Adaptive Horizons Tech Solutions B.V.

Comprehensive Manual: AHTS Database Schema Version 1.0.0

License:

Blueprint for Business: Understanding This Schema

Welcome to the foundational blueprint for our application's data! This SQL schema, currently at **Version 1.0.0**, serves as the backbone for managing crucial business information and processes.

What This Schema IS:

This schema is designed as the data structure for a robust **Customer Relationship Management (CRM) system with integrated foundational Enterprise Resource Planning (ERP) capabilities.** At its core, it provides the framework to:

- **Manage Customer Interactions:** From initial leads and opportunities (leads, tenders) through to sales (quotes_customer, orders) and post-sales support (support_tickets).

- **Centralize Key Business Entities:** Organize information about companies, contacts, and products.
- **Handle Core Sales & Procurement Cycles:** Support quoting, ordering, and invoicing for both customer-facing sales and supplier-side purchasing (purchase_orders, invoices_supplier).
- **Basic Inventory & Product Tracking:** Provide mechanisms for managing stock_levels, warehouses, and serial_numbers for products.
- **Track User Actions & Responsibilities:** Log which system users are responsible for key records and transactions.
- **Ensure Data Integrity:** Employ relational constraints, lookup tables for consistency, and database triggers for specific business rules.
- **Maintain Version Control:** Incorporate a db_schema_version table to track the evolution of the schema itself.

Essentially, it's a solid foundation for applications that need to manage the lifecycle of customer engagement, sales, basic purchasing, and product information.

What This Schema IS NOT (at Version 1.0.0):

While comprehensive in its CRM and foundational ERP aspects, this initial version does **not** yet include in-depth modules for:

- **Advanced Financial Accounting:** It lacks a general ledger, chart of accounts, or complex financial reporting structures beyond basic invoicing and payments.
- **Detailed Manufacturing Processes:** Features like multi-level Bills of Materials (BOMs), work orders, or Material Requirements Planning (MRP) are outside the current scope.
- **Comprehensive Human Resources (HR) / Payroll:** These functionalities are not part of this version.
- **Advanced Warehouse Management (WMS):** While basic inventory is covered, intricate WMS features like advanced bin management, complex picking strategies, or detailed logistics are not included.
- **Project Management or Advanced Supply Chain Management.**

This schema is intended to be a strong starting point, extensible for future enhancements and module additions. Its current focus is on delivering core CRM functionalities with essential ERP support.

This manual will guide you through the detailed structure, relationships, and considerations for interacting with this database schema.

Contents

1. Introduction

1.1 Purpose

This document describes the MySQL database schema version 1.0.0. This schema is designed to support a comprehensive Enterprise Resource Planning (ERP) or Customer Relationship Management (CRM) system, encompassing entities such as companies, contacts, products, sales processes (leads, tenders, quotes, orders), procurement, inventory, billing, and support.

1.2 Version

The schema version described in this manual is **1.0.0 - Initial Structure**.

1.3 Core Design Principles

- **Relational Integrity:** Foreign keys are used extensively to maintain data consistency. ON DELETE and ON UPDATE clauses are defined (mostly CASCADE, SET NULL, or RESTRICT) to manage referential integrity.
- **Soft Deletes:** Core entities (users, companies, departments, products, addresses, contacts) implement a soft delete pattern using is_deleted (BOOLEAN) and deleted_at (TIMESTAMP) columns. This allows data to be marked as deleted without physically removing it, preserving history.
- **User Responsibility Tracking:** Key transactional tables (e.g., leads, tenders, quotes_customer, orders, purchase_orders, invoices_customer, invoices _supplier) include fields to track the system user responsible for creating or managing the record (e.g., assigned_user_id, issued_by_user_id, created_by_user_id).
- **Generated Columns for Active Uniqueness:** Some tables use generated columns (e.g., active_email in users) to enforce unique constraints only on active, non-deleted records.
- **Centralized Entities:** Tables like companies, contacts, and addresses serve as central repositories, linked to various roles or transactions.
- **Status Lookup Tables:** Dedicated tables (e.g., order_statuses, lead_statuses) are used to manage predefined statuses for various entities, promoting consistency and allowing for descriptions and sequencing.
- **Database Triggers for Integrity:** MySQL triggers are implemented to enforce complex business rules that cannot be handled by simple constraints, such as ensuring address-department-company consistency and preventing multiple primary contacts/addresses.
- **Queryable Schema Version:** A db_schema_version table is included to programmatically track the applied schema version and its checksum.

2. Schema Setup & Configuration

2.1 Initial SQL Settings

The script begins by setting session variables for robust execution:

- SET FOREIGN_KEY_CHECKS=0;: Temporarily disables foreign key checks to allow tables to be dropped and created in any order without immediate referential integrity errors. This is re-enabled at the end.
- SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO,STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION";:
  - NO_AUTO_VALUE_ON_ZERO: Affects AUTO_INCREMENT columns; 0 will not be used for AUTO_INCREMENT.
  - STRICT_TRANS_TABLES: Enables strict mode for transactional storage engines, causing errors for invalid data values rather than warnings and coercions.
  - NO_ENGINE_SUBSTITUTION: Prevents automatic substitution of the default storage engine if the requested engine is unavailable.
- SET time_zone = "+00:00";: Sets the session time zone to UTC, promoting consistency in timestamp storage.

2.2 Drop Order

The script systematically drops existing database objects to ensure a clean environment. The order is important to avoid foreign key constraint violations during the drop process:

1. Views
2. Triggers
3. Junction tables and tables that are frequently depended upon by others.
4. Core entity tables.
5. Status/Lookup tables (including db_schema_version).

3. Reference Data & Lookup Tables

These tables store relatively static data or predefined lists used by other tables, often for dropdowns or categorizations.

3.1 countries

- **Purpose:** Stores a list of countries.
- **Key Columns:**
  - country_id (PK): Unique identifier for the country.
  - country_name (UNIQUE): Full name of the country.
  - country_code_iso2 (UNIQUE): Standard 2-letter ISO country code.
- **Used By:** addresses, leads, warehouses.

3.2 currencies

- **Purpose:** Stores a list of currencies and their properties.
- **Key Columns:**
  - currency_code (PK): Standard 3-letter currency code (e.g., USD, EUR).
  - currency_name: Full name of the currency.
  - symbol: Currency symbol (e.g., $, €).
- **Used By:** Many transactional tables like products, orders, invoices, etc.

3.3 market_segments

- **Purpose:** Defines market segments for classifying companies.
- **Key Columns:**
  - segment_id (PK): Unique identifier for the market segment.
  - segment_name (UNIQUE): Name of the segment (e.g., Enterprise, SMB).
- **Used By:** company_market_segments (junction table).

3.4 expertise_tags

- **Purpose:** Defines areas of expertise for contacts.
- **Key Columns:**
  - tag_id (PK): Unique identifier for the tag.
  - tag_name (UNIQUE): Name of the expertise tag (e.g., Technical Support).
- **Used By:** contact_expertise (junction table).

3.5 roles

- **Purpose:** Defines user roles within the system (e.g., Administrator, Sales).
- **Key Columns:**
  - role_id (PK): Unique identifier for the role.
  - role_name (UNIQUE): Name of the role.
- **Used By:** user_roles (junction table).

3.6 permissions

- **Purpose:** Defines specific permissions that can be granted (e.g., manage_all, manage_sales).
- **Key Columns:**
  - permission_id (PK): Unique identifier for the permission.
  - permission_name (UNIQUE): Name of the permission.
- **Used By:** role_permissions (junction table).

3.7 product_categories

- **Purpose:** Stores categories for products, allowing for a hierarchical structure.
- **Key Columns:**
  - category_id (PK): Unique identifier for the category.
  - category_name (UNIQUE): Name of the product category.
  - parent_category_id (FK to product_categories.category_id): Allows for sub-categories.
- **Used By:** products.

3.8 warehouses

- **Purpose:** Defines physical or logical warehouse locations.
- **Key Columns:**
  - warehouse_id (PK): Unique identifier for the warehouse.
  - warehouse_name (UNIQUE): Name of the warehouse.
  - country_id (FK to countries.country_id): Country where the warehouse is located.
- **Used By:** stock_levels, orders (preferred warehouse), purchase_orders (deliver to).

