for the `orders` table ,as well as the `offices` table.
  - Order history maintaining and managing system.
  - Office unit system and management for employees.
  - Complex analytical queries relating office performance.
  - Search and filter implementations.
  - Complete password authentication system.
  - UI modifications.

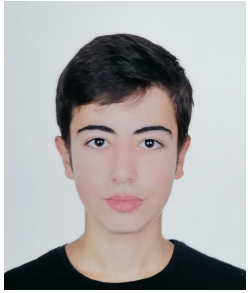


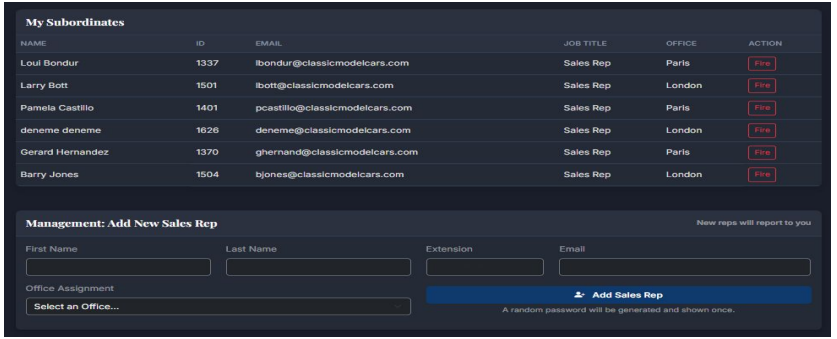






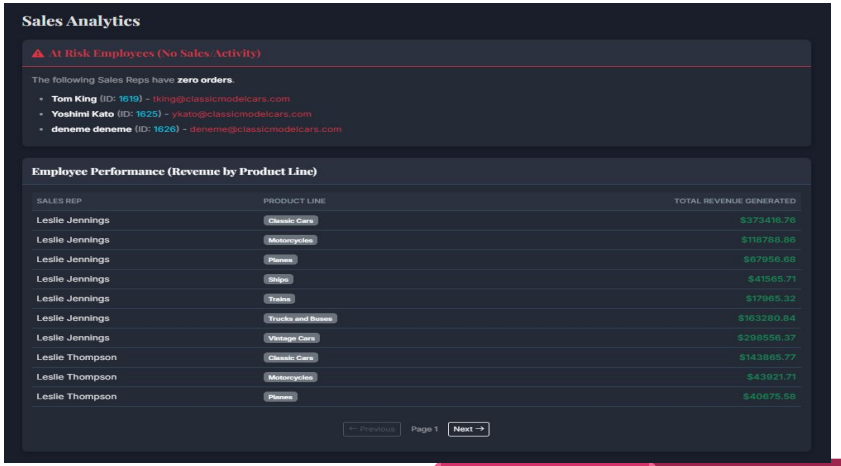
Introduction

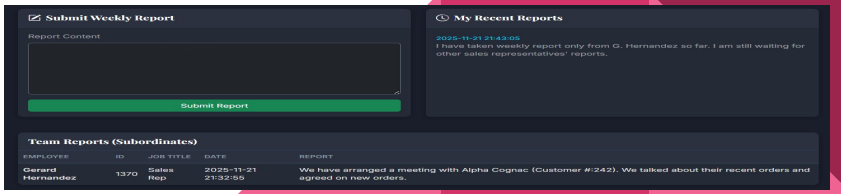




Lords of Data consists of 5 members:

- **Hüseyin Hayri DEDE (150220002)**
  - Mainly responsible for the `employees` table, as well as the `employee_reports` table.
  - Dashboard page
  - Employee hierarchy
  - Employee reporting system
  - Sales and employee performance analytics

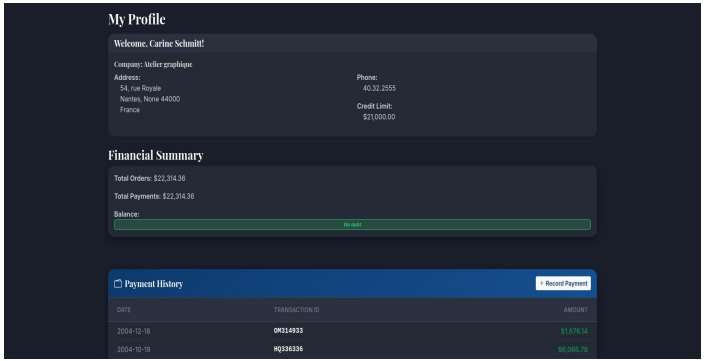


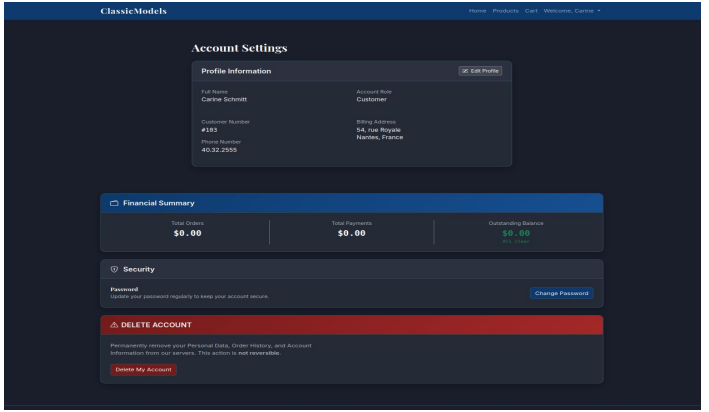


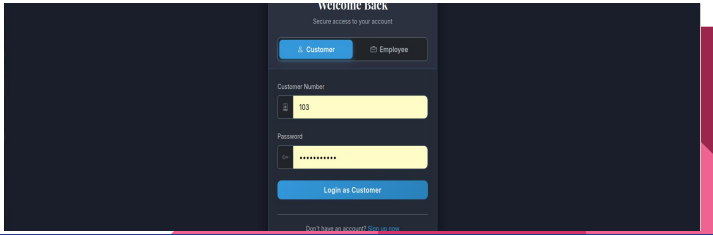
Introduction

Lords of Data consists of 5 members:

- **Ahmet Said Genç (150220092)**
  - Mainly responsible for the `customer` table.
  - Login page
  - Feel lucky page
  - Search and sort mechanism for employees' customer table
  - Customers CRUD
  - Product sort logic
  - UI enhancements



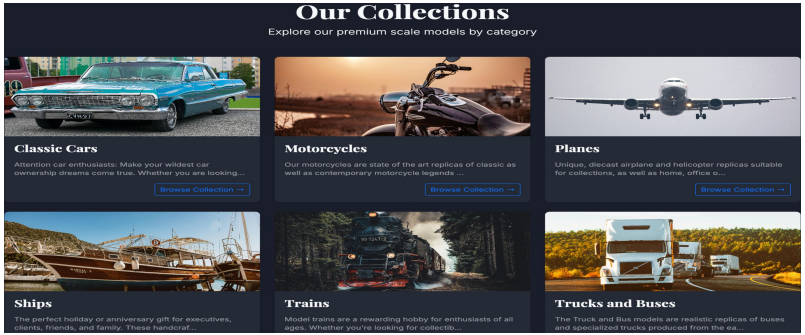


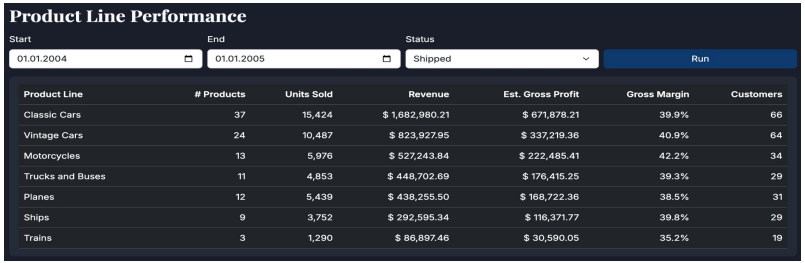


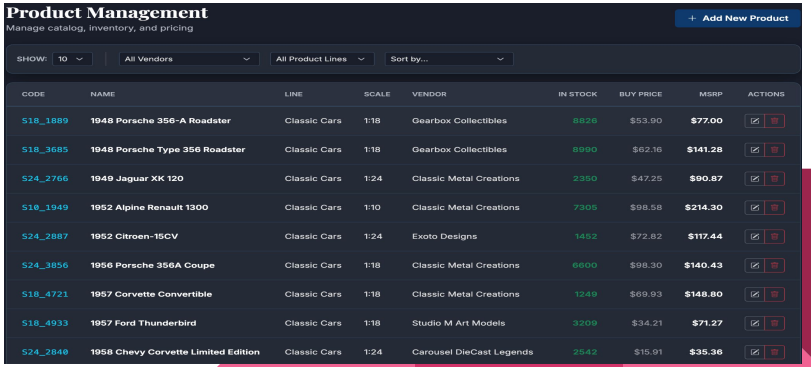
Introduction

Lords of Data consists of 5 members:

- **Erce Baran Keskin (070220716)**
  - Mainly responsible for the `products` table, as well as the `productlines` table.
  - Contributed to the Products, Manage Products, and Product Line Report pages.
  - Performed performance analytics per product line.



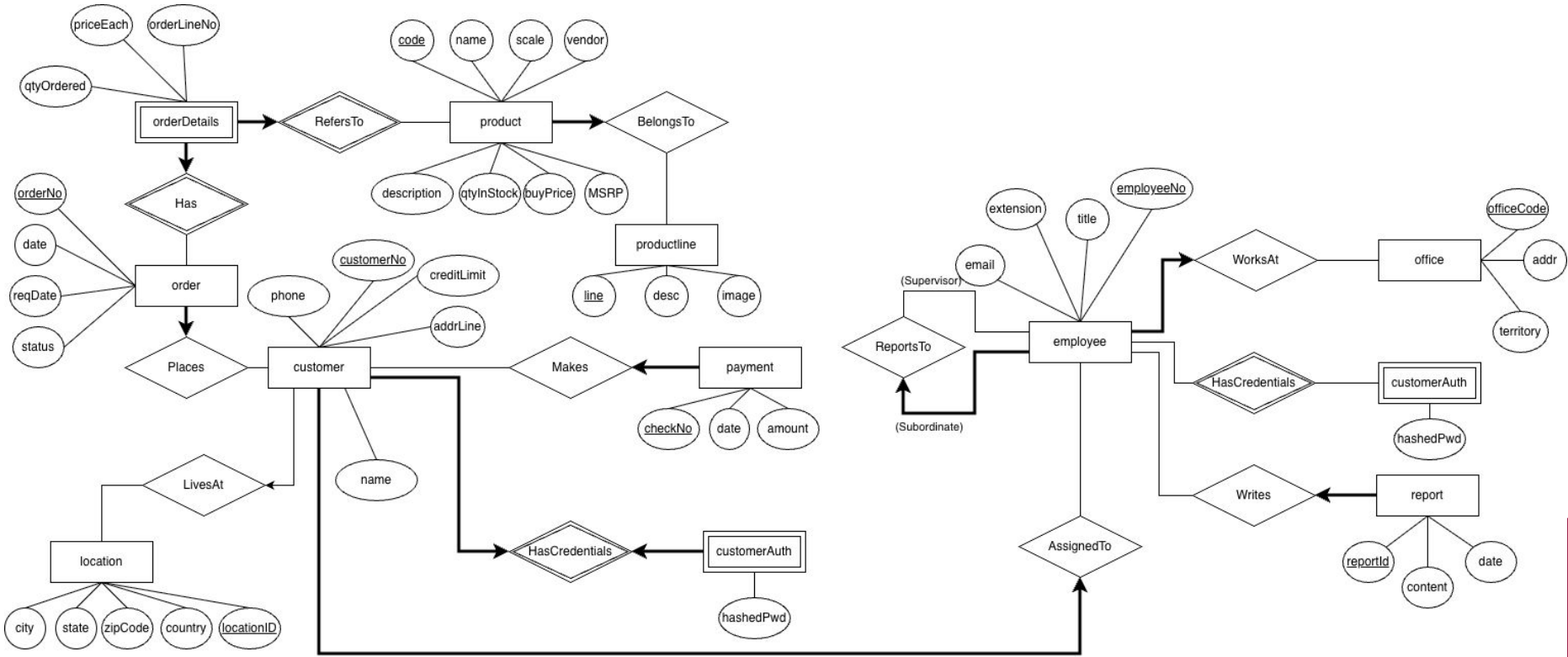




Overview of ClassicModels

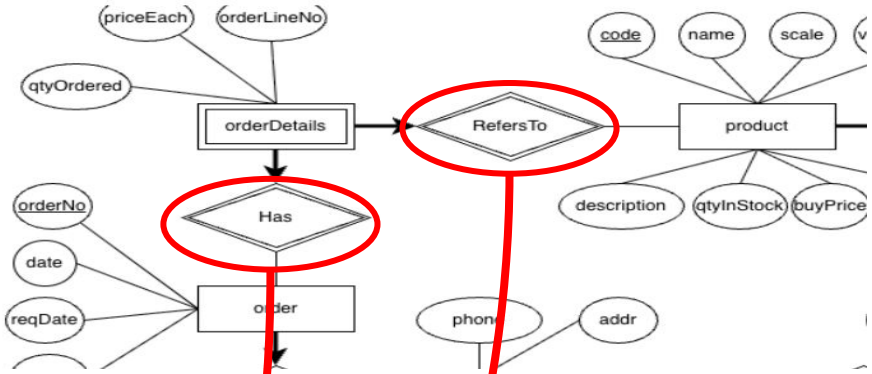
- ClassicModels is a relational database-powered full-stack web application for a retailer of scale models (classic cars, motorcycles, planes).
- The motivation behind its creation was to automate an otherwise poorly managed system. Our application utilises MySQL and Flask to fasten procedures.
- The goals of the project were:
  - To build distinct portals for **Customers**, **Employees**, and **Managers** for security.
  - To create data-driven real-time analytics depending on sales data to make it easier for managers to interpret their data.
  - To enforce business rules at the database level.
  - To remove manual calculation errors.
  - To optimize storage and access.

