SYSTEM DOCUMENTATION

1. Project Information

Project Name: MercaFacil
Student Name: Juan Sebastian Iguaran Davila
Course: DESARROLLO WEB
Semester: 8
Date: 18/11/25
Instructor: JAIDER QUINTERO

Short Project Description:
MercaFacil is a comprehensive marketplace web application that facilitates interactions between clients and sellers. It manages the entire e-commerce lifecycle, including product cataloging, order processing, payment handling, shipment tracking, and a review system. The system features a robust Role-Based Access Control (RBAC) system to secure resources and manage user permissions (Admins, Sellers, Clients).

2. System Architecture Overview

2.1 Architecture Description

The system follows a standard **Client-Server Architecture** using a RESTful API communication model.

- **The Frontend (Client)** is a Single Page Application (SPA) built with Angular. It consumes JSON data from the backend.
- **The Backend (Server)** is built with Node.js and Express, utilizing Sequelize ORM for database interactions. It exposes secure API endpoints protected by JWT (JSON Web Tokens).
- **The Database** is a relational database (SQL) storing all persistent data.

2.2 Technologies Used

- **Frontend:** Angular (v20.3.7), TypeScript, TailwindCSS, PrimeNG (UI Components).
- **Backend:** Node.js, Express.js, TypeScript.
- **Database Engine:** MySQL (via `mysql2` driver) / Sequelize ORM.
- **Additional Libraries / Tools:**
  - *Auth:* `jsonwebtoken` (JWT), `bcryptjs` (Password Hashing).
  - *Utilities:* `morgan` (Logging), `cors`, `dotenv`.
  - *Dev:* `nodemon`, `ts-node`.

2.3 Visual explanation of the system's operation

```
+----------------+          +----------------+          +----------------+
|   User (Web)   | <------> |    REST API    | <------> |    Database    |
|   (Angular)    |   JSON   | (Node/Express) |   SQL    | (MySQL/Sequelize)|
+----------------+          +----------------+          +----------------+
       ^                          |
       |                          |
       +----[ Auth Middleware ]----+
            (JWT Validation)
```

3. Database Documentation (ENGLISH)

3.1 Database Description

The database powering **MercaFacil** is structured to support a **multi-vendor marketplace**, where multiple sellers can publish products and clients can place orders, execute payments, receive shipments, and submit reviews.

It is implemented in **MySQL** using **Sequelize ORM**, with a normalized relational structure and a complementary **Role-Based Access Control (RBAC)** module to secure backend resources.

The schema is divided into two main areas:

A) E-commerce Core Entities

- **Client:** Stores customer account data.
- **Seller:** Stores vendor accounts and includes password hashing hooks.
- **Product:** Items listed by sellers.
- **Category:** Product categorization system.
- **Tag:** Labels used for product filtering.
- **ProductTag:** Pivot table for Product–Tag (N:N).
- **Order:** Represents a purchase made by a client.
- **OrderDetail:** Contains products inside each order.
- **Payment:** Payment related to an order (1:1).
- **Shipment:** Shipping information for an order (1:1).
- **Review:** Customer product reviews.

B) RBAC – Role-Based Access Control Module

- **User:** Authentication identity.
- **Role:** Permission definitions (Admin, Seller, Client).
- **RoleUser:** User–Role assignments (N:N).
- **Resource:** System endpoints or protected functionalities.
- **ResourceRole:** Maps roles to resources for access validation.

This module allows dynamic and secure permission management across the API.

3.2 ERD – Entity Relationship Diagram

The following describes the relationships defined inside the Sequelize models:

- **Client —< Order:** A client can have many orders.
- **Client —< Review:** A client can write multiple reviews.
- **Seller —< Product:** A seller can publish many products.
- **Product — Category:** Each product belongs to one category.
- **Product — Seller:** Each product belongs to one seller.
- **Product >—< Tag (via ProductTag):** Products can have many tags and vice versa.
- **Order —< OrderDetail:** An order contains many order items.
- **Order >—< Product (via OrderDetail):** Many-to-many through the order details.
- **Order — Payment (1:1):** Each order has one payment record.
- **Order — Shipment (1:1):** Each order has one shipment.
- **User >—< Role (via RoleUser):** A user can have multiple roles.
- **Role >—< Resource (via ResourceRole):** Roles define access to system resources.

