

Databases and Application Development

Module Code: ACCA5026

Module Title: Database Design and Implementation

Student Name: David Lopez Rodriguez

Student ID: 2336542

Submission Date: May 2026

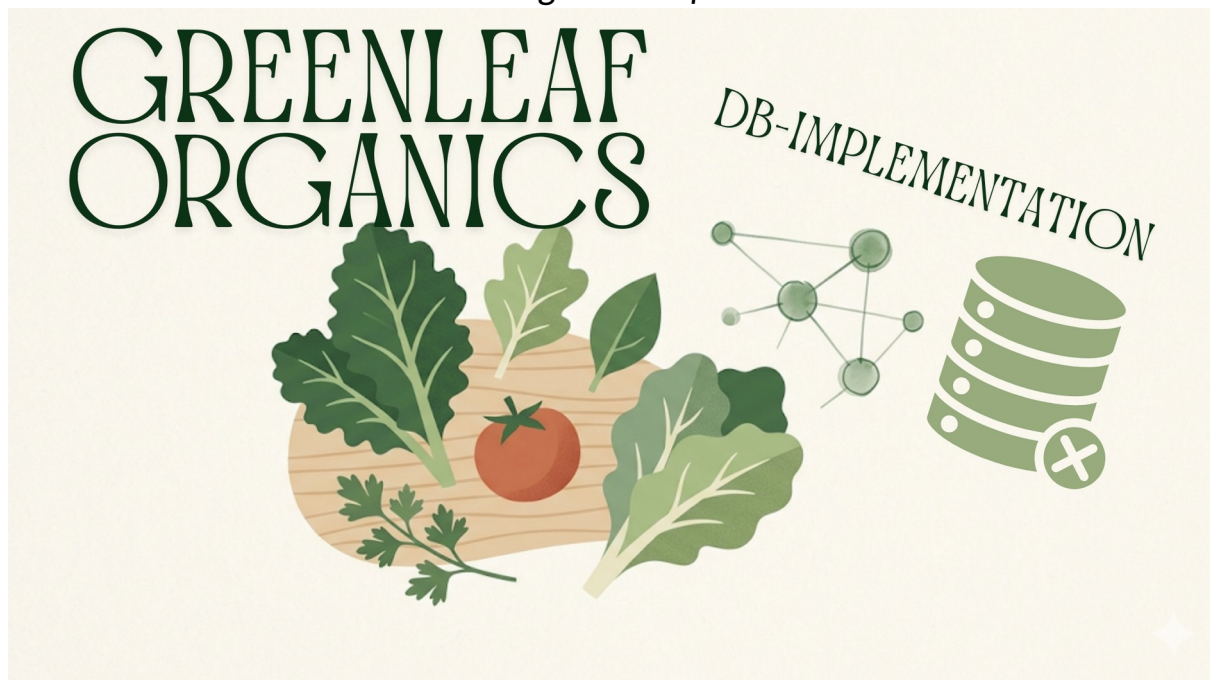
Table of Contents

Part A — GreenLeaf Organics.....	3
1. Introduction.....	4
2. Database Design.....	5
2.1 Entities and Attributes.....	5
2.2 Relationships and Cardinalities.....	6
2.3 Entity Relationship Diagram.....	7
2.4 Normalisation to Third Normal Form.....	8
3. Implementation Documentation.....	9
3.1 Table Creation.....	9
3.2 Constraints.....	10
3.3 Sample Data and Validation.....	11
4. Stored Procedures and Transactional Scenario.....	13
4.1 sp_RegisterCustomer.....	13
4.2 sp_PlaceOrder and the Rollback Scenario.....	14
5. Triggers and Auditing.....	16
6. Performance Tuning.....	17
7. Conclusion and Reflection.....	19
8. References (Part A).....	20
Part B — Retro VideoClub: SQL Injection Report.....	21
1. Introduction.....	22
2. Why it is a problem.....	22
3. Vulnerable patterns.....	23
4. Defences implemented.....	24
5. Why these defences work — live demonstration.....	25
6. Conclusion.....	26
7. References (Part B).....	26

PART A

GreenLeaf Organics

Database Design and Implementation



1. Introduction

GreenLeaf Organics is a UK wholesaler selling organic foods, environmentally friendly skincare products, and biodegradable household items. They have several organic suppliers that stock goods under their own certification, as well as customers that are a mix of private individuals and small businesses. As they scale up operations they have become tired of spreadsheets and hand taking orders. They want to migrate to a real online storefront. Under that website they need a database system capable of handling suppliers, products, customers, orders, stock and reviews in one central place.

I have been tasked with designing and constructing a relational database to power said system. The work included submitting the schema, inserting sample data, creating stored procedures with transaction handling, triggers for auditing purposes, and defining an indexing strategy. MySQL 8 was my platform of choice as the InnoDB engine supports full ACID transactions. Foreign key constraints were needed to ensure data integrity. It also supports stored procedures and triggers, which were also required by the specification. I completed this work through phpMyAdmin as it allows me to visually confirm that my SQL has the expected result in each table after running tests.

I believe a relational model is perfect for this scenario. The entities described in the brief are concise. The relationships are defined as mostly one-to-many, with some many-to-many relationships. Those are precisely the kinds of relationships that are cleanly represented using relational tables. Adding money as a factor and needing to keep track of stock quantities also means we need to focus on ACID compliance. A customer cannot place an order if that would reduce stock levels to less than zero. If that check passes but the order is only recorded without decreasing the stock then our data is corrupt. ACID compliance can help us avoid that scenario by treating the entire process as a single unit of work [1].

This report will cover the design process first, followed by implementation details. SQL for procedures and triggers will be provided along with the reasoning behind indexing choices.

2. Database Design

2.1 Entities and Attributes

The first step was to read through the brief and identify the entities involved. Eight primary entities and two junction / log tables was the final solution. The primary ones are Suppliers, Certifications, Products, Categories, Customers, Orders, OrderItems and Reviews. Next it was added ProductCategoryMap as a junction table to break the many-to-many relationship between Products and Categories, along with an InventoryLog table to audit changes to stock levels.

I debated for a while whether Certifications should be its own table or just an attribute of Suppliers. I decided to go with its own table as a supplier can have multiple certifications (such as Soil Association, Fair Trade, EU Organic etc) and each certification has additional attributes such as issuer and expiry date. Having multiple certifications in one column would either require comma separated values or repeated columns, both of which violate first normal form.

All tables have a primary key, mostly an INT AUTO_INCREMENT for simplicity. Foreign keys are prefixed with their table name to avoid ambiguity. The two junction tables have composite primary keys consisting of the two foreign keys, as is typical of an associative entity. Below is a list of the entities and their key attributes.

Suppliers	SupplierID (PK), SupplierName, ContactEmail (UNIQUE), Website, Status
Certifications	CertID (PK), SupplierID (FK), CertName, Issuer, ExpiryDate, DocumentURL
Categories	CategoryID (PK), CategoryName (UNIQUE), Description
Products	ProductID (PK), SupplierID (FK), ProductName, Price, StockQuantity, ExpiryDate
ProductCategoryMap	ProductID (FK), CategoryID (FK), composite PK
Customers	CustomerID (PK), FirstName, LastName, Email (UNIQUE), Password, ShippingAddress
Orders	OrderID (PK), CustomerID (FK), OrderDate, TotalAmount, OrderStatus, PaymentStatus
OrderItems	OrderID (FK), ProductID (FK), Quantity, PriceAtPurchase, composite PK
Reviews	ReviewID (PK), ProductID (FK), CustomerID (FK), Rating, Comment, ReviewDate
InventoryLog	AuditID (PK), ProductID (FK), LogTimestamp, ChangeAmount, ChangeType, UserAccount

2.2 Relationships and Cardinalities

A Supplier can have many Products (but a Product can have only one Supplier) so that is a one-to-many. A Supplier can also have many Certifications which is one-to-many. Categories and Products are many-to-many. A Product can exist in multiple Categories (ie organic kale can be placed in both "Organic Vegetables" and a "Featured" category) and Categories will have many Products. To solve that relationship I created the ProductCategoryMap junction table.