3.9 db_schema_version

- **Purpose:** Tracks the applied versions of this database schema itself, allowing for programmatic version checking.

- **Key Columns:**
  - version_id (PK): Auto-incrementing ID for the version record.
  - version_tag (UNIQUE): The human-readable version string (e.g., "1.0.0").
  - description: A brief text description of the changes in this version.
  - applied_by: User or script that applied this schema version.
  - applied_at (TIMESTAMP): When this schema version was applied.
  - script_checksum_algo: The algorithm (e.g., 'SHA256', 'MD5') used to calculate the checksum.
  - script_checksum: The checksum of the SQL script file (with placeholders) that defined this version, used for verification.
- **Important Note:** The script_checksum_algo and script_checksum values (specifically the placeholders _YOUR_SCRIPT_CHECKSUM_ALGO_HERE_ and %%SCRIPT_CHECKSUM_PLACEHOLDER%%) in the INSERT statement must be replaced with actual values before running the script. The checksum should be calculated on the script file *while it still contains these placeholders*.

3.10 Status Tables (General)

A series of tables are used to define enumerated status values for different entities. This approach provides flexibility to add descriptions or other metadata to statuses, and to manage them as data rather than hardcoding them in application logic or ENUM types directly in transactional tables (though ENUM is used in addresses.address_type for simplicity).

- **Structure:** Typically include a status_id (PK, TINYINT UNSIGNED AUTO_INCREMENT) and status_name (VARCHAR, UNIQUE). Some might include a description or sequence for ordering.
- **Examples:**
  - customer_account_statuses: (e.g., Active, Inactive)
  - lead_statuses: (e.g., New, Contacted, Qualified)
  - tender_statuses: (e.g., Qualification, Proposal)
  - quote_statuses: (e.g., Draft, Sent, Accepted) - Used for both customer and supplier quotes.
  - order_statuses: (e.g., Pending, Processing, Shipped)
  - payment_statuses: (e.g., Unpaid, Paid)
  - shipment_statuses: (e.g., Preparing, Shipped, Delivered)
  - po_statuses (Purchase Order Statuses): (e.g., Draft, Sent, Fully Received)
  - invoice_statuses (Customer Invoice Statuses): (e.g., Draft, Sent, Paid)

- supplier_invoice_statuses: (e.g., Received, Approved, Paid)
- serial_number_statuses: (e.g., In Stock, Allocated, Shipped)
- rma_statuses (Return Merchandise Authorization Statuses): (e.g., Requested, Approved)
- rma_item_conditions: (e.g., New, Used, Damaged)
- rma_item_actions: (e.g., Refund, Replacement, Repair)
- ticket_statuses (Support Ticket Statuses): (e.g., Open, In Progress, Resolved)
- ticket_priorities: (e.g., Low, Medium, High) - Includes a level for ordering.

3.11 partner_types

- **Purpose:** Defines different types of partnerships (e.g., Reseller, Technology Partner).
- **Key Columns:**
  - type_id (PK): Unique identifier.
  - type_name (UNIQUE): Name of the partner type.
- **Used By:** partners.

4. Core Entity Tables

These tables represent the fundamental entities in the system.

4.1 users

- **Purpose:** Stores information about internal system users (employees).
- **Key Columns:**
  - user_id (PK): Unique identifier.
  - email (NOT NULL): User's email, raw value.
  - password_hash (NOT NULL): Hashed password for security.
  - is_active (BOOLEAN): Whether the user account is active.
  - is_deleted, deleted_at: For soft deletes.
  - active_email (GENERATED, UNIQUE): Stores email if not deleted, NULL otherwise. Used for unique constraint on active user emails.
- **Relationships:**
  - Linked to user_roles to define permissions.
  - Referenced by many tables for user responsibility tracking (e.g., customers.account_manager_id, orders.created_by_user_id).
- **User Tracking:** This table *is* the source for user identity.

4.2 companies

- **Purpose:** Central table for all external organizations (customers, suppliers, partners, etc.).
- **Key Columns:**
  - company_id (PK): Unique identifier.
  - company_name (NOT NULL): Name of the company.
  - vat_number, website, default_payment_terms: Additional company details.
  - preferred_currency_code (FK to currencies.currency_code): Default currency for the company.
  - is_deleted, deleted_at: For soft deletes.
  - active_company_name (GENERATED, UNIQUE): Stores company_name if not deleted, NULL otherwise. For unique constraint on active company names.
- **Relationships:**
  - Parent to departments, addresses.
  - Linked to role tables (customers, suppliers, partners) via company_id.

- Linked to company_market_segments, company_contacts.
- Referenced in transactional tables like orders, invoices_customer, etc.

4.3 departments

- **Purpose:** Represents departments within a company.
- **Key Columns:**
  - department_id (PK): Unique identifier.
  - company_id (FK to companies.company_id, NOT NULL): The company this department belongs to.
  - department_name (NOT NULL): Name of the department.
  - parent_department_id (FK to departments.department_id): For hierarchical department structures.
  - default_contact_id (FK to contacts.contact_id): A primary contact for the department.
  - is_deleted, deleted_at: For soft deletes.
- **Relationships:**
  - Child of companies.
  - Can be self-referencing for hierarchy.
  - Linked to addresses (as owner_department_id), department_contacts.

4.4 products

- **Purpose:** Stores details about products or services offered or procured.
- **Key Columns:**
  - product_id (PK): Unique identifier.
  - sku (NOT NULL): Stock Keeping Unit.
  - product_name (NOT NULL): Name of the product.
  - category_id (FK to product_categories.category_id): Product category.
  - unit_price, currency_code (FK to currencies.currency_code): Default selling price.
  - default_cost_price, default_cost_currency_code (FK to currencies.currency_code): Default cost.
  - track_serial_number (BOOLEAN): Indicates if individual serial numbers should be tracked.
  - is_active (BOOLEAN): Whether the product is currently active for sale/purchase.

- is_deleted, deleted_at: For soft deletes.
- active_sku (GENERATED, UNIQUE): Stores sku if not deleted, NULL otherwise. For unique constraint on active SKUs.

- **Relationships:**
  - Linked to product_categories.
  - Referenced by stock_levels, quote_customer_items, order_items, quote_supplier_items, purchase_order_items, serial_numbers, rma_items, support_tickets.

4.5 addresses

- **Purpose:** Stores physical or mailing addresses.
- **Key Columns:**
  - address_id (PK): Unique identifier.
  - company_id (FK to companies.company_id, NOT NULL): The company this address primarily belongs to.
  - owner_department_id (FK to departments.department_id): Optionally, the specific department using this address. A trigger ensures this department belongs to company_id.
  - address_type (ENUM): Type of address (Billing, Shipping, Office, Other).
  - is_primary_billing_for_owner, is_primary_shipping_for_owner (BOOLEAN): Flags to indicate primary addresses. Triggers enforce uniqueness of these flags per owner (company or department).
  - country_id (FK to countries.country_id): Country of the address.
  - is_deleted, deleted_at: For soft deletes.
- **Relationships:**
  - Belongs to a companies.
  - Optionally linked to a departments.
  - Referenced by contacts, quotes_customer (billing/shipping), orders (billing/shipping), purchase_orders (ship from).

4.6 contacts

- **Purpose:** Central table for contact persons.
- **Key Columns:**
  - contact_id (PK): Unique identifier.

  - first_name (NOT NULL), last_name, job_title.
  - email: Contact's email, raw value.
  - address_id (FK to addresses.address_id): Optional link to a specific address for the contact.
  - is_deleted, deleted_at: For soft deletes.
  - active_email (GENERATED, UNIQUE): Stores email if not deleted and not NULL, otherwise NULL. For unique constraint on active, non-null emails.
- **Relationships:**
  - Can be linked to addresses.
  - Referenced by departments (default contact).
  - Linked to various roles/entities via junction tables: customer_contacts, supplier_contacts, partner_contacts, company_contacts, department_contacts.
  - Also linked to contact_expertise, tenders (primary contact), quotes_customer, orders, quotes_supplier, purchase_orders, support_tickets.