Full ER Diagram (Chen Notation)



ER-to-Relational Mapping

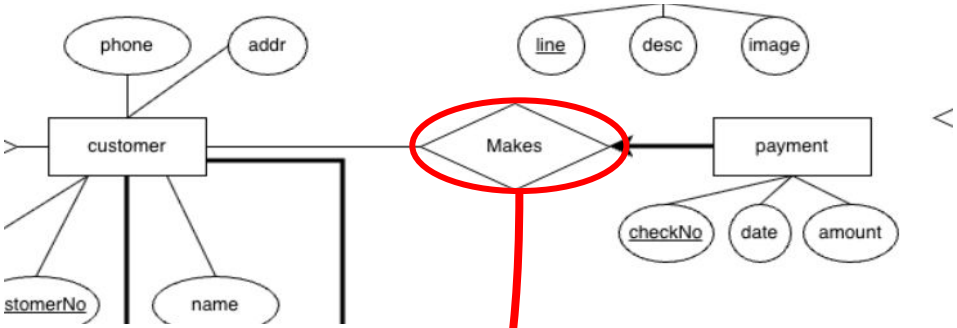
- The orderdetails entity is a **weak entity** that is used to **resolve the many-to-many relationship** between `product` and `order`.
- The `Has` and `RefersTo` identifying relationships were built into this table using the `orderNumber` and `productCode` FKs respectively.
- When mapping, we used a **surrogate key** (`orderDetailsNumber`) instead of relying on the candidate key alone to simplify operations.
- Integrity constraints were also included to cascade changes in identification data of the order or product in the entry.




```
CREATE TABLE `orderdetails` (
    `orderDetailsNumber`  INT          NOT NULL AUTO_INCREMENT,
    `orderNumber`         INT          NOT NULL,
    `productCode`         VARCHAR(15) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci,
    `quantityOrdered`     INT          NOT NULL,
    `priceEach`           double       NOT NULL,
    `orderLineNumber`     SMALLINT     NOT NULL,
    PRIMARY KEY (`orderDetailsNumber`),
    UNIQUE KEY `unique_line_item` (`orderNumber`, `productCode`),
    FOREIGN KEY (`orderNumber`) REFERENCES `orders`(`orderNumber`)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    FOREIGN KEY (`productCode`) REFERENCES `products`(`productCode`)
        ON DELETE CASCADE
        ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

ER-to-Relational Mapping

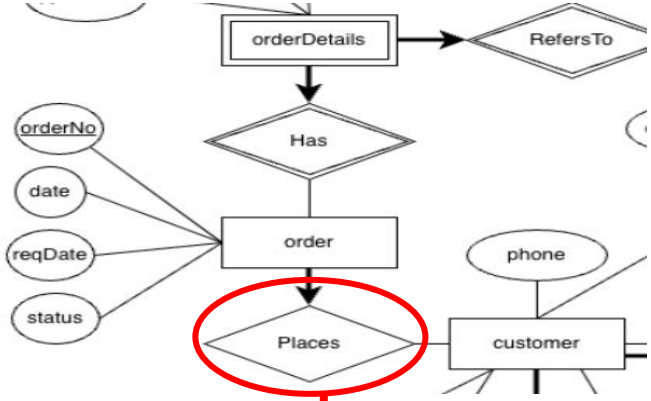
- The **`payment`** entity and the `Makes` one-to-many relationship were mapped to the `payments` table.
- The relationship itself is covered by the `customerNumber` foreign key.
- Referential integrity was enforced using `ON DELETE/UPDATE CASCADE`.
- Since `checkNumber` is **globally unique**, we marked payments as a **strong entity**.




```
CREATE TABLE payments (
    customerNumber  INT           NOT NULL,
    checkNumber     VARCHAR(50)   NOT NULL,
    paymentDate     DATE          NOT NULL,
    amount          DECIMAL(10,2) NOT NULL,
    PRIMARY KEY (checkNumber),
    FOREIGN KEY (customerNumber)
        REFERENCES customers(customerNumber)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

ER-to-Relational Mapping

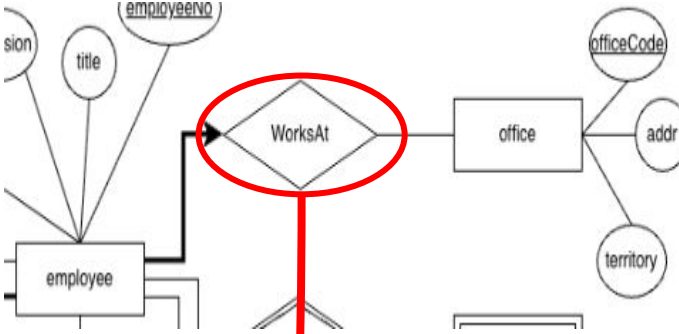
- The `orders` is a **strong entity** that represents the core purchase transaction in the system.
- **`Places`** is **an one-to-many relationship** between `orders` and `customers`
- `orderNumber` serves as the unique identifier derived from the ER model.Foreign Key constraints guarantee that no order exists without a valid customer.
- Referential integrity was enforced using `ON DELETE/UPDATE CASCADE.`
- Nullable attributes like `shippedDate` and `comments` correctly model **optional information** that may not be known at order creation time.




```
CREATE TABLE `orders` (
  `orderNumber` int(11) NOT NULL,
  `orderDate` date NOT NULL,
  `requiredDate` date NOT NULL,
  `shippedDate` date DEFAULT NULL,
  `status` varchar(15) NOT NULL,
  `comments` text DEFAULT NULL,
  `customerNumber` int(11) NOT NULL,
  PRIMARY KEY (`orderNumber`),
  KEY `customerNumber` (`customerNumber`),
  CONSTRAINT `orders_ibfk_1`
  FOREIGN KEY (`customerNumber`)
  REFERENCES `customers` (`customerNumber`)
  ON DELETE RESTRICT
  ON UPDATE CASCADE
```

ER-to-Relational Mapping

- `offices` is an **independent entity** that represents physical company locations.
- `officeCode` serves as the unique, stable identifier derived from the ER model.Requires no Foreign Keys, ensuring existence without external dependencies.
- `WorksAt` implements the One-to-Many relationship by placing the `officeCode` Foreign Key in the `employees` table

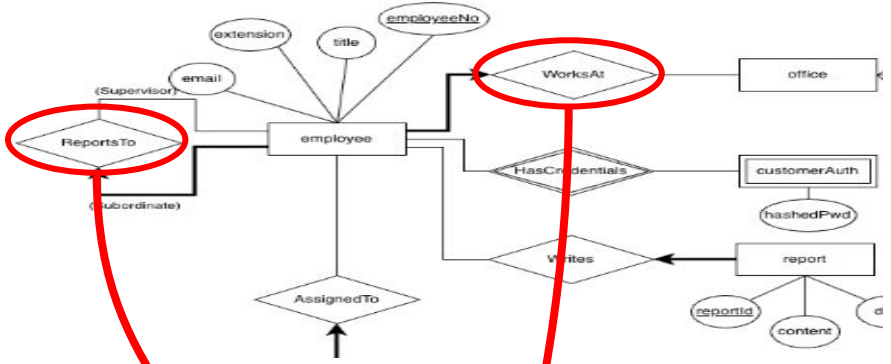



```sql
CREATE TABLE `offices` (
  `officeCode` varchar(10) NOT NULL,
  `city` varchar(50) NOT NULL,
  `phone` varchar(50) NOT NULL,
  `addressLine1` varchar(50) NOT NULL,
  `addressLine2` varchar(50) DEFAULT NULL,
  `state` varchar(50) DEFAULT NULL,
  `country` varchar(50) NOT NULL,
  `postalCode` varchar(15) NOT NULL,
  `territory` varchar(10) NOT NULL,
  PRIMARY KEY (`officeCode`)
```

```sql
CREATE TABLE `employees` (
  `employeeNumber` int(11) NOT NULL,
  `lastName` varchar(50) NOT NULL,
  `firstName` varchar(50) NOT NULL,
  `extension` varchar(10) NOT NULL,
  `email` varchar(100) NOT NULL,
  `officeCode` varchar(10) NOT NULL,
  `reportsTo` int(11) DEFAULT NULL,
  `jobTitle` varchar(50) NOT NULL,
  PRIMARY KEY (`employeeNumber`),
  KEY `reportsTo` (`reportsTo`),
  CONSTRAINT `fk_employees_reports_to`
  FOREIGN KEY (`reportsTo`) REFERENCES `employees` (`employeeNumber`),
  CONSTRAINT `fk_employees_offi
  FOREIGN KEY (`officeCode`) REFERENCES `offices` (`officeCode`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

ER-to-Relational Mapping

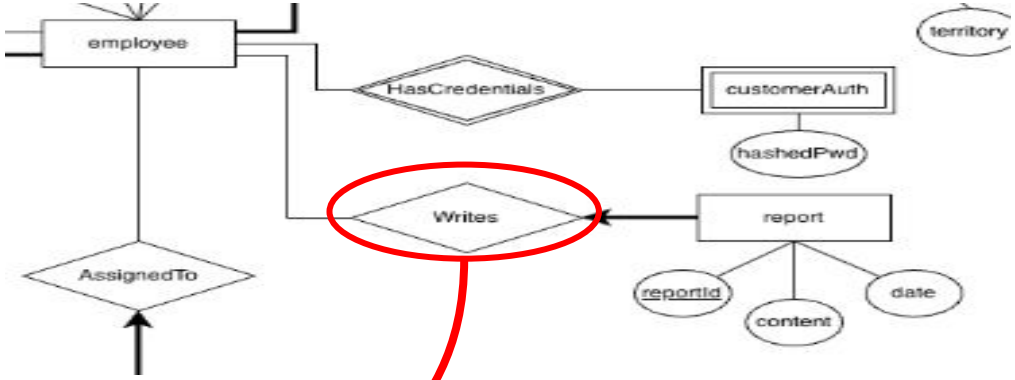
- The `WorksAt` one-to-many relationship were mapped to the offices table.
- The relationship itself is covered by the `officeCode` foreign key.
- Referential integrity was enforced using `ON DELETE SET NULL` and `ON UPDATE CASCADE`.
- The `ReportsTo` one-to-many relationship were mapped to employees table itself. It forms a self referencing relationship.
- Referential integrity was enforced using `ON UPDATE CASCADE`.