On the customer side of things, a Customer can write many Orders but each Order is placed by only one Customer. Each Order has many OrderItems and each OrderItem is tied to exactly one Product. Lastly, a Customer can author many Reviews and Products can have many Reviews but each Review is written by one Customer about one Product so that is many-to-many, solved by using the Reviews table itself (the reviews table holds its own data like Rating and Comment so it's not just a link table, it's an associative entity).

InventoryLog is attached to Products one-to-many. Any time stock movement occurs a row is added to the log. This allows us to see the full history of changes to StockQuantity.

2.3 Entity Relationship Diagram

The diagram below shows the full schema with the primary keys, foreign keys and cardinalities. Crow's foot notation is used to indicate the one and many sides of each relationship.

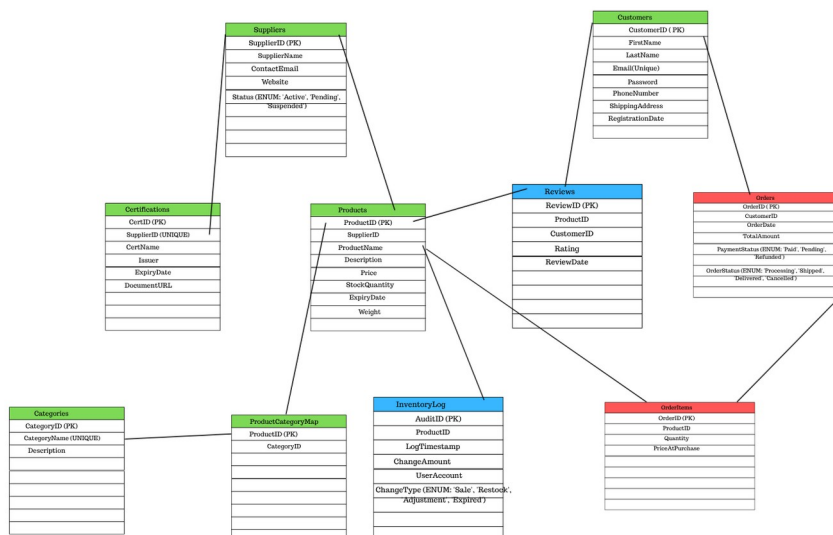


Figure 1. ER diagram for the GreenLeaf Organics database.

2.4 Normalisation to Third Normal Form

Normalisation is the process of organising the columns and tables of a relational database to reduce redundancy and avoid update anomalies. It was first formalised by Codd in the 1970s as part of the relational model [4]. There are several normal forms but the standard target for a transactional system like this one is third normal form.

First normal form means that every cell holds a single atomic value and there are no repeating groups. All my tables satisfy this. For example, instead of having a single column on Suppliers that lists all certifications separated by commas, I split that out into a Certifications table where each row holds one certification.

Second normal form mandates that all non-key columns be dependent on the entire primary key. This only applies in cases where there is more than one column in a primary key (ie. a composite key). In OrderItems above, the primary key is (OrderID, ProductID) and the non-key columns (Quantity and PriceAtPurchase) depend on the entire key; they can't be identified with just OrderID or ProductID. ProductCategoryMap also satisfies 2NF, because it technically has a composite key but no non-key attributes at all.

Third normal form extends this to say that no column should be dependent on another non-key column. A common violation would be storing SupplierName on the Products table, because then it would depend on SupplierID, which is itself a non-key attribute of that row [3]. I avoided this by keeping SupplierID as a foreign key only, and joining on the Suppliers table whenever the supplier name is needed. Orders suffers from the same problem. When writing INSERT queries I never put the customer's name or address on the Orders row; I store CustomerID there.

There is a small trade-off here. Since the data is more spread-out across tables, common queries ("give me every product and its supplier name") will need joins. This can cause read performance to suffer on a large database, but proper normalization ensures data integrity. That's more important than read-performance for a system like this, since stock levels and prices will be changing constantly [4]. Also, the indexing strategy in section 6 helps keep those joins speedy.

3. Implementation Documentation

3.1 Table Creation

The complete SQL script is provided in the attached file GreenLeafOrganics.sql. The script begins by dropping the database GreenLeafOrganics if it already exists and creates a new one. Then it creates the tables in dependency order (parents before children) so that the foreign key constraints can resolve properly. For instance, Suppliers must be created before Products because Products has a foreign key into Suppliers.

After executing the script there will be ten tables in the database. Figure 2 shows the results of running the script in phpMyAdmin and clicking on the 'Tables' option to see the list of tables. Notice that all tables were created without any errors.

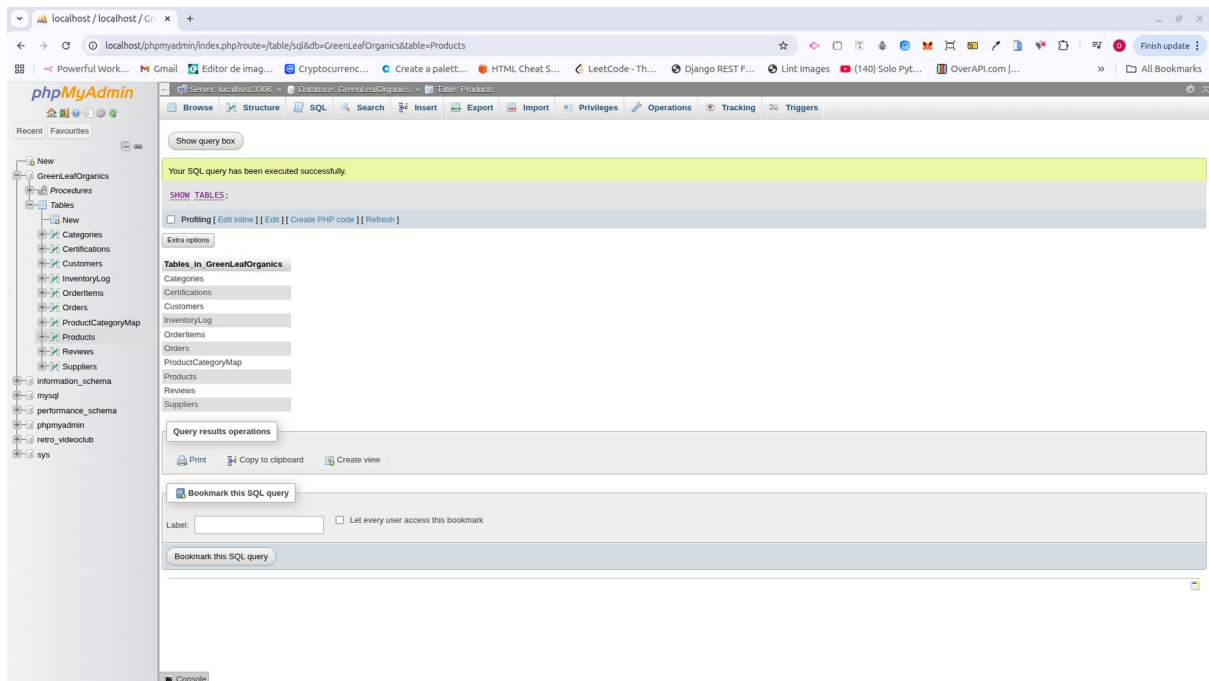


Figure 2. phpMyAdmin showing the GreenLeafOrganics database with all ten tables created.

3.2 Constraints

Several constraints are applied to maintain data integrity. PRIMARY KEY is used on every table to enforce row uniqueness. NOT NULL is applied to columns that our business rules dictate must always have a value. In this example those are ProductName, Price and Email. UNIQUE is applied to Customer Email and SupplierContactEmail and CategoryName because it would not make sense to have duplicates in those columns.

CHECK constraints was implemented to enforce simple business rules. Price has CHECK (Price > 0) to prevent someone accidentally (or deliberately) listing a product at zero or a negative price. StockQuantity has CHECK (StockQuantity >= 0) to prevent the quantity going below

zero. OrderItems.Quantity has CHECK (Quantity > 0). Reviews.Rating has CHECK (Rating BETWEEN 1 AND 5) to prevent users giving more than 5 stars.

By far the trickiest design decisions were around the foreign keys and deciding what should happen when the parent is deleted. MySQL allows you to specify CASCADE, RESTRICT, SET NULL or NO ACTION [5].