5. Company Role Tables

These tables define specific roles a companies entity can play. A company can be a customer, a supplier, a partner, or any combination.

5.1 customers

- **Purpose:** Designates a company as a customer and stores customer-specific information.
- **Key Columns:**
  - customer_id (PK): Unique identifier for the customer role instance.
  - company_id (FK to companies.company_id, UNIQUE, NOT NULL): Links to the central company record. Ensures a company can only be a customer once.
  - account_manager_id (FK to users.user_id): The internal user responsible for this customer account.
  - account_status_id (FK to customer_account_statuses.status_id, NOT NULL): Current status of the customer account.
- **Relationships:**
  - Extends companies.
  - Linked to customer_contacts.
  - Implied link to orders, invoices_customer, etc., through company_id.
- **User Tracking:** account_manager_id.

5.2 suppliers

- **Purpose:** Designates a company as a supplier.
- **Key Columns:**
  - supplier_id (PK): Unique identifier for the supplier role instance.
  - company_id (FK to companies.company_id, UNIQUE, NOT NULL): Links to the central company record.
- **Relationships:**
  - Extends companies.
  - Linked to supplier_contacts.
  - Implied link
    to quotes_supplier, purchase_orders, invoices_supplier through company_id.

5.3 partners

- **Purpose:** Designates a company as a partner and stores partnership details.

- **Key Columns:**
  - partner_id (PK): Unique identifier for the partner role instance.
  - company_id (FK to companies.company_id, UNIQUE, NOT NULL): Links to the central company record.
  - partner_type_id (FK to partner_types.type_id): Type of partnership.
- **Relationships:**
  - Extends companies.
  - Linked to partner_contacts.

6. Junction & Dependent Tables

These tables primarily establish many-to-many relationships or store data dependent on other primary entities.

6.1 user_roles

- **Purpose:** Many-to-many relationship between users and roles.
- **Key Columns:** user_id (FK), role_id (FK). Primary Key is (user_id, role_id).

6.2 role_permissions

- **Purpose:** Many-to-many relationship between roles and permissions.
- **Key Columns:** role_id (FK), permission_id (FK). Primary Key is (role_id, permission_id).

6.3 company_market_segments

- **Purpose:** Many-to-many relationship between companies and market_segments.
- **Key Columns:** company_id (FK), segment_id (FK). Primary Key is (company_id, segment_id).

6.4 Contact Linkage Tables

These tables link contacts to specific roles or entities. Triggers enforce uniqueness for primary contacts.

- **customer_contacts**: Links customers (via customer_id) and contacts (via contact_id).
  - is_primary_for_customer (BOOLEAN): Marks the primary contact for the customer role.
- **supplier_contacts**: Links suppliers (via supplier_id) and contacts (via contact_id).
  - is_primary_for_supplier (BOOLEAN): Marks the primary contact for the supplier role.
- **partner_contacts**: Links partners (via partner_id) and contacts (via contact_id).
  - is_primary_for_partner (BOOLEAN): Marks the primary contact for the partner role.
- **company_contacts**: Links companies directly to contacts for general company contacts.
- **department_contacts**: Links departments to contacts.

6.5 contact_expertise

- **Purpose:** Many-to-many relationship between contacts and expertise_tags.
- **Key Columns:** contact_id (FK), tag_id (FK). Primary Key is (contact_id, tag_id).

6.6 stock_levels

- **Purpose:** Tracks inventory levels of products in specific warehouses.
- **Key Columns:**

- stock_level_id (PK).
- product_id (FK to products.product_id, NOT NULL).
- warehouse_id (FK to warehouses.warehouse_id, NOT NULL).
- quantity_on_hand, quantity_reserved, quantity_on_order: Core quantity fields.
- quantity_available, quantity_projected: Generated columns for calculated stock figures.
- Unique Key on (product_id, warehouse_id).

- **Relationships:** Links products and warehouses. Referenced by serial_numbers.

6.7 exchange_rates

- **Purpose:** Stores historical exchange rates between currencies.
- **Key Columns:**
  - rate_id (PK).
  - from_currency_code (FK to currencies.currency_code).
  - to_currency_code (FK to currencies.currency_code).
  - rate (DECIMAL): The exchange rate.
  - effective_date (DATE): The date from which this rate is effective.
  - Unique Key on (from_currency_code, to_currency_code, effective_date).

7. CRM & Sales Tables

These tables manage the sales pipeline and customer order processing.

7.1 leads

- **Purpose:** Captures potential sales opportunities or prospective customers.
- **Key Columns:**
  - lead_id (PK).
  - lead_status_id (FK to lead_statuses.status_id, NOT NULL): Current status of the lead.
  - assigned_user_id (FK to users.user_id): The user responsible for this lead.
  - country_id (FK to countries.country_id).
  - converted_company_id (FK to companies.company_id, UNIQUE): If converted, the resulting company.
  - converted_tender_id (FK to tenders.tender_id, UNIQUE): If converted, the resulting tender.
- **Relationships:** Can be converted into companies and/or tenders.
- **User Tracking:** assigned_user_id.

7.2 tenders

- **Purpose:** Represents a formal sales opportunity or response to a request for proposal (RFP).
- **Key Columns:**
  - tender_id (PK).
  - company_id (FK to companies.company_id): The prospective or existing customer company.
  - lead_id (FK to leads.lead_id): The lead from which this tender originated (if any).
  - primary_contact_id (FK to contacts.contact_id): Main contact at the customer for this tender.
  - assigned_user_id (FK to users.user_id): The user responsible for this tender.
  - tender_status_id (FK to tender_statuses.status_id, NOT NULL): Current status.
  - currency_code (FK to currencies.currency_code).
- **Relationships:** Can originate from a leads. Can lead to quotes_customer.
- **User Tracking:** assigned_user_id.

7.3 quotes_customer

- **Purpose:** Stores customer quotations.

- **Key Columns:**
  - quote_id (PK).
  - quote_number (UNIQUE, NOT NULL): Unique identifier for the quote document.
  - company_id (FK to companies.company_id, NOT NULL): The customer company.
  - customer_contact_id (FK to contacts.contact_id).
  - billing_address_id (FK to addresses.address_id), shipping_address_id (FK to addresses.address_id).
  - tender_id (FK to tenders.tender_id): Tender this quote relates to.
  - issued_by_user_id (FK to users.user_id): User who issued the quote.
  - quote_status_id (FK to quote_statuses.status_id, NOT NULL).
  - currency_code (FK to currencies.currency_code, NOT NULL).
  - related_order_id (FK to orders.order_id, UNIQUE): If converted to an order.
- **Relationships:** Linked to companies, contacts, addresses, tenders, users. Parent to quote_customer_items. Can be converted to an orders.
- **User Tracking:** issued_by_user_id.

7.4 quote_customer_items

- **Purpose:** Line items for a customer quote.
- **Key Columns:**
  - quote_item_id (PK).
  - quote_id (FK to quotes_customer.quote_id, NOT NULL).
  - product_id (FK to products.product_id, NOT NULL).
  - quantity, unit_price_quoted, line_total.
  - Unique key on (quote_id, product_id).
- **Relationships:** Child of quotes_customer. Links to products.

7.5 orders

- **Purpose:** Customer sales orders.
- **Key Columns:**
  - order_id (PK).
  - company_id (FK to companies.company_id, NOT NULL): The customer company.
  - related_quote_id (FK to quotes_customer.quote_id, UNIQUE): Quote this order originated from.

  - customer_contact_id (FK to contacts.contact_id).
  - billing_address_id (FK to addresses.address_id), shipping_address_id (FK to addresses.address_id).
  - created_by_user_id (FK to users.user_id): User who created the order.
  - order_status_id (FK to order_statuses.status_id, NOT NULL).
  - payment_status_id (FK to payment_statuses.status_id, NOT NULL).
  - currency_code (FK to currencies.currency_code, NOT NULL).
  - preferred_warehouse_id (FK to warehouses.warehouse_id).
- **Relationships:** Linked to companies, quotes_customer, contacts, addresses, users, warehouses. Parent to order_items, shipments, invoices_customer.
- **User Tracking:** created_by_user_id.