```
CREATE TABLE `employees` (
  `employeeNumber` int(11) NOT NULL,
  `lastName` varchar(50) NOT NULL,
  `firstName` varchar(50) NOT NULL,
  `extension` varchar(10) NOT NULL,
  `email` varchar(100) NOT NULL,
  `officeCode` varchar(10) NOT NULL,
  `reportsTo` int(11) DEFAULT NULL,
  `jobTitle` varchar(50) NOT NULL,
  PRIMARY KEY (`employeeNumber`),
  KEY `reportsTo` (`reportsTo`),
  CONSTRAINT `fk_employees_reports_to`
  FOREIGN KEY (`reportsTo`) REFERENCES `employees` (`employeeNumber`)
  ON DELETE SET NULL
  ON UPDATE CASCADE,
  CONSTRAINT `fk_employees_office`
  FOREIGN KEY (`officeCode`) REFERENCES `offices` (`officeCode`)
  ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

ER-to-Relational Mapping

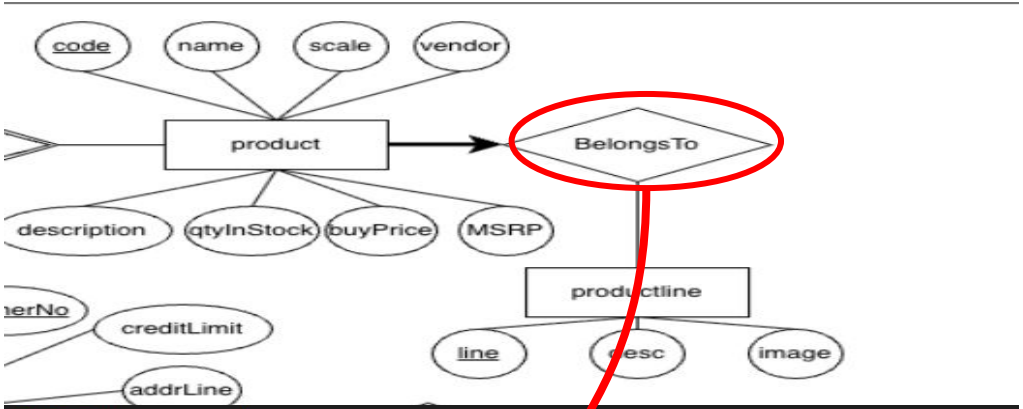
- The `Writes` one-to-many relationship were mapped to the employees table.
- The relationship itself is covered by the `employeeNumber` foreign key.
- Referential integrity was enforced using `ON DELETE/UPDATE CASCADE`.




```
CREATE TABLE IF NOT EXISTS `employee_reports` (
  `reportId` int(11) NOT NULL AUTO_INCREMENT,
  `employeeNumber` int(11) NOT NULL,
  `reportContent` text NOT NULL,
  `reportDate` datetime DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`reportId`),
  KEY `employeeNumber` (`employeeNumber`),
  FOREIGN KEY (`employeeNumber`) REFERENCES `employees` (`employeeNumber`)
  ON DELETE CASCADE
  ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

ER-to-Relational Mapping

- The `BelongsTo` **one-to-many** relationship was integrated into the employees table via the `productLine` FK.
- **`productlines`** is a **strong entity** that represents the types of products the retailer sells.

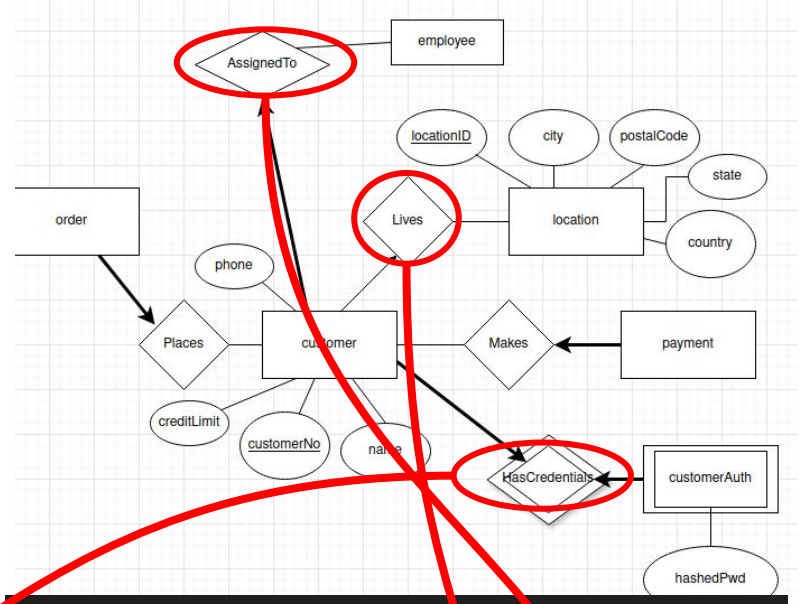



```
CREATE TABLE `productlines` (
  `productLine`      varchar(50)      NOT NULL,
  `textDescription`  varchar(4000)    DEFAULT NULL,
  `htmlDescription`  mediumtext       DEFAULT NULL,
  `image`            mediumblob       DEFAULT NULL,
  PRIMARY KEY (`productLine`)
);
```

```
CREATE TABLE `products` (
  `productCode` varchar(15) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL,
  `productName` varchar(70) NOT NULL,
  `productLine` varchar(50) NOT NULL,
  `productScale` varchar(10) NOT NULL,
  `productVendor` varchar(50) NOT NULL,
  `productDescription` text NOT NULL,
  `quantityInStock` smallint(6) NOT NULL,
  `buyPrice` double NOT NULL,
  `MSRP` double NOT NULL,
  PRIMARY KEY (`productCode`),
  KEY `productline` (`productLine`),
  CONSTRAINT `products_ibfk_1` FOREIGN KEY (`productLine`) REFERENCES `productlines` (`productLine`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

ER-to-Relational Mapping

- The **Lives** one-to-many relationship was mapped by placing the primary key of **Location** as a foreign key in the **Customers** table.
- This enforces referential integrity while allowing multiple customers to share the same location, resulting in a normalized design




```
CREATE TABLE `customer_auth` (
  `customerNumber` int NOT NULL,
  `hashedPassword` varchar(255) NOT NULL,
  PRIMARY KEY (`customerNumber`),
  CONSTRAINT `fk_customer_auth_num` FOREIGN KEY (`customerNumber`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
/*!40101 SET character_set_client = @saved_cs_client */;
```

```
-- Create the Normalized Customers Table
CREATE TABLE `customers` (
  `customerNumber` int(11) NOT NULL,
  `customerName` varchar(50) NOT NULL,
  `contactLastName` varchar(50) NOT NULL,
  `contactFirstName` varchar(50) NOT NULL,
  `phone` varchar(50) NOT NULL,
  `addressLine1` varchar(50) NOT NULL,
  `addressLine2` varchar(50) DEFAULT NULL,
  `locationID` int(11) DEFAULT NULL,
  `salesRepEmployeeNumber` int(11) DEFAULT NULL,
  `creditLimit` double DEFAULT NULL,
  PRIMARY KEY (`customerNumber`),
  KEY `salesRepEmployeeNumber` (`salesRepEmployeeNumber`),
  KEY `fk_customer_location` (`locationID`),
  CONSTRAINT `customers_ibfk_1` FOREIGN KEY (`salesRepEmployeeNumber`)
  REFERENCES `employees` (`employeeNumber`) ON DELETE SET NULL,
  CONSTRAINT `customers_ibfk_2` FOREIGN KEY (`locationID`)
  REFERENCES `locations` (`locationID`)
```

Normalization

- The orderdetails table is already in BCNF, because:
  - ✓ **1NF:**
    - All columns contain atomic values.
    - All rows are unique.
    - There are no repeating groups.
  - ✓ **2NF:**
    - There are no partial dependencies, this is because it has only one primary key.
    - Even if we considered the candidate key `(orderNumber, productCode)`, it is also free of partial dependencies.
    - The table is in 1NF.
  - ✓ **3NF:**
    - There are no transitive dependencies (non-key to non-key FDs).
    - The table is in 2NF.
  - ✓ **BCNF:**
    - Every determinant is a candidate key.
    - The table is in 3NF.

```
CREATE TABLE `orderdetails` (
    `orderDetailsNumber` INT         NOT NULL AUTO_INCREMENT,
    `orderNumber`        INT         NOT NULL,
    `productCode`        VARCHAR(15) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci,
    `quantityOrdered`    INT         NOT NULL,
    `priceEach`          double      NOT NULL,
    `orderLineNumber`    SMALLINT    NOT NULL,
    PRIMARY KEY (`orderDetailsNumber`),
    UNIQUE KEY `unique_line_item` (`orderNumber`, `productCode`),
    FOREIGN KEY (`orderNumber`) REFERENCES `orders`(`orderNumber`)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    FOREIGN KEY (`productCode`) REFERENCES `products`(`productCode`)
        ON DELETE CASCADE
        ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

Normalization

- The initial payments table's PK was `(customerNumber, checkNumber)`. This made it 1NF.
  - ✓ **1NF:**
    - All columns contain atomic values.
    - All rows are unique.
    - There are no repeating groups.
  - ✗ **2NF:**
    - The table is in 1NF.
    - `checkNumber` **is already unique on its own, creating partial dependencies.**
    - **The Fix:** `checkNumber` as the PK.
- This approach was chosen because `checkNumber` is unique for each payment.
  - ✓ **3NF:**
    - There are no transitive dependencies. (non-key to non-key FDs).
    - The table is in 2NF.
  - ✓ **BCNF:**
    - Every determinant is a candidate key.
    - The table is in 3NF.

```sql
-- Unnormalized version
-- CREATE TABLE payments (
--     customerNumber INT NOT NULL,
--     checkNumber VARCHAR(50) NOT NULL,
--     paymentDate DATE NOT NULL,
--     amount DECIMAL(10,2) NOT NULL,
--     PRIMARY KEY (customerNumber, checkNumber),
--     FOREIGN KEY (customerNumber) REFERENCES customers(customerNumber)
--         ON DELETE CASCADE
--         ON UPDATE CASCADE
-- );

-- Normalized
CREATE TABLE payments (
    customerNumber  INT          NOT NULL,
    checkNumber     VARCHAR(50)  NOT NULL,
    paymentDate     DATE         NOT NULL,
    amount          DECIMAL(10,2) NOT NULL,
    PRIMARY KEY (checkNumber),
    FOREIGN KEY (customerNumber)
        REFERENCES customers(customerNumber)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

Normalization

- The orders table is already in BCNF, because:
  - ✓ **1NF:**
    - All columns contain **atomic values**.
    - All rows are unique.
    - There are no repeating groups.
  - ✓ **2NF:**
    - The table is in 1NF.
    - There are **no partial dependencies**, because the table has a single-attribute primary key (`orderNumber`).
  - ✓ **3NF:**
    - The table is in 2NF.
    - There are no transitive dependencies (non-key to non-key FDs).
  - ✓ **BCNF**:
    - Every determinant is a candidate key.
    - The table is in 3NF.

```
--Normalized (BCNF)
CREATE TABLE `orders` (
    `orderNumber`    int(11)        NOT NULL,
    `orderDate`      date           NOT NULL,
    `requiredDate`   date           NOT NULL,
    `shippedDate`    date           DEFAULT NULL,
    `status`         varchar(15)    NOT NULL,
    `comments`       text           DEFAULT NULL,
    `customerNumber` int(11)        NOT NULL,
    PRIMARY KEY (`orderNumber`),
    KEY `customerNumber` (`customerNumber`),
    CONSTRAINT `orders_ibfk_1`
    FOREIGN KEY (`customerNumber`)
      REFERENCES `customers` (`customerNumber`)
      ON DELETE RESTRICT
      ON UPDATE CASCADE
);
```

Normalization

- The offices table is already in BCNF, because:
  - ✓ **1NF:**
    - All columns (`city`, `state`, `country` etc.) contain **atomic values**.
    - All rows are unique.
    - There are no repeating groups.
  - ✓ **2NF:**
    - The table is in 1NF.
    - There are **no partial dependencies**, because the table has a single-attribute primary key (`officeCode`).
  - ✓ **3NF:**
    - The table is in 2NF.
    - There are **no transitive dependencies**. All non-key attributes (phone, territory, location details) rely *only* on the `officeCode`.
  - ✓ BCNF:
    - Every determinant is a candidate key.
    - The table is in 3NF.

```sql
-- Normalized (BCNF)
CREATE TABLE offices (
  officeCode    VARCHAR(10)    NOT NULL,
  city          VARCHAR(50)    NOT NULL,
  phone         VARCHAR(50)    NOT NULL,
  addressLine1  VARCHAR(50)    NOT NULL,
  addressLine2  VARCHAR(50)    DEFAULT NULL,
  'state'       VARCHAR(50)    DEFAULT NULL,
  country       VARCHAR(50)    NOT NULL,
  postalCode    VARCHAR(15)    NOT NULL,
  territory     VARCHAR(10)    NOT NULL,
  PRIMARY KEY (officeCode)
);
```