Foreign key	Action	Why
Certifications -> Suppliers	ON DELETE CASCADE	If a supplier gets removed, their certifications no longer mean anything.
Products -> Suppliers	ON DELETE RESTRICT	Stops you accidentally deleting a supplier that still has live products on the shop.
ProductCategoryMap -> Products / Categories	ON DELETE CASCADE	The link table is just an association. If either side is gone the link should be too.
Orders -> Customers	ON DELETE RESTRICT	Order history has to be preserved, even if a customer record is being removed.
OrderItems -> Orders	ON DELETE CASCADE	Order lines are part of the order, not standalone records.
OrderItems -> Products	ON DELETE RESTRICT	You cannot delete a product that has been sold, otherwise the sales history is broken.
Reviews -> Products / Customers	ON DELETE CASCADE	A review with no product or no customer is meaningless.
InventoryLog -> Products	ON DELETE CASCADE	If a product is removed, its log rows go with it.

RESTRICT was the choice I went with most often because for a real shop, losing order history would be a much bigger problem than having to manually unlink a few records before a delete. CASCADE was reserved for cases where the child rows really do not have any meaning without the parent.

3.3 Sample Data and Validation

After the tables were built I populated them with realistic sample data covering 5 suppliers, 5 categories, 5 products, 3 customers, 2 orders, 3 order items, 2 reviews and the supporting category links. The data is small but covers each table, which is enough to test queries and prove the constraints work. Figure 3 shows a count of rows in each table after the inserts ran.

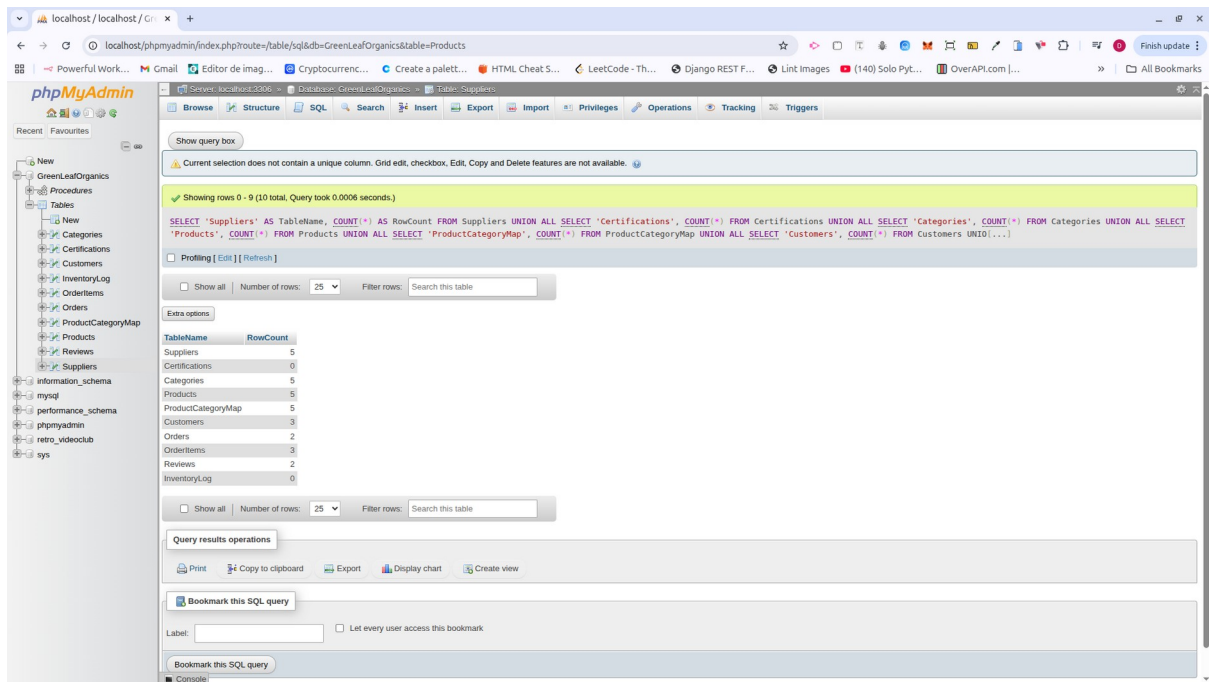


Figure 3. Row counts across all tables after sample data was inserted.

To check the joins work I ran a query that lists every product with its supplier name. This needs the Products to Suppliers foreign key to resolve, so it confirms the relationship is wired up properly.

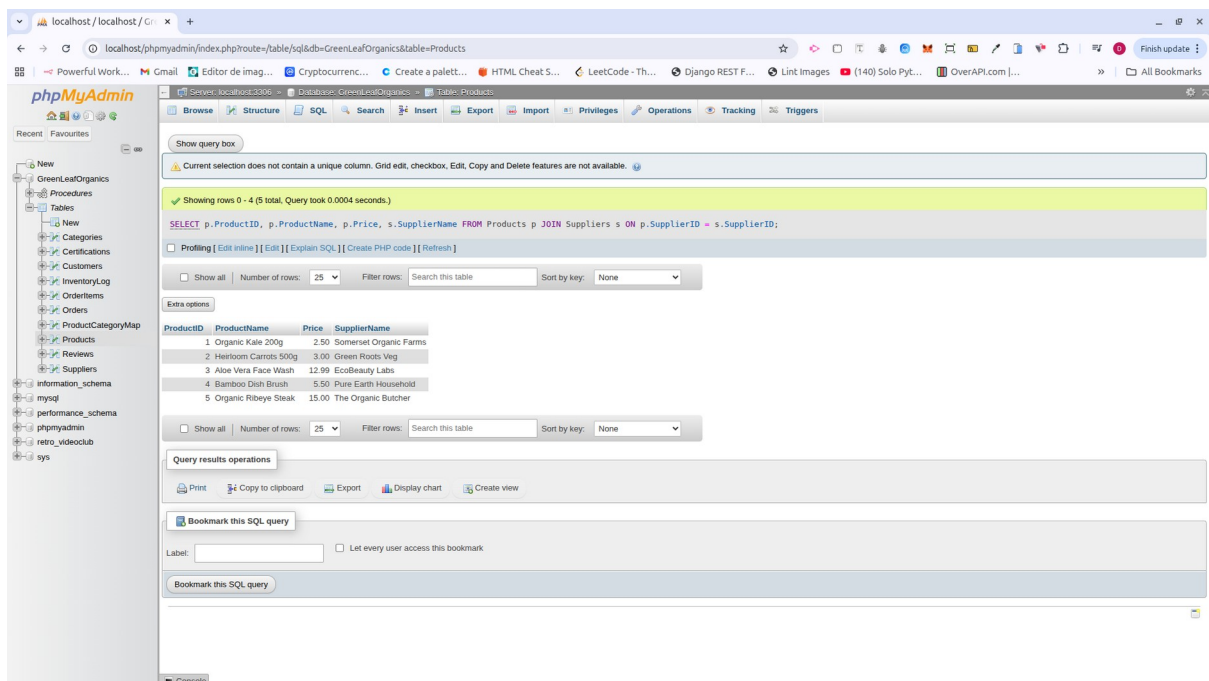


Figure 4. Products joined with Suppliers, returning each product alongside its supplier name.

A more involved check was joining four tables to get a full view of orders. The query joins Orders with Customers, OrderItems and Products and pulls the customer's name, the product name, the

quantity and the line total. This is the kind of query the website would run when a customer views their order history.