7.6 order_items

- **Purpose:** Line items for a customer order.
- **Key Columns:**
  - order_item_id (PK).
  - order_id (FK to orders.order_id, NOT NULL).
  - product_id (FK to products.product_id, NOT NULL).
  - quantity, unit_price_at_order, line_total.
  - Unique key on (order_id, product_id).
- **Relationships:** Child of orders. Links to products. Referenced by shipment_items, rma_items.

8. Warehousing & Shipping Tables

8.1 shipments

- **Purpose:** Tracks shipments made against customer orders.
- **Key Columns:**
  - shipment_id (PK).
  - order_id (FK to orders.order_id, NOT NULL).
  - warehouse_id (FK to warehouses.warehouse_id): Warehouse shipped from.
  - shipment_status_id (FK to shipment_statuses.status_id, NOT NULL).
  - carrier, tracking_number.
- **Relationships:** Linked to orders, warehouses. Parent to shipment_items.
- **User Tracking:** None directly; responsibility implied via the associated orders.created_by_user_id.

8.2 shipment_items

- **Purpose:** Line items for a shipment, linking back to order items.
- **Key Columns:**
  - shipment_item_id (PK).
  - shipment_id (FK to shipments.shipment_id, NOT NULL).
  - order_item_id (FK to order_items.order_item_id, NOT NULL).
  - quantity_shipped.
  - Unique key on (shipment_id, order_item_id).
- **Relationships:** Child of shipments. Links to order_items. Referenced by serial_numbers.

9. Procurement Tables

These tables manage the process of purchasing goods or services from suppliers.

9.1 quotes_supplier

- **Purpose:** Stores quotations received from suppliers.
- **Key Columns:**
  - supplier_quote_id (PK).
  - company_id (FK to companies.company_id, NOT NULL): The supplier company.
  - supplier_contact_id (FK to contacts.contact_id).
  - recorded_by_user_id (FK to users.user_id): User who recorded this supplier quote.
  - quote_status_id (FK to quote_statuses.status_id, NOT NULL).
  - currency_code (FK to currencies.currency_code, NOT NULL).
- **Relationships:** Linked to companies, contacts, users. Parent to quote_supplier_items. Can lead to purchase_orders.
- **User Tracking:** recorded_by_user_id.

9.2 quote_supplier_items

- **Purpose:** Line items for a supplier quote.
- **Key Columns:**
  - supplier_quote_item_id (PK).
  - supplier_quote_id (FK to quotes_supplier.supplier_quote_id, NOT NULL).
  - product_id (FK to products.product_id, NOT NULL).
  - quantity, unit_cost, line_total.
  - Unique key on (supplier_quote_id, product_id).
- **Relationships:** Child of quotes_supplier. Links to products.

9.3 purchase_orders

- **Purpose:** Purchase orders issued to suppliers.
- **Key Columns:**
  - po_id (PK).
  - po_number (UNIQUE, NOT NULL): Unique purchase order document number.
  - company_id (FK to companies.company_id, NOT NULL): The supplier company.
  - supplier_contact_id (FK to contacts.contact_id).

  - placed_by_user_id (FK to users.user_id): User who placed the PO.
  - po_status_id (FK to po_statuses.status_id, NOT NULL).
  - currency_code (FK to currencies.currency_code, NOT NULL).
  - deliver_to_warehouse_id (FK to warehouses.warehouse_id).
  - related_supplier_quote_id (FK to quotes_supplier.supplier_quote_id).
- **Relationships:** Linked to companies, contacts, users, warehouses, quotes_supplier. Parent to purchase_order_items, invoices_supplier.
- **User Tracking:** placed_by_user_id.

9.4 purchase_order_items

- **Purpose:** Line items for a purchase order.
- **Key Columns:**
  - po_item_id (PK).
  - po_id (FK to purchase_orders.po_id, NOT NULL).
  - product_id (FK to products.product_id, NOT NULL).
  - quantity_ordered, unit_cost, line_total.
  - Unique key on (po_id, product_id).
- **Relationships:** Child of purchase_orders. Links to products. Referenced by serial_numbers.

10. Billing & Payments Tables

10.1 invoices_customer

- **Purpose:** Invoices issued to customers.
- **Key Columns:**
  - invoice_id (PK).
  - invoice_number (UNIQUE, NOT NULL): Unique invoice document number.
  - company_id (FK to companies.company_id, NOT NULL): The customer company.
  - order_id (FK to orders.order_id, NOT NULL): The order this invoice relates to.
  - created_by_user_id (FK to users.user_id): User who created/issued this invoice.
  - billing_address_id (FK to addresses.address_id).
  - invoice_status_id (FK to invoice_statuses.status_id, NOT NULL).
  - currency_code (FK to currencies.currency_code, NOT NULL).
  - balance_due (GENERATED): Calculated as total_amount - amount_paid.
- **Relationships:** Linked to companies, orders, users, addresses. Referenced by payments. Parent to invoice_customer_attachments.
- **User Tracking:** created_by_user_id.

10.2 invoices_supplier

- **Purpose:** Invoices received from suppliers.
- **Key Columns:**
  - supplier_invoice_id (PK).
  - company_id (FK to companies.company_id, NOT NULL): The supplier company.
  - po_id (FK to purchase_orders.po_id): The PO this invoice relates to.
  - entered_by_user_id (FK to users.user_id): User who entered this invoice.
  - supplier_invoice_number (NOT NULL): Supplier's invoice number. Unique per company.
  - invoice_status_id (FK to supplier_invoice_statuses.status_id, NOT NULL).
  - currency_code (FK to currencies.currency_code, NOT NULL).
  - balance_due (GENERATED): Calculated as total_amount - amount_paid.
  - Unique key on (company_id, supplier_invoice_number).

- **Relationships:** Linked to companies, purchase_orders, users. Referenced by payments. Parent to invoice_supplier_attachments.
- **User Tracking:** entered_by_user_id.

10.3 payments

- **Purpose:** Records payments made or received.
- **Key Columns:**
  - payment_id (PK).
  - customer_invoice_id (FK to invoices_customer.invoice_id): If for a customer invoice.
  - supplier_invoice_id (FK to invoices_supplier.supplier_invoice_id): If for a supplier invoice.
  - processed_by_user_id (FK to users.user_id): User who processed the payment.
  - currency_code (FK to currencies.currency_code, NOT NULL).
  - chk_payment_link (CHECK constraint): Ensures payment is linked to either a customer OR supplier invoice, but not both.
- **Relationships:** Links to invoices_customer or invoices_supplier, and users.
- **User Tracking:** processed_by_user_id.

11. Other Operational Tables

11.1 serial_numbers

- **Purpose:** Tracks individual serial numbers for products where products.track_serial_number is true.
- **Key Columns:**
  - serial_number_id (PK).
  - product_id (FK to products.product_id, NOT NULL).
  - serial_number (VARCHAR, NOT NULL). Unique per product.
  - status_id (FK to serial_number_statuses.status_id, NOT NULL).
  - current_stock_level_id (FK to stock_levels.stock_level_id): Current stock location if applicable.
  - po_item_id_received (FK): PO item through which it was received.
  - shipment_item_id_shipped (FK): Shipment item through which it was shipped.
  - rma_item_id_returned (FK): RMA item if returned.
  - Unique key on (product_id, serial_number).
- **Relationships:** Links to products, serial_number_statuses, stock_levels, purchase_order_items, shipment_items, rma_items.

11.2 rmas

- **Purpose:** Manages Return Merchandise Authorizations.
- **Key Columns:**
  - rma_id (PK).
  - rma_number (UNIQUE, NOT NULL).
  - company_id (FK to companies.company_id, NOT NULL): The customer company returning items.
  - order_id (FK to orders.order_id): Original order if known.
  - rma_status_id (FK to rma_statuses.status_id, NOT NULL).
  - approved_by_user_id (FK to users.user_id): User who approved the RMA.
- **Relationships:** Linked to companies, orders, users. Parent to rma_items, support_tickets.
- **User Tracking:** approved_by_user_id.