Normalization

- The employees table is already in BCNF, because:
  - ✓ **1NF:**
    - All columns contain atomic values.
    - All rows are unique.
    - There are no repeating groups.
  - ✓ **2NF:**
    - There are no partial dependencies, this is because it has only one primary key.
    - The table is in 1NF.
  - ✓ **3NF:**
    - There are no transitive dependencies (non-key to non-key FDs).
    - The table is in 2NF.
  - ✓ **BCNF:**
    - Every determinant is a candidate key.
    - The table is in 3NF.

```
CREATE TABLE `employees` (
  `employeeNumber` int(11) NOT NULL,
  `lastName` varchar(50) NOT NULL,
  `firstName` varchar(50) NOT NULL,
  `extension` varchar(10) NOT NULL,
  `email` varchar(100) NOT NULL,
  `officeCode` varchar(10) NOT NULL,
  `reportsTo` int(11) DEFAULT NULL,
  `jobTitle` varchar(50) NOT NULL,
  PRIMARY KEY (`employeeNumber`),
  KEY `reportsTo` (`reportsTo`),
  CONSTRAINT `fk_employees_reports_to`
  FOREIGN KEY (`reportsTo`) REFERENCES `employees` (`employeeNumber`)
  ON DELETE SET NULL
  ON UPDATE CASCADE,
  CONSTRAINT `fk_employees_office`
  FOREIGN KEY (`officeCode`) REFERENCES `offices` (`officeCode`)
  ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

Normalization

- The products table is already in BCNF, because:
  - ✓ **1NF:**
    - All columns contain atomic values.
    - All rows are unique.
    - There are no repeating groups.
  - ✓ **2NF:**
    - There are no partial dependencies, because the table has a single-attribute primary key (`productCode`).
    - All non-key attributes are fully functionally dependent on `productCode`.
    - The table is in 1NF.
  - ✓ **3NF:**
    - There are no transitive dependencies (non-key to non-key FDs).
    - Descriptive attributes depend directly on `productCode`.
    - The table is in 2NF.
  - ✓ **BCNF:**
    - Every determinant is a candidate key.
    - The table is in 3NF.

```
5   CREATE TABLE `products` (
6     `productCode` varchar(15) CHARACTER SET
7     utf8mb4 COLLATE utf8mb4_general_ci NOT NULL,
8     `productName` varchar(70) NOT NULL,
9     `productLine` varchar(50) NOT NULL,
10    `productScale` varchar(10) NOT NULL,
11    `productVendor` varchar(50) NOT NULL,
12    `productDescription` text NOT NULL,
13    `quantityInStock` smallint(6) NOT NULL,
14    `buyPrice` double NOT NULL,
15    `MSRP` double NOT NULL,
16    PRIMARY KEY (`productCode`),
17    KEY `productLine` (`productLine`),
18    CONSTRAINT `products_ibfk_1` FOREIGN KEY (`productLine`)
19    REFERENCES `productlines` (`productLine`)
20  ) ENGINE=InnoDB DEFAULT
21  CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

Normalization

- The productlines table is already in BCNF, because:
  - ✓ **1NF:**
    - All columns contain atomic values.
    - All rows are unique.
    - There are no repeating groups.
  - ✓ **2NF:**
    - There are no partial dependencies, because the table has a single-attribute primary key (`productLine`).
    - All non-key attributes are fully functionally dependent on `productLine`.
    - The table is in 1NF.
  - ✓ **3NF:**
    - There are no transitive dependencies (non-key to non-key FDs).
    - Descriptive attributes depend directly on `productLine`.
    - The table is in 2NF.
  - ✓ **BCNF:**
    - Every determinant is a candidate key.
    - The table is in 3NF.

```
CREATE TABLE `productlines` (
  `productLine` varchar(50) NOT NULL,
  `textDescription` varchar(4000) DEFAULT NULL,
  `htmlDescription` mediumtext DEFAULT NULL,
  `image` mediumblob DEFAULT NULL,
  PRIMARY KEY (`productLine`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_general_ci;
```

Normalization

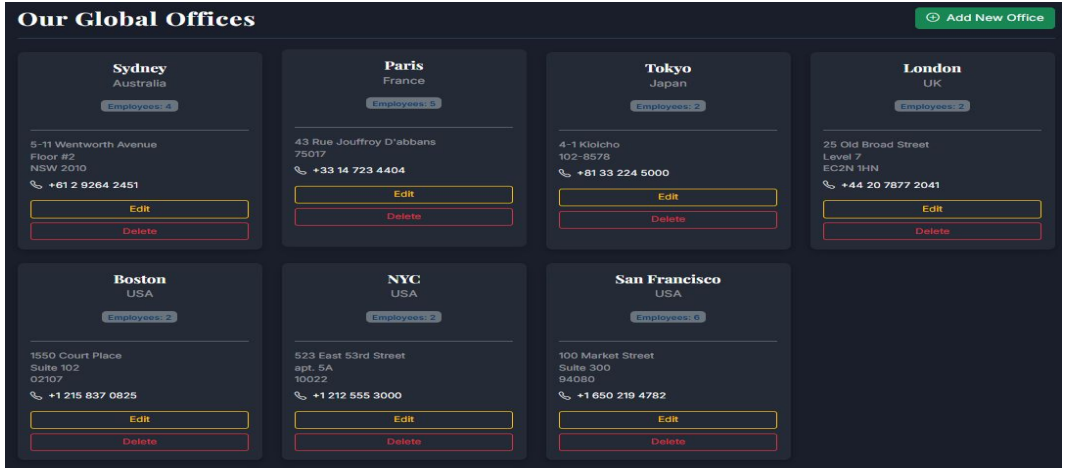
- ✓ **1NF:**
  - All columns contain atomic values (e.g., one phone number per cell).
  - All rows are unique via `customerNumber`.
  - There are no repeating groups of data.
- ✓ **2NF:**
  - The table is in 1NF.
  - Since there is a single-column Primary Key (`customerNumber`), there are **no partial functional dependencies**. Every non-key attribute depends on the whole key.
- ✓ **3NF:**
  - The table is in 2NF.
  - All non-key attributes (e.g., `customerName`, `phone`) are now functionally dependent **only** on the Primary Key (`customerNumber`) and nothing else.
  - The geographic attributes live in a `locations` table, referenced by `locationID`.

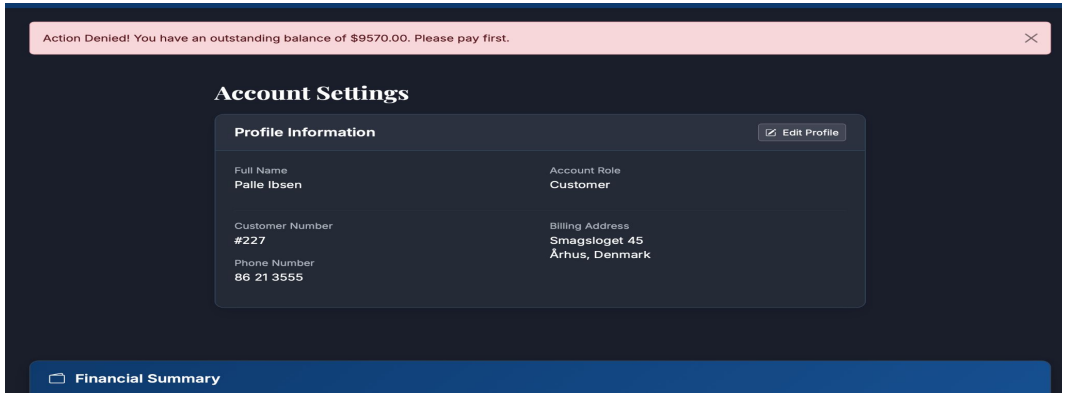
```
CREATE TABLE `locations` (
  `locationID` int(11) NOT NULL AUTO_INCREMENT,
  `city` varchar(50) NOT NULL,
  `state` varchar(50) DEFAULT NULL,
  `postalCode` varchar(15) DEFAULT NULL,
  `country` varchar(50) NOT NULL,
  PRIMARY KEY (`locationID`),
  UNIQUE KEY `unique_geo` (`city`, `state`, `postalCode`, `country`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

-- Create the Normalized Customers Table
CREATE TABLE `customers` (
  `customerNumber` int(11) NOT NULL,
  `customerName` varchar(50) NOT NULL,
  `contactLastName` varchar(50) NOT NULL,
  `contactFirstName` varchar(50) NOT NULL,
  `phone` varchar(50) NOT NULL,
  `addressLine1` varchar(50) NOT NULL,
  `addressLine2` varchar(50) DEFAULT NULL,
  `locationID` int(11) DEFAULT NULL,
  `salesRepEmployeeNumber` int(11) DEFAULT NULL,
  `creditLimit` double DEFAULT NULL,
  PRIMARY KEY (`customerNumber`),
  KEY `salesRepEmployeeNumber` (`salesRepEmployeeNumber`),
  KEY `fk_customer_location` (`locationID`),
  CONSTRAINT `customers_ibfk_1` FOREIGN KEY (`salesRepEmployeeNumber`)
  REFERENCES `employees` (`employeeNumber`) ON DELETE SET NULL,
  CONSTRAINT `customers_ibfk_2` FOREIGN KEY (`locationID`)
  REFERENCES `locations` (`locationID`)
```

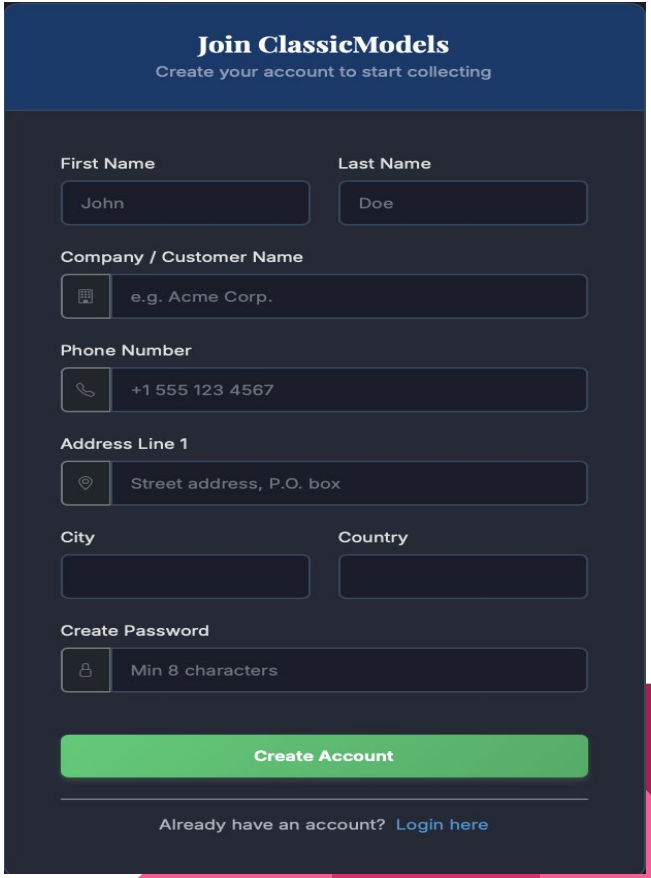
Dataset Description

- **What does the data include?**
  - The dataset is a **synthetic dataset** used to simulate business data for **a retailer of scale models** of cars, ships, etc.
  - **Typical relational tables:** customers, products, product lines, orders, order details, payments, employees, and offices which describe sales operations, products, and organizational structure.
- **What is the source?**
  - The dataset comes from the ClassicModels sample database, originally provided by MySQLTutorial.
    It can be downloaded directly from places like the MySQL sample DB documentation.
- **How large is it?**
  - The entire database is about 500 KB in size.
  - It consists of **8 tables** with a total of approximately **3,864 rows and about 59 columns** across the schema.
- **Did you perform any parsing or scraping?**
  - No, we did not perform any **automated** parsing or scraping. **All images used were manually sourced** from copyright-free repositories and **included as static assets**.
- **Did you use synthetic data?**
  - The ClassicModels dataset is an educational dataset, and all of its data is synthetic.
  - However, we did not inject any extra data into the database other than testing its functionality.

