E-commerce RESTful API - Full Documentation

---

1. Overview

This documentation describes the E-commerce API developed to manage users, products, categories, shopping carts, orders, clients (for OAuth2), addresses, payment methods, coupons, notifications, and dashboard metrics. The API adheres to RESTful principles and uses JSON for data exchange. Security is implemented with Spring Security, covering form-based authentication, OAuth2 for social login, and JWT for resource protection.

---

2. Architecture

The application follows a layered architecture with the following main responsibilities:

- **Models** (io.github.gabrielpetry23.ecommerceapi.model): Represent the application's domain entities and are mapped to database tables using JPA (Jakarta Persistence API).
- **Repositories** (io.github.gabrielpetry23.ecommerceapi.repository): Provide abstraction for accessing and manipulating data persisted in the database, utilizing Spring Data JPA.
- **Services** (io.github.gabrielpetry23.ecommerceapi.service): Contain the application's business logic, orchestrating operations and interacting with repositories. They include services for managing coupons, notifications, and dashboard metrics, as well as a service for sending abandoned cart reminders.
- **Controllers** (io.github.gabrielpetry23.ecommerceapi.controller): Receive HTTP requests, delegate logic to services, and return responses to the client.
- **DTOs** (io.github.gabrielpetry23.ecommerceapi.controller.dto): Data Transfer Objects, used to transport data between layers and expose specific information in API requests and responses.
- **Mappers** (io.github.gabrielpetry23.ecommerceapi.controller.mappers): Responsible for converting between model entities and DTOs, facilitating separation of concerns.
- **Validators** (io.github.gabrielpetry23.ecommerceapi.validators): Implement validation logic to ensure the integrity of input data.
- **Security** (io.github.gabrielpetry23.ecommerceapi.security): Configurations and components related to authentication and authorization, including JWT, OAuth2, and role-based access control.
- **Configurations** (io.github.gabrielpetry23.ecommerceapi.configuration): Configuration classes for Spring, including database, security, OAuth2, and Swagger.
- **Exceptions** (io.github.gabrielpetry23.ecommerceapi.exceptions): Custom exception classes for handling specific application errors.

- **Exception Handlers** (io.github.gabrielpetry23.ecommerceapi.controller.common.GlobalExceptionHandler): Centralize exception handling to return consistent and informative error responses.

---

3. Data Model (Entity-Relationship Diagram)

---

4. API Endpoints

Below is a detailed list of API endpoints, with their HTTP methods, URIs, and access permissions:

USERS (/users)

Method	Endpoint	Description	Authorization
POST	/users	Create a new user	Public
GET	/users/{id}	Get a specific user	USER (self), ADMIN, MANAGER
PUT	/users/{id}	Update a user	USER (self), ADMIN, MANAGER
DELETE	/users/{id}	Delete a user	ADMIN, MANAGER
POST	/users/login	Login (Form-based authentication)	Public

PRODUCTS (/products)

Method	Endpoint	Description	Authorization
POST	/products	Create a new product	ADMIN, MANAGER
GET	/products	List all products (with pagination)	Public
GET	/products/search	Search products (with filters)	Public
GET	/products/{id}	Get a specific product	Public
PUT	/products/{id}	Update a product	ADMIN, MANAGER
DELETE	/products/{id}	Delete a product	ADMIN, MANAGER
GET	/products/{id}/reviews	Get product reviews	Public
POST	/products/{id}/reviews	Create a review	USER
DELETE	/products/{id}/reviews/{reviewId}	Delete a review	USER (self), ADMIN, MANAGER
POST	/products/{id}/images	Add image to product	ADMIN, MANAGER

DELETE	/products/{id}/images/{imageId}	Remove image from product	ADMIN, MANAGER

CATEGORIES (/categories)

Method	Endpoint	Description	Authorization
GET	/categories	List all categories	Public
GET	/categories/{id}	Get a specific category	Public
POST	/categories	Create a category	ADMIN, MANAGER
PUT	/categories/{id}	Update a category	ADMIN, MANAGER
DELETE	/categories/{id}	Delete a category	ADMIN, MANAGER
GET	/categories/{id}/products	List products by category	Public

CARTS (/carts)

