Technical Report No.1 - Project Database Foundations

Daniel Alonso Chavarro Chipatecua 20241020066

February 15, 2025

This report presents the design from the ground up of the database of the application called Mercado Libre. This app is a trading platform where users can buy and sell products ranging from technology and clothing to vehicles and property. Products can be new or used and are offered by both private sellers and businesses.

1 Business Model

The application, being a commerce platform, the main objective is to facilitate electronic commerce so that it is accessible for anyone to buy and sell products, and as a consequence of the above, it allows small and medium-sized companies to expand and improve their visibility.

2 Processes and Information Required

In order to build the application we require more information apart from the business model, such as user stories and from there we can extract functionalities to turn them into entities and finally make the entity relationship model.

2.1 User Stories

The first phase to be able to design the database is to know the necessities of the user. For this, first an interview was conducted with people asking what they would like to see in a Commerce applications, where they mentioned their point of view based on personal experiences with similar applications. These interviews were transformed into user stories where all the interviews are summarized, these user stories are in the format.

As a (role), I want (action), so what (impact)

- **As a new user**, I want to create an account, so that I can start buying or selling products on the platform.

- **As a buyer**, I want to search for products using filters like price, category, and location, so that I can find the items that best match my needs.
- **As a seller**, I want to publish new products with descriptions, prices, and images, so that potential buyers can easily find and purchase them.
- **As a buyer**, I want to add items to my shopping cart, so that I can review and purchase multiple products at once.
- **As a buyer**, I want to see the total cost of the products in my cart, so that I can make purchasing decisions based on my budget.
- **As a buyer**, I want to pay using my preferred payment method (credit card, debit), so that I can complete my purchase securely and conveniently.
- **As a seller**, I want to offer free or discounted shipping options, so that I can attract more buyers to my products.
- **As a seller**, I want to create my own online store through Mercado Shops, so that I can showcase my brand and products more effectively.

2.2 Functionalities

Based on the user stories presented above, the application functionalities were generated, these functionalities are:

- **Advanced product search**: Filter by categories, price, location, condition (new or used), sellers, among others.
- **Shopping cart**: Possibility of adding multiple products and paying for them in a single transaction.
- **Shipping nationwide**: Through delivery provider, buyers can receive products at home or at pick-up points.
- **Purchase history**: View all purchases made on the platform.
- **Favorites**: Add products to a favorites list to review them later.
- **Product ratings**: Rate products and sellers.
- **Purchase guarantee**: Buyer protection that ensures the refund or exchange of defective products or products that do not match the description.
- **Product posting**: Easy creation and management of posts with options for categories, photos, videos, and detailed descriptions.
- **Shops**: Possibility of creating customized online stores.
- **Discounts and promotions**: Possibility of offering specific discounts or promotions on products.

3 Database design process

Using methodology given in class by Carlos Sierra, the design is divided in 10 steps:

3.1 Define components

The components are the application modules; in this case they are:

- Transactions
- Client Service
- Products to Buy
- Delivery Service
- Sale of products

3.2 Define Entities And Atributes

With the components defined, we can identify main entities found from the components:

- User
- Account Status User
- Delivery
- Delivery Provider
- Status Delivery
- Product
- Product Status
- Shopping Car
- Status Car
- Transaction
- Payment Method
- Transaction Status
- Receipt
- Favorites List
- Seller

- Store
- Client Report
- Report Category
- Status Report

3.3 Define Atributes

Now that we have the main entities we can define their attributes

1. User
    - ID_User (PK)
    - Name
    - Last_Name
    - Username
    - Email
    - Address
    - Phone
    - Date_Registered
    - Password
    - Date_Birth
    - ID_Account_Status_User (FK)
2. Delivery
    - ID_Delivery (PK)
    - ID_User (FK)
    - ID_Product (FK)
    - Mailing Address
    - Date_Created
    - Date_Estimated_Arrive
    - ID_Status_Delivery (FK)
    - ID_Provider_Delivery (FK)
    - Delivery_Cost
3. Delivery Provider
    - ID_Provider_Delivery (PK)

- Name
- Description

4. Status Delivery
    - ID_Status_Delivery (PK)
    - Name
    - Description
5. Product
    - ID_Product (PK)
    - ID_Seller
    - Name
    - Description
    - Price
    - Quantity_Stock
    - Category
    - ID_Product_Status (FK)
    - Date_Published
    - Brand
    - Rating
6. Product Status
    - ID_Product_Status (PK)
    - Name
    - Description
7. Shopping Car
    - ID_Shopping_Car (PK)
    - ID_User (FK)
    - Date_Created
    - ID_Status_Cart (FK)
8. Status Car
    - ID_Status_Shopping_Car (PK)
    - Name
    - Description

9. Transaction
    - ID_Transaction (PK)
    - ID_User (FK)
    - ID_Product (FK)
    - ID_Payment_Method (FK)
    - Amount
    - Date_Bought
    - ID_Transaction_Status (FK)
10. Payment Method
    - ID_Payment_Method (PK)
    - Type
    - Provider
    - Card_Number
    - Due_Date
    - Owner
11. Transaction Status
    - ID_Transaction_Status (PK)
    - Name
    - Description
12. Receipt
    - ID_Receipt (PK)
    - ID_Transaction (FK)
    - ID_User (FK)
    - Date_Created
    - Details
    - Amount_Paid
    - Taxes
13. Favorites List
    - ID_Favorites_List (PK)
    - ID_User (FK)
    - Name

- Date_Created

14. Seller
    - ID_Seller (PK)
    - First_Name
    - Last_Name
    - Email
    - Address
    - Phone
    - Date_Registered
    - Password
    - Date_Birth
    - ID_Account_Status (FK)
    - Rating
    - Number_Sales
15. Store
    - ID_Store (PK)
    - ID_Seller (FK)
    - Name
    - Date_Created
    - Number_Products
    - Rating
16. Client Report
    - ID_Report (PK)
    - ID_User (FK)
    - ID_Delivery (FK)
    - ID_Category_Report (FK)
    - Description
    - Date_Report
    - ID_Status_Report (FK)
17. Category Report
    - ID_Category_Report (PK)
    - Name

- Description

18. Status Report
    - ID_Status_Report (PK)
    - Name
    - Description

3.4 Relationship Table

Now I set the relationship of every entity

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	🟩	✓	✓		✓		✓		✓			✓	✓	✓		✓		
2	✓	🟩	✓	✓	✓							✓		✓		✓		
3	✓	✓	🟩															
4	✓	✓		🟩														
5	✓	✓			🟩	✓	✓		✓			✓	✓	✓	✓	✓		
6					✓	🟩												
7	✓				✓		🟩	✓	✓					✓				
8							✓	🟩										
9	✓				✓		✓		🟩	✓	✓	✓						
10									✓	🟩		✓						
11									✓		🟩							
12	✓	✓			✓				✓	✓		🟩		✓				
13	✓				✓								🟩	✓				
14	✓	✓			✓		✓					✓	✓	🟩	✓	✓		
15						✓							✓		🟩	✓		
16	✓				✓									✓	✓	🟩	✓	✓
17																✓	🟩	
18																✓		🟩

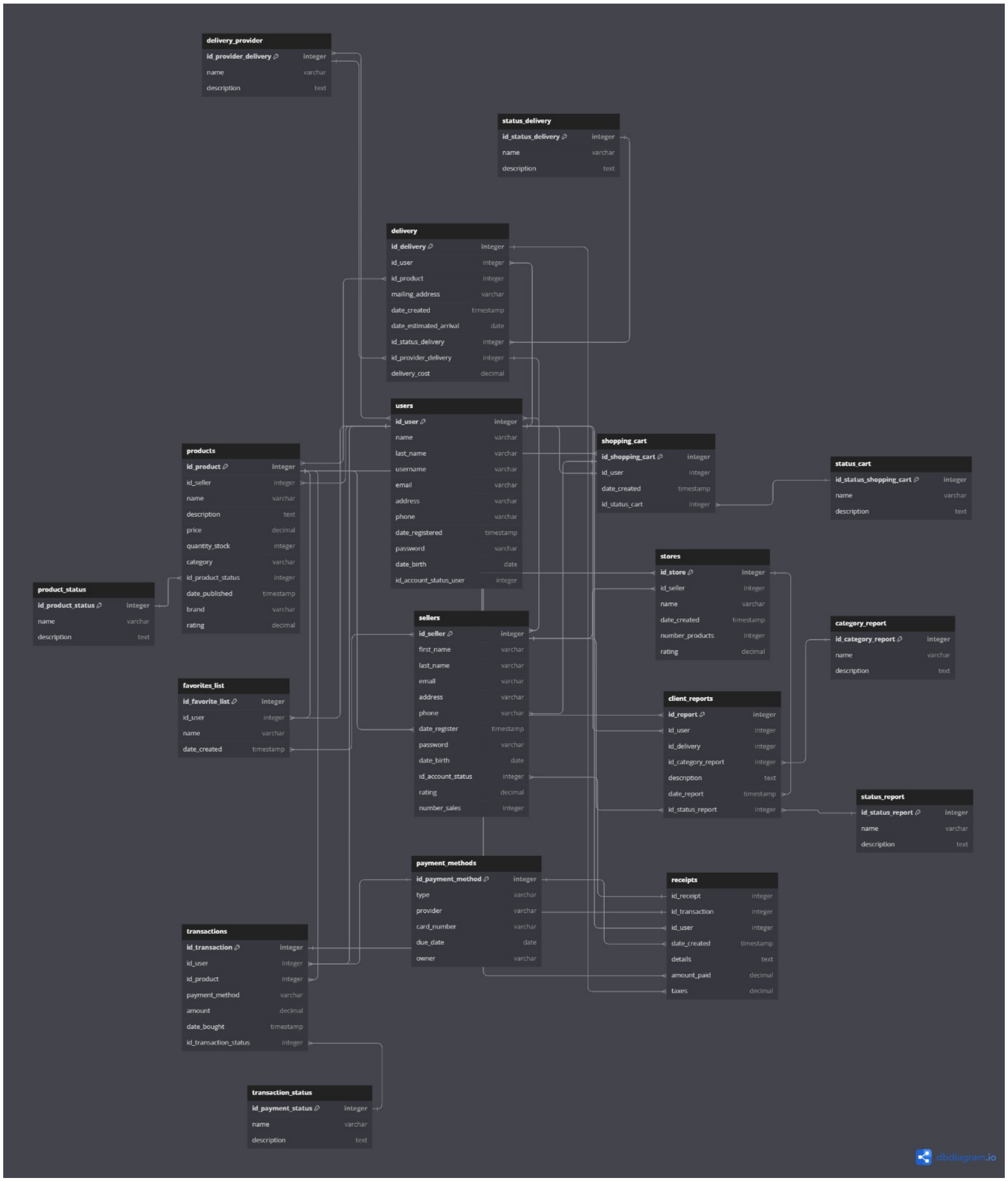
3.5 Define Relationship Types

Now I have all the relationships then assign the type of it.

- **User** – *Receipt* (One – Many)
- **User** – *Favorites List* (One – Many)
- **User** – *Seller* (Many – Many)
- **User** – *Client Report* (One – Many)
- **Delivery** – *Delivery Provider* (One – Many)

- **Delivery** – *Status Delivery* (Many – One)
- **Delivery** – *Product* (Many – Many)
- **Delivery** – *Receipt* (One – Many)
- **Delivery** – *Seller* (Many – One)
- **Product** – *Product Status* (Many – One)
- **Product** – *Shopping Car* (Many – One)
- **Product** – *Transaction* (Many – One)
- **Product** – *Receipt* (Many – One)
- **Product** – *Favorites List* (Many – One)
- **Product** – *Seller* (Many – One)
- **Product** – *Store* (Many – One)
- **Product** – *Client Report* (One – Many)
- **Shopping Car** – *Status Car* (Many – One)
- **Shopping Car** – *Seller* (One – Many)
- **Transaction** – *Payment Method* (Many – One)
- **Transaction** – *Transaction Status* (Many – One)
- **Transaction** – *Receipt* (One – One)
- **Payment Method** – *Receipt* (One – Many)
- **Receipt** – *Seller* (Many – One)
- **Favorites List** – *Seller* (Many – Many)
- **Seller** – *Store* (One – Many)
- **Seller** – *Client Report* (One – Many)
- **Store** – *Client Report* (One – Many)
- **Client Report** – *Report Category* (Many – One)
- **Client Report** – *Status Report* (Many – One)

3.6 First Model Design



3.7 Many-Many consideration

- **User** – *Delivery Provider* (Many – Many)

- **User** – *Seller* (Many – Many)
- **Delivery** – *Product* (Many – Many)
- **Favorites List** – *Seller* (Many – Many)

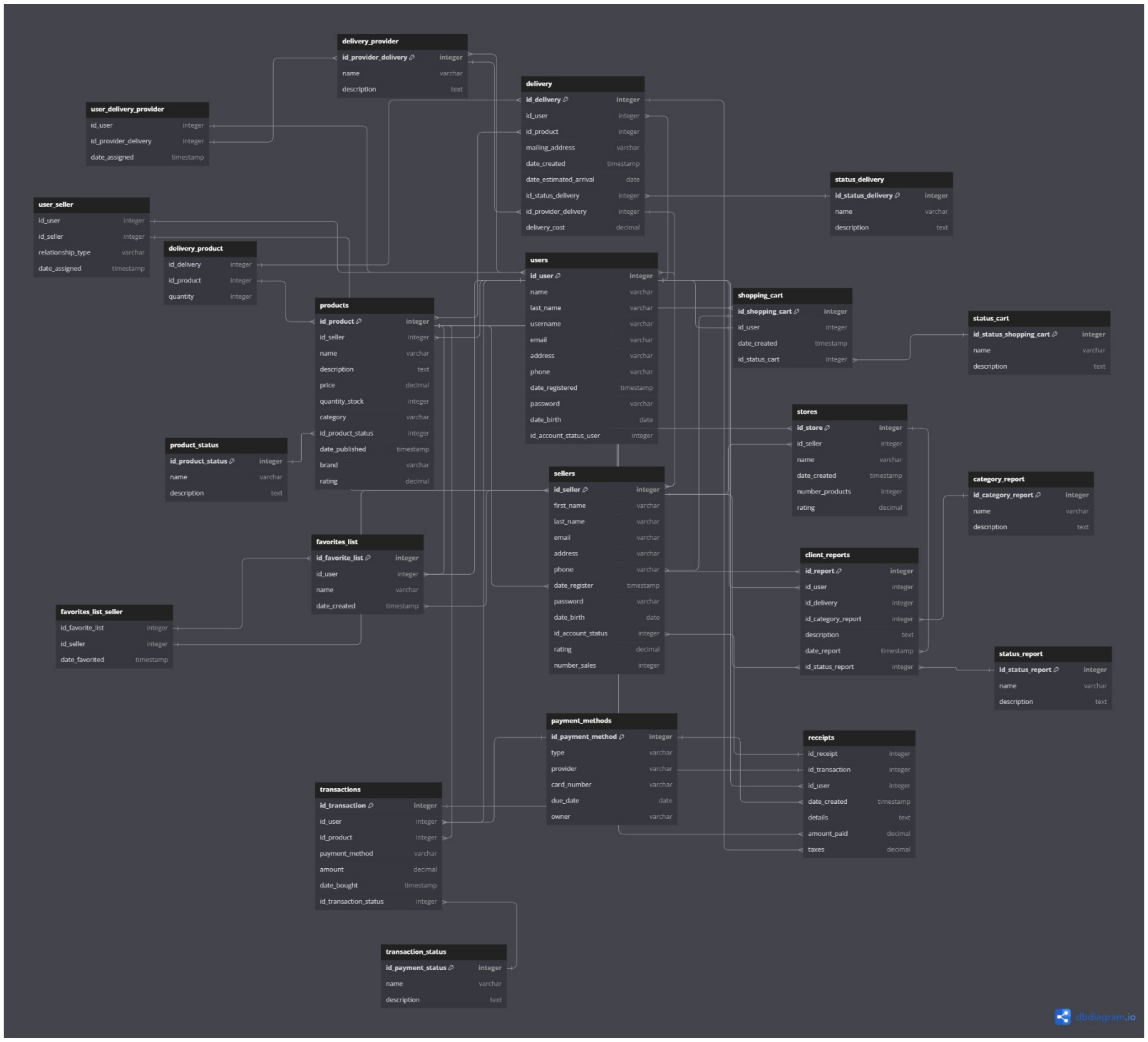
New Entities

- **User/Delivery Provider**
- **User/Seller**
- **Delivery/Product**
- **Favorites List/Seller**