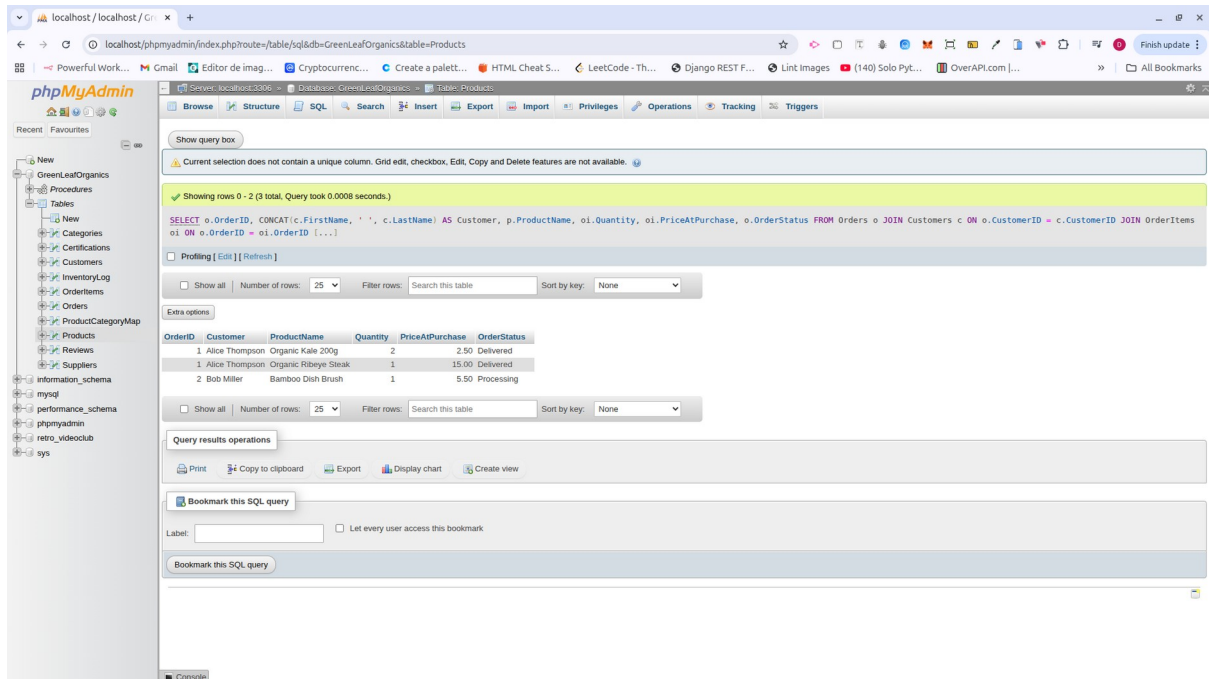


Figure 5. Multi-table join showing order details with customer and product names.

I also tested the constraints by trying to break them. Figure 6 shows what happens when I tried to insert a customer with an email address that already existed. MySQL rejects the row with a duplicate entry error, which is exactly what the UNIQUE constraint is supposed to do. I ran similar tests for negative prices and a stock quantity below zero, both of which were rejected by the CHECK constraints.

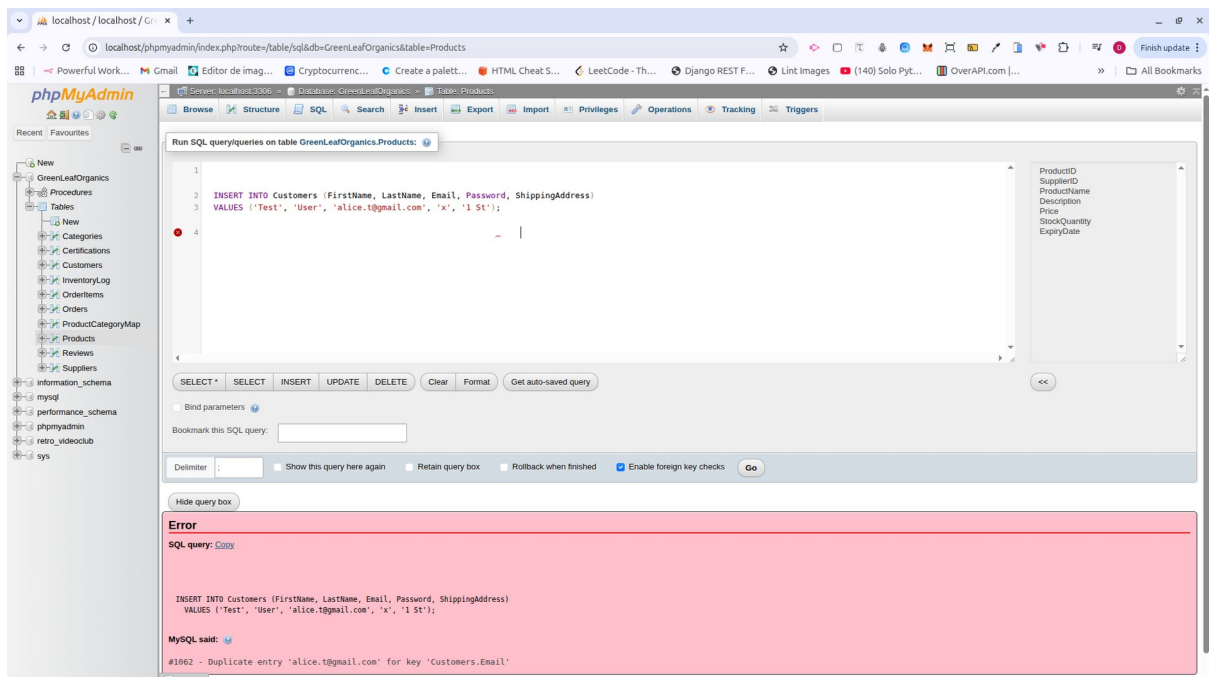


Figure 6. UNIQUE constraint rejecting a duplicate customer email.

4. Stored Procedures and Transactional Scenario

Stored procedures are blocks of SQL that live inside the database and can be called by name. They have a few advantages over running raw SQL from the application. First, the logic stays in one place, so if the rules change later you only update one procedure rather than every web page that talks to the database. Second, they are precompiled which means repeated calls do not pay the parsing cost every time. Third, they can be granted as permissions, so an application user can be allowed to call `sp_PlaceOrder` without being given direct write access to the underlying tables [6].

I wrote two procedures. `sp_RegisterCustomer` covers the customer sign up flow, and `sp_PlaceOrder` handles the order placement transaction. Both follow the standard MySQL pattern with `DELIMITER`, `CREATE PROCEDURE`, parameter list and a `BEGIN ... END` block.

4.1 sp_RegisterCustomer

This procedure takes the new customer's details as parameters and inserts them into `Customers`. There is already a `UNIQUE` constraint on `Email` that would catch duplicates anyway, but I added an explicit `IF EXISTS` check at the top so the error message is human readable rather than the raw MySQL constraint violation. If the email is already registered the procedure raises `SQLSTATE '45000'` with a friendly message; otherwise it does the `INSERT`.

Figure 7 shows the procedure being called with a brand new email address. The customer is added and a `SELECT` confirms the row is in the table.

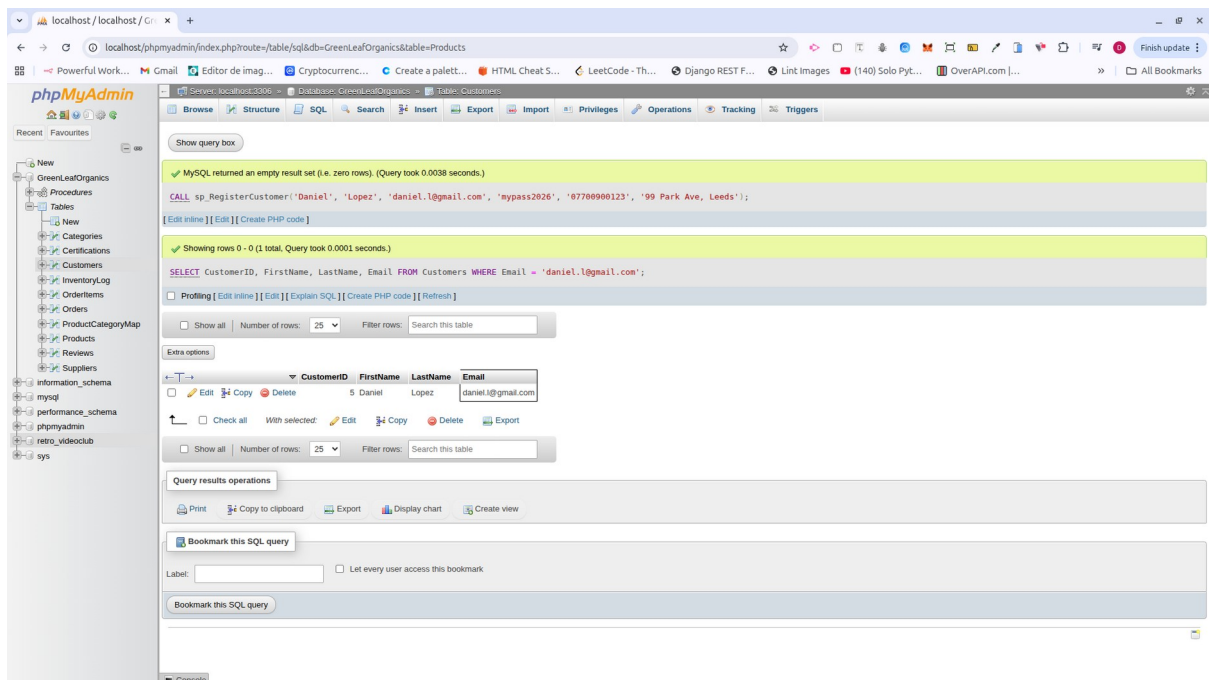


Figure 7. `sp_RegisterCustomer` adding a new customer successfully.