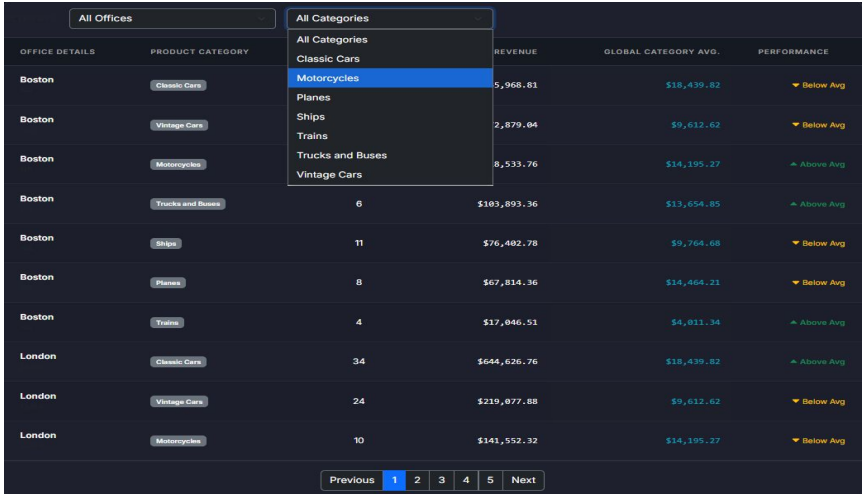
Features

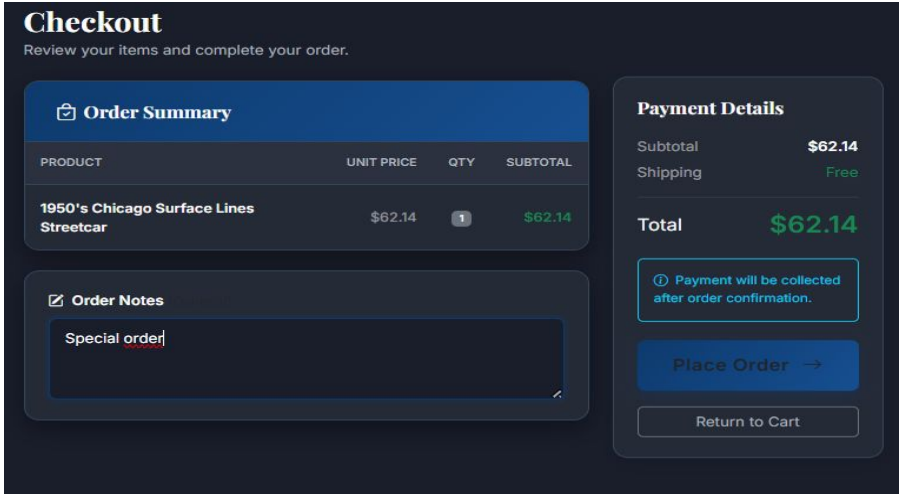

Our Global Offices
Add New Office
Sydney
Australia
Employees: 4
5-11 Wentworth Avenue
Floor #2
NSW 2010
+61 2 9264 2451
Edit
Delete
Paris
France
Employees: 5
43 Rue Jouffroy D'abbans
75017
+33 14 723 4404
Edit
Delete
Tokyo
Japan
Employees: 2
4-1 Kioicho
102-8578
+81 33 224 5000
Edit
Delete
London
UK
Employees: 2
25 Old Broad Street
Level 7
EC2N 1HN
+44 20 7877 2041
Edit
Delete
Boston
USA
Employees: 2
1550 Court Place
Suite 102
02107
+1 215 837 0825
Edit
Delete
NYC
USA
Employees: 2
523 East 53rd Street
apt. 5A
10022
+1 212 555 3000
Edit
Delete
San Francisco
USA
Employees: 6
100 Market Street
Suite 300
94080
+1 650 219 4782
Edit
Delete


Action Denied! You have an outstanding balance of $9570.00. Please pay first.
Account Settings
Profile Information
Edit Profile
Full Name
Palle Ibsen
Account Role
Customer
Customer Number
#227
Billing Address
Smagsloget 45
Århus, Denmark
Phone Number
86 21 3555
Financial Summary

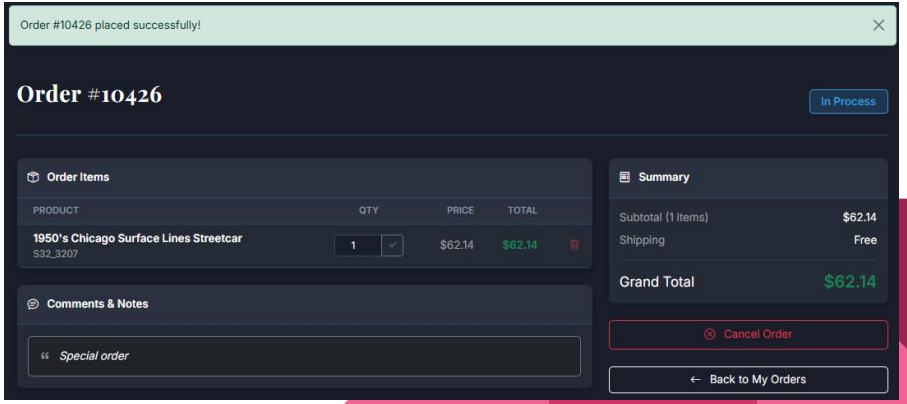

Join ClassicModels
Create your account to start collecting
First Name
John
Last Name
Doe
Company / Customer Name
e.g. Acme Corp.
Phone Number
+1 555 123 4567
Address Line 1
Street address, P.O. box
City
Country
Create Password
Min 8 characters
Create Account
Already have an account? Login here

Features

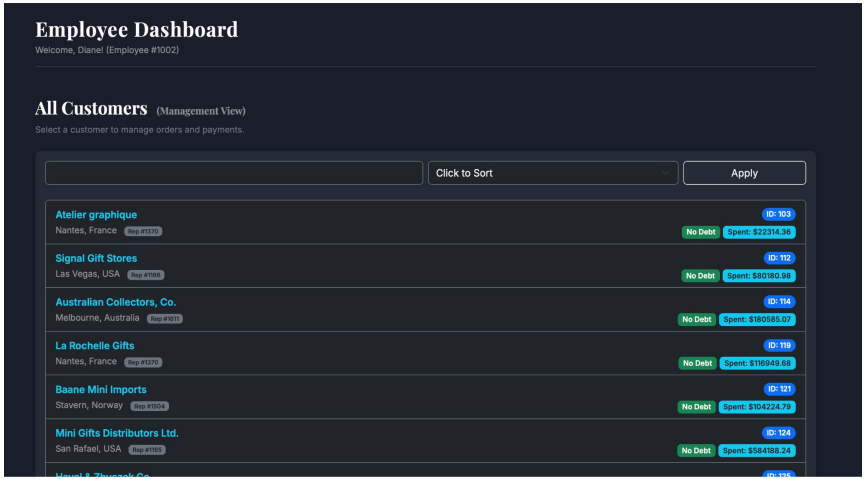

All Offices
All Categories
All Categories
Classic Cars
Motorcycles
Planes
Ships
Trains
Trucks and Buses
Vintage Cars
OFFICE DETAILS
PRODUCT CATEGORY
REVENUE
GLOBAL CATEGORY AVG.
PERFORMANCE
Boston Classic Cars $18,439.82 Below Avg
Boston Vintage Cars $9,612.62 Below Avg
Boston Motorcycles $14,195.27 Above Avg
Boston Trucks and Buses 6 $103,893.36 $13,654.85 Above Avg
Boston Ships 11 $76,402.78 $9,764.68 Below Avg
Boston Planes 8 $67,814.36 $14,464.21 Below Avg
Boston Trains 4 $17,046.51 $4,011.34 Above Avg
London Classic Cars 34 $644,626.76 $18,439.82 Above Avg
London Vintage Cars 24 $219,077.88 $9,612.62 Below Avg
London Motorcycles 10 $141,552.32 $14,195.27 Below Avg
Previous 1 2 3 4 5 Next

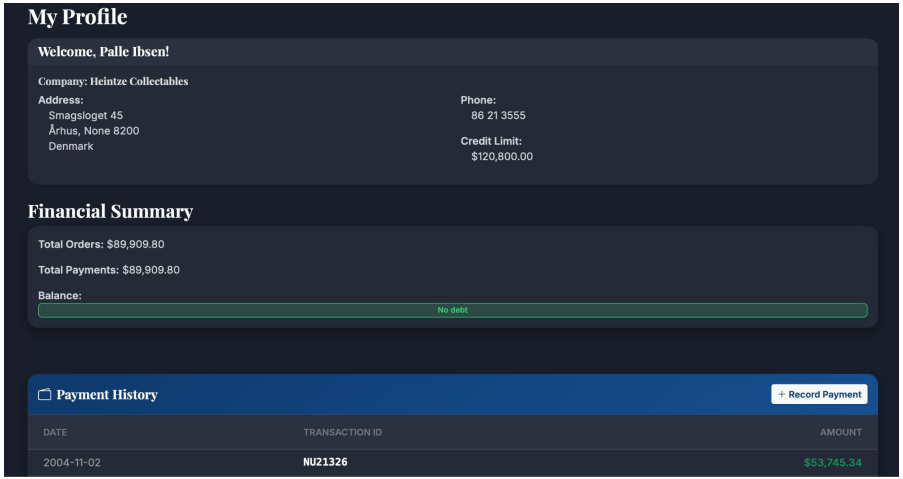

Checkout
Review your items and complete your order.
Order Summary
PRODUCT
UNIT PRICE
QTY
SUBTOTAL
1950's Chicago Surface Lines Streetcar
$62.14
1
$62.14
Order Notes
Special order
Payment Details
Subtotal $62.14
Shipping Free
Total $62.14
Payment will be collected after order confirmation.
Place Order
Return to Cart

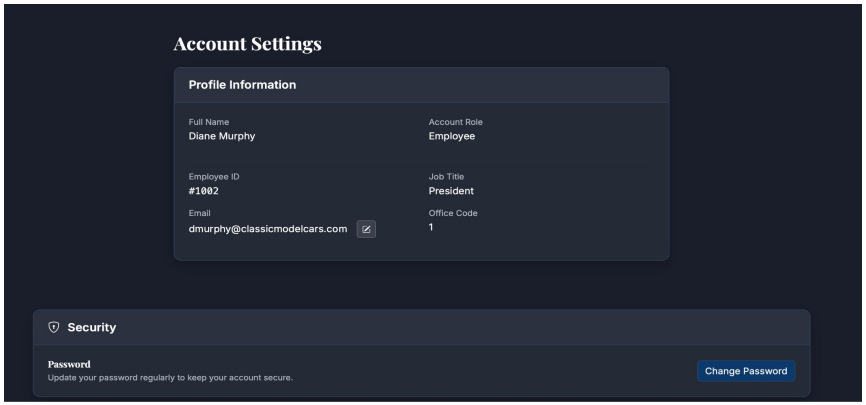

Classic Cars
Sort by
1952 Alpine Renault 1300
Scale: 1:10
Buy Price: $98.58
MSRP: $214.3
Stock: 7305 pieces
1972 Alfa Romeo GTA
Scale: 1:10
Buy Price: $85.68
MSRP: $136.0
Stock: 3252 pieces
1962 LanciaA Delta 16V
Scale: 1:10
Buy Price: $103.42
MSRP: $147.74
Stock: 6791 pieces

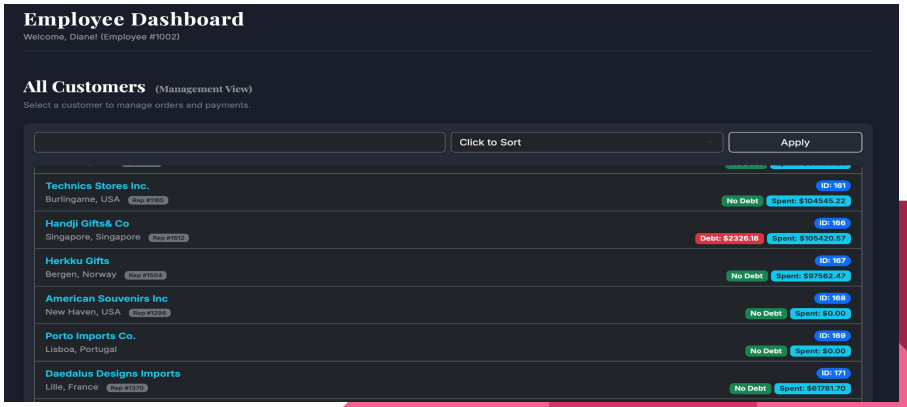

Order #10426 placed successfully!
Order #10426
In Process
Order Items
PRODUCT
QTY
PRICE
TOTAL
1950's Chicago Surface Lines Streetcar
S32_3207
1
$62.14
$62.14
Comments & Notes
Special order
Summary
Subtotal (1 items) $62.14
Shipping Free
Grand Total $62.14
Cancel Order
Back to My Orders