New Relationships

- **User** – *User/Delivery Provider* (One – Many)
- **Delivery Provider** – *User/Delivery Provider* (One – Many)
- **User** – *User/Seller* (One – Many)
- **Seller** – *User/Seller* (One – Many)
- **Delivery** – *Delivery/Product* (One – Many)
- **Product** – *Delivery/Product* (One – Many)
- **Favorites List** – *Favorites List/Seller* (One – Many)
- **Seller** – *Favorites List/Seller* (One – Many)

3.8 Final Design



3.9 Get Data-Structured

User

- **ID_User**: int (Unique identifier for the user, PK)
- **Name**: varchar (User's first name)
- **Last_Name**: varchar (User's last name)
- **Username**: varchar (Unique username)
- **Email**: varchar (Email address)
- **Address**: varchar (Mailing address)

- **Phone**: varchar (Phone number)
- **Date_Registered**: timestamp (Date the user registered)
- **Password**: varchar (User's password)
- **Date_Birth**: date (User's date of birth)
- **ID_Account_Status_User**: int (FK to Account Status)

Delivery

- **ID_Delivery**: int (Unique identifier for the delivery, PK)
- **ID_User**: int (FK to User)
- **ID_Product**: int (FK to Product)
- **Mailing_Address**: varchar (Delivery mailing address)
- **Date_Created**: timestamp (Date the delivery was created)
- **Date_Estimated_Arrive**: date (Estimated delivery arrival date)
- **ID_Status_Delivery**: int (FK to Status Delivery)
- **ID_Provider_Delivery**: int (FK to Delivery Provider)
- **Delivery_Cost**: decimal (Cost of the delivery)

Delivery Provider

- **ID_Provider_Delivery**: int (Unique identifier for the delivery provider, PK)
- **Name**: varchar (Provider name)
- **Description**: text (Description of the provider)

Status Delivery

- **ID_Status_Delivery**: int (Unique identifier for the status delivery, PK)
- **Name**: varchar (Status name)
- **Description**: text (Description of the delivery status)

Product

- **ID_Product**: int (Unique identifier for the product, PK)
- **ID_Seller**: int (FK to Seller)
- **Name**: varchar (Product name)
- **Description**: text (Description of the product)
- **Price**: decimal (Product price)
- **Quantity_Stock**: int (Quantity of product in stock)
- **Category**: varchar (Product category)
- **ID_Product_Status**: int (FK to Product Status)
- **Date_Published**: timestamp (Date the product was published)
- **Brand**: varchar (Brand of the product)
- **Rating**: decimal (Average rating of the product)

Product Status

- **ID_Product_Status**: int (Unique identifier for the product status, PK)
- **Name**: varchar (Status name)
- **Description**: text (Description of the product status)

Shopping Cart

- **ID_Shopping_Cart**: int (Unique identifier for the shopping cart, PK)
- **ID_User**: int (FK to User)
- **Date_Created**: timestamp (Date the shopping cart was created)
- **ID_Status_Cart**: int (FK to Status Cart)

Status Cart

- **ID_Status_Shopping_Cart**: int (Unique identifier for the shopping cart status, PK)
- **Name**: varchar (Status name)
- **Description**: text (Description of the cart status)

Transaction

- **ID_Transaction**: int (Unique identifier for the transaction, PK)
- **ID_User**: int (FK to User)
- **ID_Product**: int (FK to Product)
- **Payment_Method**: varchar (Payment method used, e.g., "Card", "PayPal")
- **Amount**: decimal (Transaction amount)
- **Date_Bought**: timestamp (Date the transaction was made)
- **ID_Transaction_Status**: int (FK to Transaction Status)

Payment Method

- **ID_Payment_Method**: int (Unique identifier for the payment method, PK)
- **Type**: varchar (Payment type, e.g., "Credit Card")
- **Provider**: varchar (Payment provider, e.g., "Visa")
- **Card_Number**: varchar (Card number)
- **Due_Date**: date (Card expiration date)
- **Owner**: varchar (Cardholder's name)

Transaction Status

- **ID_Payment_Status**: int (Unique identifier for the payment status, PK)
- **Name**: varchar (Status name)
- **Description**: text (Description of the status)

Receipt

- **ID_Receipt**: int (Unique identifier for the receipt, PK)
- **ID_Transaction**: int (FK to Transaction)
- **ID_User**: int (FK to User)
- **Date_Created**: timestamp (Date the receipt was created)
- **Details**: text (Receipt details)
- **Amount_Paid**: decimal (Amount paid)
- **Taxes**: decimal (Taxes applied to the receipt)

Favorites List

- **ID_Favorite_List**: int (Unique identifier for the favorites list, PK)
- **ID_User**: int (FK to User)
- **Name**: varchar (Name of the favorites list)
- **Date_Created**: timestamp (Date the list was created)

Seller

- **ID_Seller**: int (Unique identifier for the seller, PK)
- **First_Name**: varchar (Seller's first name)
- **Last_Name**: varchar (Seller's last name)
- **Email**: varchar (Seller's email)
- **Address**: varchar (Seller's address)
- **Phone**: varchar (Seller's phone number)
- **Date_Register**: timestamp (Date the seller registered)
- **Password**: varchar (Seller's password)
- **Date_Birth**: date (Seller's date of birth)
- **ID_Account_Status**: int (FK to Account Status)
- **Rating**: decimal (Seller rating)
- **Number_Sales**: int (Number of sales made by the seller)

Store

- **ID_Store**: int (Unique identifier for the store, PK)
- **ID_Seller**: int (FK to Seller)
- **Name**: varchar (Store name)
- **Date_Created**: timestamp (Date the store was created)
- **Number_Products**: int (Number of products in the store)
- **Rating**: decimal (Store rating)

Client Report

- **ID_Report**: int (Unique identifier for the client report, PK)
- **ID_User**: int (FK to User)
- **ID_Delivery**: int (FK to Delivery)
- **ID_Category_Report**: int (FK to Report Category)
- **Description**: text (Description of the report)
- **Date_Report**: timestamp (Date the report was created)
- **ID_Status_Report**: int (FK to Status Report)

Category Report

- **ID_Category_Report**: int (Unique identifier for the report category, PK)
- **Name**: varchar (Category name)
- **Description**: text (Description of the category)

Status Report

- **ID_Status_Report**: int (Unique identifier for the report status, PK)
- **Name**: varchar (Status name)
- **Description**: text (Description of the status)

User/Delivery Provider

- **ID_User**: int (Unique identifier for the user, FK to User)
- **ID_Provider_Delivery**: int (Unique identifier for the delivery provider, FK to Delivery Provider)
- **Date_Assigned**: timestamp (Optional, tracks when the user was assigned to the delivery provider)

User/Seller

- **ID_User**: int (Unique identifier for the user, FK to User)
- **ID_Seller**: int (Unique identifier for the seller, FK to Seller)
- **Relationship_Type**: varchar (Optional, describes the user-seller relationship)
- **Date_Assigned**: timestamp (Optional, tracks when the user was connected to the seller)

Delivery/Product

- **ID_Delivery**: int (Unique identifier for the delivery, FK to Delivery)
- **ID_Product**: int (Unique identifier for the product, FK to Product)
- **Quantity**: int (Optional, tracks the quantity of the product in the delivery)

Favorites List/Seller

- **ID_Favorite_List**: int (Unique identifier for the favorites list, FK to Favorites List)
- **ID_Seller**: int (Unique identifier for the seller, FK to Seller)
- **Date_Favorited**: timestamp (Optional, tracks when the seller was added to the favorites list)

3.10 Constraints and Properties

User

- **ID_User**: `int`
  - Constraints: `PRIMARY KEY`, `NOT NULL`, `AUTO_INCREMENT`
  - Properties: Must be unique, automatically generated.
- **Name**: `string`
  - Constraints: `NOT NULL`, `VARCHAR(100)`
  - Properties: Must contain only alphabetic characters and spaces, minimum length of 2 characters.
- **Last_Name**: `string`
  - Constraints: `NOT NULL`, `VARCHAR(100)`
  - Properties: Must contain only alphabetic characters and spaces, minimum length of 2 characters.
- **Username**: `string`
  - Constraints: `NOT NULL`, `UNIQUE`, `VARCHAR(50)`
  - Properties: Must be unique, minimum length of 3 characters, can contain alphanumeric characters and underscores.
- **Email**: `string`
  - Constraints: `NOT NULL`, `UNIQUE`, `VARCHAR(100)`
  - Properties: Must follow valid email format (e.g., name@domain.com).

- **Address**: `string`
  - Constraints: `NOT NULL`, `VARCHAR(200)`
  - Properties: Must be a valid address.
- **Phone**: `string`
  - Constraints: `NOT NULL`, `VARCHAR(15)`
  - Properties: Must be a valid phone number, can contain digits, dashes, or spaces (regex check).
- **Date_Registered**: `timestamp`
  - Constraints: `NOT NULL`
  - Properties: Automatically set when the user is registered.
- **Password**: `string`
  - Constraints: `NOT NULL`, `VARCHAR(100)`
  - Properties: Must be securely hashed, minimum length of 8 characters.
- **Date_Birth**: `date`
  - Constraints: `NOT NULL`
  - Properties: Must be a valid date, user must be over 18 years old.
- **ID_Account_Status_User**: `int`
  - Constraints: `FOREIGN KEY`, `NOT NULL`
  - Properties: References the Account Status of the user.

Delivery

- **ID_Delivery**: `int`
  - Constraints: `PRIMARY KEY`, `NOT NULL`, `AUTO_INCREMENT`
  - Properties: Must be unique, automatically generated.
- **ID_User**: `int`
  - Constraints: `FOREIGN KEY`, `NOT NULL`
  - Properties: References the associated user.
- **ID_Product**: `int`
  - Constraints: `FOREIGN KEY`, `NOT NULL`
  - Properties: References the associated product.

- **Mailing_Address**: `string`
  - Constraints: `NOT NULL`, `VARCHAR(200)`
  - Properties: Must be a valid address.
- **Date_Created**: `timestamp`
  - Constraints: `NOT NULL`
  - Properties: Automatically set when the delivery is created.
- **Date_Estimated_Arrive**: `date`
  - Constraints: `NOT NULL`
  - Properties: Must be a valid date in the future.
- **ID_Status_Delivery**: `int`
  - Constraints: `FOREIGN KEY`, `NOT NULL`
  - Properties: References the delivery status.
- **ID_Provider_Delivery**: `int`
  - Constraints: `FOREIGN KEY`, `NOT NULL`
  - Properties: References the delivery provider.
- **Delivery_Cost**: `decimal(10, 2)`
  - Constraints: `NOT NULL`
  - Properties: Must be a positive value.

Delivery Provider

- **ID_Provider_Delivery**: `int`
  - Constraints: `PRIMARY KEY`, `NOT NULL`, `AUTO_INCREMENT`
  - Properties: Must be unique, automatically generated.
- **Name**: `string`
  - Constraints: `NOT NULL`, `VARCHAR(100)`
  - Properties: Must contain only alphabetic characters and spaces, minimum length of 2 characters.
- **Description**: `text`
  - Constraints: `NULL`
  - Properties: Optional description of the provider.

Status Delivery

- **ID_Status_Delivery**: `int`
  - Constraints: `PRIMARY KEY`, `NOT NULL`, `AUTO_INCREMENT`
  - Properties: Must be unique, automatically generated.
- **Name**: `string`
  - Constraints: `NOT NULL`, `VARCHAR(50)`
  - Properties: Must contain only alphabetic characters and spaces.
- **Description**: `text`
  - Constraints: `NULL`
  - Properties: Optional description of the status.

Product

- **ID_Product:** int
  - Constraints: PRIMARY KEY, NOT NULL, AUTO_INCREMENT
  - Properties: Must be unique, automatically generated.
- **ID_Seller:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the seller of the product.
- **Name:** string
  - Constraints: NOT NULL, VARCHAR(100)
  - Properties: Must contain only alphabetic characters and spaces, minimum length of 2 characters.
- **Description:** text
  - Constraints: NOT NULL
  - Properties: Detailed product description.
- **Price:** decimal(10, 2)
  - Constraints: NOT NULL
  - Properties: Must be a positive value.
- **Quantity_Stock:** int
  - Constraints: NOT NULL

  - Properties: Must be a non-negative integer.

- **Category:** string
  - Constraints: NOT NULL, VARCHAR(50)
  - Properties: Product category, minimum length of 2 characters.
- **ID_Product_Status:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the product status.
- **Date_Published:** timestamp
  - Constraints: NOT NULL
  - Properties: Automatically set when the product is published.
- **Brand:** string
  - Constraints: NOT NULL, VARCHAR(50)
  - Properties: Product brand name.
- **Rating:** decimal(2, 1)
  - Constraints: NOT NULL
  - Properties: Must be between 0.0 and 5.0.

Product Status

- **ID_Product_Status:** int
  - Constraints: PRIMARY KEY, NOT NULL, AUTO_INCREMENT
  - Properties: Must be unique, automatically generated.
- **Name:** string
  - Constraints: NOT NULL, VARCHAR(50)
  - Properties: Must contain only alphabetic characters and spaces.
- **Description:** text
  - Constraints: NULL
  - Properties: Optional description of the status.

Shopping Cart

- **ID_Shopping_Cart:** int
  - Constraints: PRIMARY KEY, NOT NULL, AUTO_INCREMENT
  - Properties: Must be unique, automatically generated.
- **ID_User:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the user who owns the cart.
- **Date_Created:** timestamp
  - Constraints: NOT NULL
  - Properties: Automatically set when the shopping cart is created.
- **ID_Status_Cart:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the status of the cart.

Status Cart

- **ID_Status_Shopping_Cart:** int
  - Constraints: PRIMARY KEY, NOT NULL, AUTO_INCREMENT
  - Properties: Must be unique, automatically generated.
- **Name:** string
  - Constraints: NOT NULL, VARCHAR(50)
  - Properties: Must contain only alphabetic characters and spaces.
- **Description:** text
  - Constraints: NULL
  - Properties: Optional description of the status.

Transaction

- **ID_Transaction:** int
  - Constraints: PRIMARY KEY, NOT NULL, AUTO_INCREMENT
  - Properties: Must be unique, automatically generated.
- **ID_User:** int

  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the user who made the transaction.

- **ID_Product:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the product in the transaction.

- **Payment_Method:** string
  - Constraints: NOT NULL, VARCHAR(50)
  - Properties: Indicates the payment method (e.g., "Card", "PayPal").

- **Amount:** decimal(10, 2)
  - Constraints: NOT NULL
  - Properties: Must be a positive value.

- **Date_Bought:** timestamp
  - Constraints: NOT NULL
  - Properties: Automatically set when the transaction is completed.

- **ID_Transaction_Status:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the transaction status.

Payment Method

- **ID_Payment_Method:** int
  - Constraints: PRIMARY KEY, NOT NULL, AUTO_INCREMENT
  - Properties: Must be unique, automatically generated.

- **Type:** string
  - Constraints: NOT NULL, VARCHAR(50)
  - Properties: Payment method type (e.g., "Credit Card").

- **Provider:** string
  - Constraints: NOT NULL, VARCHAR(50)
  - Properties: Payment provider (e.g., "Visa").

- **Card_Number:** string
  - Constraints: NOT NULL, VARCHAR(16)

- Properties: Must be a valid card number, only digits allowed.

- **Due_Date:** date
  - Constraints: NOT NULL
  - Properties: Must be a valid expiration date.
- **Owner:** string
  - Constraints: NOT NULL, VARCHAR(100)
  - Properties: Name of the cardholder.

Transaction Status

- **ID_Payment_Status:** int
  - Constraints: PRIMARY KEY, NOT NULL, AUTO_INCREMENT
  - Properties: Must be unique, automatically generated.
- **Name:** string
  - Constraints: NOT NULL, VARCHAR(50)
  - Properties: Must contain only alphabetic characters and spaces.
- **Description:** text
  - Constraints: NULL
  - Properties: Optional description of the status.

Receipt

- **ID_Receipt:** int
  - Constraints: PRIMARY KEY, NOT NULL, AUTO_INCREMENT
  - Properties: Must be unique, automatically generated.
- **ID_Transaction:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the associated transaction.
- **ID_User:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the user associated with the receipt.
- **Date_Created:** timestamp

  - Constraints: NOT NULL
  - Properties: Automatically set when the receipt is created.

- **Details:** text
  - Constraints: NOT NULL
  - Properties: Detailed information of the transaction.
- **Amount_Paid:** decimal(10, 2)
  - Constraints: NOT NULL
  - Properties: Must be a positive value.
- **Taxes:** decimal(10, 2)
  - Constraints: NOT NULL
  - Properties: Must be a positive value.