Figure 8 shows what happens when the same email is reused. The procedure raises the friendly error and no row is inserted.

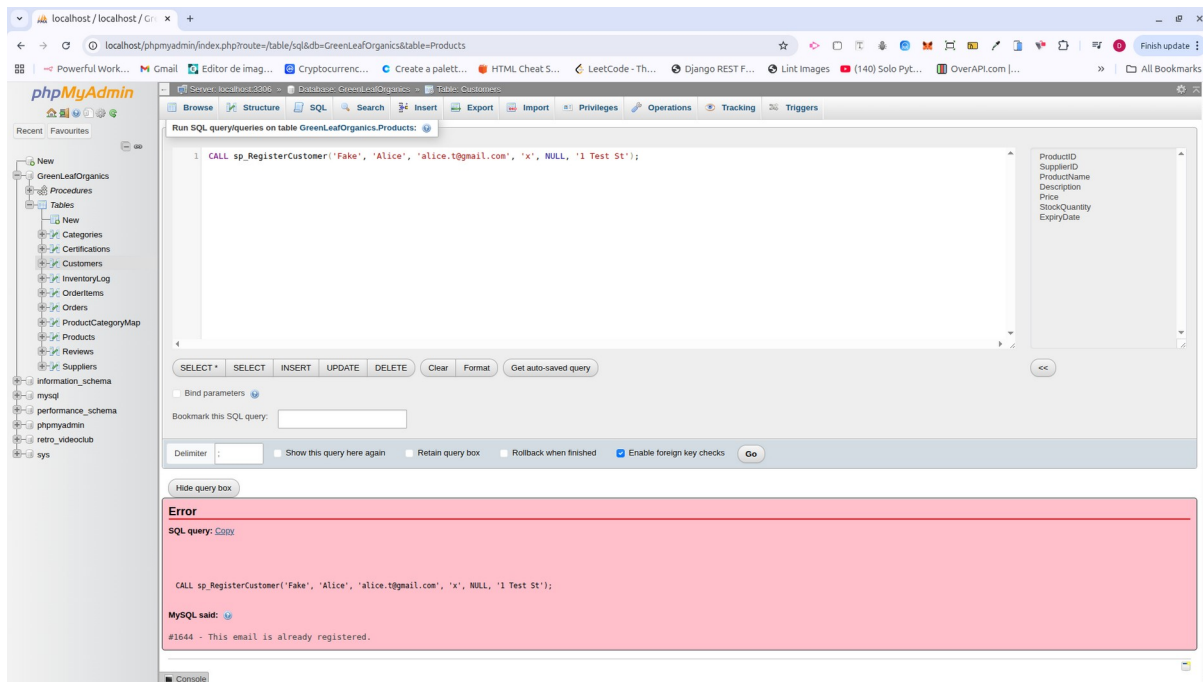


Figure 8. `sp_RegisterCustomer` rejecting a duplicate email with a friendly message.

4.2 `sp_PlaceOrder` and the Rollback Scenario

`sp_PlaceOrder` does several things that all must succeed or fail together. It inserts a new Orders row, inserts an OrderItems row, subtracts from StockQuantity on the product and updates the orders TotalAmount. If any of these statements fail the previous ones must be undone. Otherwise you end up with orders with no order items or with product whose StockQuantity doesn't match what was actually sold. This sort of problem is exactly what ACID transactions are meant to solve [1][2].

`sp_PlaceOrder` starts with `START TRANSACTION` and ends with `COMMIT`. Inbetween there is `DECLARE EXIT HANDLER FOR SQLEXCEPTION` code that executes `ROLLBACK` and `RESIGNAL` if anything fails. This ensures that `COMMIT` will only be executed if all statements before it were successful and if any `SQLEXCEPTION` occurred (for example `CHECK` constraint fails) the transaction will rollback. The first thing inside the transaction is I read the StockQuantity and Price columns using `SELECT ... FOR UPDATE`. `FOR UPDATE` places a row level lock on the product row preventing two customers from buying the last piece of stock simultaneously. Without that lock there would be a race condition where customer A reads `stock=1` and customer B reads `stock=1` (the same value). Both customers check and find that YES they can buy the product, both `UPDATE` the row to set `stock=0` and you end up selling 2 of something you only had 1 in stock. The row level lock forces customer B to wait until customer A's transaction `COMMITs` or rolls back, maintaining isolation.

With the stock safely read and locked I check: does the product exist, is there sufficient stock, does the customer exist. On any failure I `SIGNAL SQLSTATE '45000'` with a descriptive message. If I `SIGNAL` then the exit handler will trap it and execute `ROLLBACK`.

I tested both paths. Figure 9 shows a successful order placement. Customer 1 buys 2 units of product 1 (organic kale). The stock level on that product drops by 2, the order header is created, the order line is inserted and the total is set. The whole thing is one COMMIT.

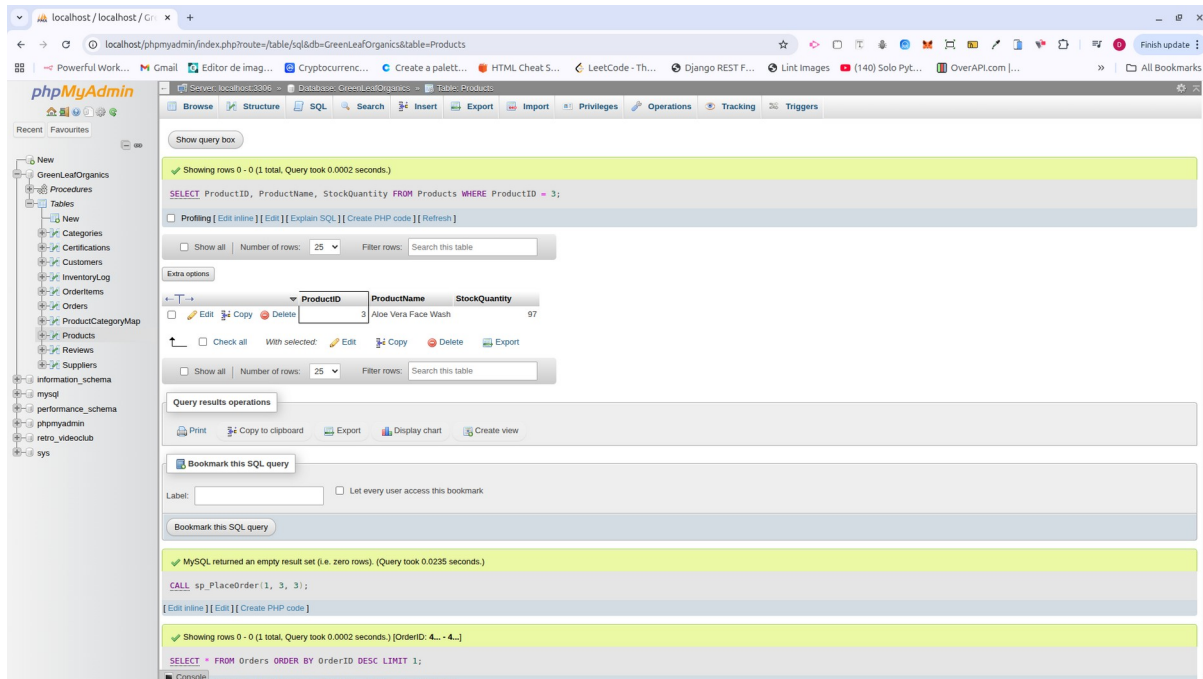


Figure 9. `sp_PlaceOrder` running successfully and decreasing stock.