11.3 rma_items

- **Purpose:** Line items for an RMA.
- **Key Columns:**
  - rma_item_id (PK).
  - rma_id (FK to rmas.rma_id, NOT NULL).
  - order_item_id (FK to order_items.order_item_id): Original order item if applicable.
  - product_id (FK to products.product_id, NOT NULL).
  - serial_number_id (FK to serial_numbers.serial_number_id, UNIQUE): If a serialized item is returned.
  - condition_id (FK to rma_item_conditions.condition_id).
  - action_requested_id (FK to rma_item_actions.action_id), action_taken_id (FK to rma_item_actions.action_id).
- **Relationships:** Child of rmas. Links to order_items, products, serial_numbers, and RMA lookup tables.

11.4 support_tickets

- **Purpose:** Tracks customer support tickets.
- **Key Columns:**
  - ticket_id (PK).
  - ticket_number (UNIQUE, NOT NULL).
  - company_id (FK to companies.company_id, NOT NULL): Customer company.
  - contact_id (FK to contacts.contact_id): Customer contact.
  - ticket_status_id (FK to ticket_statuses.status_id, NOT NULL).
  - priority_id (FK to ticket_priorities.priority_id, NOT NULL).
  - assigned_user_id (FK to users.user_id): User assigned to the ticket.
  - related_order_id (FK), related_product_id (FK), related_rma_id (FK): Links to other relevant entities.
- **Relationships:** Linked to companies, contacts, users, and optionally orders, products, rmas.
- **User Tracking:** assigned_user_id.

11.5 attachments

- **Purpose:** Central table to store metadata about uploaded files.
- **Key Columns:**

  - attachment_id (PK).
  - file_name, file_path, file_type, file_size_bytes.
  - uploaded_by_user_id (FK to users.user_id): User who uploaded the file.
- **Relationships:** Referenced by various attachment junction tables.
- **User Tracking:** uploaded_by_user_id.

11.6 Attachment Junction Tables

These tables create many-to-many relationships between various entities and the attachments table.

- **Structure:** Typically entity_id (FK), attachment_id (FK). Primary Key is (entity_id, attachment_id).
- **Examples:**
  - lead_attachments
  - tender_attachments
  - quote_customer_attachments
  - purchase_order_attachments
  - invoice_customer_attachments
  - invoice_supplier_attachments

12. Database Triggers

Triggers are used to enforce data integrity and business rules automatically at the database level.

12.1 Address Consistency Triggers

- **addr_consistency_before_insert**:
  - **Event:** BEFORE INSERT ON addresses.
  - **Purpose:** If owner_department_id is provided, ensures that the department belongs to the company_id specified for the address.
- **addr_consistency_before_update**:
  - **Event:** BEFORE UPDATE ON addresses.
  - **Purpose:** If owner_department_id or company_id is changed and owner_department_id is set, ensures the updated department still belongs to the specified (or updated) company.

12.2 Single Primary Address Triggers

These triggers ensure that a company (if owner_department_id is NULL) or a department (if owner_department_id is set) has at most one primary billing address and at most one primary shipping address among its non-deleted addresses.

- **addr_enforce_single_primary_insert**:
  - **Event:** BEFORE INSERT ON addresses.
  - **Purpose:** Prevents inserting a new address flagged as primary if another primary of the same type (billing/shipping) already exists for that owner.
- **addr_enforce_single_primary_update**:
  - **Event:** BEFORE UPDATE ON addresses.
  - **Purpose:** Prevents updating an address to become primary if another primary of the same type already exists for that owner (excluding the address being updated itself).

12.3 Single Primary Contact Triggers

These triggers ensure that a customer, supplier, or partner role has at most one primary contact.

- **cust_contact_single_primary_insert / update**: For customer_contacts.
- **supp_contact_single_primary_insert / update**: For supplier_contacts.
- **part_contact_single_primary_insert / update**: For partner_contacts.
  - **Event (Insert):** BEFORE INSERT.
  - **Purpose (Insert):** If is_primary_for_X is TRUE, checks if another primary contact already exists for that specific customer/supplier/partner.

- **Event (Update):** BEFORE UPDATE.
- **Purpose (Update):** If is_primary_for_X is being changed to TRUE, checks if another primary contact already exists for that specific customer/supplier/partner (excluding the contact link being updated).

13. Reporting Views

Views are predefined queries that simplify data retrieval for reporting or application use.

13.1 Active Entity Views

These views typically filter out soft-deleted records.

- **view_active_users**: Selects non-deleted users.
- **view_active_companies**: Selects non-deleted companies.
- **view_active_departments**: Selects non-deleted departments belonging to active companies.
- **view_active_addresses**: Selects non-deleted addresses belonging to active companies.
- **view_active_contacts**: Selects non-deleted contacts.
- **view_active_customers**:
  Joins customers with view_active_companies and customer_account_statuses for a comprehensive view of active customers.
- **view_active_suppliers**: Joins suppliers with view_active_companies.
- **view_active_partners**: Joins partners with view_active_companies and partner_types.
- **view_active_products**: Selects non-deleted products.

13.2 Operational Views

- **view_low_stock_alert**: Identifies active products where
  the quantity_projected in stock_levels is at or below the reorder_level.
- **view_sales_by_country_month**: Aggregates sales data from orders and order_items, grouping by country (from billing address), year, month, and currency. Excludes cancelled/pending orders.
- **view_overdue_customer_invoices**: Lists customer invoices that are past their due_date and not yet fully paid or voided. Includes customer contact and account manager details, and the user who created the invoice.

14. GDPR & Data Privacy Considerations

The General Data Protection Regulation (GDPR) and similar data privacy laws impose significant obligations on how personal data is collected, processed, stored, and protected. While this schema provides a structure for data, the application built upon it, along with organizational processes, must ensure compliance. This section outlines how the schema design relates to GDPR principles and what further considerations are necessary at the application and process level.

Disclaimer: This section provides general information and is not legal advice. Consult with a legal professional specializing in data privacy for specific GDPR compliance guidance.

14.1 Personal Data Identification

Personal data is any information relating to an identified or identifiable natural person. Within this schema, personal data is primarily located in:

- **users table:** first_name, last_name, email.
- **contacts table:** first_name, last_name, job_title, email, phone_work, phone_mobile, and potentially notes if they contain personal identifiers.
- **leads table:** first_name, last_name, email, phone, and potentially notes.
- **addresses table:** While often company addresses, if linked to an individual contact or a sole proprietorship, address details can become personal data.
- **attachments table:** If uploaded files contain personal data (e.g., CVs, identification documents), their content is personal data. The uploaded_by_user_id also links an action to a user.
- **User ID fields:** Fields like account_manager_id, assigned_user_id, issued_by_user_id, created_by_user_id, placed_by_user_id, entered_by_user_id, processed_by_user_id, approved_by_user_id, uploaded_by_user_id link activities to specific system users, whose details are personal data.

The application logic must clearly identify all fields that store or could store personal data.

14.2 Lawful Basis for Processing

Under GDPR, processing of personal data must have a lawful basis. Common bases relevant to this schema might include:

- **Contract:** Processing necessary for the performance of a contract with the data subject (e.g., processing an order for a customer contact).
- **Legitimate Interests:** Processing for the legitimate interests of the data controller (your organization), provided these are not overridden by the rights and freedoms of the data subject (e.g., internal user data for system operation, B2B contact details for business communication).
- **Consent:** Freely given, specific, informed, and unambiguous consent from the data subject for one or more specific purposes (e.g., marketing emails to leads who are individuals).

- **Legal Obligation:** Processing necessary for compliance with a legal obligation (e.g., retaining financial records like invoices for tax purposes).

The application and organizational processes must document the lawful basis for each type of personal data processing.

14.3 Data Minimization

Collect and process only the personal data that is adequate, relevant, and limited to what is necessary in relation to the purposes for which it is processed.