Favorites List

- **ID_Favorite_List:** int
  - Constraints: PRIMARY KEY, NOT NULL, AUTO_INCREMENT
  - Properties: Must be unique, automatically generated.
- **ID_User:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the user who owns the list.
- **Name:** string
  - Constraints: NOT NULL, VARCHAR(100)
  - Properties: Name of the favorites list, must contain alphabetic characters.
- **Date_Created:** timestamp
  - Constraints: NOT NULL
  - Properties: Automatically set when the list is created.

Seller

- **ID_Seller:** int
  - Constraints: PRIMARY KEY, NOT NULL, AUTO_INCREMENT
  - Properties: Must be unique, automatically generated.
- **First_Name:** string
  - Constraints: NOT NULL, VARCHAR(100)
  - Properties: Must contain only alphabetic characters, minimum length of 2 characters.
- **Last_Name:** string
  - Constraints: NOT NULL, VARCHAR(100)
  - Properties: Must contain only alphabetic characters, minimum length of 2 characters.
- **Email:** string
  - Constraints: NOT NULL, UNIQUE, VARCHAR(100)
  - Properties: Must follow valid email format (e.g., name@domain.com).
- **Address:** string
  - Constraints: NOT NULL, VARCHAR(200)
  - Properties: Must be a valid address.
- **Phone:** string
  - Constraints: NOT NULL, VARCHAR(15)
  - Properties: Must be a valid phone number.
- **Date_Register:** timestamp
  - Constraints: NOT NULL
  - Properties: Automatically set when the seller registers.
- **Password:** string
  - Constraints: NOT NULL, VARCHAR(100)
  - Properties: Must be securely hashed, minimum length of 8 characters.
- **Date_Birth:** date
  - Constraints: NOT NULL
  - Properties: Must be a valid date.

- **ID_Account_Status:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the seller's account status.
- **Rating:** decimal(2, 1)
  - Constraints: NOT NULL
  - Properties: Must be between 0.0 and 5.0.
- **Number_Sales:** int
  - Constraints: NOT NULL
  - Properties: Total number of sales made by the seller.

Store

- **ID_Store:** int
  - Constraints: PRIMARY KEY, NOT NULL, AUTO_INCREMENT
  - Properties: Must be unique, automatically generated.
- **ID_Seller:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the seller who owns the store.
- **Name:** string
  - Constraints: NOT NULL, VARCHAR(100)
  - Properties: Must be unique and descriptive.
- **Date_Created:** timestamp
  - Constraints: NOT NULL
  - Properties: Automatically set when the store is created.
- **Number_Products:** int
  - Constraints: NOT NULL
  - Properties: Number of products in the store.
- **Rating:** decimal(2, 1)
  - Constraints: NOT NULL
  - Properties: Must be between 0.0 and 5.0.

Client Report

- **ID_Report:** int
  - Constraints: PRIMARY KEY, NOT NULL, AUTO_INCREMENT
  - Properties: Must be unique, automatically generated.
- **ID_User:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the user who submitted the report.
- **ID_Delivery:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the delivery related to the report.
- **ID_Category_Report:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the report category.
- **Description:** text
  - Constraints: NOT NULL
  - Properties: Detailed description of the report.
- **Date_Report:** timestamp
  - Constraints: NOT NULL
  - Properties: Automatically set when the report is created.
- **ID_Status_Report:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the report status.

Category Report

- **ID_Category_Report:** int
  - Constraints: PRIMARY KEY, NOT NULL, AUTO_INCREMENT
  - Properties: Must be unique, automatically generated.
- **Name:** string
  - Constraints: NOT NULL, VARCHAR(100)
  - Properties: Must contain only alphabetic characters and spaces.

- **Description:** text
  - Constraints: NULL
  - Properties: Optional description of the category.

Status Report

- **ID_Status_Report:** int
  - Constraints: PRIMARY KEY, NOT NULL, AUTO_INCREMENT
  - Properties: Must be unique, automatically generated.
- **Name:** string
  - Constraints: NOT NULL, VARCHAR(50)
  - Properties: Must contain only alphabetic characters and spaces.
- **Description:** text
  - Constraints: NULL
  - Properties: Optional description of the status.

User/Delivery Provider

- **ID_User:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the user involved in the delivery provider.
- **ID_Provider_Delivery:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the delivery provider.
- **Date_Assigned:** timestamp
  - Constraints: NULL
  - Properties: Tracks when the user was assigned to the delivery provider.

User/Seller

- **ID_User:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the user.
- **ID_Seller:** int

  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the seller.
- **Relationship_Type:** string
  - Constraints: NULL, VARCHAR(50)
  - Properties: Describes the user-seller relationship.
- **Date_Assigned:** timestamp
  - Constraints: NULL
  - Properties: Tracks when the user was connected to the seller.

Delivery/Product

- **ID_Delivery:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the delivery.
- **ID_Product:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the product.
- **Quantity:** int
  - Constraints: NULL
  - Properties: Tracks the quantity of the product in the delivery.

Favorites List/Seller

- **ID_Favorite_List:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the favorites list.
- **ID_Seller:** int
  - Constraints: FOREIGN KEY, NOT NULL
  - Properties: References the seller.
- **Date_Favorited:** timestamp
  - Constraints: NULL
  - Properties: Tracks when the seller was added to the favorites list.

3.11 Second Iteration

A second iteration is done to see the effectiveness of the entity relationship model according to the needs.

3.11.1 Deleted Entities

Seller: It is too similar to the users entity.
Receipts: The objective of this entity was to relate the users entity and the products, however the "transaction" entity already fulfills that objective.
UserSeller: Since the "seller" entity is removed, there is no point in making a relationship with something that does not exist.
DeliveryProduct:Resulta que es una relacion uno a muchos considerando al producto como un objeto unico y fisico, no como un identificador.

3.12 Modified entities

ShoppingCart: Changed the card code and expiration date attribute for security reasons, replaced by the token attribute (INT NOT NULL)
And all foreing key's entity added to end "_FK"

3.13 Added entity

Address: Since many entities had "address" as VARCHAR, I would say it is more scalable to have it as an entity.

- **ID_Address**: int
  - **Constraints**: PRIMARY KEY, AUTO_INCREMENT, NOT NULL
  - **Properties**: Unique identifier for each address.
- **Street**: varchar(100)
  - **Constraints**: NOT NULL
  - **Properties**: Stores the street name and number.
- **City**: varchar(50)
  - **Constraints**: NOT NULL
  - **Properties**: Stores the city name.
- **State**: varchar(50)
  - **Constraints**: NOT NULL
  - **Properties**: Stores the state or region name.
- **Country**: varchar(50)

  - **Constraints**: NOT NULL
  - **Properties**: Stores the country name.

- **ZipCode**: varchar(20)
  - **Constraints**: NOT NULL
  - **Properties**: Stores the postal or ZIP code.

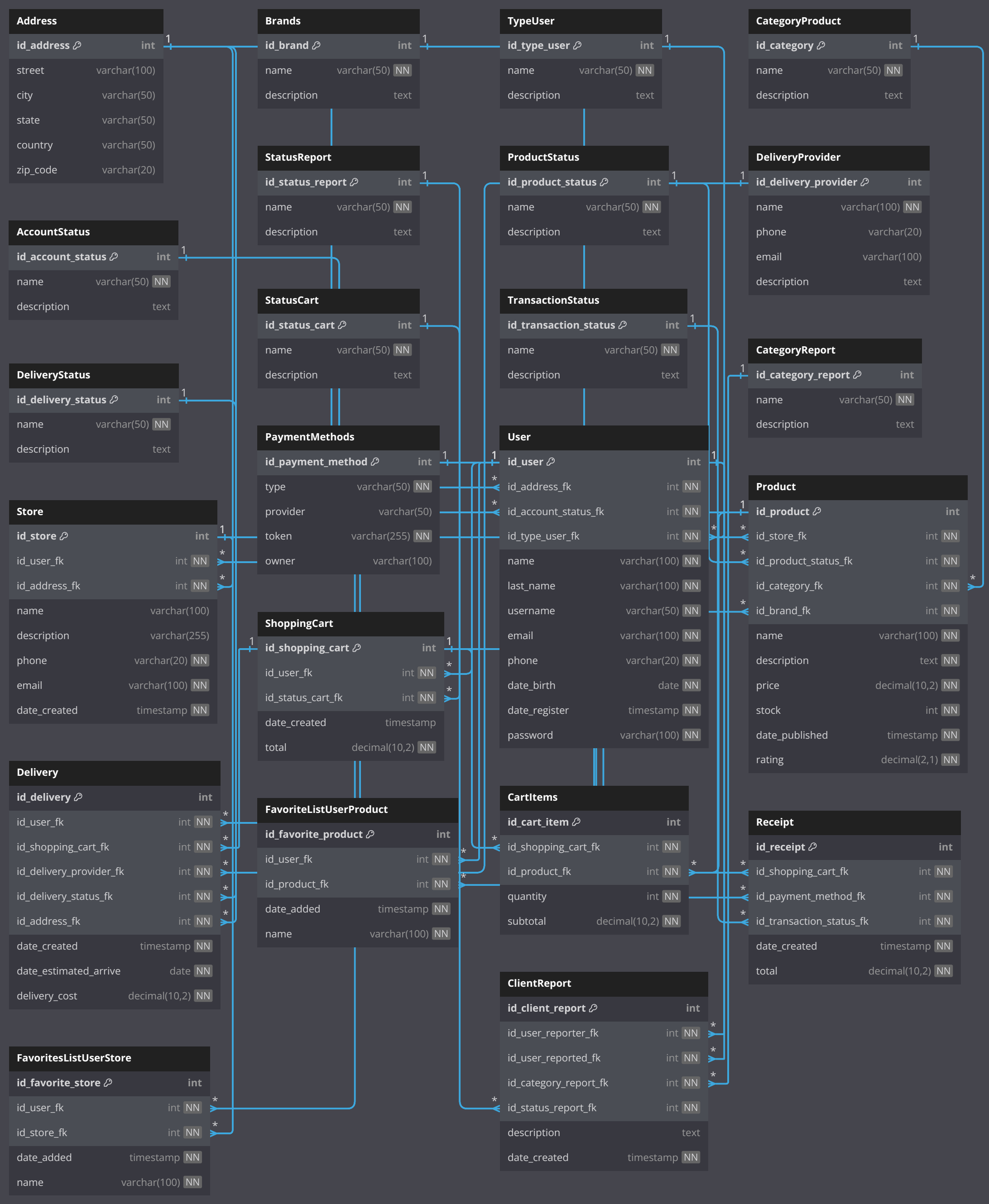
CategoryProduct: For more flexibility in category products.

- **ID_Category**: int
  - **Constraints**: PRIMARY KEY, AUTO_INCREMENT, NOT NULL
  - **Properties**: Unique identifier for each category.
- **Name**: varchar(50)
  - **Constraints**: NOT NULL
  - **Properties**: Name of the category.
- **Description**: text
  - **Constraints**: NULL
  - **Properties**: Description of the category.

Brands: It will be necessary for queries by brand.

- **ID_Brand**: int
  - **Constraints**: PRIMARY KEY, AUTO_INCREMENT, NOT NULL
  - **Properties**: Unique identifier for each brand.
- **Name**: varchar(50)
  - **Constraints**: NOT NULL
  - **Properties**: Name of the brand.
- **Description**: text
  - **Constraints**: NULL
  - **Properties**: Description of the brand.

4 Final Design



dbdiagram.io

5 Data Manipulation Language (DML)

In Project MercadoLbreXS, DML operations are used to modify the data stored in the database. These operations include inserting new records, updating existing records, and deleting records. The primary DML commands used in this project are:

- **INSERT**: Adds new records to a table.
- **UPDATE**: Modifies existing records in a table.
- **DELETE**: Removes records from a table.

Each entity in the project, such as users, products, orders, and more, has corresponding CRUD (Create, Read, Update, Delete) operations implemented using DML commands. These operations are encapsulated in the CRUD classes and exposed through the FastAPI endpoints.

6 Data Query Language (DQL)

DQL operations are used to query and retrieve data from the database. The primary DQL command used in this project is:

SELECT: Retrieves records from one or more tables. DQL commands are used to fetch data based on specific criteria, such as retrieving a user by their ID, getting all products, or finding orders within a certain date range. These queries are implemented in the CRUD classes and are accessible through the FastAPI endpoints.

7 Entities with CRUD operations

7.1 AccountStatusCRUD

7.1.1 create

Query:

```
INSERT INTO AccountStatus (name, description)
VALUES (%s, %s);
```

Parameters:

- `status.name` - The name of the account status.
- `status.description` - The description of the account status.

7.1.2 update

Query:

```
UPDATE AccountStatus
SET name = %s, description = %s
WHERE id_account_status = %s;
```

Parameters:

- `status.name` - The updated name of the account status.
- `status.description` - The updated description of the account status.
- `status.id_account_status` - The ID of the account status to update.

7.1.3 delete

Query:

```
DELETE FROM AccountStatus
WHERE id_account_status = %s;
```

Parameters:

- `status_id` - The ID of the account status to delete.

7.1.4 get_by_id

Query:

```
SELECT * FROM AccountStatus
WHERE id_account_status = %s;
```

Parameters:

- `status_id` - The ID of the account status to retrieve.

7.1.5 get_all

Query:

```
SELECT * FROM AccountStatus;
```

Parameters:

- None

7.2 AddressCRUD

7.2.1 create

Query:

```
INSERT INTO Address (street, city, state, country, zip_code)
VALUES (%s, %s, %s, %s, %s);
```

Parameters:

- `address.street` - The street name.
- `address.city` - The city name.
- `address.state` - The state name.
- `address.country` - The country name.
- `address.zip_code` - The zip code.

7.2.2 update

Query:

```
UPDATE Address
SET street = %s, city = %s, state = %s, country = %s, zip_code = %s
WHERE id_address = %s;
```

Parameters:

- `address.street` - The updated street.
- `address.city` - The updated city.
- `address.state` - The updated state.
- `address.country` - The updated country.
- `address.zip_code` - The updated zip code.
- `address.id_address` - The ID of the address to update.

7.2.3 delete

Query:

```
DELETE FROM Address
WHERE id_address = %s;
```

Parameters:

- `address_id` - The ID of the address to delete.

7.2.4 get_by_id

Query:

```
SELECT * FROM Address
WHERE id_address = %s;
```

Parameters:

- `address_id` - The ID of the address to retrieve.

7.2.5 get_all

Query:

```
SELECT * FROM Address;
```

Parameters:

- None

7.2.6 get_by_city

Query:

```
SELECT * FROM Address
WHERE city = %s;
```

Parameters:

- `city` - The city to filter addresses by.

7.2.7 get_by_state

Query:

```
SELECT * FROM Address
WHERE state = %s;
```

Parameters:

- `state` - The state to filter addresses by.

7.2.8 get_by_country

Query:

```
SELECT * FROM Address
WHERE country = %s;
```

Parameters:

- `country` - The country to filter addresses by.

7.2.9 get_by_zip_code

Query:

```
SELECT * FROM Address
WHERE zip_code = %s;
```

Parameters:

- `zip_code` - The zip code to filter addresses by.

7.2.10 get_by_city_and_state

Query:

```
SELECT * FROM Address
WHERE city = %s AND state = %s;
```

Parameters:

- `city` - The city to filter addresses by.
- `state` - The state to filter addresses by.

7.2.11 get_by_zip_code_and_country

Query:

```
SELECT * FROM Address
WHERE zip_code = %s AND country = %s;
```

Parameters:

- `zip_code` - The zip code to filter addresses by.
- `country` - The country to filter addresses by.

7.2.12 get_by_zip_code_and_state

Query:

```
SELECT * FROM Address
WHERE zip_code = %s AND state = %s;
```

Parameters:

- `zip_code` - The zip code to filter addresses by.
- `state` - The state to filter addresses by.

7.2.13 get_by_country_and_state_and_street

Query:

```
SELECT * FROM Address
WHERE country = %s AND state = %s AND street = %s;
```

Parameters:

- `country` - The country to filter addresses by.
- `state` - The state to filter addresses by.
- `street` - The street name to filter addresses by.

7.2.14 get_address_with_user

Query:

```
SELECT Address.*, Users.name AS user_name, Users.email AS user_email
FROM Address
JOIN Users ON Address.id_address = Users.id_address_fk
WHERE Address.id_address = %s;
```

Parameters:

- `address_id` - The ID of the address to retrieve, along with associated user information.