Features


Employee Dashboard
Welcome, Diane! (Employee #1002)
All Customers (Management View)
Select a customer to manage orders and payments.
Click to Sort
Apply
Atelier graphique
Nantes, France
ID: 103
No Debt
Spent: $22314.36
Signal Gift Stores
Las Vegas, USA
ID: 112
No Debt
Spent: $80180.98
Australian Collectors, Co.
Melbourne, Australia
ID: 114
No Debt
Spent: $180585.07
La Rochelle Gifts
Nantes, France
ID: 119
No Debt
Spent: $116949.68
Baane Mini Imports
Stavern, Norway
ID: 121
No Debt
Spent: $104224.79
Mini Gifts Distributors Ltd.
San Rafael, USA
ID: 124
No Debt
Spent: $584188.24


My Profile
Welcome, Palle Ibsen!
Company: Heintze Collectables
Address:
Smagsloget 45
Århus, None 8200
Denmark
Phone:
86 21 3555
Credit Limit:
$120,800.00
Financial Summary
Total Orders: $89,909.80
Total Payments: $89,909.80
Balance:
No debt
Payment History
+ Record Payment
DATE
TRANSACTION ID
AMOUNT
2004-11-02
NU21326
$53,745.34


Account Settings
Profile Information
Full Name
Diane Murphy
Account Role
Employee
Employee ID
#1002
Job Title
President
Email
dmurphy@classicmodelcars.com
Office Code
1
Security
Password
Update your password regularly to keep your account secure.
Change Password


Employee Dashboard
Welcome, Diane! (Employee #1002)
All Customers (Management View)
Select a customer to manage orders and payments.
Click to Sort
Apply
Technics Stores Inc.
Burlingame, USA
ID: 161
No Debt
Spent: $104545.22
Handji Gifts& Co
Singapore, Singapore
ID: 166
Debt: $2326.18
Spent: $105420.57
Herkku Gifts
Bergen, Norway
ID: 167
No Debt
Spent: $97562.47
American Souvenirs Inc
New Haven, USA
ID: 168
No Debt
Spent: $0.00
Porto Imports Co.
Lisboa, Portugal
ID: 169
No Debt
Spent: $0.00
Daedalus Designs Imports
Lille, France
ID: 171
No Debt
Spent: $61781.70

Complex Queries

- This complex query is a data analysis function that **joins Offices, Employees, Customers, Orders, OrderDetails** to link office data to individual sales transactions.
- The subquery dynamically calculates the average revenue of other employees in the same office for every row, allowing for a fair, location-based performance comparison. It aggregates revenue for all peers in that office first, then calculates the AVG of those totals.
- The inner GROUP BY statement helps to calculate the average revenue by aggregating individual employee revenue. The outer GROUP BY calculates the total for that individual sales representative.
- It also uses COALESCE to handle NULL values (employees with no sales appear as $0.00).

```
def get_sales_rep_vs_office_average(self):

    query = """
        SELECT
            o.city AS office_city,
            e.employeeNumber,
            CONCAT(e.firstName, ' ', e.lastName) AS sales_rep,
            COALESCE(SUM(od.quantityOrdered * od.priceEach), 0) AS rep_revenue,

            (
                SELECT AVG(rep_total)
                FROM (
                    SELECT
                        COALESCE(SUM(od2.quantityOrdered * od2.priceEach), 0) AS rep_total
                    FROM employees e2
                    LEFT JOIN customers c2
                        ON e2.employeeNumber = c2.salesRepEmployeeNumber
                    LEFT JOIN orders o2
                        ON c2.customerNumber = o2.customerNumber
                    LEFT JOIN orderdetails od2
                        ON o2.orderNumber = od2.orderNumber
                    WHERE e2.officeCode = e.officeCode
                    GROUP BY e2.employeeNumber
                ) office_rep_totals
            ) AS office_avg_revenue

        FROM offices o
        JOIN employees e
            ON o.officeCode = e.officeCode
        LEFT JOIN customers c
            ON e.employeeNumber = c.salesRepEmployeeNumber
        LEFT JOIN orders ord
            ON c.customerNumber = ord.customerNumber
        LEFT JOIN orderdetails od
            ON ord.orderNumber = od.orderNumber

        GROUP BY o.city, e.employeeNumber, sales_rep
        ORDER BY office_city, rep_revenue DESC;
    """
```

Complex Queries

Sales Rep Performance vs Office Average

All Offices | Filter

OFFICE	SALES REP	REP REVENUE	OFFICE AVG	Δ
Boston	Steve Patterson	$505875.42	$446269.31	+$59606.11
Boston	Julie Firrelli	$386663.20	$446269.31	$-59606.11
London	Larry Bott	$732096.79	$718475.35	+$13621.44
London	Barry Jones	$704853.91	$718475.35	$-13621.44
NYC	George Vanauf	$669377.05	$578794.86	+$90582.19
NYC	Foon Yue Tseng	$488212.67	$578794.86	$-90582.19
Paris	Gerard Hernandez	$1258577.81	$616752.32	+$641825.49
Paris	Pamela Castillo	$868220.55	$616752.32	+$251468.23
Paris	Loui Bondur	$569485.75	$616752.32	$-47266.57
Paris	Martin Gerard	$387477.47	$616752.32	$-229274.85

← Previous | Page 1 | Next →

Complex Queries

- This complex query examines the offices in a comparative manner in each product type. It **JOINS Offices, Employees, Customers, Orders, OrderDetails , products , productLines.**
- It utilizes a **correlated nested subquery** to calculate global average. For each record, the subquery calculates the *worldwide* average revenue for that category, allowing for performance comparison.
- It relies on **Dynamic SQL generation** to apply user-defined filters and supports **Pagination** to manage large result sets efficiently.
- It uses `COALESCE` functions to handle `NULL` values, ensuring accurate financial calculations even for product lines that haven't generated sales in specific regions yet.

```python
def get_ultimate_analysis_paginated(self, limit=10, offset=0, filter_office=None, filter_category=None):
    """
    THE ULTIMATE QUERY (PAGINATED & FILTERED):
    Analyzes Office performance per Product Category vs Global Averages.
    Includes 7-Table Joins, Nested Subqueries, and Dynamic Filtering.
    """
    # Dynamic WHERE clause construction
    where_clauses = ["pl.productLine IS NOT NULL"]
    params = []

    if filter_office and filter_office != 'All':
        where_clauses.append("o.city = %s")
        params.append(filter_office)

    if filter_category and filter_category != 'All':
        where_clauses.append("pl.productLine = %s")
        params.append(filter_category)

    where_sql = " AND ".join(where_clauses)

    query = f"""
        SELECT
            o.city AS Office,
            o.territory AS Region,
            pl.productLine AS Category,

            COUNT(DISTINCT ord.orderNumber) AS Order_Count,
            COALESCE(SUM(od.quantityOrdered * od.priceEach), 0) AS Total_Revenue,

            -- COMPLEX NESTED SUBQUERY: Global Average Calculation
            (
                SELECT AVG(sub_sum)
                FROM (
                    SELECT SUM(od2.quantityOrdered * od2.priceEach) as sub_sum
                    FROM orderdetails od2
                    JOIN products p2 ON od2.productCode = p2.productCode
                    WHERE p2.productLine = pl.productLine -- Correlated Reference
                    GROUP BY od2.orderNumber
                ) AS global_stats
            ) AS Global_Category_Avg

        FROM offices o
        JOIN employees e
            ON o.officeCode = e.officeCode
        LEFT JOIN customers c
            ON e.employeeNumber = c.salesRepEmployeeNumber
        LEFT JOIN orders ord
            ON c.customerNumber = ord.customerNumber
        LEFT JOIN orderdetails od
            ON ord.orderNumber = od.orderNumber
        LEFT JOIN products p
            ON od.productCode = p.productCode
        LEFT JOIN productlines pl
            ON p.productLine = pl.productLine

        WHERE {where_sql}

        GROUP BY o.officeCode, pl.productLine
        ORDER BY o.city, Total_Revenue DESC
        LIMIT %s OFFSET %s
    """
```

Complex Queries

All Offices | All Categories

OFFICE DETAILS	PRODUCT CATEGORY	ORDER COUNT	TOTAL REVENUE	GLOBAL CATEGORY AVG.	PERFORMANCE
Boston	Classic Cars	19	$335,968.81	$18,439.82	Below Avg
Boston	Vintage Cars	19	$172,879.04	$9,612.62	Below Avg
Boston	Motorcycles	6	$118,533.76	$14,195.27	Above Avg
Boston	Trucks and Buses	6	$103,893.36	$13,654.85	Above Avg
Boston	Ships	11	$76,402.78	$9,764.68	Below Avg
Boston	Planes	8	$67,814.36	$14,464.21	Below Avg
Boston	Trains	4	$17,046.51	$4,011.34	Above Avg
London	Classic Cars	34	$644,626.76	$18,439.82	Above Avg
London	Vintage Cars	24	$219,077.88	$9,612.62	Below Avg
London	Motorcycles	10	$141,552.32	$14,195.27	Below Avg

Previous | 1 | 2 | 3 | 4 | 5 | Next

Complex Queries

- This query generates a comprehensive performance report by merging workforce data with sales metrics in offices manner.
- It joins **5 tables** (`Offices`, `Employees`, `Customers`, `Orders`, `OrderDetails`) using `LEFT JOINS` with preserving data integrity.
- It uses `COUNT(DISTINCT column)` to accurately count employees and customers. This is crucial to prevent the "duplicate counting caused by the multi-table joins.
- A **Scalar Subquery** is used in the `SELECT` clause to dynamically fetch the specific "Manager" or "VP" for each office without complicating the main `GROUP BY` logic.
- It implements a conditional **CASE statement** to calculate the 'Average Order Value', safely handling potential division-by-zero errors for new offices

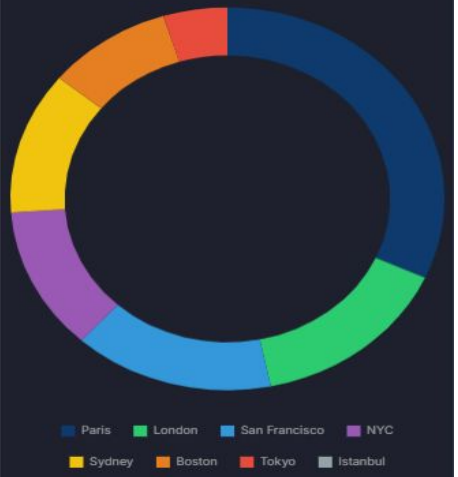
```
def get_consolidated_office_stats(self):
    """
    MERGED & COMPLEX QUERY:
    Combines Activity Report + Order Stats into one master query.
    Features: Multi-Join (5 Tables), Left Joins, Count Distinct, Nested Subquery for Manager.
    """
    query = """
        SELECT
            o.officeCode,
            o.city,
            o.country,
            o.territory,
            COUNT(DISTINCT e.employeeNumber) as active_employees,
            COUNT(DISTINCT c.customerNumber) as customer_count,
            COUNT(DISTINCT ord.orderNumber) as total_orders,
            COALESCE(SUM(od.quantityOrdered * od.priceEach), 0) as total_revenue,

            COALESCE(
                (SELECT CONCAT(m.firstName, ' ', m.lastName)
                 FROM employees m
                 WHERE m.officeCode = o.officeCode
                 AND (m.jobTitle LIKE '%Manager%' OR m.jobTitle LIKE '%VP%' OR m.jobTitle LIKE '%President%')
                 LIMIT 1
                ), 'Regional Lead'
            ) as manager_name,

            CASE
                WHEN COUNT(DISTINCT ord.orderNumber) > 0
                THEN CAST(COALESCE(SUM(od.quantityOrdered * od.priceEach), 0) AS DECIMAL(10,2)) / COUNT(DISTINCT ord.orderNumber)
                ELSE 0.00
            END as avg_ticket_size

            FROM offices o
            LEFT JOIN employees e
                ON o.officeCode = e.officeCode
            LEFT JOIN customers c
                ON e.employeeNumber = c.salesRepEmployeeNumber
            LEFT JOIN orders ord
                ON c.customerNumber = ord.customerNumber
            LEFT JOIN orderdetails od
                ON ord.orderNumber = od.orderNumber

            GROUP BY o.officeCode, o.city, o.country, o.territory
            ORDER BY total_revenue DESC;
        """
```