3.3 Logical Model

The logical data model includes several key conventions:

Soft Deletes (ACTIVE/INACTIVE)

Most tables use a `status` column defined as:

3.4 Physical Model (Tables)

Users Table

Column	Type	PK/FK	Description
id	INTEGER	PK	Unique user identifier.
username	STRING	—	User login name.
email	STRING	—	Unique email address.
password	STRING	—	Bcrypt hashed password.

Products Table

Column	Type	PK/FK	Description
id	INTEGER	PK	Product identifier.
name	STRING	—	Product name.
price	FLOAT	—	Unit price.
id_seller	INTEGER	FK	Links to sellers table.
id_category	INTEGER	FK	Links to categories table.
status	ENUM	—	'ACTIVE' / 'INACTIVE'.

Orders Table

Column	Type	PK/FK	Description
id	INTEGER	PK	Order identifier.
id_client	INTEGER	FK	Links to clients table.
total	FLOAT	—	Calculated total of the order.
status	ENUM	—	'PENDING', 'PAID', 'SHIPPED'.

Order Details Table

Column	Type	PK/FK	Description
id	INTEGER	PK	Line item ID.
id_order	INTEGER	FK	Links to orders.
id_product	INTEGER	FK	Links to products.
quantity	INTEGER	—	Number of units purchased.

Clients Table

Column	Type	PK/FK	Description
id	INTEGER	PK	Client identifier.
name	STRING	—	Client's full name.
code	STRING	—	Unique ID code (CC/TI).

Sellers Table

Column	Type	PK/FK	Description
id	INTEGER	PK	Seller identifier.
name	STRING	—	Store or seller name.
phone	STRING	—	Contact number.

---

4. Use Cases – CRUD

4.1 Use Case: Create Product

Actor: Seller / Administrator
Description: Registers a new product to make it available for purchase.
Preconditions: User must be authenticated and have an 'ACTIVE' status.
Postconditions: A new product record is added to the database.

Main Flow:

1. User navigates to the “New Product” form.
2. User enters Name, Price, Description.
3. User selects a Category and Seller via dropdown.

4. User clicks **Guardar** (Save).
5. Frontend validates inputs.
6. Frontend sends `POST /api/ocul/Products`.
7. Backend validates JWT.
8. Backend inserts the record.
9. System displays a success toast.

4.2 Use Case: Read Products

Actor: Any User
Description: View a paginated list of available products.

Main Flow:

1. User opens the “Products” view.
2. Frontend requests `GET /api/ocul/Products`.
3. Backend returns products with status **ACTIVE**, including Category and Seller names.
4. Frontend displays them in a PrimeNG table.

4.3 Use Case: Update Client

Actor: Admin / Client
Description: Update personal information such as address or phone number.

Main Flow:

1. User selects a client to edit.
2. Frontend loads current data using `getClientById`.
3. User modifies the address or email.
4. System sends `PATCH /api/ocul/Clientes/:id`.
5. Database updates the record if the client is ACTIVE.

4.4 Use Case: Delete Order (Logical)

Actor: Admin
Description: Marks an order as inactive rather than physically removing it.

Main Flow:

1. User clicks the **Delete** button.
2. System shows a confirmation dialog.
3. If accepted: Frontend sends `DELETE /api/ocul/Orders/:id/logic`.
4. Backend sets the order status to **INACTIVE**.
5. The list refreshes hiding inactive orders.

---

5. Backend Documentation

5.1 Backend Architecture

The backend is developed with **Node.js + Express**, using **Sequelize ORM** for database management.

Modules:

- **Routes:** API endpoints.
- **Controllers:** Business logic.
- **Models:** Table schemas.
- **Middleware:** JWT authentication + validation.

5.2 Folder Structure

```
marketplace/src/
├── config/              # Server configuration
├── controllers/         # Logic for Auth, Products, Orders, etc.
│   └── authorization/  # User, Role and Auth controllers
├── database/            # Sequelize connection setup
├── http/                # .http REST Client files
├── middleware/          # JWT verification
├── models/              # Sequelize Models
│   └── authorization/  # RBAC models
├── routes/              # Route definitions
└── server.ts            # Application entry point
```

5.3 API Documentation (REST)

Endpoint

Method & Path: `POST /api/ocul/Orders`
Purpose: Create a new purchase order linked to a client.