Method	Endpoint	Description	Authorization
GET	/carts/{id}	Get a specific cart	USER (self), ADMIN, MANAGER
POST	/carts	Create a new cart (associated with the logged-in USER)	USER
POST	/carts/{id}/items	Add item to cart	USER (owner)
PUT	/carts/{id}/items/{itemId}	Update item in cart	USER (owner)
DELETE	/carts/{id}/items/{itemId}	Remove item from cart	USER (owner)
DELETE	/carts/{id}	Empty/Delete the cart	USER (self), ADMIN, MANAGER
GET	/users/{userId}/cart	Get a user's cart	USER (self), ADMIN, MANAGER

ORDERS (/orders)

Method	Endpoint	Description	Authorization
POST	/orders	Create new order	USER

GET	/orders	List all orders (with pagination)	ADMIN, MANAGER
GET	/orders/{id}	Get a specific order	USER (self), ADMIN, MANAGER
GET	/users/{userId}/orders	Get a user's orders (with pagination)	USER (self), ADMIN, MANAGER
PUT	/orders/{id}/status	Update an order's status	ADMIN, MANAGER
GET	/orders/{orderId}/tracking	Get order tracking information	USER (self), ADMIN, MANAGER
POST	/orders/{orderId}/coupon	Apply discount coupon to an order	USER (self)

CLIENTS (OAuth) (/clients)

Method	Endpoint	Description	Authorization
POST	/clients	Create a new Client	ADMIN, MANAGER
GET	/clients/{id}	Get a specific Client	ADMIN, MANAGER
PUT	/clients/{id}	Update a Client	ADMIN, MANAGER
DELETE	/clients/{id}	Delete a Client	ADMIN, MANAGER

ADDRESSES (/users/{userId}/addresses)

Method	Endpoint	Description	Authorization
GET	/users/{userId}/addresses	Get a user's addresses	USER (self), ADMIN, MANAGER
GET	/users/{userId}/addresses/{addressId}	Get a specific address	USER (self), ADMIN, MANAGER
POST	/users/{userId}/addresses	Create address	USER (self)
PUT	/users/{userId}/addresses/{addressId}	Update address	USER (self), ADMIN, MANAGER
DELETE	/users/{userId}/addresses/{addressId}	Delete address	USER (self), ADMIN, MANAGER

PAYMENT METHODS (/users/{userId}/payment-methods)

Method	Endpoint	Description	Authorization
GET	/users/{userId}/payment-methods	Get payment methods	USER (self), ADMIN, MANAGER
GET	/users/{userId}/payment-methods/{paymentMethodId}	Get payment method	USER (self), ADMIN, MANAGER
POST	/users/{userId}/payment-methods	Create payment method	USER (self)
PUT	/users/{userId}/payment-methods/{paymentMethodId}	Update payment method	USER (self), ADMIN, MANAGER
DELETE	/users/{userId}/payment-methods/{paymentMethodId}	Delete payment method	USER (self), ADMIN, MANAGER

NOTIFICATIONS (/users/{userId}/notifications)

Method	Endpoint	Description	Authorization
GET	/users/{userId}/notifications	Get all notifications for a user	USER (self), ADMIN, MANAGER
GET	/users/{userId}/notifications/unread	Get unread notifications for a user	USER (self), ADMIN, MANAGER
PUT	/users/{userId}/notifications/mark-all-as-read	Mark all notifications for a user as read	USER (self)
PUT	/users/{userId}/notifications/{notificationId}/mark-as-read	Mark a specific notification as read	USER (self)

COUPONS (/coupons)

Method	Endpoint	Description	Authorization
POST	/coupons	Create a new coupon	ADMIN, MANAGER

GET	/coupons	List all coupons (with pagination)	ADMIN, MANAGER

DASHBOARD (/dashboard)

Method	Endpoint	Description	Authorization
GET	/dashboard/metrics	Get dashboard metrics (orders by month, best-selling products, total sales value)	ADMIN, MANAGER

---

5. Authentication and Authorization

The API uses Spring Security for authentication and authorization, with the following strategies:

- **Form-based Authentication** (/login or /users/login): For administrative and managerial users to access specific functionalities through a web interface (if implemented).
- **Social Login (OAuth2)**: Allows users to authenticate using their identity provider accounts (e.g., Google, Facebook). The /oauth2/login endpoint initiates the authentication flow, and /authorized receives the authorization code.
- **JWT (JSON Web Tokens)**: Used to protect API resources. After authentication (via form or OAuth2), a JWT can be generated and sent in subsequent requests in the Authorization: Bearer <token> header. The JwtCustomAuthenticationFilter processes these tokens, and the user roles contained in the token are used for authorization.