7.3 BrandCRUD

7.3.1 create

Query:

```
INSERT INTO Brands (name, description)
VALUES (%s, %s);
```

Parameters:

- `brand.name` - The name of the brand.
- `brand.description` - The description of the brand.

7.3.2 update

Query:

```
UPDATE Brands
SET name = %s, description = %s
WHERE id_brand = %s;
```

Parameters:

- `brand.name` - The updated name of the brand.
- `brand.description` - The updated description of the brand.
- `brand.id_brand` - The ID of the brand to update.

7.3.3 delete

Query:

```
DELETE FROM Brands
WHERE id_brand = %s;
```

Parameters:

- `brand_id` - The ID of the brand to delete.

7.3.4 get_by_id

Query:

```
SELECT * FROM Brands
WHERE id_brand = %s;
```

Parameters:

- `brand_id` - The ID of the brand to retrieve.

7.3.5 get_all

Query:

```
SELECT * FROM Brands;
```

Parameters:

- None

7.3.6 get_by_name

Query:

```
SELECT * FROM Brands
WHERE name = %s;
```

Parameters:

- `name` - The name of the brand to search for.

7.4 CartItemCRUD

7.4.1 create

Query:

```
INSERT INTO CartItems (id_shopping_cart_fk, id_product_fk, quantity,
    subtotal)
VALUES (%s, %s, %s, %s);
```

Parameters:

- `item.id_shopping_cart_fk` - The shopping cart ID.
- `item.id_product_fk` - The product ID.
- `item.quantity` - The quantity of the product in the cart.
- `item.subtotal` - The subtotal price for the product in the cart.

7.4.2 update

Query:

```
UPDATE CartItems
SET id_shopping_cart_fk = %s, id_product_fk = %s, quantity = %s,
   subtotal = %s
WHERE id_cart_item = %s;
```

Parameters:

- `item.id_shopping_cart_fk` - The updated shopping cart ID.
- `item.id_product_fk` - The updated product ID.
- `item.quantity` - The updated quantity.
- `item.subtotal` - The updated subtotal price.
- `item.id_cart_item` - The ID of the cart item to update.

7.4.3 delete

Query:

```
DELETE FROM CartItems
WHERE id_cart_item = %s;
```

Parameters:

- `item_id` - The ID of the cart item to delete.

7.4.4 get_by_id

Query:

```
SELECT * FROM CartItems
WHERE id_cart_item = %s;
```

Parameters:

- `item_id` - The ID of the cart item to retrieve.

7.4.5 get_all

Query:

```
SELECT * FROM CartItems;
```

Parameters:

- None

7.4.6 get_by_shopping_cart_id

Query:

```
SELECT * FROM CartItems
WHERE id_shopping_cart_fk = %s;
```

Parameters:

- `cart_id` - The shopping cart ID to filter cart items.

7.4.7 get_item_with_product

Query:

```
SELECT CartItems.*, Product.name AS product_name, Product.
   description AS product_description
FROM CartItems
JOIN Product ON CartItems.id_product_fk = Product.id_product
WHERE CartItems.id_cart_item = %s;
```

Parameters:

- `item_id` - The ID of the cart item to retrieve along with product details.

7.4.8 get_item_with_cart

Query:

```
SELECT CartItems.*, ShoppingCart.total AS cart_total, ShoppingCart.
   date_created AS cart_date_created
FROM CartItems
JOIN ShoppingCart ON CartItems.id_shopping_cart_fk = ShoppingCart.
   id_shopping_cart
WHERE CartItems.id_cart_item = %s;
```

Parameters:

- `item_id` - The ID of the cart item to retrieve along with cart details.

7.5 CategoryReportCRUD

7.5.1 create

Query:

```
INSERT INTO CategoryReport (name, description)
VALUES (%s, %s);
```

Parameters:

- `category.name` - The name of the category report.
- `category.description` - The description of the category report.

7.5.2 update

Query:

```
UPDATE CategoryReport
SET name = %s, description = %s
WHERE id_category_report = %s;
```

Parameters:

- `category.name` - The updated name of the category report.
- `category.description` - The updated description of the category report.
- `category.id_category_report` - The ID of the category report to update.

7.5.3 delete

Query:

```
DELETE FROM CategoryReport
WHERE id_category_report = %s;
```

Parameters:

- `category_id` - The ID of the category report to delete.

7.5.4 get_by_id

Query:

```
SELECT * FROM CategoryReport
WHERE id_category_report = %s;
```

Parameters:

- `category_id` - The ID of the category report to retrieve.

7.5.5 get_all

Query:

```
SELECT * FROM CategoryReport;
```

Parameters:

- None

7.6 ClientReportCRUD

7.6.1 create

Query:

```
INSERT INTO ClientReport (id_user_reporter_fk, id_user_reported_fk,
id_category_report_fk, id_status_report_fk, description,
   date_created)
VALUES (%s, %s, %s, %s, %s, %s);
```

Parameters:

- `report.id_user_reporter_fk` - The ID of the user who made the report.
- `report.id_user_reported_fk` - The ID of the user being reported.
- `report.id_category_report_fk` - The category of the report.
- `report.id_status_report_fk` - The status of the report.
- `report.description` - The description of the report.
- `report.date_created` - The date the report was created.

7.6.2 update

Query:

```
UPDATE ClientReport
SET id_user_reporter_fk = %s, id_user_reported_fk = %s,
id_category_report_fk = %s, id_status_report_fk = %s,
description = %s, date_created = %s
WHERE id_client_report = %s;
```

Parameters:

- Same as `create`, with the addition of:
- `report.id_client_report` - The ID of the report to update.

7.6.3 delete

Query:

```
DELETE FROM ClientReport
WHERE id_client_report = %s;
```

Parameters:

- `report_id` - The ID of the report to delete.

7.6.4 get_by_id

Query:

```
SELECT * FROM ClientReport
WHERE id_client_report = %s;
```

Parameters:

- `report_id` - The ID of the report to retrieve.

7.6.5 get_all

Query:

```
SELECT * FROM ClientReport;
```

Parameters:

- None

7.6.6 get_report_with_user

Query:

```
SELECT ClientReport.*, Users.name AS user_name, Users.email AS
   user_email
FROM ClientReport
JOIN Users ON ClientReport.id_user_reporter_fk = Users.id_user
WHERE ClientReport.id_client_report = %s;
```

Parameters:

- `report_id` - The ID of the report to retrieve, along with the user details.

7.6.7 get_report_with_category

Query:

```
SELECT ClientReport.*, CategoryReport.name AS category_name,
CategoryReport.description AS category_description
FROM ClientReport
JOIN CategoryReport ON ClientReport.id_category_report_fk =
   CategoryReport.id_category_report
WHERE ClientReport.id_client_report = %s;
```

Parameters:

- `report_id` - The ID of the report to retrieve, along with the category details.

7.6.8 get_report_with_status

Query:

```
SELECT ClientReport.*, StatusReport.name AS status_name,
StatusReport.description AS status_description
FROM ClientReport
JOIN StatusReport ON ClientReport.id_status_report_fk = StatusReport
   .id_status_report
WHERE ClientReport.id_client_report = %s;
```

Parameters:

- `report_id` - The ID of the report to retrieve, along with the status details.

7.6.9 get_by_user_and_category

Query:

```
SELECT * FROM ClientReport
WHERE id_user_reporter_fk = %s AND id_category_report_fk = %s;
```

Parameters:

- `user_id` - The ID of the user who made the report.
- `category_id` - The ID of the report category.

7.6.10 get_by_user_and_status

Query:

```
SELECT * FROM ClientReport
WHERE id_user_reporter_fk = %s AND id_status_report_fk = %s;
```

Parameters:

- `user_id` - The ID of the user who made the report.
- `status_id` - The ID of the report status.

7.6.11 get_by_category_and_status

Query:

```
SELECT * FROM ClientReport
WHERE id_category_report_fk = %s AND id_status_report_fk = %s;
```

Parameters:

- `category_id` - The ID of the report category.
- `status_id` - The ID of the report status.

7.7 DeliveryProviderCRUD

7.7.1 create

Query:

```
INSERT INTO DeliveryProvider (name, phone, email, description)
VALUES (%s, %s, %s, %s);
```

Parameters:

- `provider.name` - The name of the delivery provider.
- `provider.phone` - The phone number of the provider.
- `provider.email` - The email of the provider.
- `provider.description` - The description of the provider.

7.7.2 update

Query:

```
UPDATE DeliveryProvider
SET name = %s, phone = %s, email = %s, description = %s
WHERE id_delivery_provider = %s;
```

Parameters:

- `provider.name` - The updated name of the provider.
- `provider.phone` - The updated phone number of the provider.
- `provider.email` - The updated email of the provider.
- `provider.description` - The updated description of the provider.
- `provider.id_delivery_provider` - The ID of the provider to update.

7.7.3 delete

Query:

```
DELETE FROM DeliveryProvider
WHERE id_delivery_provider = %s;
```

Parameters:

- `provider_id` - The ID of the provider to delete.

7.7.4 get_by_id

Query:

```
SELECT * FROM DeliveryProvider
WHERE id_delivery_provider = %s;
```

Parameters:

- `provider_id` - The ID of the provider to retrieve.

7.7.5 get_all

Query:

```
SELECT * FROM DeliveryProvider;
```

Parameters:

- None

7.7.6 get_by_name

Query:

```
SELECT * FROM DeliveryProvider
WHERE LOWER(name) LIKE LOWER(%s);
```

Parameters:

- `name` - The name (or partial name) of the provider to search for.

7.7.7 get_by_phone

Query:

```
SELECT * FROM DeliveryProvider
WHERE phone = %s;
```

Parameters:

- `phone` - The phone number of the provider.

7.7.8 get_by_email

Query:

```
SELECT * FROM DeliveryProvider
WHERE email = %s;
```

Parameters:

- `email` - The email of the provider.

7.8 DeliveryStatusCrud

7.8.1 create

Query:

```
INSERT INTO delivery_status (name, description)
VALUES (%s, %s)
```

Parameters:

- `delivery_status.name` - The name of the delivery status.
- `delivery_status.description` - The description of the delivery status.

7.8.2 update

Query:

```
UPDATE delivery_status
SET name = %s, description = %s
WHERE id_delivery_status = %s
```

Parameters:

- `delivery_status.name` - The updated name of the delivery status.
- `delivery_status.description` - The updated description of the delivery status.
- `delivery_status.id` - The ID of the delivery status to update.

7.8.3 delete

Query:

```
DELETE FROM delivery_status
WHERE id_delivery_status = %s
```

Parameters:

- `delivery_status_id` - The ID of the delivery status to delete.

7.8.4 get_by_id

Query:

```
SELECT * FROM delivery_status
WHERE id_delivery_status = %s
```

Parameters:

- `delivery_status_id` - The ID of the delivery status to retrieve.

7.8.5 get_all

Query:

```
SELECT * FROM delivery_status
```

Parameters:

- None

7.8.6 get_by_status_name

Query:

```
SELECT * FROM delivery_status
WHERE name LIKE %s
```

Parameters:

- `status_name` - The name (or partial name) of the delivery status to search for.

7.9 DeliveryCRUD

7.9.1 create

Query:

```
INSERT INTO Delivery (id_user_fk, id_shopping_cart_fk,
   id_delivery_provider_fk,
id_delivery_status_fk, id_address_fk, date_created,
   date_estimated_arrive, delivery_cost)
VALUES (%s, %s, %s, %s, %s, %s, %s, %s);
```

Parameters:

- `delivery.id_user_fk` - The ID of the user.
- `delivery.id_shopping_cart_fk` - The ID of the shopping cart.
- `delivery.id_delivery_provider_fk` - The ID of the delivery provider.
- `delivery.id_delivery_status_fk` - The ID of the delivery status.
- `delivery.id_address_fk` - The ID of the delivery address.
- `delivery.date_created` - The creation date of the delivery.
- `delivery.date_estimated_arrive` - The estimated arrival date.
- `delivery.delivery_cost` - The cost of the delivery.

7.9.2 update

Query:

```
UPDATE Delivery
SET id_user_fk = %s, id_shopping_cart_fk = %s,
   id_delivery_provider_fk = %s,
id_delivery_status_fk = %s, id_address_fk = %s, date_created = %s,
date_estimated_arrive = %s, delivery_cost = %s
WHERE id_delivery = %s;
```

Parameters:

- Same as `create`, with the addition of:
- `delivery.id_delivery` - The ID of the delivery to update.

7.9.3 delete

Query:

```
DELETE FROM Delivery
WHERE id_delivery = %s;
```

Parameters:

- `delivery_id` - The ID of the delivery to delete.

7.9.4 get_by_id

Query:

```
SELECT * FROM Delivery
WHERE id_delivery = %s;
```

Parameters:

- `delivery_id` - The ID of the delivery to retrieve.

7.9.5 get_all

Query:

```
SELECT * FROM Delivery;
```

Parameters:

- None

7.9.6 get_by_user_id

Query:

```
SELECT * FROM Delivery
WHERE id_user_fk = %s;
```

Parameters:

- `user_id` - The ID of the user.

7.9.7 get_by_status_id

Query:

```
SELECT * FROM Delivery
WHERE id_delivery_status_fk = %s;
```

Parameters:

- `status_id` - The ID of the delivery status.

7.9.8 get_by_provider_id

Query:

```
SELECT * FROM Delivery
WHERE id_delivery_provider_fk = %s;
```

Parameters:

- `provider_id` - The ID of the delivery provider.

7.9.9 get_delivery_with_user

Query:

```
SELECT Delivery.*, Users.name AS user_name, Users.email AS
   user_email
FROM Delivery
JOIN Users ON Delivery.id_user_fk = Users.id_user
WHERE Delivery.id_delivery = %s;
```

Parameters:

- `delivery_id` - The ID of the delivery to retrieve, along with user details.

7.9.10 get_delivery_with_status

Query:

```
SELECT Delivery.*, DeliveryStatus.name AS status_name,
   DeliveryStatus.description AS status_description
FROM Delivery
JOIN DeliveryStatus ON Delivery.id_delivery_status_fk =
   DeliveryStatus.id_delivery_status
WHERE Delivery.id_delivery = %s;
```

Parameters:

- `delivery_id` - The ID of the delivery to retrieve, along with status details.

7.9.11 get_by_user_and_status

Query:

```
SELECT * FROM Delivery
WHERE id_user_fk = %s AND id_delivery_status_fk = %s;
```

Parameters:

- `user_id` - The ID of the user.
- `status_id` - The ID of the delivery status.

7.9.12 get_by_provider_and_status

Query:

```
SELECT * FROM Delivery
WHERE id_delivery_provider_fk = %s AND id_delivery_status_fk = %s;
```

Parameters:

- `provider_id` - The ID of the delivery provider.
- `status_id` - The ID of the delivery status.

7.10 FavoriteListUserProductCRUD

7.10.1 create

Query:

```
INSERT INTO FavoriteListUsersProduct (id_user_fk, id_product_fk,
   date_added, name)
VALUES (%s, %s, %s, %s);
```

Parameters:

- `favorite.id_user_fk` - The ID of the user who added the product to favorites.
- `favorite.id_product_fk` - The ID of the product that was added.
- `favorite.date_added` - The date when the product was added to favorites.
- `favorite.name` - The name of the favorite list.

7.10.2 update

Query:

```
UPDATE FavoriteListUsersProduct
SET id_user_fk = %s, id_product_fk = %s, date_added = %s, name = %s
WHERE id_favorite_product = %s;
```

Parameters:

- `favorite.id_user_fk` - The updated user ID.
- `favorite.id_product_fk` - The updated product ID.
- `favorite.date_added` - The updated date of addition.
- `favorite.name` - The updated name of the favorite list.
- `favorite.id_favorite_product` - The ID of the favorite list item to update.

7.10.3 delete

Query:

```
DELETE FROM FavoriteListUsersProduct
WHERE id_favorite_product = %s;
```

Parameters:

- favorite_id - The ID of the favorite list item to delete.

7.10.4 get_by_id

Query:

```
SELECT * FROM FavoriteListUsersProduct
WHERE id_favorite_product = %s;
```

Parameters:

- favorite_id - The ID of the favorite list item to retrieve.

7.10.5 get_all

Query:

```
SELECT * FROM FavoriteListUsersProduct;
```

Parameters:

- None

7.11 FavoriteListUserStoreCRUD

7.11.1 create

Query:

```
INSERT INTO FavoriteListUsersStore (id_user_fk, id_store_fk,
   date_added, name)
VALUES (%s, %s, %s, %s);
```

Parameters:

- favorite.id_user_fk - The ID of the user who added the store to favorites.
- favorite.id_store_fk - The ID of the store that was added.
- favorite.date_added - The date when the store was added to favorites.
- favorite.name - The name of the favorite list.