- **Schema:** The schema provides various fields. The application should only populate those necessary for the defined purpose. For example, not all contact fields might be mandatory for every type of contact.
- **Application:** Ensure forms and processes only request essential personal data.

14.4 Accuracy

Personal data must be accurate and, where necessary, kept up to date.

- **Schema:** Provides fields for storing data.
- **Application:** Implement mechanisms for data subjects (e.g., users, contacts through a portal) or administrators to review and correct inaccurate personal data.

14.5 Storage Limitation (Retention)

Personal data should be kept in a form which permits identification of data subjects for no longer than is necessary for the purposes for which the personal data are processed.

- **Schema:** Timestamps like created_at and updated_at can help track data age.
  The deleted_at field for soft deletes indicates when data was marked for removal.
- **Application/Process:** Define and implement data retention policies. This includes processes for:
  - Identifying data that has exceeded its retention period.
  - Securely anonymizing or permanently deleting such data, including from soft-deleted records and backups, when the lawful basis for retention expires.

14.6 Integrity and Confidentiality (Security)

Process personal data in a manner that ensures appropriate security, including protection against unauthorized or unlawful processing and against accidental loss, destruction, or damage, using appropriate technical or organizational measures.

- **Schema:** Does not directly enforce this but relies on the database system's security (MySQL user permissions, encryption at rest/transit) and application-level security.
- **Considerations:**
  - Strong password hashing (password_hash in users).

- Database access controls (least privilege).
- Encryption of sensitive data at rest and in transit.
- Regular security audits and updates.
- Access logs within the application for sensitive data.

14.7 Data Subject Rights

GDPR grants several rights to data subjects. The schema and application must facilitate these.

14.7.1 Right to Access

Data subjects have the right to obtain confirmation as to whether or not personal data concerning them is being processed, and, where that is the case, access to the personal data.

- **Schema:** Data is structured; queries can be built to retrieve all personal data related to a specific individual (e.g., a contact's details, their associated orders, support tickets).
- **Application:** Provide a mechanism (e.g., a user profile page, an admin tool) to export or display this information upon request.

14.7.2 Right to Rectification

Data subjects have the right to obtain the rectification of inaccurate personal data.

- **Schema:** Supports UPDATE statements.
- **Application:** Provide interfaces for users/contacts to update their information or for administrators to do so on their behalf.

14.7.3 Right to Erasure (Right to be Forgotten)

Data subjects have the right to have their personal data erased without undue delay under certain conditions (e.g., data no longer necessary, consent withdrawn, processed unlawfully).

- **Schema:**
    - The soft-delete mechanism (is_deleted, deleted_at) is a first step.
    - True erasure requires DELETE statements and potentially anonymization of foreign key references if complete deletion would break referential integrity critical for non-personal data (e.g., anonymizing created_by_user_id on an order if the user is erased, rather than deleting the order).
- **Application/Process:**
    - A process to handle erasure requests.
    - Considerations for data in backups.
    - Distinguish between soft delete (for operational recovery) and permanent erasure for GDPR.

14.7.4 Right to Restrict Processing

Data subjects have the right to obtain restriction of processing under certain circumstances.

- **Schema:** An additional flag (e.g., is_processing_restricted BOOLEAN) could be added to tables like contacts or users.
- **Application:** Application logic would need to check this flag and limit processing accordingly (e.g., not sending marketing emails, not including in certain reports).

14.7.5 Right to Data Portability

Data subjects have the right to receive the personal data concerning them, which they have provided to a controller, in a structured, commonly used, and machine-readable format and have the right to transmit those data to another controller.

- **Schema:** Well-structured data facilitates this.
- **Application:** Provide export functionality in common formats (e.g., CSV, JSON).

14.8 Soft Deletes and GDPR

The soft delete mechanism (is_deleted, deleted_at) helps maintain historical data and allows for "undeleting" accidental removals. However, for GDPR's Right to Erasure:

- Soft deleting is **not** equivalent to erasure. Personal data still exists in the database.
- A separate process is needed for permanent deletion or anonymization of soft-deleted personal data when an erasure request is validated or retention periods expire.
- The application's access queries must consistently filter by is_deleted = FALSE for general operations.

14.9 Consent Management (If Applicable)

If consent is the lawful basis for processing any personal data (e.g., marketing communications to individual leads/contacts):

- **Schema:** May require additional tables or fields to track:
    - Specific consent given (e.g., consent_type_id FK).
    - Timestamp of consent.
    - Source of consent (how it was obtained).
    - Version of privacy policy/terms agreed to.
    - Mechanism for withdrawal of consent.
- **Application:** Must provide clear, granular consent options and an easy way for users to withdraw consent.

14.10 Data Protection by Design and by Default

Privacy considerations should be integrated into the system design from the outset.

- **Schema:** The current design attempts to separate concerns and identify user responsibilities.
- **Application:** Implement features that support privacy (e.g., role-based access control, audit trails for access to personal data, minimizing data display where not necessary).

15. SQL Access Code Considerations

When writing application code that interacts with this SQL database, several considerations are crucial for security, performance, data integrity, and maintainability.

15.1 Security

15.1.1 SQL Injection Prevention

This is paramount. Never construct SQL queries by directly concatenating user-supplied input into query strings.

- **Always use Prepared Statements (Parameterized Queries):** This is the most effective defense. The database driver handles proper escaping of input.
  - Example (Conceptual - language specific):
    SELECT * FROM users WHERE email = ? AND password_hash = ? (and then bind parameters).
- **Input Validation and Sanitization:** Validate all inputs on the application side for expected type, length, format, and range, even when using prepared statements. Sanitize output when displaying data to prevent XSS if data is rendered in web pages.
- **Stored Procedures (with caution):** If used, ensure they are written securely and do not themselves construct dynamic SQL from parameters in an unsafe way.

15.1.2 Principle of Least Privilege

Database users connecting from the application should only have the minimum necessary permissions.

- Avoid using a database superuser (like root) for regular application operations.
- Create specific database users for your application.
- Grant only necessary privileges (SELECT, INSERT, UPDATE, DELETE) on specific tables or views. Avoid granting ALL PRIVILEGES or DDL permissions (CREATE, DROP, ALTER) to the application user.
- Consider different users for read-only operations versus write operations if applicable.

15.2 Performance

15.2.1 Efficient Queries & Index Utilization

- **Write WHERE clauses effectively:** Ensure WHERE clauses can utilize existing indexes. This schema includes indexes on commonly queried columns (foreign keys, unique keys, specific fields like email, order_date).
  - Analyze query performance using EXPLAIN in MySQL to understand how indexes are being used.

- Avoid functions on indexed columns in the WHERE clause if possible (e.g., WHERE YEAR(order_date) = 2023 won't use an index on order_date as efficiently as WHERE order_date >= '2023-01-01' AND order_date < '2024-01-01').

- **Appropriate JOINs:** Use INNER JOIN when records must exist in both tables. Use LEFT JOIN when you want all records from the "left" table even if there's no match in the "right" table. Ensure join conditions are on indexed columns.
- **Limit Results:** Use LIMIT when you don't need all matching rows, especially for pagination.

15.2.2 Avoiding SELECT *

- Only select the columns you actually need. SELECT * can:
  - Increase data transfer between the database and application.
  - Prevent the database from using "covering indexes" (where all needed data is in the index itself).
  - Make code brittle if table structures change.

15.2.3 Batch Operations

For bulk inserts, updates, or deletes, use batch operations provided by your database driver/ORM where possible, instead of executing many individual statements in a loop. This reduces network overhead and can be more performant.

15.2.4 Connection Pooling

Applications should use connection pooling to manage database connections efficiently. Opening and closing connections for every query is resource-intensive. Connection pools reuse existing connections.

15.3 Data Integrity & Transactions

15.3.1 Using Transactions

When a logical operation involves multiple DML (Data Manipulation Language - INSERT, UPDATE, DELETE) statements that must succeed or fail together, use database transactions.

- Start a transaction.
- Execute your SQL statements.
- If all succeed, COMMIT the transaction.
- If any fail, ROLLBACK the transaction to undo all changes within that transaction.
- Example: Creating an order and its order items should be within a single transaction.