Figure 10 shows the rollback path. I called `sp_PlaceOrder` asking for 50 units of a product that only has 15 in stock. The validation fails, the SIGNAL fires, the exit handler rolls back the transaction. The error message "Not enough stock available" is shown to the caller.

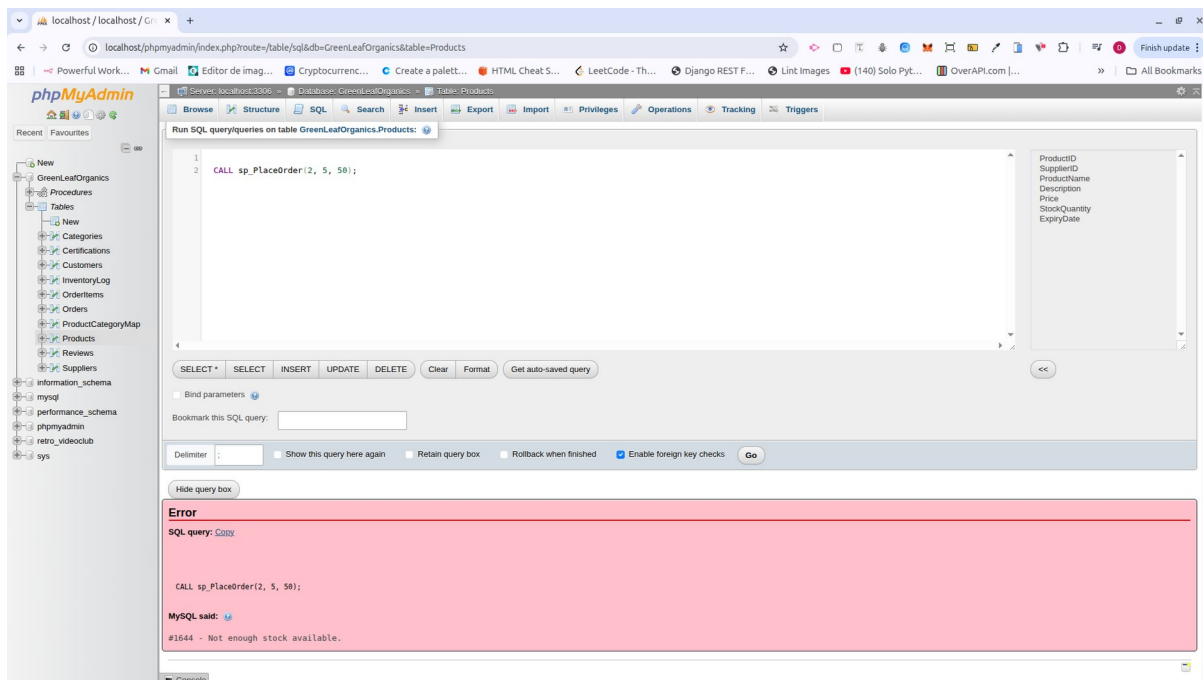


Figure 10. `sp_PlaceOrder` rejecting an order because stock is insufficient.

Importantly, after the rollback nothing in the database has changed. Figure 11 shows the StockQuantity on the affected product before and after the failed call. It is exactly the same. No Orders row exists for that attempt, no OrderItems row was created and no log entry was written. This is the atomicity property of ACID in action [1]: the whole transaction is treated as a single unit, and either all of it happens or none of it does.

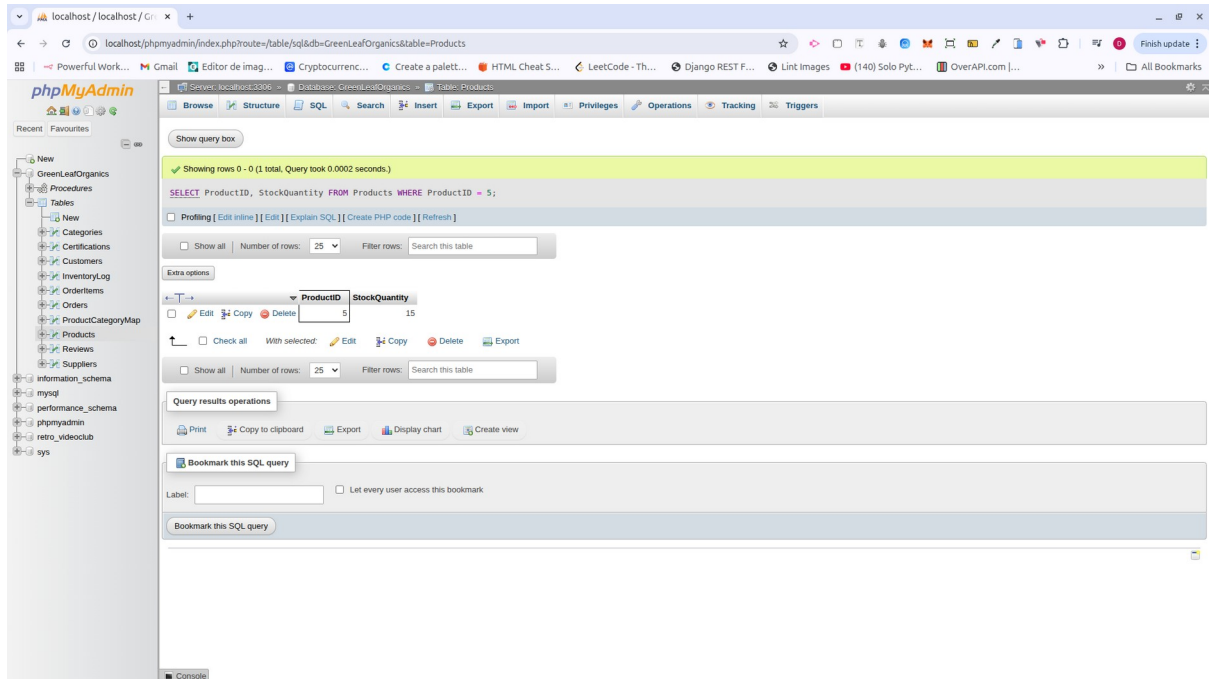


Figure 11. Stock level unchanged after the failed transaction confirms the rollback worked.

The decision to commit or roll back is therefore not arbitrary. I commit when (a) the customer exists, (b) the product exists, (c) the requested quantity is available in stock, and (d) all four writes (Orders insert, OrderItems insert, Products update, Orders update) succeed without raising an exception. If any of those conditions are false, I roll back, because committing a half-done order would leave the database in an inconsistent state where stock counts or order totals no longer match reality.

5. Triggers and Auditing

A trigger is piece of SQL code which is automatically executed when some event occurs on a table. The event can be an INSERT, UPDATE or DELETE, and occurs BEFORE or AFTER the triggering event [7]. Triggers have a major advantage when it comes to auditing; they cannot be avoided. As long as the desired change originates through the database itself, the trigger fires REGARDLESS of whether the change originated from a stored procedure, a hand-typed UPDATE statement or an admin application.

I created one trigger, which I called **tr_AfterStockUpdate**. It fires AFTER UPDATE on Products. Within the trigger body, I check to see if the **StockQuantity** column actually changed value (by comparing OLD.StockQuantity to NEW.StockQuantity). If it did change, I calculate the quantity changed and determine whether it is positive or negative. If the NEW value is lower than the OLD one, then the change is a Sale. Otherwise it must be a Restock. I then INSERT a row into **InventoryLog** with the ProductID, change amount, type and user account that made the change.

I used AFTER UPDATE rather than BEFORE UPDATE because I only care to log changes that successfully happened. If an UPDATE were to violate a CHECK constraint, a BEFORE trigger would execute but the change would not actually occur, leading to misleading information in the log. With AFTER, the trigger only executes if the row was successfully updated.