Complex Queries

Global Office Performance

LOCATION	MANAGER	STAFF / CLIENTS	TOTAL ORDERS	REVENUE	AVG. ORDER VALUE
4 Paris	Gerard Bondur	5 29	105	$3,036,696	$28,921
7 London	Regional Lead	2 17	47	$1,436,951	$30,573
1 San Francisco	Diane Murphy	6 12	48	$1,429,064	$29,772
3 NYC	Regional Lead	2 15	39	$1,157,590	$29,682
6 Sydney	William Patterson	4 10	38	$1,147,176	$30,189
2 Boston	Regional Lead	2 12	36	$894,026	$24,834
5 Tokyo	Regional Lead	2 5	16	$457,110	$28,569
9 Istanbul	Regional Lead	0 0	0	$0	$0

Revenue Distribution

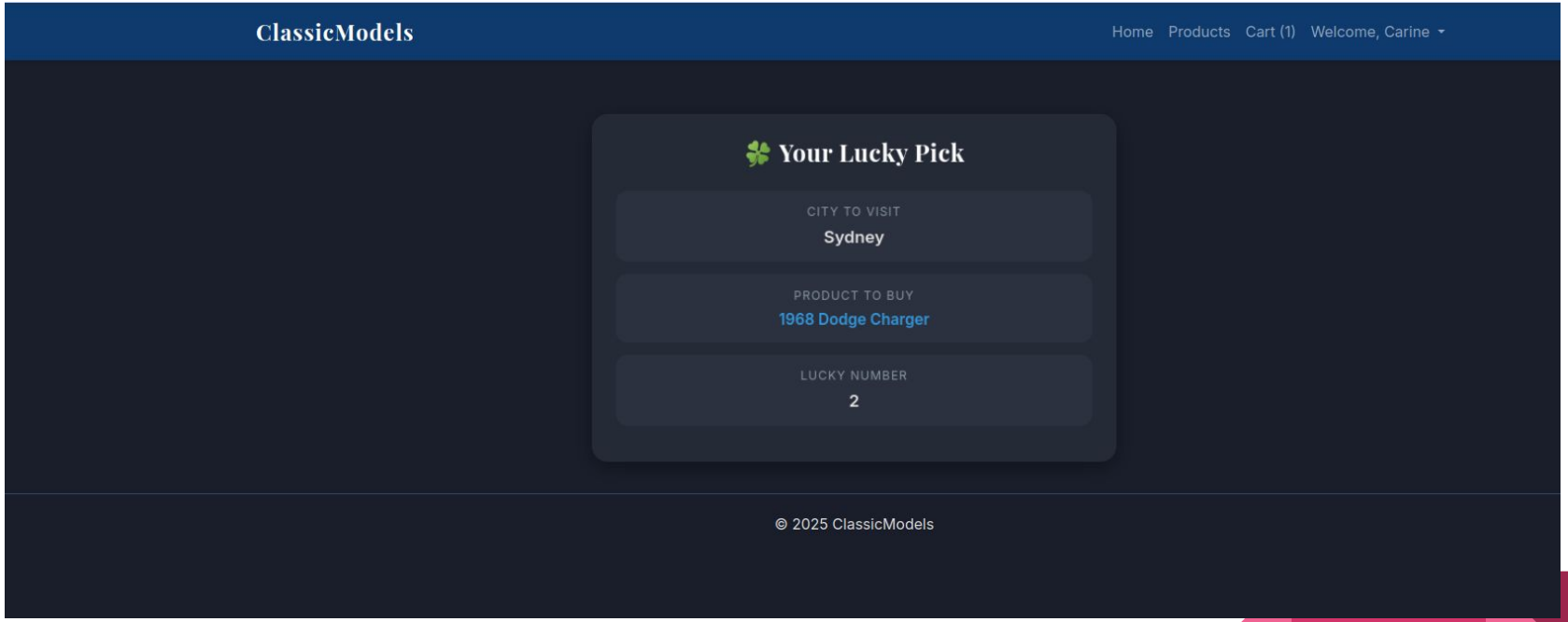


Complex Queries

- The "Feel Lucky" query is a complex analytical tool designed to provide a personalized, randomized recommendation for customers. It traverses seven different tables to find a connection between a customer's history, their assigned support staff, and a random product recommendation.
- The query prioritizes the city of the **Sales Representative** assigned to the customer (`o.city`). If a customer is not assigned a representative, the system dynamically falls back to a **completely random office city** (`off.city`). This ensures the user always receives a travel suggestion.
- To generate a valid recommendation, the query executes a **7-table join**:
    1. **Customer** → **Employee**: Identifies the Sales Rep.
    2. **Employee** → **Office**: Finds the Rep's home base.
    3. **Customer** → **Order**: Looks at buying history.
    4. **Order** → **OrderDetails** → **Product**: Finds the specific items purchased.

```
def feel_lucky(self):
    query = """
            SELECT
                COALESCE(o.city, off.city) AS city_to_visit,
                p.productName AS product_to_buy,
                p.productCode,
                CAST(FLOOR(RAND() * 100) AS SIGNED) + 1 AS lucky_number
            FROM customers c

            LEFT JOIN employees e
                ON c.salesRepEmployeeNumber = e.employeeNumber

            LEFT JOIN offices o
                ON e.officeCode = o.officeCode

            JOIN orders ord
                ON c.customerNumber = ord.customerNumber

            JOIN orderdetails od
                ON ord.orderNumber = od.orderNumber

            JOIN products p
                ON od.productCode = p.productCode

            JOIN (
                SELECT city
                FROM offices
                ORDER BY RAND()
                LIMIT 1
            ) off
            WHERE ord.orderNumber = (
                SELECT orderNumber
                FROM orders
                ORDER BY RAND()
                LIMIT 1
            )
            GROUP BY
                o.city,
                off.city,
                p.productName,
                p.productCode

            ORDER BY RAND()
            LIMIT 1;


    """
    return self.execute_query(query)
```

Complex Queries

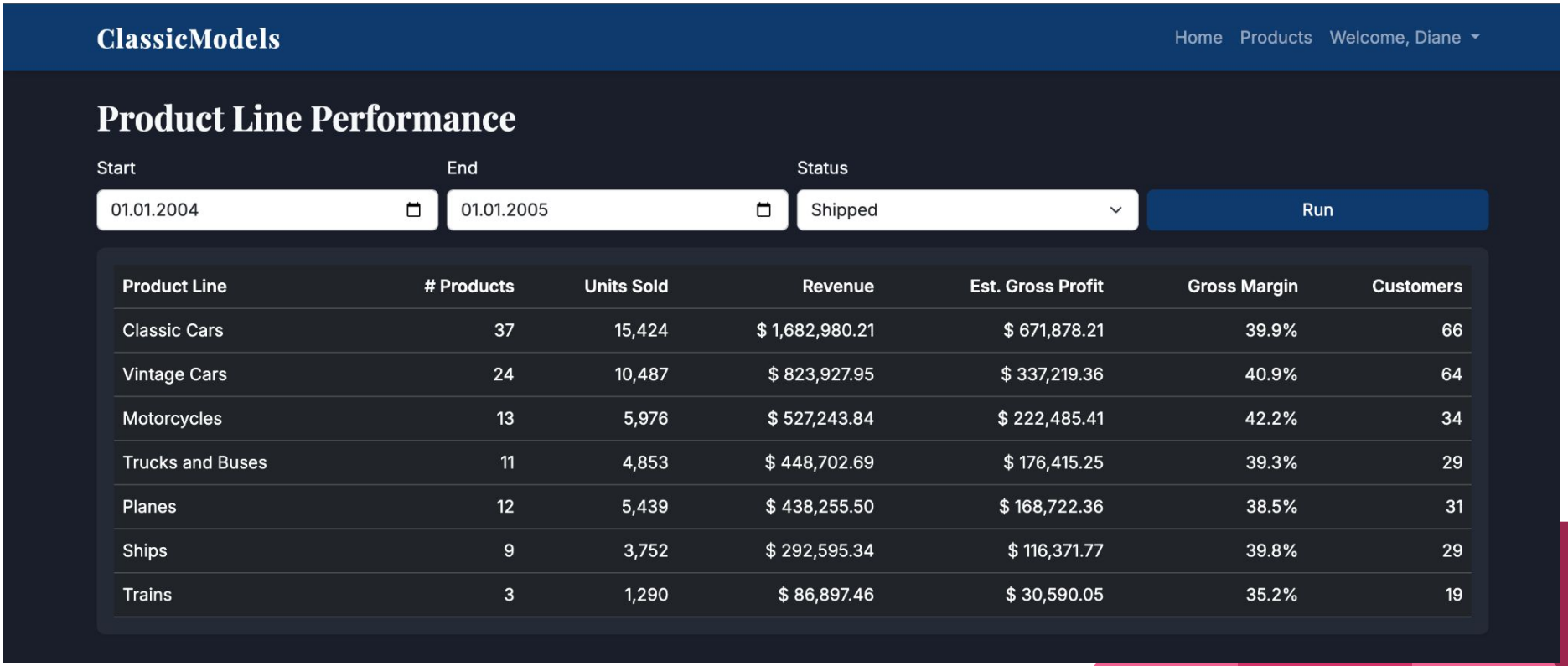


Complex Queries

- This complex query is a product-line–level analytics reporting function.
- It joins `productlines`, `products`, `orderdetails`, and `orders` to **connect product categories to individual sales transactions**.
- It uses `SUM` and `COUNT` aggregations to calculate units sold, revenue, estimated cost, gross profit, gross margin, and distinct customers per product line.
- The inner `GROUP BY` **aggregates transactional data** at the product-line level.
- The outer `LEFT JOIN` ensures that product lines with no qualifying sales are still included in the report.
- `COALESCE` is used to handle `NULL` values so that missing sales data is displayed as zero.

```
def report_productlines():

    sql = """
    SELECT
        pl.productLine,
        COALESCE(s.numProducts, 0)        AS numProducts,
        COALESCE(s.unitsSold, 0)          AS unitsSold,
        COALESCE(s.revenue, 0)            AS revenue,
        COALESCE(s.estCOGS, 0)            AS estCOGS,
        COALESCE(s.estGrossProfit, 0)     AS estGrossProfit,
        COALESCE(s.estGrossMargin, 0)     AS estGrossMargin,
        COALESCE(s.distinctCustomers, 0)  AS distinctCustomers
    FROM productlines pl
    LEFT JOIN (
        SELECT
            pl2.productLine,
            COUNT(DISTINCT pr.productCode) AS numProducts,
            SUM(od.quantityOrdered) AS unitsSold,
            SUM(od.quantityOrdered * od.priceEach) AS revenue,
            SUM(od.quantityOrdered * pr.buyPrice) AS estCOGS,
            SUM(od.quantityOrdered * od.priceEach) - SUM(od.quantityOrdered * pr.buyPrice) AS estGrossProfit,
            CASE
            WHEN SUM(od.quantityOrdered * od.priceEach) = 0 THEN 0
            ELSE
                (SUM(od.quantityOrdered * od.priceEach) - SUM(od.quantityOrdered * pr.buyPrice))
                / SUM(od.quantityOrdered * od.priceEach)
            END AS estGrossMargin,
            COUNT(DISTINCT o.customerNumber) AS distinctCustomers
        FROM productlines pl2
        JOIN products pr
        ON pr.productLine = pl2.productLine
        JOIN orderdetails od
        ON od.productCode = pr.productCode
        JOIN orders o
        ON o.orderNumber = od.orderNumber
        WHERE o.status = %s
        AND o.orderDate >= %s
        AND o.orderDate <  %s
        GROUP BY pl2.productLine
    ) s
    ON s.productLine = pl.productLine

    ORDER BY revenue DESC;
"""
```