15.3.2 Error Handling

Robustly handle potential database errors in your application code.

- Check return codes or exceptions from database operations.

- Log errors appropriately.
- Provide user-friendly feedback if an operation fails.
- Implement retry mechanisms for transient errors if appropriate (e.g., deadlocks, temporary network issues), but with backoff strategies.

15.4 Maintainability & Readability

15.4.1 Consistent Formatting

Adopt a consistent style for writing SQL queries in your application code (e.g., capitalization of keywords, indentation). This improves readability.

15.4.2 Use of ORMs or Query Builders (Optional)

Object-Relational Mappers (ORMs) or query builders can abstract away some raw SQL, potentially improving developer productivity and reducing the risk of SQL injection (if used correctly).

- **Pros:** Can map database tables to application objects, often handle boilerplate SQL, can be database-agnostic to some extent.
- **Cons:** Can have a learning curve, might generate inefficient SQL for complex queries if not understood well, can add an abstraction layer that sometimes makes debugging harder.
- Evaluate if an ORM (like SQLAlchemy for Python, Hibernate for Java, TypeORM for TypeScript/JavaScript) or a query builder (like Knex.js) fits your project's needs and team expertise.

15.4.3 Commenting Complex Queries

If you have complex SQL queries embedded in your application, comment them to explain their purpose and logic, especially non-obvious joins or conditions.

15.5 Handling Soft Deletes

When querying data from tables that use soft deletes (is_deleted column):

- **Always include WHERE is_deleted = FALSE** in your queries unless you specifically intend to retrieve or operate on "deleted" records (e.g., for an admin "recycle bin" feature or GDPR erasure process).
- The provided views (view_active_companies, etc.) already incorporate this filtering. Using these views can simplify application queries.
- When "deleting" a record, your application code should execute an UPDATE statement to set is_deleted = TRUE and deleted_at = CURRENT_TIMESTAMP.

15.6 Data Type Matching

Ensure that data types used in your application code (e.g., when binding parameters to prepared statements) match or are compatibly convertible to the column data types in the database schema to avoid errors or implicit conversions that might affect performance or correctness.

15.7 Character Set and Collation Awareness

This schema uses utf8mb4 character set and utf8mb4_unicode_ci collation, which supports a wide range of characters.

- Ensure your application's database connection is configured to use utf8mb4.
- Be mindful of this when handling string data to prevent character encoding issues.

16. Application Layer Software Considerations

While the database schema provides the foundational structure for data storage and integrity, the application layer built on top of it is responsible for implementing business logic, user interaction, and ensuring overall system functionality, security, and performance. This chapter outlines key considerations when developing software that interacts with this database schema.

16.1 Application Architecture & Design

A well-thought-out architecture is crucial for building a scalable, maintainable, and robust application.

16.1.1 Layered Architecture

Adopting a layered architecture helps in separating concerns and improving code organization. Common layers include:

- **Presentation Layer (UI/API):** This layer is responsible for all user interactions or API request/response handling. It should not contain business logic but delegate tasks to the application/service layer.
- **Application/Service Layer (Business Logic):** This core layer orchestrates application workflows, enforces business rules (those not handled by database triggers), validates data, and coordinates with the Data Access Layer.
- **Data Access Layer (DAL) / Repository Pattern:** This layer abstracts and encapsulates all database interaction logic. It translates application-level requests into SQL queries (or ORM operations) and maps database results back to application domain objects or Data Transfer Objects (DTOs).
- **Domain Model / Entities:** These are the objects or data structures that represent the core entities of your application (e.g., User class, Product class, Order class). DTOs are often used for transferring data between layers, especially for API boundaries, to decouple the internal domain model from external contracts.

16.1.2 Modularity & Separation of Concerns

Design the application as a collection of loosely coupled, highly cohesive modules. Each module should have a clear responsibility (e.g., User Management, Product Catalog, Order Processing, Inventory Management, Billing). This improves maintainability, testability, and allows different teams to work on different parts of the application concurrently.

16.1.3 API Design (If Applicable)

If the application exposes an API (e.g., for a web frontend, mobile app, or third-party integrations):

- **Style:** Choose a suitable API style (e.g., RESTful, GraphQL) based on project requirements.
- **Versioning:** Implement API versioning (e.g., URI path /api/v1/..., custom headers) from the outset to manage changes without breaking existing clients.
- **Consistency:** Maintain consistent naming conventions, request/response structures, and error handling across all endpoints.

- **Security:** Secure API endpoints using appropriate authentication (e.g., OAuth 2.0, API Keys) and authorization mechanisms.
- **Data Formats:** Standardize on a data format like JSON.
- **Error Handling:** Use standard HTTP status codes and provide clear, structured error messages.
- **Idempotency:** Design mutating operations (POST, PUT, DELETE) to be idempotent where appropriate.
- **Rate Limiting & Throttling:** Implement measures to prevent API abuse and ensure fair usage.

16.2 Business Logic Implementation

The application layer is where most of the system's intelligence resides.

16.2.1 Comprehensive Validation

While the database has constraints and triggers, the application layer must perform more extensive validation:

- **Input Validation:** Validate all incoming data for type, format, length, range, and presence before processing.
- **Business Rule Validation:** Enforce rules specific to the business domain (e.g., "a customer cannot place an order if their account is inactive," "a discount percentage must be between 0 and a configured maximum").
- **Cross-Field Validation:** Validate relationships between different input fields.

16.2.2 Workflow Management

Many business processes are multi-step workflows (e.g., order fulfillment: payment, stock check, shipment, notification).

- Clearly define these workflows.
- For complex workflows, consider using state machines, workflow engines, or event-driven patterns.
- Ensure that operations spanning multiple database updates are handled atomically, typically using database transactions managed by the Data Access Layer.

16.2.3 Calculations and Derived Data

While the database schema includes some generated columns (e.g., balance_due on invoices, quantity_available on stock levels), more complex calculations or aggregations will typically be handled in the application layer.

- Examples: Calculating complex pricing based on customer tiers and promotions, sales tax calculations based on jurisdiction rules, generating aggregate financial reports.
- Ensure these calculations are accurate, consistently applied, and well-tested.

16.2.4 Application-Level Roles & Permissions Enforcement

The database schema defines roles and permissions. The application layer is responsible for:

- Authenticating users.
- Determining the authenticated user's roles and associated permissions.
- Enforcing these permissions by controlling access to features, data, and specific actions (e.g., only a user with "Approve RMA" permission can change an RMA status to "Approved").

16.3 Data Handling & Management

Effective data handling is key to performance and integrity.

16.3.1 Concurrency Control

When multiple users or processes access and modify data simultaneously, concurrency issues can arise.

- **Optimistic Locking:** Suitable for most web applications. Add a version column (e.g., row_version INT or use the updated_at TIMESTAMP carefully) to frequently updated tables. Before an UPDATE, check if the version/timestamp of the record in the database matches the version/timestamp when the data was initially read. If not, a concurrent modification occurred, and the application should handle the conflict (e.g., by informing the user and asking them to retry).
- **Pessimistic Locking:** (e.g., SELECT ... FOR UPDATE) Locks database records when read, preventing other transactions from modifying them until the lock is released. This can reduce concurrency and should be used judiciously for critical, short-lived operations where optimistic locking is insufficient.

16.3.2 Caching Strategies

To improve performance and reduce database load:

- Identify data that is frequently accessed but changes infrequently (e.g., data from lookup tables like countries, currencies, product_categories; user permissions).
- Implement caching using in-memory caches (for single-instance applications) or distributed caches (e.g., Redis, Memcached) for multi-instance deployments.
- Develop clear cache invalidation strategies (e.g., time-to-live (TTL), event-based invalidation) to ensure data consistency.

16.3.3 Application-Level Data Auditing

Beyond the created_at and updated_at timestamps in the schema:

- For sensitive data or critical entities, implement a more detailed audit trail. This might involve separate audit tables that log:
  - Who made the change (user_id).
  - What entity/record was changed (table_name, record_id).