7.11.2 update

Query:

```
UPDATE FavoriteListUsersStore
SET id_user_fk = %s, id_store_fk = %s, date_added = %s, name = %s
WHERE id_favorite_store = %s;
```

Parameters:

- `favorite.id_user_fk` - The updated user ID.
- `favorite.id_store_fk` - The updated store ID.
- `favorite.date_added` - The updated date of addition.
- `favorite.name` - The updated name of the favorite list.
- `favorite.id_favorite_store` - The ID of the favorite list item to update.

7.11.3 delete

Query:

```
DELETE FROM FavoriteListUsersStore
WHERE id_favorite_store = %s;
```

Parameters:

- `favorite_id` - The ID of the favorite list item to delete.

7.11.4 get_by_id

Query:

```
SELECT * FROM FavoriteListUsersStore
WHERE id_favorite_store = %s;
```

Parameters:

- `favorite_id` - The ID of the favorite list item to retrieve.

7.11.5 get_all

Query:

```
SELECT * FROM FavoriteListUsersStore;
```

Parameters:

- None

7.12 PaymentMethodCRUD

7.12.1 create

Query:

```
INSERT INTO PaymentMethods (type, provider, token, owner)
VALUES (%s, %s, %s, %s);
```

Parameters:

- `payment_method.type` - The type of payment method (e.g., credit card, PayPal).
- `payment_method.provider` - The provider of the payment method (e.g., Visa, MasterCard).
- `payment_method.token` - The secure token representing the payment method.
- `payment_method.owner` - The owner of the payment method.

7.12.2 update

Query:

```
UPDATE PaymentMethods
SET type = %s, provider = %s, token = %s, owner = %s
WHERE id_payment_method = %s;
```

Parameters:

- `payment_method.type` - The updated type of payment method.
- `payment_method.provider` - The updated provider of the payment method.
- `payment_method.token` - The updated secure token.
- `payment_method.owner` - The updated owner.
- `payment_method.id_payment_method` - The ID of the payment method to update.

7.12.3 delete

Query:

```
DELETE FROM PaymentMethods
WHERE id_payment_method = %s;
```

Parameters:

- `payment_method_id` - The ID of the payment method to delete.

7.12.4 get_by_id

Query:

```
SELECT * FROM PaymentMethods
WHERE id_payment_method = %s;
```

Parameters:

- `payment_method_id` - The ID of the payment method to retrieve.

7.12.5 get_all

Query:

```
SELECT * FROM PaymentMethods;
```

Parameters:

- None

7.12.6 get_by_type

Query:

```
SELECT * FROM PaymentMethods
WHERE type = %s;
```

Parameters:

- `type` - The type of payment method to search for.

7.12.7 get_by_provider

Query:

```
SELECT * FROM PaymentMethods
WHERE provider = %s;
```

Parameters:

- `provider` - The provider of the payment method to search for.

7.12.8 get_by_token

Query:

```
SELECT * FROM PaymentMethods
WHERE token = %s;
```

Parameters:

- `token` - The token associated with the payment method.

7.13 ProductStatusCrud

7.13.1 create

Query:

```
INSERT INTO product_status (id_product_status, name, description)
VALUES (%s, %s, %s);
```

Parameters:

- `product_status.id_product_status` - The ID of the product status.
- `product_status.name` - The name of the product status.
- `product_status.description` - The description of the product status.

7.13.2 update

Query:

```
UPDATE product_status
SET name = %s, description = %s
WHERE id_product_status = %s;
```

Parameters:

- `product_status.name` - The updated name of the product status.
- `product_status.description` - The updated description of the product status.
- `product_status.id_product_status` - The ID of the product status to update.

7.13.3 delete

Query:

```
DELETE FROM product_status
WHERE id_product_status = %s;
```

Parameters:

- `product_status_id` - The ID of the product status to delete.

7.13.4 get_by_id

Query:

```
SELECT * FROM product_status
WHERE id_product_status = %s;
```

Parameters:

- `product_status_id` - The ID of the product status to retrieve.

7.13.5 get_all

Query:

```
SELECT * FROM product_status;
```

Parameters:

- None

7.13.6 get_by_name

Query:

```
SELECT * FROM product_status
WHERE name LIKE %s;
```

Parameters:

- `status_name` - The name (or partial name) of the product status to search for.

7.14 ProductCRUD

7.14.1 create

Query:

```
INSERT INTO Product (id_store_fk, id_product_status_fk,
   id_category_fk,
id_brand_fk, name, description, price, stock, date_published, rating
   )
VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s);
```

Parameters:

- `product.id_store_fk` - The store ID where the product is listed.
- `product.id_product_status_fk` - The product status ID.
- `product.id_category_fk` - The category ID of the product.
- `product.id_brand_fk` - The brand ID of the product.
- `product.name` - The name of the product.
- `product.description` - The description of the product.
- `product.price` - The price of the product.
- `product.stock` - The stock quantity of the product.
- `product.date_published` - The date the product was published.
- `product.rating` - The product's rating.

7.14.2 update

Query:

```
UPDATE Product
SET id_store_fk = %s, id_product_status_fk = %s, id_category_fk = %s
    ,
id_brand_fk = %s, name = %s, description = %s, price = %s, stock = %
    s,
date_published = %s, rating = %s
WHERE id_product = %s;
```

Parameters:

- Same as `create`, with the addition of:
- `product.id_product` - The ID of the product to update.

7.14.3 delete

Query:

```
DELETE FROM Product
WHERE id_product = %s;
```

Parameters:

- `product_id` - The ID of the product to delete.

7.14.4 get_by_id

Query:

```
SELECT * FROM Product
WHERE id_product = %s;
```

Parameters:

- `product_id` - The ID of the product to retrieve.

7.14.5 get_all

Query:

```
SELECT * FROM Product;
```

Parameters:

- None

7.14.6 get_by_store_id

Query:

```
SELECT * FROM Product
WHERE id_store_fk = %s;
```

Parameters:

- `store_id` - The ID of the store to filter products.

7.14.7 get_by_category_id

Query:

```
SELECT * FROM Product
WHERE id_category_fk = %s;
```

Parameters:

- `category_id` - The ID of the category to filter products.

7.14.8 get_by_brand_id

Query:

```
SELECT * FROM Product
WHERE id_brand_fk = %s;
```

Parameters:

- `brand_id` - The ID of the brand to filter products.

7.14.9 get_by_status_id

Query:

```
SELECT * FROM Product
WHERE id_product_status_fk = %s;
```

Parameters:

- `status_id` - The ID of the product status to filter products.

7.14.10 get_product_with_store

Query:

```
SELECT Product.*, Store.name AS store_name, Store.description AS
   store_description
FROM Product
JOIN Store ON Product.id_store_fk = Store.id_store
WHERE Product.id_product = %s;
```

7.14.11 get_product_with_category

Query:

```
SELECT Product.*, CategoryProduct.name AS category_name,
CategoryProduct.description AS category_description
FROM Product
JOIN CategoryProduct ON Product.id_category_fk = CategoryProduct.
   id_category
WHERE Product.id_product = %s;
```

7.14.12 get_product_with_brand

Query:

```
SELECT Product.*, Brands.name AS brand_name, Brands.description AS
   brand_description
FROM Product
JOIN Brands ON Product.id_brand_fk = Brands.id_brand
WHERE Product.id_product = %s;
```

7.14.13 get_product_with_status

Query:

```
SELECT Product.*, ProductStatus.name AS status_name, ProductStatus.
   description AS status_description
FROM Product
JOIN ProductStatus ON Product.id_product_status_fk = ProductStatus.
   id_product_status
WHERE Product.id_product = %s;
```

7.14.14 get_by_store_and_category

Query:

```
SELECT * FROM Product
WHERE id_store_fk = %s AND id_category_fk = %s;
```

7.14.15 get_by_brand_and_status

Query:

```
SELECT * FROM Product
WHERE id_brand_fk = %s AND id_product_status_fk = %s;
```

7.14.16 get_by_store_and_status

Query:

```
SELECT * FROM Product
WHERE id_store_fk = %s AND id_product_status_fk = %s;
```

7.14.17 get_by_category_and_brand

Query:

```
SELECT * FROM Product
WHERE id_category_fk = %s AND id_brand_fk = %s;
```

7.15 ReceiptCRUD

7.15.1 create

Query:

```
INSERT INTO Receipt (id_shopping_cart_fk, id_payment_method_fk,
   id_transaction_status_fk, date_created, total)
VALUES (%s, %s, %s, %s, %s);
```

Parameters:

- `receipt.id_shopping_cart_fk` - The shopping cart ID associated with the receipt.
- `receipt.id_payment_method_fk` - The payment method ID used for the transaction.
- `receipt.id_transaction_status_fk` - The transaction status ID.
- `receipt.date_created` - The date when the receipt was created.
- `receipt.total` - The total amount for the transaction.

7.15.2 update

Query:

```
UPDATE Receipt
SET id_shopping_cart_fk = %s, id_payment_method_fk = %s,
   id_transaction_status_fk = %s, date_created = %s, total = %s
WHERE id_receipt = %s;
```

Parameters:

- Same as `create`, with the addition of:
- `receipt.id_receipt` - The ID of the receipt to update.

7.15.3 delete

Query:

```
DELETE FROM Receipt
WHERE id_receipt = %s;
```

Parameters:

- `receipt_id` - The ID of the receipt to delete.

7.15.4 get_by_id

Query:

```
SELECT * FROM Receipt
WHERE id_receipt = %s;
```

Parameters:

- `receipt_id` - The ID of the receipt to retrieve.

7.15.5 get_all

Query:

```
SELECT * FROM Receipt;
```

Parameters:

- None

7.15.6 get_receipt_with_cart

Query:

```
SELECT Receipt.*, ShoppingCart.total AS cart_total, ShoppingCart.
   date_created AS cart_date_created
FROM Receipt
JOIN ShoppingCart ON Receipt.id_shopping_cart_fk = ShoppingCart.
   id_shopping_cart
WHERE Receipt.id_receipt = %s;
```

Parameters:

- `receipt_id` - The ID of the receipt to retrieve, along with shopping cart details.

7.15.7 get_receipt_with_payment_method

Query:

```
SELECT Receipt.*, PaymentMethods.type AS payment_type,
   PaymentMethods.provider AS payment_provider
FROM Receipt
JOIN PaymentMethods ON Receipt.id_payment_method_fk = PaymentMethods
   .id_payment_method
WHERE Receipt.id_receipt = %s;
```

7.15.8 get_receipt_with_transaction_status

Query:

```
SELECT Receipt.*, TransactionStatus.name AS status_name,
   TransactionStatus.description AS status_description
FROM Receipt
JOIN TransactionStatus ON Receipt.id_transaction_status_fk =
   TransactionStatus.id_transaction_status
WHERE Receipt.id_receipt = %s;
```

7.15.9 get_by_cart_and_payment_method

Query:

```
SELECT * FROM Receipt
WHERE id_shopping_cart_fk = %s AND id_payment_method_fk = %s;
```

Parameters:

- `cart_id` - The shopping cart ID.
- `payment_method_id` - The payment method ID.

7.15.10 get_by_transaction_status_and_total

Query:

```
SELECT * FROM Receipt
WHERE id_transaction_status_fk = %s AND total = %s;
```

Parameters:

- `status_id` - The transaction status ID.
- `total` - The total transaction amount.

7.15.11 get_by_date_created

Query:

```
SELECT * FROM Receipt
WHERE date_created = %s;
```

Parameters:

- `date_created` - The creation date of the receipt.

7.15.12 get_by_total

Query:

```
SELECT * FROM Receipt
WHERE total = %s;
```

Parameters:

- `total` - The total transaction amount.

7.15.13 get_by_cart_and_status

Query:

```
SELECT * FROM Receipt
WHERE id_shopping_cart_fk = %s AND id_transaction_status_fk = %s;
```

Parameters:

- `cart_id` - The shopping cart ID.
- `status_id` - The transaction status ID.

7.15.14 get_by_payment_method_and_status

Query:

```
SELECT * FROM Receipt
WHERE id_payment_method_fk = %s AND id_transaction_status_fk = %s;
```

Parameters:

- `payment_method_id` - The payment method ID.
- `status_id` - The transaction status ID.

7.16 ShoppingCartCRUD

7.16.1 create

Query:

```
INSERT INTO ShoppingCart (id_status_cart_fk, date_created, total)
VALUES (%s, %s, %s);
```

Parameters:

- `cart.id_status_cart_fk` - The status ID of the shopping cart.
- `cart.date_created` - The date when the shopping cart was created.
- `cart.total` - The total amount in the shopping cart.

7.16.2 update

Query:

```
UPDATE ShoppingCart
SET id_status_cart_fk = %s, date_created = %s, total = %s
WHERE id_shopping_cart = %s;
```

Parameters:

- `cart.id_status_cart_fk` - The updated status ID.
- `cart.date_created` - The updated creation date.
- `cart.total` - The updated total amount.
- `cart.id_shopping_cart` - The ID of the shopping cart to update.

7.16.3 delete

Query:

```
DELETE FROM ShoppingCart
WHERE id_shopping_cart = %s;
```

Parameters:

- `cart_id` - The ID of the shopping cart to delete.

7.16.4 get_by_id

Query:

```
SELECT * FROM ShoppingCart
WHERE id_shopping_cart = %s;
```

Parameters:

- `cart_id` - The ID of the shopping cart to retrieve.

7.16.5 get_all

Query:

```
SELECT * FROM ShoppingCart;
```

Parameters:

- None

7.16.6 get_cart_with_user

Query:

```
SELECT ShoppingCart.*, Users.name AS user_name, Users.email AS
   user_email
FROM ShoppingCart
JOIN Users ON ShoppingCart.id_user_fk = Users.id_user
WHERE ShoppingCart.id_shopping_cart = %s;
```

7.16.7 get_cart_with_status

Query:

```
SELECT ShoppingCart.*, StatusCart.name AS status_name, StatusCart.
   description AS status_description
FROM ShoppingCart
JOIN StatusCart ON ShoppingCart.id_status_cart_fk = StatusCart.
   id_status_cart
WHERE ShoppingCart.id_shopping_cart = %s;
```

7.16.8 get_by_user_and_status

Query:

```
SELECT * FROM ShoppingCart
WHERE id_user_fk = %s AND id_status_cart_fk = %s;
```

7.16.9 get_by_date_and_total

Query:

```
SELECT * FROM ShoppingCart
WHERE date_created = %s AND total = %s;
```

7.16.10 get_by_user_id

Query:

```
SELECT * FROM ShoppingCart
WHERE id_user_fk = %s;
```

7.16.11 get_by_status_id

Query:

```
SELECT * FROM ShoppingCart
WHERE id_status_cart_fk = %s;
```

7.16.12 get_by_total

Query:

```
SELECT * FROM ShoppingCart
WHERE total = %s;
```

7.16.13 get_by_user_and_date

Query:

```
SELECT * FROM ShoppingCart
WHERE id_user_fk = %s AND date_created = %s;
```

7.16.14 get_by_status_and_total

Query:

```
SELECT * FROM ShoppingCart
WHERE id_status_cart_fk = %s AND total = %s;
```

7.17 StatusCartCrud

7.17.1 create

Query:

```
INSERT INTO status_cart (name, description)
VALUES (%s, %s);
```

Parameters:

- `status_cart.name` - The name of the cart status.
- `status_cart.description` - The description of the cart status.

7.17.2 update

Query:

```
UPDATE status_cart
SET name = %s, description = %s
WHERE id_status_cart = %s;
```

Parameters:

- `status_cart.name` - The updated name of the cart status.
- `status_cart.description` - The updated description of the cart status.
- `status_cart.id` - The ID of the cart status to update.

7.17.3 delete

Query:

```
DELETE FROM status_cart
WHERE id_status_cart = %s;
```

Parameters:

- `status_cart_id` - The ID of the cart status to delete.

7.17.4 get_by_id

Query:

```
SELECT * FROM status_cart
WHERE id_status_cart = %s;
```

Parameters:

- `status_cart_id` - The ID of the cart status to retrieve.

7.17.5 get_all

Query:

```
SELECT * FROM status_cart;
```

Parameters:

- None

7.17.6 get_by_status_name

Query:

```
SELECT * FROM status_cart
WHERE LOWER(name) LIKE LOWER(%s);
```

Parameters:

- `status_name` - The name (or partial name) of the cart status to search for.