Figure 12 shows InventoryLog after I executed sp_PlaceOrder a few times. There is a single row created in the table for each stock change, showing the time it happened, the ProductID affected, the amount of the change (negative if it was a sale) and the user account that caused it. Notice how the log was filled out automatically; I never INSERTed a row into it myself. That is the beauty of a trigger in this situation: the auditing is forced by the database itself. Even a developer who forgot to log his stock changes into the application would be unable to leave a change unlogged.

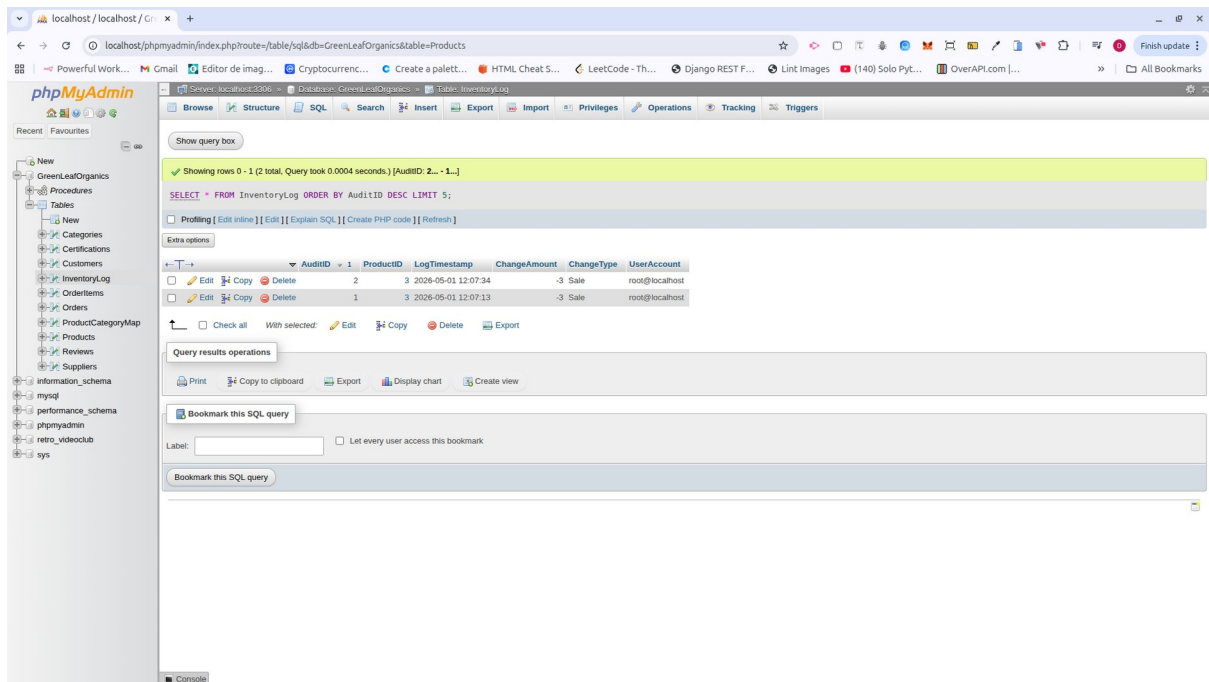


Figure 12. InventoryLog rows automatically generated by tr_AfterStockUpdate.

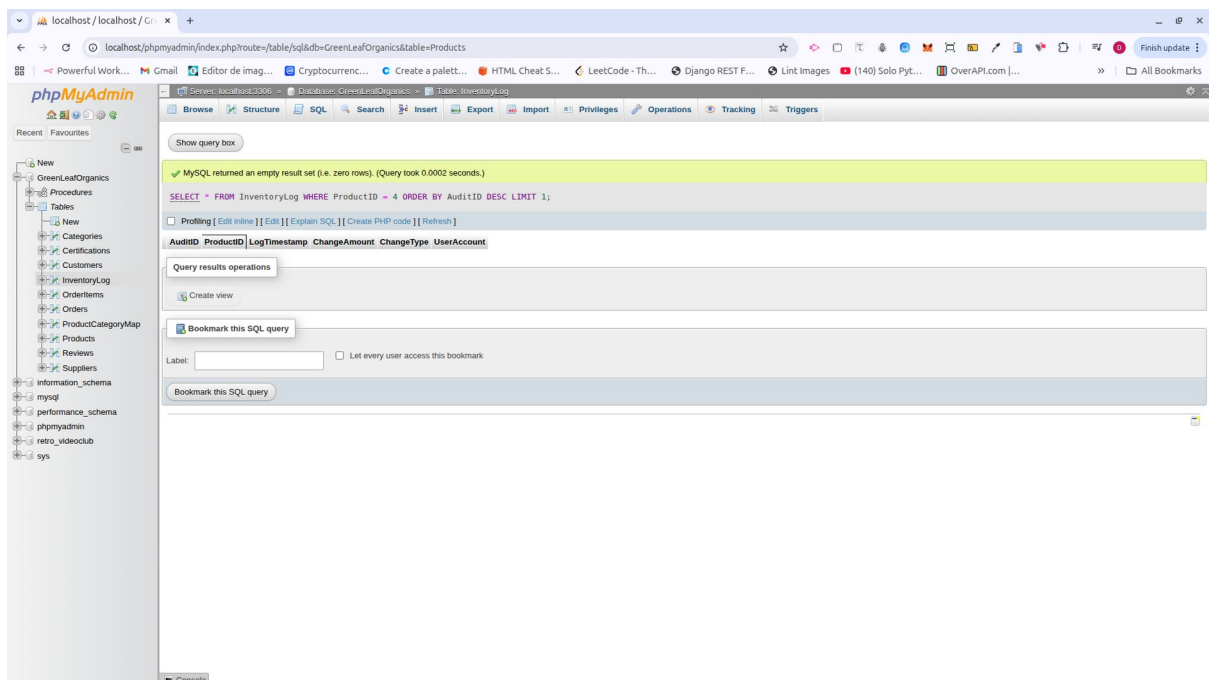


Figure 13. Detailed view of InventoryLog showing each stock movement classified as a Sale or Restock.

The OLD and NEW pseudo rows inside an UPDATE trigger are useful because they let you compare the value before and after the change in the same statement. Without them I would have to read the table twice or do the calculation in application code, both of which are slower and less reliable.

6. Performance Tuning

Indexes allow SELECT queries to be sped up dramatically by providing an ordered data structure (usually a B-tree) that can be traversed quickly instead of scanning every row in the table [8]. Without an index looking up all the orders placed by a given customer in a table with a million rows involves reading all million rows. With an index on CustomerID it becomes similar to looking up a word in the index of a book.

That said, indexes come at a cost. Every index must be updated whenever rows are inserted, updated or deleted. Indexes speed up read operations but slow down writes. Writes are less common for an online shop though (every pageview is a read; only the occasional order is a write), so it's usually worth it to add carefully chosen secondary indexes. MySQL also adds an index automatically on every primary key and every foreign key column, so joins between related tables already get a free speed boost just from the foreign key constraint [5].

In addition to those implicit indexes that MySQL provides automatically, I decided to add five secondary indexes on columns that I expect to query frequently:

idx_products_supplier on Products(SupplierID)

The shop will frequently show "all products from this supplier", such as on a supplier-detail page or admin reports. MySQL's foreign key constraint automatically adds this index for us when using InnoDB, but I include it in the script anyway as an explicitly named index for clarity.

idx_orders_customer on Orders(CustomerID)

When a logged-in customer views their order history, the website needs to run `SELECT * FROM Orders WHERE CustomerID = ?`. Without an index this query would become slow as the order table grows. MySQL again provides this automatically because of the foreign key constraint, but it is worthwhile to name the index so its intent is clear.

idx_orders_date on Orders(OrderDate)

Running management reports such as "all orders this week" or "sales for April" will require filtering on OrderDate. Searching and sorting by date without an index will force MySQL to scan the entire Orders table, becoming slower as the table grows.

idx_reviews_product on Reviews(ProductID)

Every product view includes customer reviews for that product, which means a `SELECT WHERE ProductID = ?`. MySQL's foreign key constraint covers us here too, but experience shows that reviews grow much faster than products so this index will have the most benefit as the site scales.

idx_products_name on Products(ProductName)

Site visitors will search for products using fancy filters like `SELECT * FROM Products WHERE ProductName LIKE 'organic%'`. A B-tree index can't do much to help searches with a leading wildcard (e.g. '%kale'), but it can speed up queries that have a prefix to search for, which will be most product browsing searches.