Complex Queries



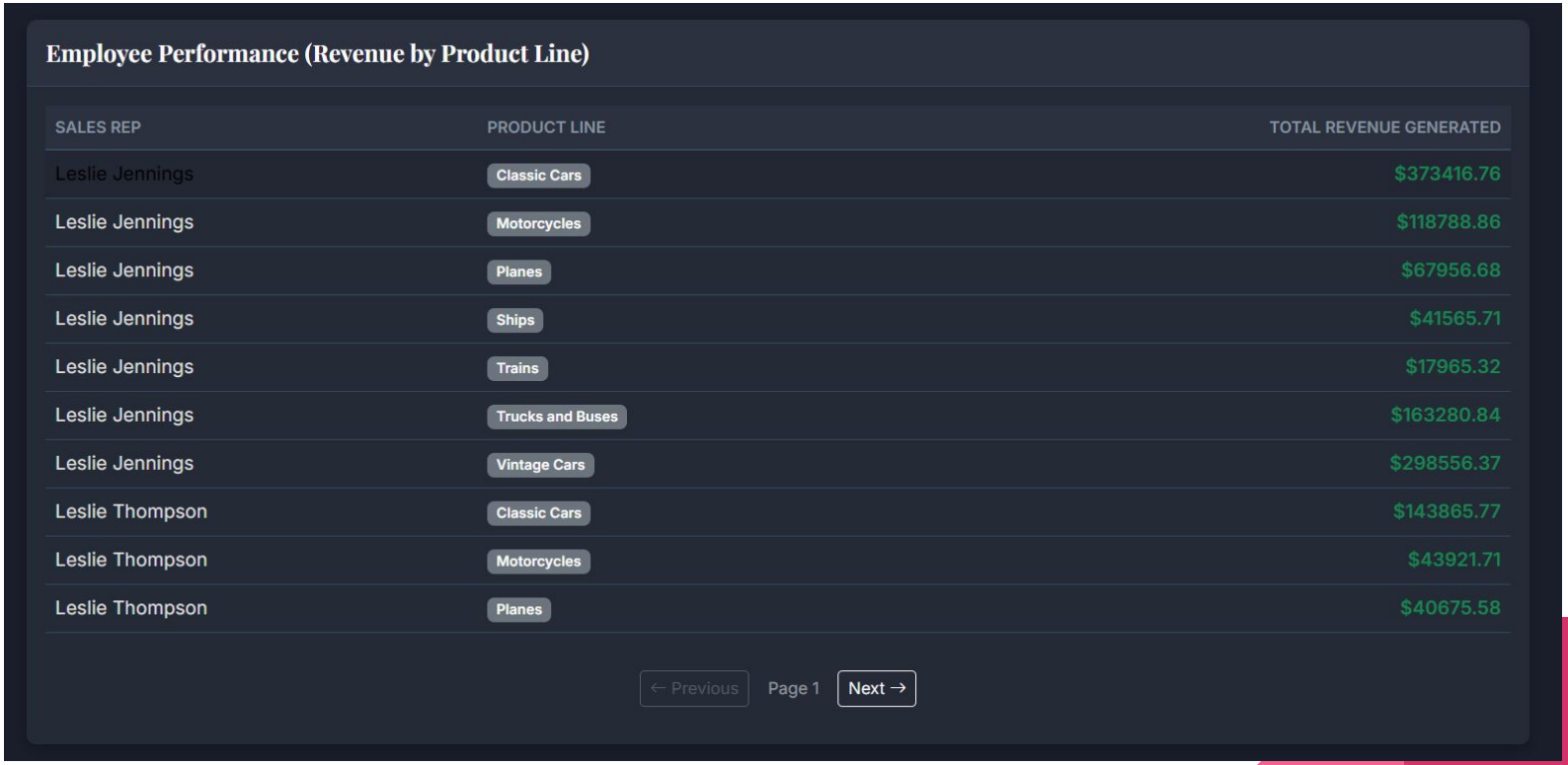
Product Line	# Products	Units Sold	Revenue	Est. Gross Profit	Gross Margin	Customers
Classic Cars	37	15,424	$ 1,682,980.21	$ 671,878.21	39.9%	66
Vintage Cars	24	10,487	$ 823,927.95	$ 337,219.36	40.9%	64
Motorcycles	13	5,976	$ 527,243.84	$ 222,485.41	42.2%	34
Trucks and Buses	11	4,853	$ 448,702.69	$ 176,415.25	39.3%	29
Planes	12	5,439	$ 438,255.50	$ 168,722.36	38.5%	31
Ships	9	3,752	$ 292,595.34	$ 116,371.77	39.8%	29
Trains	3	1,290	$ 86,897.46	$ 30,590.05	35.2%	19

Complex Queries

- This complex query is an employee-performance analytics reporting function.
- It joins `employees, customers, orders, order_details`, and `products` to **connect sales representatives to their specific product-line sales transactions.**
- It uses SUM aggregations to calculate the total revenue generated by each sales representative per product category.
- The WHERE clause filters the dataset to focus exclusively on employees holding the 'Sales Rep' job title.
- The GROUP BY **aggregates transactional data** at the individual employee and product-line levels.
- The LIMIT and OFFSET clauses are utilized to implement **pagination**.

```
def get_employee_performance_matrix(self, limit=10, offset=0):
    query = """
        SELECT
            e.firstName,
            e.lastName,
            p.productLine,
            SUM(od.quantityOrdered * od.priceEach) as revenue
        FROM employees e
        JOIN customers c ON e.employeeNumber = c.salesRepEmployeeNumber
        JOIN orders o ON c.customerNumber = o.customerNumber
        JOIN orderdetails od ON o.orderNumber = od.orderNumber
        JOIN products p ON od.productCode = p.productCode
        WHERE e.jobTitle = 'Sales Rep'
        GROUP BY e.employeeNumber, e.firstName, e.lastName, p.productLine
        ORDER BY e.employeeNumber
        LIMIT %s OFFSET %s
    """
    return self.execute_query(query, (limit, offset))
```

Complex Queries



Employee Performance (Revenue by Product Line)

SALES REP	PRODUCT LINE	TOTAL REVENUE GENERATED
Leslie Jennings	Classic Cars	$373416.76
Leslie Jennings	Motorcycles	$118788.86
Leslie Jennings	Planes	$67956.68
Leslie Jennings	Ships	$41565.71
Leslie Jennings	Trains	$17965.32
Leslie Jennings	Trucks and Buses	$163280.84
Leslie Jennings	Vintage Cars	$298556.37
Leslie Thompson	Classic Cars	$143865.77
Leslie Thompson	Motorcycles	$43921.71
Leslie Thompson	Planes	$40675.58

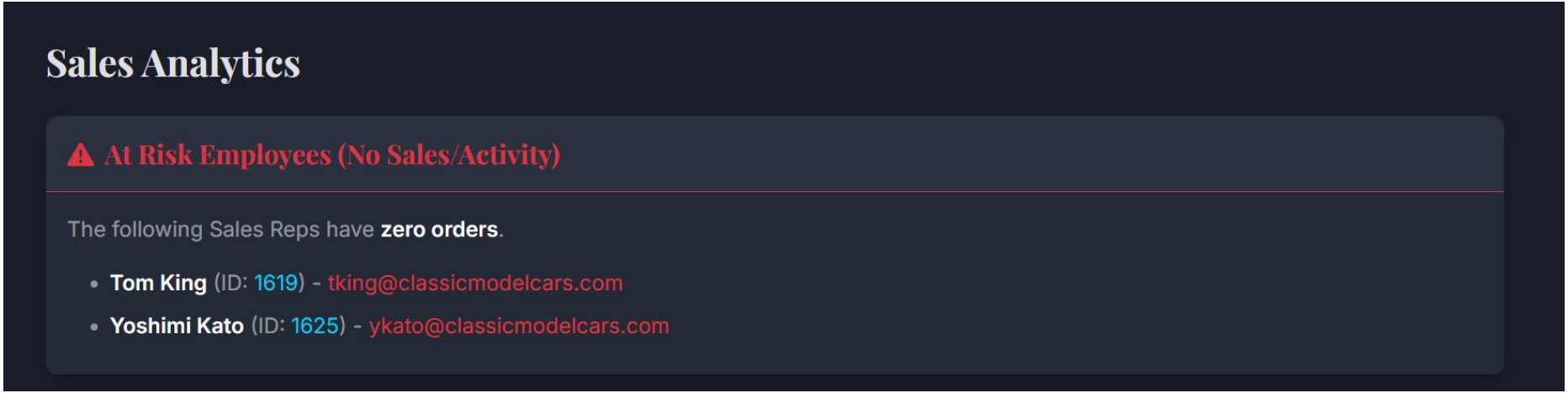
← Previous Page 1 Next →

Complex Queries

- This query is an employee productivity and efficiency tracking function.
- It performs a sequence of `LEFT JOIN`s starting from the employees table to customers and orders to map the relationship between sales staff and their transaction history.
- It uses the `COUNT` aggregation to calculate both the total number of customers assigned to a representative and their orders.
- The `WHERE` clause filters the dataset to focus exclusively on employees holding the 'Sales Rep' job title.
- The `GROUP BY` **aggregates transactional data** at the individual employee level.
- The `HAVING` clause serves as a critical filter to choose "unproductive" representatives who have zero successful orders (order_count = 0).

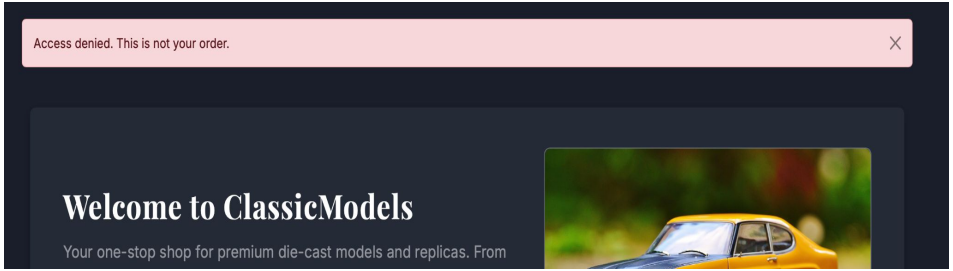
```
def get_unproductive_employees(self):
    query = """
        SELECT
            e.employeeNumber,
            e.firstName,
            e.lastName,
            e.email,
            e.jobTitle,
            COUNT(c.customerNumber) as customer_count,
            COUNT(o.orderNumber) as order_count
        FROM employees e
        LEFT JOIN customers c ON e.employeeNumber = c.salesRepEmployeeNumber
        LEFT JOIN orders o ON c.customerNumber = o.customerNumber
        WHERE e.jobTitle = 'Sales Rep'
        GROUP BY e.employeeNumber, e.firstName, e.lastName, e.email, e.jobTitle
        HAVING order_count = 0
    """
    return self.execute_query(query)
```

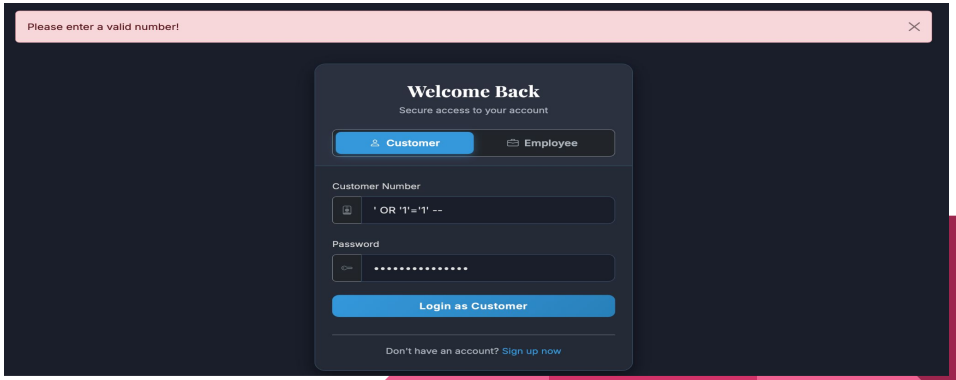
Complex Queries



Testing

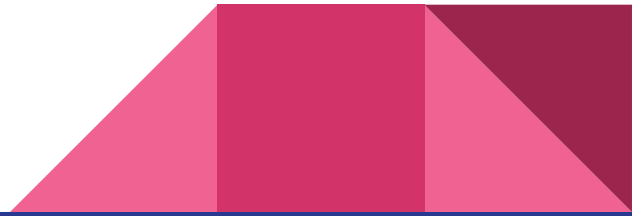
- The application is functional and secure.
- It **preserves database integrity** as needed.
- The database was tested against **SQL injections.** It handles injections.
- The app was also tested against bad data, and it passed this test as well because it **validates user inputs server-side** (type checks, required fields, etc.) and forbids bad data.
- It enforces role-based access control: **only employees can manage products and only authorized roles can access reports/analytics pages**. We attempted to break into other customers' orders or employee pages, and failed to do so.
- It prevents inconsistent references **using normalized tables**, as well as database integrity.





Questions

If you have any questions, we would love to answer them!



THANK YOU