7.18 StatusReportCrud

7.18.1 create

Query:

```
INSERT INTO status_report (name, description)
VALUES (%s, %s);
```

Parameters:

- `status_report.name` - The name of the report status.
- `status_report.description` - The description of the report status.

7.18.2 update

Query:

```
UPDATE status_report
SET name = %s, description = %s
WHERE id_status_report = %s;
```

Parameters:

- `status_report.name` - The updated name of the report status.
- `status_report.description` - The updated description of the report status.
- `status_report.id` - The ID of the report status to update.

7.18.3 delete

Query:

```
DELETE FROM status_report
WHERE id_status_report = %s;
```

Parameters:

- `status_report_id` - The ID of the report status to delete.

7.18.4 get_by_id

Query:

```
SELECT * FROM status_report
WHERE id_status_report = %s;
```

Parameters:

- `status_report_id` - The ID of the report status to retrieve.

7.18.5 get_all

Query:

```
SELECT * FROM status_report;
```

Parameters:

- None

7.18.6 get_by_name

Query:

```
SELECT * FROM status_report
WHERE name LIKE %s;
```

Parameters:

- `status_name` - The name (or partial name) of the report status to search for.

7.19 StoreCRUD

7.19.1 create

Query:

```
INSERT INTO Store (id_user_fk, id_address_fk, name, description,
   phone, email, date_created)
VALUES (%s, %s, %s, %s, %s, %s, %s);
```

Parameters:

- `store.id_user_fk` - The ID of the user who owns the store.
- `store.id_address_fk` - The address ID of the store.
- `store.name` - The name of the store.
- `store.description` - A description of the store.
- `store.phone` - The contact phone number of the store.
- `store.email` - The contact email of the store.
- `store.date_created` - The date when the store was created.

7.19.2 update

Query:

```
UPDATE Store
SET id_address_fk = %s, name = %s, description = %s, phone = %s,
   email = %s
WHERE id_store = %s;
```

Parameters:

- `store.id_address_fk` - The updated address ID.
- `store.name` - The updated store name.
- `store.description` - The updated store description.
- `store.phone` - The updated store phone number.
- `store.email` - The updated store email.
- `store.id_store` - The ID of the store to update.

7.19.3 delete

Query:

```
DELETE FROM Store
WHERE id_store = %s;
```

Parameters:

- `store_id` - The ID of the store to delete.

7.19.4 get_by_id

Query:

```
SELECT * FROM Store
WHERE id_store = %s;
```

Parameters:

- `store_id` - The ID of the store to retrieve.

7.19.5 get_all

Query:

```
SELECT * FROM Store;
```

Parameters:

- None

7.19.6 get_by_user_id

Query:

```
SELECT * FROM Store
WHERE id_user_fk = %s;
```

Parameters:

- `user_id` - The ID of the user who owns the store.

7.19.7 get_by_address_id

Query:

```
SELECT * FROM Store
WHERE id_address_fk = %s;
```

Parameters:

- `address_id` - The ID of the address associated with the store.

7.19.8 get_store_with_address

Query:

```
SELECT Store.*, Address.*
FROM Store
JOIN Address ON Store.id_address_fk = Address.id_address
WHERE Store.id_store = %s;
```

Parameters:

- `store_id` - The ID of the store to retrieve, along with its address details.

7.20 TransactionStatusCrud

7.20.1 create

Query:

```
INSERT INTO transaction_status (name, description)
VALUES (%s, %s);
```

Parameters:

- `transaction_status.name` - The name of the transaction status.
- `transaction_status.description` - The description of the transaction status.

7.20.2 update

Query:

```
UPDATE transaction_status
SET name = %s, description = %s
WHERE id_transaction_status = %s;
```

Parameters:

- `transaction_status.name` - The updated name of the transaction status.
- `transaction_status.description` - The updated description of the transaction status.
- `transaction_status.id` - The ID of the transaction status to update.

7.20.3 delete

Query:

```
DELETE FROM transaction_status
WHERE id_transaction_status = %s;
```

Parameters:

- `transaction_status_id` - The ID of the transaction status to delete.

7.20.4 get_by_id

Query:

```
SELECT * FROM transaction_status
WHERE id_transaction_status = %s;
```

Parameters:

- `transaction_status_id` - The ID of the transaction status to retrieve.

7.20.5 get_all

Query:

```
SELECT * FROM transaction_status;
```

Parameters:

- None

7.20.6 get_by_status_name

Query:

```
SELECT * FROM transaction_status
WHERE name LIKE %s;
```

Parameters:

- `status_name` - The name (or partial name) of the transaction status to search for.

7.21 TypeUserCRUD

7.21.1 create

Query:

```
INSERT INTO TypeUsers (name, description)
VALUES (%s, %s);
```

Parameters:

- `type_user.name` - The name of the user type.
- `type_user.description` - The description of the user type.

7.21.2 update

Query:

```
UPDATE TypeUsers
SET name = %s, description = %s
WHERE id_type_user = %s;
```

Parameters:

- `type_user.name` - The updated name of the user type.
- `type_user.description` - The updated description of the user type.
- `type_user.id_type_user` - The ID of the user type to update.

7.21.3 delete

Query:

```
DELETE FROM TypeUsers
WHERE id_type_user = %s;
```

Parameters:

- `type_user_id` - The ID of the user type to delete.

7.21.4 get_by_id

Query:

```
SELECT * FROM TypeUsers
WHERE id_type_user = %s;
```

Parameters:

- `type_user_id` - The ID of the user type to retrieve.

7.21.5 get_all

Query:

```
SELECT * FROM TypeUsers;
```

Parameters:

- None

7.21.6 get_by_name

Query:

```
SELECT * FROM TypeUsers
WHERE name = %s;
```

Parameters:

- `name` - The name of the user type to search for.

7.22 UsersCRUD

7.22.1 create

Query:

```
INSERT INTO Users (id_user, id_address_fk, id_account_status_fk,
   id_type_user_fk,
name, last_name, username, email, phone, date_birth, date_register,
   password)
VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s);
```

Parameters:

- User-related details including ID, address, status, type, name, contact, and login information.

7.22.2 update

Query:

```
UPDATE Users
SET id_address_fk = %s, id_account_status_fk = %s, id_type_user_fk =
    %s,
name = %s, last_name = %s, username = %s, email = %s, phone = %s,
date_birth = %s, date_register = %s, password = %s
WHERE id_user = %s;
```

7.22.3 delete

Query:

```
DELETE FROM Users
WHERE id_user = %s;
```

7.22.4 get_by_id

Query:

```
SELECT Users.*, AccountStatus.name AS status, TypeUsers.name AS role
FROM Users
JOIN TypeUsers ON Users.id_type_user_fk = TypeUsers.id_type_user
JOIN AccountStatus ON Users.id_account_status_fk = AccountStatus.
   id_account_status
WHERE Users.id_user = %s;
```

7.22.5 get_all

Query:

```
SELECT Users.*, AccountStatus.name AS status, TypeUsers.name AS role
FROM Users
JOIN TypeUsers ON Users.id_type_user_fk = TypeUsers.id_type_user
JOIN AccountStatus ON Users.id_account_status_fk = AccountStatus.
   id_account_status;
```

7.22.6 get_by_username

Query:

```
SELECT Users.*, AccountStatus.name AS status, TypeUsers.name AS role
FROM Users
JOIN TypeUsers ON Users.id_type_user_fk = TypeUsers.id_type_user
JOIN AccountStatus ON Users.id_account_status_fk = AccountStatus.
   id_account_status
WHERE LOWER(Users.username) LIKE LOWER(%s);
```

7.22.7 get_by_email

Query:

```
SELECT Users.*, AccountStatus.name AS status, TypeUsers.name AS role
FROM Users
JOIN TypeUsers ON Users.id_type_user_fk = TypeUsers.id_type_user
JOIN AccountStatus ON Users.id_account_status_fk = AccountStatus.
   id_account_status
WHERE Users.email LIKE %s;
```

7.22.8 get_by_phone

Query:

```
SELECT Users.*, AccountStatus.name AS status, TypeUsers.name AS role
FROM Users
JOIN TypeUsers ON Users.id_type_user_fk = TypeUsers.id_type_user
JOIN AccountStatus ON Users.id_account_status_fk = AccountStatus.
   id_account_status
WHERE Users.phone LIKE %s;
```

7.22.9 get_by_name

Query:

```
SELECT Users.*, AccountStatus.name AS status, TypeUsers.name AS role
FROM Users
JOIN TypeUsers ON Users.id_type_user_fk = TypeUsers.id_type_user
JOIN AccountStatus ON Users.id_account_status_fk = AccountStatus.
   id_account_status
WHERE LOWER(Users.name) LIKE LOWER(%s);
```

7.22.10 get_by_last_name

Query:

```
SELECT Users.*, AccountStatus.name AS status, TypeUsers.name AS role
FROM Users
JOIN TypeUsers ON Users.id_type_user_fk = TypeUsers.id_type_user
JOIN AccountStatus ON Users.id_account_status_fk = AccountStatus.
   id_account_status
WHERE LOWER(Users.last_name) LIKE LOWER(%s);
```

7.22.11 get_by_role

Query:

```
SELECT Users.*, AccountStatus.name AS status, TypeUsers.name AS role
FROM Users
JOIN TypeUsers ON Users.id_type_user_fk = TypeUsers.id_type_user
JOIN AccountStatus ON Users.id_account_status_fk = AccountStatus.
   id_account_status
WHERE LOWER(TypeUsers.name) LIKE LOWER(%s);
```

7.22.12 get_by_id_status

Query:

```
SELECT Users.*, AccountStatus.name AS status, TypeUsers.name AS role
FROM Users
JOIN TypeUsers ON Users.id_type_user_fk = TypeUsers.id_type_user
JOIN AccountStatus ON Users.id_account_status_fk = AccountStatus.
   id_account_status
WHERE AccountStatus.id_account_status = %s;
```

7.22.13 get_by_id_type_user

Query:

```
SELECT Users.*, AccountStatus.name AS status, TypeUsers.name AS role
FROM Users
JOIN TypeUsers ON Users.id_type_user_fk = TypeUsers.id_type_user
JOIN AccountStatus ON Users.id_account_status_fk = AccountStatus.
   id_account_status
WHERE TypeUsers.id_type_user = %s;
```

7.22.14 get_user_with_address

Query:

```
SELECT Users.*, AccountStatus.name AS status, TypeUsers.name AS role
   ,
Address.country AS country, Address.city AS city
FROM Users
JOIN TypeUsers ON Users.id_type_user_fk = TypeUsers.id_type_user
JOIN AccountStatus ON Users.id_account_status_fk = AccountStatus.
   id_account_status
JOIN Address ON Users.id_address_fk = Address.id_address
WHERE Users.id_user = %s;
```

7.22.15 get_all_users_with_address

Query:

```
SELECT Users.*, AccountStatus.name AS status, TypeUsers.name AS role
   ,
Address.country AS country, Address.city AS city
FROM Users
JOIN TypeUsers ON Users.id_type_user_fk = TypeUsers.id_type_user
JOIN AccountStatus ON Users.id_account_status_fk = AccountStatus.
   id_account_status
JOIN Address ON Users.id_address_fk = Address.id_address;
```

7.22.16 validate_seller

Query:

```
SELECT TypeUsers.name
FROM Users
JOIN TypeUsers ON Users.id_type_user_fk = TypeUsers.id_type_user
WHERE Users.id_user = %s AND TypeUsers.id_type_user = 1;
```

8 Managing Data types

Overview In Project MercadoLibreXS, managing data types is crucial for ensuring data integrity, consistency, and efficient storage. Each entity in the project, such as users, products, orders, and more, has specific attributes that require appropriate data types. Properly defining and handling these data types is essential for the correct functioning of the application and database operations.

8.1 Data Types in Project MercadoLibreXS

The project uses a variety of data types to represent different kinds of information. These data types are defined in the data models and are used in the database schema, CRUD operations, and API endpoints. The primary data types used in Project MercadoLibreXS include:

- Integer: Used for numeric values without decimal points, such as IDs and quantities.
- **String**: Used for textual data, such as names, descriptions, and email addresses.
- **Float**: Used for numeric values with decimal points, such as prices and ratings.
- **Boolean**: Used for true/false values, such as flags for default status or verification.
- **Optional**: Used to indicate that a field can be null or missing, providing flexibility in data representation.

8.2 Data Types According to each Entity

8.3 UserData

- **id_user**: int
- **id_address_fk**: int
- **id_account_status_fk**: int
- **id_type_user_fk**: int
- **name**: str
- **last_name**: str
- **username**: str
- **email**: str
- **phone**: str
- **date_birth**: str
- **date_register**: str
- **password**: str
- **status**: Optional[str]
- **role**: Optional[str]
- **country**: Optional[str]
- **city**: Optional[str]

8.4 TypeUserData

- **id_type_user**: int
- **name**: str
- **description**: Optional[str]

8.5 StatusCartData

- **id_status_cart**: int
- **name**: str
- **description**: Optional[str]

8.6 AccountStatusData

- **id_account_status**: int
- **name**: str
- **description**: Optional[str]

8.7 ProductData

- **id_product**: int
- **id_store_fk**: int
- **id_product_status_fk**: int
- **id_category_fk**: int
- **id_brand_fk**: int
- **name**: str
- **description**: Optional[str]
- **price**: float
- **stock**: Optional[int]
- **date_published**: str
- **rating**: Optional[float]
- **store**: Optional[StoreData]
- **status**: Optional[ProductStatusData]
- **category**: Optional[CategoryProductData]
- **brand**: Optional[BrandData]

8.8 StoreData

- **id_store**: int
- **id_user_fk**: int
- **id_address_fk**: int
- **name**: str
- **description**: Optional[str]

- **phone**: str
- **email**: str
- **date_created**: str
- **owner**: Optional[UserData]
- **address**: Optional[AddressData]
- **products**: List[ProductData]

8.9 AddressData

- **id_address**: int
- **street**: str
- **city**: str
- **state**: str
- **country**: str
- **zip_code**: str

8.10 BrandData

- **id_brand**: int
- **name**: str
- **description**: Optional[str]

8.11 CategoryProductData

- **id_category**: int
- **name**: str
- **description**: Optional[str]

8.12 ProductStatusData

- **id_product_status**: int
- **name**: str
- **description**: Optional[str]

8.13 DeliveryProviderData

- **id_delivery_provider**: int
- **name**: str
- **phone**: str
- **email**: Optional[str]
- **description**: Optional[str]

8.14 ReceiptData

- **id_receipt**: int
- **id_shopping_cart_fk**: int
- **id_payment_method_fk**: int
- **id_transaction_status_fk**: int
- **date_created**: str
- **total**: float

8.15 PaymentMethodData

- **id_payment_method**: int
- **name**: str
- **description**: Optional[str]

8.16 TransactionStatusData

- **id_transaction_status**: int
- **name**: str
- **description**: Optional[str]

8.17 ClientReportData

- **id_client_report**: int
- **id_user_reporter_fk**: int
- **id_user_reported_fk**: int
- **id_category_report_fk**: int

- **id_status_report_fk**: int
- **description**: str
- **date_created**: str

8.18 StatusReportData

- **id_status_report**: int
- **name**: str
- **description**: Optional[str]

8.19 CategoryReportData

- **id_category_report**: int
- **name**: str
- **description**: Optional[str]

9 Managing Conections

In Project MercadoLibreXS, managing database connections is crucial for ensuring efficient and reliable communication between the application and the database. Proper connection management helps maintain data integrity, optimize performance, and handle concurrent access to the database. This section provides an overview of how connections are managed in Project MercadoLibreXS.

The project uses a dedicated connection management module to handle database connections. This module is responsible for establishing, maintaining, and closing connections to the database. It ensures that connections are reused efficiently and that resources are released when no longer needed.

Connection Module The connection management is implemented in the connections/-database_connection.py file. This module provides a DatabaseConnection class that encapsulates the logic for managing database connections.

Special thanks to Carlos Andres Sierra for providing the database connection module

9.1 Conection to MySQL

Description: This class is responsible for managing the connection to a MySQL database. It provides methods for connecting, disconnecting, and executing common database operations such as retrieving, inserting, updating, and deleting records.

9.1.1 Attributes

- **_dbname** (str): The name of the database (default: "mercadolibrexs").
- **_duser** (str): The username for database authentication (default: "root").
- **_dpass** (str): The password for database authentication (default: "admin12345").
- **_dhost** (str): The hostname of the MySQL server (default: "localhost").
- **_dport** (int): The port number for the MySQL server (default: 3306).
- **connection** (mysql.connector connection object): Stores the active database connection.