  - What fields were changed.
  - The old and new values.
  - Timestamp of the change.
- This can be implemented via application logic (often in the service or data access layer) or, for simpler cases, database triggers (though application-level auditing can capture more business context).

16.3.4 Advanced Reporting & Analytics

While the schema includes some reporting views:

- Complex analytical queries or business intelligence (BI) requirements might necessitate a separate reporting database, data warehouse, or integration with BI tools.
- The application might need to periodically ETL (Extract, Transform, Load) data into such systems.
- Be mindful of the performance impact of running large analytical queries directly against the operational transactional database. Consider using read replicas.

16.4 User Experience & Interface (UX/UI)

The application's interface should be intuitive and efficient.

16.4.1 User-Friendly Forms & Input

- Design clear and concise forms for data entry.
- Use appropriate input controls (dropdowns populated from lookup tables like countries or order_statuses, date pickers, number inputs with validation).
- Provide real-time or near real-time validation feedback to users.

16.4.2 Search, Filtering, and Pagination

- Implement effective search functionality across key entities (e.g., products, companies, orders).
- Provide intuitive filtering options based on relevant attributes (e.g., order status, date ranges).
- Always use pagination when displaying lists of records that can grow large, to ensure good performance and user experience.

16.4.3 Notifications & Alerts

Design mechanisms to notify users of important system events or actions requiring their attention:

- In-app notifications.
- Email notifications (e.g., order confirmation, shipment updates, overdue invoice reminders, support ticket responses).

- Consider user preferences for notification channels and frequency.

16.5 Operations & Maintainability

Long-term success depends on how easy the application is to operate and maintain.

16.5.1 Configuration Management

- Externalize all application configurations (database connection strings, API keys for external services, feature flags, email server settings, etc.) from the codebase.
- Use environment variables, configuration files (e.g., JSON, YAML, .env), or dedicated configuration management services.

16.5.2 Comprehensive Logging

Implement robust logging throughout the application:

- Log important events, errors, user actions, and system performance metrics.
- Use structured logging (e.g., JSON format) to make logs easier to parse and analyze.
- Configure appropriate log levels (DEBUG, INFO, WARNING, ERROR, CRITICAL) for different environments.
- Ensure logs do not contain sensitive personal data unless absolutely necessary and properly secured/anonymized.

16.5.3 Robust Error Handling & Resilience

- Anticipate and gracefully handle exceptions and errors (e.g., database connection issues, external API failures, invalid data).
- Provide informative (but not overly technical) error messages to users.
- Implement retry mechanisms with backoff strategies for transient failures when communicating with external services or the database.
- Consider circuit breaker patterns for integrations with potentially unreliable services.

16.5.4 Background Jobs & Scheduled Tasks

For operations that are long-running, resource-intensive, or need to run on a schedule:

- Use a background job processing system (e.g., Celery for Python, Hangfire for .NET, Sidekiq for Ruby, Quartz for Java) or system-level schedulers like cron.
- Examples: Generating large reports, sending bulk emails, data synchronization, nightly data cleanup tasks.

16.5.5 Thorough Testing Strategy

Implement a multi-layered testing approach:

- **Unit Tests:** Test individual functions, methods, and classes in isolation, mocking dependencies.

- **Integration Tests:** Test the interaction between different components, including the Data Access Layer's interaction with a test database instance.
- **End-to-End (E2E) / UI Tests:** Test complete user flows through the application's interface.
- **API Tests:** If an API is exposed, test its endpoints for correctness, security, and performance.
- **Performance Tests:** Load test the application to identify bottlenecks and ensure it meets performance requirements.

16.5.6 Deployment (CI/CD)

Automate the build, test, and deployment process using a Continuous Integration/Continuous Deployment (CI/CD) pipeline.

- This pipeline should also manage database schema migrations (applying new versions of your SQL script like db_schema_vX.Y.Z.sql) in a controlled and repeatable manner. Tools like Flyway or Liquibase can assist with this, integrating with the db_schema_version table concept.

16.5.7 Monitoring & Operational Alerting

Continuously monitor the application and database for:

- Performance metrics (response times, throughput, resource utilization).
- Error rates and types.
- System health (availability, disk space, database connection pool status).
- Set up automated alerts for critical issues to enable proactive problem resolution.

16.6 Schema-Specific Application Logic

The application needs to correctly interact with specific features of this schema.

16.6.1 Soft Delete Handling

- **Querying:** All standard data retrieval queries must include WHERE is_deleted = FALSE (or use the provided view_active_... views) unless specifically intended to access "deleted" records (e.g., for an admin "recycle bin" feature or a GDPR-compliant permanent erasure process).
- **Deleting:** When a user "deletes" a record, the application should execute an UPDATE statement setting is_deleted = TRUE and deleted_at = CURRENT_TIMESTAMP for the relevant table.

16.6.2 Populating User Responsibility Fields

The application must accurately populate fields like created_by_user_id, assigned_user_id, issued_by_user_id, etc., with the user_id of the authenticated system user performing the action. This is typically obtained from the user's session or authentication token.

16.6.3 Enforcing Status Transitions

While status lookup tables define possible statuses, the application logic must enforce valid transitions between them. For example, an order cannot typically transition from "Pending" directly to "Delivered" without going through "Processing" and "Shipped". This logic resides in the application/service layer.

16.6.4 Managing Stock Level Updates

The stock_levels table has generated columns for quantity_available and quantity_projected. The application is responsible for correctly updating the base columns:

- quantity_on_hand: Increased upon PO item receipt, decreased upon shipment.
- quantity_reserved: Increased when an order is placed/confirmed, decreased upon shipment or order cancellation.
- quantity_on_order: Increased when a PO is placed, decreased upon PO item receipt.
  These updates should typically occur within database transactions to ensure consistency (e.g., reserving stock and creating an order item should be atomic).

16.6.5 Consistent Currency Handling

- When creating or updating records with monetary values (e.g., orders, invoices), ensure the correct currency_code is stored alongside the amounts.
- When displaying monetary values, always present them with their associated currency symbol or code.
- For financial reporting or operations involving multiple currencies, use the exchange_rates table to perform conversions, being mindful of the effective_date of the rates. Implement clear logic for which exchange rate to use if an exact date match isn't found (e.g., most recent prior rate).

17. Data Population Strategy

The SQL script includes INSERT statements to populate reference tables (like currencies, countries, various status tables) and some core sample data
for users, companies, departments, products, orders, etc.
It also sets default values for status columns in transactional tables using ALTER TABLE ... ALTER COLUMN ... SET DEFAULT ....
This sample data is intended for initial setup and testing.

18. Schema Deployment & Version Checking

The schema includes a db_schema_version table to track its evolution.

- **Deployment Process:**
    1. The SQL script (e.g., db_schema_v1.0.0.sql) should initially contain placeholders:
        - _YOUR_SCRIPT_CHECKSUM_ALGO_HERE_ for script_checksum_algo.
        - %%SCRIPT_CHECKSUM_PLACEHOLDER%% for script_checksum.
    2. Before running the script, choose a checksum algorithm (e.g., 'SHA256' or 'MD5').
    3. Calculate the checksum of the *entire script file* while it still contains these placeholders.
    4. In a temporary copy of the script (or via an automated deployment script):
        - Replace _YOUR_SCRIPT_CHECKSUM_ALGO_HERE_ with the chosen algorithm string (e.g., 'SHA256').
        - Replace %%SCRIPT_CHECKSUM_PLACEHOLDER%% with the actual checksum string calculated in step 3.
    5. Execute this modified/temporary script against the database.
- **Querying Current Version:**
- SELECT version_tag, description, applied_at, applied_by, script_checksum_algo, script_checksum
- FROM db_schema_version
- ORDER BY applied_at DESC, version_id DESC

LIMIT 1;

The script_checksum stored will be the checksum of the script *before* the placeholder was replaced, allowing verification against the version-controlled script template.

19. Final SQL Settings

The script concludes by re-enabling foreign key checks:

- SET FOREIGN_KEY_CHECKS=1;

This ensures that referential integrity is enforced for all subsequent database operations.