Request Body Example

```
{
  "id_client": 2,
  "status": "PENDING",
  "fecha": "2025-10-15T09:00:00Z",
  "total": 299.99,
  "statuss": "ACTIVE"
}
```

Responses

- **201 Created**
  JSON object of the created order.

- **400 Bad Request**
  Missing required fields.

- **401 Unauthorized**
  Missing or invalid Token.

  Source: `order.http` *(cite: 116)*

5.4 REST Client

The project includes .http files (e.g., order.http, product.http) allowing developers to test endpoints directly within VS Code without external tools like Postman.

6. Frontend Documentation

6.1 Technical Frontend Documentation

Framework Used: Angular (v20.3.0).
Styling: TailwindCSS & PrimeNG (Themes: Aura).

Folder Structure:

```
frontend/src/app/
├── components/
│ ├── auth/ # Login and Register components
│ ├── layout/ # Header, Footer, Aside navigation
│ ├── product/ # Product CRUD (create, getall, update, delete)
│ ├── client/ # Client management
│ ├── ... # Folders for Order, Payment, Shipment, Review, etc.
├── guards/ # AuthGuard to protect routes
├── models/ # TypeScript interfaces (e.g., ProductI, LoginI)
├── services/ # HTTP services (Auth, Product, Client, etc.)
├── app.routes.ts # Application routing configuration
└── app.config.ts # Global providers and PrimeNG config
```

Models, Services and Components

Services:
Centralized HTTP logic. Example: ProductService methods `getAllProducts()` and `createProduct()` inject the Auth Token via headers.

Components:
Follow a consistent structure:

- **getall** → Table
- **create** → Form
- **update** → Form
- **delete** → Delete logic

UI Library:
Uses PrimeNG components such as:

- `p-table`
- `p-dialog`

- `p-toast`
- `p-select`

6.2 Visual explanation of the system’s operation

The frontend is a **Single Page Application (SPA)**.

Navigation:
A persistent sidebar (`app-aside`) allows switching between modules (Clients, Sellers, Products, Orders).

Feedback:
Toast notifications (`p-toast`) provide immediate feedback on actions like *"Product created successfully"*.

Security:
The **AuthGuard** checks if a user is logged in before accessing restricted routes (e.g., `/products/new`), redirecting to `/login` if not.

7. Frontend–Backend Integration

Integration is achieved through Angular's **HttpClient**.

Authentication Flow:

1. User logs in via the **Login** component.
2. **AuthService** sends the credentials to the backend.
3. The backend returns a **JWT**.
4. **AuthService** stores the JWT in `localStorage`.

Authorized Requests:

Services (e.g., ClientService, OrderService) retrieve the token from `localStorage`.

They append the `Authorization: Bearer <token>` header to requests targeting protected endpoints (routes prefixed with `/api/ocul/`).

CORS:
The backend is configured with `cors` to allow requests from the Angular development server (usually port 4200).

8. Conclusions & Recommendations

Conclusions:
The project successfully implements a full-stack e-commerce management system. It demonstrates competency in:

- Building RESTful APIs with Node.js and Sequelize.
- Implementing complex relationships (Many-to-Many) between Products, Orders, and Tags.
- Securing an application using JWT and Role-Based Access Control resources.

- Developing a reactive frontend with Angular 20 and PrimeNG.

Recommendations:

- **Pagination:** While the frontend table supports pagination, the backend currently returns `findAll`. For large datasets, server-side pagination should be implemented in the controllers.
- **Error Handling:** Enhance backend error messages to be more descriptive for the frontend user (e.g., distinguishing between "Duplicate Email" and "Database Error").
- **Environment Variables:** Ensure `.env` files are strictly managed and not committed to version control in a production environment.

9. Annexes (Optional)

- **Source Code:** Available in the attached repository.
- **API Collection:** The `src/http` folder contains ready-to-use request collections for testing all modules.
- **Seed Data:** A `populate_data.ts` script using Faker.js is included to generate initial testing data for Clients, Products, and Orders.