9.1.2 Methods

connect()

- Establishes a connection to the MySQL database using the stored credentials.
- Prints an error message if the connection fails.

disconnect()

- Closes the active database connection if it exists.

list_schemas()

- Retrieves a list of all available databases (schemas) on the MySQL server.
- Returns a list of database names.
- Handles errors and prints an error message in case of failure.

create(query: str, values: tuple) → int

- Executes an `INSERT` query with the provided values.
- Commits the transaction and returns the last inserted ID.
- Handles errors and prints an error message if the query fails.

update(query: str, values: tuple)

- Executes an `UPDATE` query with the provided values.
- Commits the transaction to apply the update.
- Prints the executed query and handles errors.

delete(query: str, item_id: int)

- Executes a `DELETE` query to remove a record by its ID.
- Commits the transaction to apply the deletion.
- Handles errors and prints an error message if the query fails.

get_one(query: str, values: tuple) → ProjectModel

- Executes a `SELECT` query to retrieve a single record.
- Returns the result as a dictionary with column names as keys.
- Handles errors and prints an error message if the query fails.

get_many(query: str, values: tuple = ()) → List[ProjectModel]

- Executes a `SELECT` query to retrieve multiple records.
- Returns a list of dictionaries, where each dictionary represents a record.
- Handles errors and prints an error message if the query fails.

9.2 Conection to Postgres

Description: This class is responsible for managing the connection to a PostgreSQL database. It provides methods for connecting, disconnecting, and executing common database operations such as retrieving, inserting, updating, and deleting records.

9.2.1 Attributes

- **_dbname** (str): The name of the database (default: ”postgres”).
- **_duser** (str): The username for database authentication (default: ”postgres”).
- **_dpass** (str): The password for database authentication (default: ”P4

  w0rd”).**_dhost**(*str*) : *ThehostnameofthePostgreSQLserver*(*default* : ”*localhost*”).

- **_dport** (str): The port number for the PostgreSQL server (default: ”5432”).
- **connection** (psycopg2 connection object): Stores the active database connection.

9.2.2 Methods

connect()

- Establishes a connection to the PostgreSQL database using the stored credentials.
- Prints an error message if the connection fails.

disconnect()

- Closes the active database connection if it exists.

list_schemas()

- Retrieves a list of all available schemas in the PostgreSQL database.
- Returns a list of schema names.
- Handles errors and prints an error message in case of failure.

create(query: str, values: tuple) → int

- Executes an `INSERT` query with the provided values.
- Retrieves the ID of the newly inserted record.
- Commits the transaction and handles errors.

update(query: str, values: tuple)

- Executes an `UPDATE` query with the provided values.
- Commits the transaction to apply the update.
- Handles errors and prints an error message in case of failure.

delete(query: str, item_id: int)

- Executes a `DELETE` query to remove a record by its ID.
- Commits the transaction to apply the deletion.
- Handles errors and prints an error message if the query fails.

_convert_value(value) → str

- Converts a `datetime` object to a string format (`"YYYY-MM-DD HH:MM:SS"`).
- Returns the value as-is if it's not a `datetime` object.

get_one(query: str, values: tuple) → ProjectModel

- Executes a `SELECT` query to retrieve a single record.
- Converts the result into a dictionary where column names are keys.
- Uses `_convert_value` to handle date-time fields.
- Handles errors and prints an error message in case of failure.

get_many(query: str, values: tuple = ()) → List[ProjectModel]

- Executes a `SELECT` query to retrieve multiple records.
- Returns a list of dictionaries, where each dictionary represents a record.
- Uses `_convert_value` to handle date-time fields.
- Handles errors and prints an error message if the query fails.

10 Services

In Project MercadoLibreXS, services play a crucial role in handling the business logic and facilitating communication between the client and the database. Services are responsible for processing incoming requests, invoking the appropriate CRUD operations, and returning the results to the client. By encapsulating the business logic in services, the project ensures a clean separation of concerns, making the codebase more maintainable and scalable.

10.1 Structure of Services

Each entity in the project, such as users, products, orders, and more, has a corresponding service file. These service files define the API endpoints using FastAPI and map them to the CRUD operations. The service files handle request validation, error handling, and response formatting, ensuring a consistent and reliable API.

11 Services per Entity

11.1 Address Service

Description: This service provides API endpoints to manage addresses in the database. It includes functionalities to create, update, delete, and retrieve addresses using FastAPI.

11.1.1 Methods

create(data: AddressData) → int

- Endpoint: `POST /address/create`
- Creates a new address record in the database.
- Returns the ID of the newly created address.

update(id_: int, data: AddressData) → bool

- Endpoint: `PUT /address/update/{id_}`
- Updates an existing address record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the address is not found.

delete(id_: int) → bool

- Endpoint: `DELETE /address/delete/{id_}`
- Deletes an address record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the address is not found.

get_by_id(id_: int) → Optional[AddressData]

- Endpoint: `GET /address/get_by_id/{id_}`
- Retrieves an address record from the database using its ID.
- Returns the address data if found, otherwise returns `None`.

get_all() → List[AddressData]

- Endpoint: `GET /address/get_all`
- Retrieves all address records from the database.
- Returns a list of all stored addresses.

11.2 Brand Service

Description: This service provides API endpoints to manage brands in the database. It includes functionalities to create, update, delete, and retrieve brands using FastAPI.

11.2.1 Methods

create(data: BrandData) → int

- Endpoint: `POST /brand/create`
- Creates a new brand record in the database.
- Returns the ID of the newly created brand.
- The brand data must follow the specified structure.

update(id_: int, data: BrandData) → bool

- Endpoint: `PUT /brand/update/{id_}`
- Updates an existing brand record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the brand is not found.

delete(id_: int) → bool

- Endpoint: `DELETE /brand/delete/{id_}`
- Deletes a brand record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the brand is not found.

get_by_id(id_: int) → Optional[BrandData]

- Endpoint: `GET /brand/get_by_id/{id_}`
- Retrieves a brand record from the database using its ID.
- Returns the brand data if found, otherwise returns `None`.

get_all() → List[BrandData]

- Endpoint: `GET /brand/get_all`
- Retrieves all brand records from the database.
- Returns a list of all stored brands.

get_by_name(name: str) → Optional[BrandData]

- Endpoint: `GET /brand/get_by_name/{name}`
- Retrieves a brand record from the database using its name.
- Returns the brand data if found, otherwise returns `None`.

12 Service Class Documentation

12.1 Cart Service

Description: This service provides API endpoints to manage shopping carts in the database. It includes functionalities to create, update, delete, and retrieve carts using FastAPI.

12.1.1 Methods

create(data: CartData) → int

- Endpoint: `POST /cart/create`
- Creates a new shopping cart record in the database.
- Returns the ID of the newly created cart.

update(data: CartData) → bool

- Endpoint: `PUT /cart/update/{id}`
- Updates an existing shopping cart record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the cart is not found.

delete(id_: int) → bool

- Endpoint: `DELETE /cart/delete/{id}`
- Deletes a cart record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the cart is not found.

get_by_id(id_: int) → Optional[CartData]

- Endpoint: `GET /cart/get_by_id/{id}`
- Retrieves a cart record from the database using its ID.
- Returns the cart data if found, otherwise returns `None`.

get_all() → List[CartData]

- Endpoint: `GET /cart/get_all`
- Retrieves all shopping cart records from the database.
- Returns a list of all stored carts.

12.2 Category Product Service

Description: This service provides API endpoints to manage category products in the database. It includes functionalities to create, update, delete, and retrieve category products using FastAPI.

12.2.1 Methods

create(data: CategoryProductData) → int

- Endpoint: `POST /category_product/create`
- Creates a new category product record in the database.
- Returns the ID of the newly created category product.
- The category product data must follow the specified structure.

update(id_: int, data: CategoryProductData) → bool

- Endpoint: `PUT /category_product/update/{id}`
- Updates an existing category product record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the category product is not found.

delete(id_: int) → bool

- Endpoint: `DELETE /category_product/delete/{id}`
- Deletes a category product record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the category product is not found.

get_by_id(id_: int) → Optional[CategoryProductData]

- Endpoint: `GET /category_product/get_by_id/{id}`
- Retrieves a category product record from the database using its ID.
- Returns the category product data if found, otherwise returns `None`.

get_all() → List[CategoryProductData]

- Endpoint: `GET /category_product/get_all`
- Retrieves all category product records from the database.
- Returns a list of all stored category products.

get_by_name(name: str) → Optional[CategoryProductData]

- Endpoint: `GET /category_product/get_by_name/{name}`
- Retrieves a category product record from the database using its name.
- Returns the category product data if found, otherwise returns `None`.

12.3 Category Report Service

Description: This service provides API endpoints to manage category reports in the database. It includes functionalities to create, update, delete, and retrieve category reports using FastAPI.

12.3.1 Methods

create(data: CategoryReportData) → int

- Endpoint: `POST /category_report/create`
- Creates a new category report record in the database.
- Returns the ID of the newly created category report.

update(id_: int, data: CategoryReportData) → bool

- Endpoint: `PUT /category_report/update/{id}`
- Updates an existing category report record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the category report is not found.

delete(id_: int) → bool

- Endpoint: `DELETE /category_report/delete/{id}`
- Deletes a category report record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the category report is not found.

get_by_id(id_: int) → Optional[CategoryReportData]

- Endpoint: `GET /category_report/get_by_id/{id}`
- Retrieves a category report record from the database using its ID.
- Returns the category report data if found, otherwise returns `None`.

get_all() → List[CategoryReportData]

- Endpoint: `GET /category_report/get_all`
- Retrieves all category report records from the database.
- Returns a list of all stored category reports.

12.4 Client Report Service

Description: This service provides API endpoints to manage client reports in the database. It includes functionalities to create, update, delete, and retrieve client reports using FastAPI.

12.4.1 Methods

create(data: ClientReportData) → int

- Endpoint: `POST /client_report/create`
- Creates a new client report record in the database.
- Returns the ID of the newly created client report.

update(id_: int, data: ClientReportData) → bool

- Endpoint: `PUT /client_report/update/{id}`
- Updates an existing client report record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the client report is not found.

delete(id_: int) → bool

- Endpoint: `DELETE /client_report/delete/{id}`
- Deletes a client report record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the client report is not found.

get_by_id(id_: int) → Optional[ClientReportData]

- Endpoint: `GET /client_report/get_by_id/{id}`
- Retrieves a client report record from the database using its ID.
- Returns the client report data if found, otherwise returns `None`.

get_all() → List[ClientReportData]

- Endpoint: `GET /client_report/get_all`
- Retrieves all client report records from the database.
- Returns a list of all stored client reports.

12.5 Delivery Provider Service

Description: This service provides API endpoints to manage delivery providers in the database. It includes functionalities to create, update, delete, and retrieve delivery providers using FastAPI.

12.5.1 Methods

create(data: DeliveryProviderData) → int

- Endpoint: `POST /delivery_provider/create`
- Creates a new delivery provider record in the database.
- Returns the ID of the newly created delivery provider.
- The delivery provider data must follow the specified structure.

update(id_: int, data: DeliveryProviderData) → bool

- Endpoint: `PUT /delivery_provider/update/{id}`
- Updates an existing delivery provider record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the delivery provider is not found.

delete(id_: int) → bool

- Endpoint: `DELETE /delivery_provider/delete/{id}`
- Deletes a delivery provider record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the delivery provider is not found.

get_by_id(id_: int) → Optional[DeliveryProviderData]

- Endpoint: `GET /delivery_provider/get_by_id/{id}`
- Retrieves a delivery provider record from the database using its ID.
- Returns the delivery provider data if found, otherwise returns `None`.

get_all() → List[DeliveryProviderData]

- Endpoint: `GET /delivery_provider/get_all`
- Retrieves all delivery provider records from the database.
- Returns a list of all stored delivery providers.

get_by_name(name: str) → List[DeliveryProviderData]

- Endpoint: `GET /delivery_provider/get_by_name/{name}`
- Retrieves delivery provider records from the database using their name.
- Returns a list of matching delivery providers.

get_by_email(email: str) → List[DeliveryProviderData]

- Endpoint: `GET /delivery_provider/get_by_email/{email}`
- Retrieves delivery provider records from the database using their email.
- Returns a list of matching delivery providers.

get_by_phone(phone: str) → List[DeliveryProviderData]

- Endpoint: `GET /delivery_provider/get_by_phone/{phone}`
- Retrieves delivery provider records from the database using their phone number.
- Returns a list of matching delivery providers.

12.6 Delivery Status Service

Description: This service provides API endpoints to manage delivery statuses in the database. It includes functionalities to create, update, delete, and retrieve delivery statuses using FastAPI.

12.6.1 Methods

create(data: DeliveryStatusData) → int

- Endpoint: `POST /delivery_status/create`
- Creates a new delivery status record in the database.
- Returns the ID of the newly created delivery status.
- The delivery status data must follow the specified structure.

update(id_: int, data: DeliveryStatusData) → bool

- Endpoint: `PUT /delivery_status/update/{id}`
- Updates an existing delivery status record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the delivery status is not found.

delete(id_: int) → bool

- Endpoint: `DELETE /delivery_status/delete/{id}`
- Deletes a delivery status record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the delivery status is not found.

get_by_id(id_: int) → Optional[DeliveryStatusData]

- Endpoint: `GET /delivery_status/get_by_id/{id}`
- Retrieves a delivery status record from the database using its ID.
- Returns the delivery status data if found, otherwise returns `None`.

get_all() → List[DeliveryStatusData]

- Endpoint: `GET /delivery_status/get_all`
- Retrieves all delivery status records from the database.
- Returns a list of all stored delivery statuses.

get_by_name(name: str) → Optional[DeliveryStatusData]

- Endpoint: `GET /delivery_status/get_by_name/{name}`
- Retrieves a delivery status record from the database using its name.
- Returns the delivery status data if found, otherwise returns `None`.

Description: This service provides API endpoints to manage deliveries in the database. It includes functionalities to create, update, delete, and retrieve deliveries using FastAPI.

12.6.2 Methods

create(data: DeliveryData) → int

- Endpoint: `POST /delivery/create`
- Creates a new delivery record in the database.
- Returns the ID of the newly created delivery.

update(id_: int, data: DeliveryData) → bool

- Endpoint: `PUT /delivery/update/{id}`
- Updates an existing delivery record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the delivery is not found.

delete(id_: int) → bool

- Endpoint: `DELETE /delivery/delete/{id}`
- Deletes a delivery record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the delivery is not found.

get_by_id(id_: int) → Optional[DeliveryData]

- Endpoint: `GET /delivery/get_by_id/{id}`
- Retrieves a delivery record from the database using its ID.
- Returns the delivery data if found, otherwise returns `None`.

get_all() → List[DeliveryData]

- Endpoint: `GET /delivery/get_all`
- Retrieves all delivery records from the database.
- Returns a list of all stored deliveries.

12.7 Favorite List User Product Service

Description: This service provides API endpoints to manage favorite list user products in the database. It includes functionalities to create, update, delete, and retrieve favorite list user products using FastAPI.

12.7.1 Methods

create(data: FavoriteListUserProductData) → int

- Endpoint: `POST /favorite_list_user_product/create`
- Creates a new favorite list user product record in the database.
- Returns the ID of the newly created favorite list user product.

update(id_: int, data: FavoriteListUserProductData) → bool

- Endpoint: `PUT /favorite_list_user_product/update/{id}`
- Updates an existing favorite list user product record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the favorite list user product is not found.

delete(id_: int) → bool

- Endpoint: `DELETE /favorite_list_user_product/delete/{id}`
- Deletes a favorite list user product record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the favorite list user product is not found.

get_by_id(id_: int) → Optional[FavoriteListUserProductData]

- Endpoint: `GET /favorite_list_user_product/get_by_id/{id}`
- Retrieves a favorite list user product record from the database using its ID.
- Returns the favorite list user product data if found, otherwise returns `None`.

get_all() → List[FavoriteListUserProductData]

- Endpoint: `GET /favorite_list_user_product/get_all`
- Retrieves all favorite list user product records from the database.
- Returns a list of all stored favorite list user products.

12.8 Favorite List User Store Service

Description: This service provides API endpoints to manage favorite list user stores in the database. It includes functionalities to create, update, delete, and retrieve favorite list user stores using FastAPI.