7. Conclusion and Reflection

The finished schema addresses all requirements from the GreenLeaf spec. Suppliers, Products and Categories are handled via their own tables, with appropriate cardinalities, plus a junction table to handle their many-to-many relationship. Customers, Orders and Order Items allow customers to register and place orders. Reviews allows customers to provide feedback on purchased products. Stored procedures are used for registration and order placement with appropriate transaction handling. A trigger keeps an audit trail of all stock movements without application code having to explicitly write to the audit table.

I found a couple of concepts difficult at first. Determining the appropriate ON DELETE rules took me some time. My initial reaction was to set everything to CASCADE, as it was easy, but I quickly realized that if you CASCADE'd deletes from Customers to Orders, you would lose all history of sales made. Changing most relationships where history is downstream from RESTRICT felt safer. The downside is there are some deletions which require additional manual cleanup, however for an actual storefront that is a small price to pay rather than losing data.

Another piece I had to noodle on was the SELECT ... FOR UPDATE lock in sp_PlaceOrder. I had studied transactions before but actually implementing two orders that either both commit or one waits was some work for me [2]. Without the FOR UPDATE clause, two separate transactions could read the same stock level, both validate that there is enough available, and then both decrement the stock level resulting in a lost update. The row level lock causes the second transaction to wait.

8. References (Part A)

- [1] GeeksforGeeks, "ACID Properties in DBMS," 2026. [Online]. Available: <https://www.geeksforgeeks.org/dbms/acid-properties-in-dbms/>. [Accessed: May 1, 2026].
- [2] GeeksforGeeks, "Transaction in DBMS," 2025. [Online]. Available: <https://www.geeksforgeeks.org/dbms/transaction-in-dbms/>. [Accessed: May 1, 2026].
- [3] DataCamp, "What is Third Normal Form (3NF)? A Beginner-Friendly Guide," 2024. [Online]. Available: <https://www.datacamp.com/tutorial/third-normal-form>. [Accessed: May 1, 2026].
- [4] IBM, "What Is Database Normalization?" 2026. [Online]. Available: <https://www.ibm.com/think/topics/database-normalization>. [Accessed: May 1, 2026].
- [5] Oracle, "MySQL 8.4 Reference Manual: FOREIGN KEY Constraints," 2024. [Online]. Available: <https://dev.mysql.com/doc/refman/8.4/en/create-table-foreign-keys.html>. [Accessed: May 1, 2026].
- [6] W3Schools, "MySQL Stored Procedures," 2026. [Online]. Available: https://www.w3schools.com/mysql/mysql_stored_procedures.asp. [Accessed: May 1, 2026].
- [7] Oracle, "MySQL 8.4 Reference Manual: Trigger Syntax and Examples," 2024. [Online]. Available: <https://dev.mysql.com/doc/refman/8.4/en/trigger-syntax.html>. [Accessed: May 1, 2026].
- [8] GeeksforGeeks, "SQL Indexes," 2026. [Online]. Available: <https://www.geeksforgeeks.org/sql/sql-indexes/>. [Accessed: May 1, 2026].

PART B

Retro VideoClub

SQL Injection: Threats and Defences



SQL Injection: Threats and Defences in the Retro VideoClub Application

Module: Databases and Application Development (UWTSD Level 5)

Application audited: Retro VideoClub — PHP / PDO / MySQL game-rental system

1. Introduction

SQL injection (CWE-89) remains one of the most prevalent web application vulnerabilities, sitting at position three of the OWASP Top 10 2021 with 274,228 recorded occurrences across tested applications and a maximum incidence rate of 19% [1]. It occurs when untrusted input is concatenated into a dynamically constructed query, allowing an attacker to alter the query's intent [3]. This report analyses my Retro VideoClub PHP application to demonstrate the threat and the defences I implemented, anchored on the principle of *"keeping data separate from commands and queries"* [1].

2. Why it is a problem

SQL injection threatens four core security properties: **confidentiality** through data exposure, **authentication** through login bypass, **authorisation** through privilege escalation, and **integrity** through unauthorised modification or deletion of records [3]. The consequences are not theoretical. In October 2015, British telecoms provider TalkTalk suffered a SQL injection attack that exposed personal details of 156,959 customers, including the bank account numbers and sort codes of 15,656 of them, and resulted in the Information Commissioner's then-largest fine of £400,000 [2]. The ICO investigation found that two earlier injection attacks had gone undetected, and that the underlying database flaw had a publicly available patch more than three years old, framing SQL injection as a well-understood, preventable risk [2].

3. Vulnerable patterns

SQL injection arises because *"SQL makes no real distinction between the control and data planes"* [3]: a metacharacter in user input can introduce new commands into the query. Had I implemented the search feature in search.php unsafely, the code might have looked like:

```
// HYPOTHETICAL VULNERABLE VERSION — NOT in my code
$searchTerm = $_GET['search'];
$sql = "SELECT * FROM GAMES WHERE title LIKE '%" . $searchTerm . "%'";
$result = $pdo->query($sql);
```

An attacker submitting `' OR '1'='1` would force the WHERE clause to always evaluate true, equivalent to `SELECT * FROM GAMES`, returning every row [3]. A more destructive payload such as `'); DROP TABLE GAMES; --` could terminate the original statement and execute a destructive command, comment-out the rest, and bypass any trailing logic.

4. Defences implemented

My application avoids these patterns by using PDO **prepared statements with named placeholders**, which the PHP manual states *"helps to prevent SQL injection attacks by eliminating the need to manually quote and escape"* parameters [4]. The connection in config/Database.php (lines 17–23) sets ATTR_EMULATE_PREPARES = false to enforce real server-side prepared statements and ERRMODE_EXCEPTION for safe error propagation. Every query that touches user input, including GameService::searchGames() (lines 78–96), UserService::login() (lines 132–137), and RentalService::checkoutGame() (lines 44–53), binds values via :name placeholders rather than concatenation [4]. Defence-in-depth measures include (int) casts on all \$_GET / \$_POST ID parameters, trim() on free-text fields, htmlspecialchars() on every echoed value to prevent reflected XSS, CSRF tokens on every state-changing form, and transactional wrapping (beginTransaction / commit / rollBack) around multi-step rental operations in RentalService so that a failure mid-transaction cannot leave the database in a partial state.

5. Why these defences work — live demonstration

Prepared statements force the developer to define the SQL template first and bind parameters separately, so *"the database will always distinguish between code and data, regardless of what user input is supplied"* [5]. Submitting the canonical payload ' OR '1'='1 into the search box on search.php returns **zero results**: MySQL searches for game titles literally containing the characters ' OR '1'='1, none of which exist. This mirrors OWASP's textbook example, in which input tom' or '1'='1 is treated as a literal username rather than executable SQL [5]. The same principle protects the login form: admin' OR '1'='1' - - is matched literally against the username column and fails authentication, with password_verify() against a bcrypt hash as a final safeguard.

6. Conclusion

By layering parameterised queries, strict type casting, output escaping, CSRF tokens, and transactional integrity, the Retro VideoClub application keeps untrusted input firmly within the data plane. The defences are neither novel nor complex, yet, as the TalkTalk breach demonstrated, they remain underused in production systems. Adopting them by default is the most reliable barrier to SQL injection.

7. References (Part B)

[1] OWASP, A03:2021 – *Injection*, OWASP Top 10 2021. Available: https://owasp.org/Top10/A03_2021-Injection/

[2] Information Commissioner's Office, *TalkTalk cyber attack — how the ICO's investigation unfolded*, ICO, 2019. Available: <https://ico.org.uk/about-the-ico/media-centre/news-and-blogs/2019/10/talktalk-cyber-attack-how-the-ico-s-investigation-unfolded/>

[3] OWASP, *SQL Injection*, OWASP Community. Available:
https://owasp.org/www-community/attacks/SQL_Injection

[4] The PHP Group, *PDO::prepare*, PHP Manual. Available:
<https://www.php.net/manual/en/pdo.prepare.php>

[5] OWASP, *SQL Injection Prevention Cheat Sheet*. Available:
https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html