12.8.1 Methods

create(data: FavoriteListUserStoreData) → int

- Endpoint: `POST /favorite_list_user_store/create`
- Creates a new favorite list user store record in the database.
- Returns the ID of the newly created favorite list user store.

update(id_: int, data: FavoriteListUserStoreData) → bool

- Endpoint: `PUT /favorite_list_user_store/update/{id}`
- Updates an existing favorite list user store record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the favorite list user store is not found.

delete(id_: int) → bool

- Endpoint: `DELETE /favorite_list_user_store/delete/{id}`
- Deletes a favorite list user store record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the favorite list user store is not found.

get_by_id(id_: int) → Optional[FavoriteListUserStoreData]

- Endpoint: `GET /favorite_list_user_store/get_by_id/{id}`
- Retrieves a favorite list user store record from the database using its ID.
- Returns the favorite list user store data if found, otherwise returns `None`.

get_all() → List[FavoriteListUserStoreData]

- Endpoint: `GET /favorite_list_user_store/get_all`
- Retrieves all favorite list user store records from the database.
- Returns a list of all stored favorite list user stores.

12.9 Payment Method Service

Description: This service provides API endpoints to manage payment methods in the database. It includes functionalities to create, update, delete, and retrieve payment methods using FastAPI.

12.9.1 Methods

create(data: PaymentMethodData) → int

- Endpoint: `POST /payment_method/create`
- Creates a new payment method record in the database.
- Returns the ID of the newly created payment method.

update(id_: int, data: PaymentMethodData) → bool

- Endpoint: `PUT /payment_method/update/{id}`
- Updates an existing payment method record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the payment method is not found.

delete(id_: int) → bool

- Endpoint: `DELETE /payment_method/delete/{id}`
- Deletes a payment method record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the payment method is not found.

get_by_id(id_: int) → Optional[PaymentMethodData]

- Endpoint: `GET /payment_method/get_by_id/{id}`
- Retrieves a payment method record from the database using its ID.
- Returns the payment method data if found, otherwise returns `None`.

get_all() → List[PaymentMethodData]

- Endpoint: `GET /payment_method/get_all`
- Retrieves all payment method records from the database.
- Returns a list of all stored payment methods.

12.10 Product Status Service

Description: This service provides API endpoints to manage product statuses in the database. It includes functionalities to create, update, delete, and retrieve product statuses using FastAPI.

12.10.1 Methods

create(data: ProductStatusData) → int

- Endpoint: `POST /product_status/create`
- Creates a new product status record in the database.
- Returns the ID of the newly created product status.

update(id_: int, data: ProductStatusData) → bool

- Endpoint: `PUT /product_status/update/{id}`
- Updates an existing product status record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the product status is not found.

delete(id_: int) → bool

- Endpoint: `DELETE /product_status/delete/{id}`
- Deletes a product status record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the product status is not found.

get_by_id(id_: int) → Optional[ProductStatusData]

- Endpoint: `GET /product_status/get_by_id/{id}`
- Retrieves a product status record from the database using its ID.
- Returns the product status data if found, otherwise returns `None`.

get_all() → List[ProductStatusData]

- Endpoint: `GET /product_status/get_all`
- Retrieves all product status records from the database.
- Returns a list of all stored product statuses.

get_by_name(name: str) → Optional[ProductStatusData]

- Endpoint: `GET /product_status/get_by_name/{name}`
- Retrieves a product status record from the database based on its name.
- Returns the product status data if found, otherwise returns `None`.

12.11 Product Service

Description: This service provides API endpoints to manage products in the database. It includes functionalities to create, update, delete, and retrieve products using FastAPI.

12.11.1 Methods

create(data: ProductData) → int

- Endpoint: `POST /product/create`
- Creates a new product record in the database.
- Returns the ID of the newly created product.

update(id_: int, data: ProductData) → bool

- Endpoint: `PUT /product/update/{id}`
- Updates an existing product record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the product is not found.

delete(id_: int) → bool

- Endpoint: `DELETE /product/delete/{id}`
- Deletes a product record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the product is not found.

get_by_id(id_: int) → Optional[ProductData]

- Endpoint: `GET /product/get_by_id/{id}`
- Retrieves a product record from the database using its ID.
- Returns the product data if found, otherwise returns `None`.

get_all() → List[ProductData]

- Endpoint: `GET /product/get_all`
- Retrieves all product records from the database.
- Returns a list of all stored products.

12.12 Receipt Service

Description: This service provides API endpoints to manage receipts in the database. It includes functionalities to create, update, delete, and retrieve receipts using FastAPI.

12.12.1 Methods

create(data: ReceiptData) → int

- Endpoint: `POST /receipt/create`
- Creates a new receipt record in the database.
- Returns the ID of the newly created receipt.

update(id_: int, data: ReceiptData) → bool

- Endpoint: `PUT /receipt/update/{id}`
- Updates an existing receipt record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the receipt is not found.

delete(id_: int) → bool

- Endpoint: `DELETE /receipt/delete/{id}`
- Deletes a receipt record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the receipt is not found.

get_by_id(id_: int) → Optional[ReceiptData]

- Endpoint: `GET /receipt/get_by_id/{id}`
- Retrieves a receipt record from the database using its ID.
- Returns the receipt data if found, otherwise returns `None`.

get_all() → List[ReceiptData]

- Endpoint: `GET /receipt/get_all`
- Retrieves all receipt records from the database.
- Returns a list of all stored receipts.

12.13 Shopping Cart Service

Description: This service provides API endpoints to manage shopping carts in the database. It includes functionalities to create, update, delete, and retrieve shopping carts using FastAPI.

12.13.1 Methods

create(data: ShoppingCartData) → int

- Endpoint: `POST /shopping_cart/create`
- Creates a new shopping cart record in the database.
- Returns the ID of the newly created shopping cart.

update(id_: int, data: ShoppingCartData) → bool

- Endpoint: `PUT /shopping_cart/update/{id}`
- Updates an existing shopping cart record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the shopping cart is not found.

delete(id_: int) → bool

- Endpoint: `DELETE /shopping_cart/delete/{id}`
- Deletes a shopping cart record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the shopping cart is not found.

get_by_id(id_: int) → Optional[ShoppingCartData]

- Endpoint: `GET /shopping_cart/get_by_id/{id}`
- Retrieves a shopping cart record from the database using its ID.
- Returns the shopping cart data if found, otherwise returns `None`.

get_all() → List[ShoppingCartData]

- Endpoint: `GET /shopping_cart/get_all`
- Retrieves all shopping cart records from the database.
- Returns a list of all stored shopping carts.

12.14 Status Cart Service

Description: This service provides API endpoints to manage shopping cart statuses in the database. It includes functionalities to create, update, delete, and retrieve status carts using FastAPI.

12.14.1 Methods

create(data: StatusCartData) → int

- Endpoint: `POST /status_cart/create`
- Creates a new status cart record in the database.
- Returns the ID of the newly created status cart.

update(id_: int, data: StatusCartData) → bool

- Endpoint: `PUT /status_cart/update/{id}`
- Updates an existing status cart record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the status cart is not found.

delete(id_: int) → bool

- Endpoint: `DELETE /status_cart/delete/{id}`
- Deletes a status cart record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the status cart is not found.

get_by_id(id_: int) → Optional[StatusCartData]

- Endpoint: `GET /status_cart/get_by_id/{id}`
- Retrieves a status cart record from the database using its ID.
- Returns the status cart data if found, otherwise returns `None`.

get_all() → List[StatusCartData]

- Endpoint: `GET /status_cart/get_all`
- Retrieves all status cart records from the database.
- Returns a list of all stored status carts.

12.15 Status Report Service

Description: This service provides API endpoints to manage status reports in the database. It includes functionalities to create, update, delete, and retrieve status reports using FastAPI.

12.15.1 Methods

create(data: StatusReportData) → int

- Endpoint: `POST /status_report/create`
- Creates a new status report record in the database.
- Returns the ID of the newly created status report.

update(id_: int, data: StatusReportData) → bool

- Endpoint: `PUT /status_report/update/{id}`
- Updates an existing status report record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the status report is not found.

delete(id_: int) → bool

- Endpoint: `DELETE /status_report/delete/{id}`
- Deletes a status report record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the status report is not found.

get_by_id(id_: int) → Optional[StatusReportData]

- Endpoint: `GET /status_report/get_by_id/{id}`
- Retrieves a status report record from the database using its ID.
- Returns the status report data if found, otherwise returns `None`.

get_all() → List[StatusReportData]

- Endpoint: `GET /status_report/get_all`
- Retrieves all status report records from the database.
- Returns a list of all stored status reports.

12.16 Store Service

Description: This service provides API endpoints to manage store records in the database. It includes functionalities to create, update, delete, and retrieve stores using FastAPI.

12.16.1 Methods

create(data: StoreData) → int

- **Endpoint:** `POST /store/create`
- Creates a new store record in the database.
- Returns the ID of the newly created store.

update(id_: int, data: StoreData) → bool

- **Endpoint:** `PUT /store/update/{id}`
- Updates an existing store record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the store is not found.

delete(id_: int) → bool

- **Endpoint:** `DELETE /store/delete/{id}`
- Deletes a store record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the store is not found.

get_by_id(id_: int) → Optional[StoreData]

- **Endpoint:** `GET /store/get_by_id/{id}`
- Retrieves a store record from the database using its ID.
- Returns the store data if found, otherwise returns `None`.

get_all() → List[StoreData]

- **Endpoint:** `GET /store/get_all`
- Retrieves all store records from the database.
- Returns a list of all stored stores.

12.17 Transaction Status Service

Description: This service provides API endpoints to manage transaction statuses in the database. It allows creating, updating, deleting, and retrieving transaction status records using FastAPI.

12.17.1 Methods

create(data: TransactionStatusData) → int

- **Endpoint:** `POST /transaction_status/create`
- Creates a new transaction status record in the database.
- Returns the ID of the newly created transaction status.

update(id_: int, data: TransactionStatusData) → bool

- **Endpoint:** `PUT /transaction_status/update/{id}`
- Updates an existing transaction status record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the record is not found.

delete(id_: int) → bool

- **Endpoint:** `DELETE /transaction_status/delete/{id}`
- Deletes a transaction status record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the record is not found.

get_by_id(id_: int) → Optional[TransactionStatusData]

- **Endpoint:** `GET /transaction_status/get_by_id/{id}`
- Retrieves a transaction status record from the database using its ID.
- Returns the transaction status data if found, otherwise returns `None`.

get_all() → List[TransactionStatusData]

- **Endpoint:** `GET /transaction_status/get_all`
- Retrieves all transaction status records from the database.
- Returns a list of all stored transaction statuses.

12.18 Type User Service

Description: This service provides API endpoints to manage user types in the database. It allows creating, updating, deleting, and retrieving user type records using FastAPI.

12.18.1 Methods

create(data: TypeUserData) → int

- **Endpoint:** `POST /type_user/create`
- Creates a new type user record in the database.
- Returns the ID of the newly created type user.

update(id_: int, data: TypeUserData) → bool

- **Endpoint:** `PUT /type_user/update/{id}`
- Updates an existing type user record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the record is not found.

delete(id_: int) → bool

- **Endpoint:** `DELETE /type_user/delete/{id}`
- Deletes a type user record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the record is not found.

get_by_id(id_: int) → Optional[TypeUserData]

- **Endpoint:** `GET /type_user/get_by_id/{id}`
- Retrieves a type user record from the database using its ID.
- Returns the type user data if found, otherwise returns `None`.

get_all() → List[TypeUserData]

- **Endpoint:** `GET /type_user/get_all`
- Retrieves all type user records from the database.
- Returns a list of all stored type users.

12.19 User Service

Description: This service provides API endpoints to manage users in the database. It allows creating, updating, deleting, and retrieving user records using FastAPI.

12.19.1 Methods

create(data: UserData) → int

- **Endpoint:** `POST /user/create`
- Creates a new user record in the database.
- Returns the ID of the newly created user.

update(id_: int, data: UserData) → bool

- **Endpoint:** `PUT /user/update/{id}`
- Updates an existing user record based on the provided ID.
- Returns `True` if the update is successful.
- Raises an HTTP 404 exception if the record is not found.

delete(id_: int) → bool

- **Endpoint:** `DELETE /user/delete/{id}`
- Deletes a user record from the database using the given ID.
- Returns `True` if the deletion is successful.
- Raises an HTTP 404 exception if the record is not found.

get_by_id(id_: int) → Optional[UserData]

- **Endpoint:** `GET /user/get_by_id/{id}`
- Retrieves a user record from the database using its ID.
- Returns the user data if found, otherwise returns `None`.

get_all() → List[UserData]

- **Endpoint:** `GET /user/get_all`
- Retrieves all user records from the database.
- Returns a list of all stored users.

get_by_username(username: str) → Optional[UserData]

- **Endpoint:** `GET /user/get_by_username/{username}`
- Retrieves a user from the database based on their username.

get_by_name(name: str) → List[UserData]

- **Endpoint:** `GET /user/get_by_name/{name}`
- Retrieves users from the database that match the given name.

get_by_email(email: str) → Optional[UserData]

- **Endpoint:** `GET /user/get_by_email/{email}`
- Retrieves a user from the database based on their email.

get_by_phone(phone: str) → Optional[UserData]

- **Endpoint:** `GET /user/get_by_phone/{phone}`
- Retrieves a user from the database based on their phone number.

get_by_last_name(last_name: str) → List[UserData]

- **Endpoint:** `GET /user/get_by_last_name/{last_name}`
- Retrieves users from the database that match the given last name.

get_by_role(role: str) → List[UserData]

- **Endpoint:** `GET /user/get_by_role/{role}`
- Retrieves users from the database that match the given role.

get_by_id_status(id_status: int) → List[UserData]

- **Endpoint:** `GET /user/get_by_id_status/{id_status}`
- Retrieves users from the database based on their account status ID.

get_by_id_type_user(type_user: int) → List[UserData]

- **Endpoint:** `GET /user/get_by_id_type_user/{type_user}`
- Retrieves users from the database based on their user type ID.

get_user_with_address(id_user: int) → Optional[UserData]

- **Endpoint:** `GET /user/get_user_with_address/{id_user}`
- Retrieves a user along with their address from the database.

get_all_users_with_address() → List[UserData]

- **Endpoint:** `GET /user/get_all_users_with_address`
- Retrieves all users along with their addresses from the database.

validate_seller(id_user: int) → bool

- **Endpoint:** `GET /user/validate_seller/{id_user}`
- Validates if a user is a seller based on their user type.
- Returns `True` if the user is a seller, otherwise `False`.

13 Main Execution

13.1 Overview

This module sets up a FastAPI application for the **Project DBF API**. It configures Cross-Origin Resource Sharing (CORS) settings and includes various API routers for different services.

13.2 FastAPI Application Initialization

The application is created using:

```
app = FastAPI()
```

This initializes the FastAPI framework, which handles HTTP requests and manages the API endpoints.

13.3 CORS Configuration

Cross-Origin Resource Sharing (CORS) is configured using the `CORSMiddleware`. This allows frontend applications hosted on different domains (such as `localhost` or `localhost:8000`) to access the API. The configuration:

- **allow_origins**: Specifies the domains that are allowed to make requests.
- **allow_credentials**: Enables the use of cookies and authentication headers.
- **allow_methods**: Allows all HTTP methods (GET, POST, PUT, DELETE, etc.).
- **allow_headers**: Allows all headers in requests.

13.4 API Routers

The application includes multiple routers from different service modules, each handling a specific part of the system. The routers are registered with a common prefix `/api/v1` to maintain a consistent API structure.

Included Routers:

- `user_router` - Handles user management.
- `product_router` - Manages product-related operations.
- `cart_router` - Manages shopping carts.
- `category_router` - Handles product categories.
- `address_router` - Manages user addresses.
- `store_router` - Handles store-related operations.
- `transaction_status_router` - Manages transaction statuses.
- `status_report_router` - Manages status reports.
- `category_report_router` - Handles category reports.
- `client_report_router` - Manages client reports.
- `delivery_router` - Handles deliveries.
- `favorite_list_user_product_router` - Manages user favorite product lists.
- `favorite_list_user_store_router` - Manages user favorite store lists.
- `payment_method_router` - Handles payment methods.
- `receipt_router` - Manages receipts and transactions.
- `shopping_cart_router` - Handles shopping cart functionalities.
- `status_cart_router` - Manages shopping cart statuses.

Each router is included using:

```
app.include_router(user_router, prefix="/api/v1")
```

This ensures that all endpoints follow a structured namespace under `/api/v1`.

13.5 Root Endpoint

The root endpoint `/` is defined using:

```
@app.get("/")
def read_root():
    return {"message": "Welcome to Project DBF API"}
```

This provides a simple welcome message when accessing the base URL of the API.

13.6 Running the Application

The application is run using Uvicorn:

```
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

This starts the API server, listening on all network interfaces (`0.0.0.0`) at port `8000`, making the API accessible over the local network.