The defined user roles are:

- **USER**: Regular platform user.
- **MANAGER**: User with permissions to manage products, categories, and coupons.
- **ADMIN**: User with full administrative permissions, including OAuth client management and dashboard metrics access.

Access to endpoints is controlled through @PreAuthorize annotations in controllers, verifying if the authenticated user has the necessary role.

---

6. Error Handling

The API implements global exception handling through GlobalExceptionHandler. Error responses follow a consistent format (ResponseError) and include the HTTP status code, an error message, and, in some cases, additional details (e.g., invalid fields in validation).

The main error status codes used include:

- **400 Bad Request**: Malformed or invalid request (IllegalArgumentException, OperationNotAllowedException).

- **401 Unauthorized**: Not authenticated.
- **403 Forbidden**: Access denied due to lack of permissions (AccessDeniedException).
- **404 Not Found**: Resource not found (EntityNotFoundException, ResourceNotFoundException).
- **409 Conflict**: Data integrity violation, such as duplicate records (DuplicateRecordException).
- **422 Unprocessable Entity**: Validation errors (MethodArgumentNotValidException, InvalidFieldException, InvalidCouponException).
- **500 Internal Server Error**: Unexpected server error (RuntimeException, EmailSendingException).

---

7. Configurations

The application uses Spring configuration files (e.g., application.properties or application.yml) to define properties such as database URL, credentials, security settings, and OAuth2 properties. Configuration classes in io.github.gabrielpetry23.ecommerceapi.configuration include:

- AuthorizationServerConfiguration: Configures the OAuth2 authorization server.
- DatabaseConfiguration: Configures the database connection (using HikariCP).
- OpenApiConfiguration: Configures Swagger/OpenAPI documentation.
- ResourceServerConfiguration: Configures the resource server, protecting API endpoints with JWT and defining authorization rules.
- WebConfiguration: Configures general web aspects, such as the view controller for the login page.
- WebSocketConfig: Configuration for WebSocket support, used for real-time notifications.

---

8. Security

Security is a fundamental aspect of this API, implemented through:

- **HTTPS**: It is highly recommended that the API be served over HTTPS to ensure encryption of communications between the client and the server, protecting sensitive data.
- **Input Validation**: The API uses Jakarta Bean Validation annotations (@NotNull, @NotBlank, @Size, etc.) on DTOs, which are verified by Spring (@Valid) in controllers. This helps prevent injection attacks and ensures data integrity. Validation errors are handled by GlobalExceptionHandler and return a 422 Unprocessable Entity status with details of invalid fields.
- **CSRF (Cross-Site Request Forgery) Protection**: Although disabled in the configuration (csrf(AbstractHttpConfigurer::disable)), in production scenarios with traditional web interfaces, CSRF protection is crucial to prevent attacks where a

malicious site tries to perform actions on behalf of an authenticated user without their consent. If a web interface is integrated, reactivating and configuring CSRF protection is essential.
- **Password Security**: User passwords are securely stored in the database using the bcrypt hashing algorithm, configured in the PasswordEncoder bean. This ensures that real passwords are not stored in plain text.
- **OAuth2 Client Secrets**: OAuth2 client secrets (client_secret) are securely stored in the database and must be treated confidentially.
- **Granular Access Control**: The use of roles and the @PreAuthorize annotation allows precise access control to API resources, ensuring that only authorized users can perform certain actions.
- **Security Updates**: Keeping project dependencies (Spring Boot, security libraries, etc.) updated is crucial to mitigate known security vulnerabilities.

---

9. Swagger/OpenAPI

The API documentation is available through Swagger/OpenAPI, accessible at the following endpoints (configured in WebSecurityCustomizer to be ignored by security):

- /v3/api-docs/**
- /swagger-ui.html/**
- /swagger-ui/**

The OpenAPI configuration is done in the OpenApiConfiguration class, which defines general API information (title, version, description, contact) and configures the Bearer JWT security scheme for authentication. Use the Swagger UI interface to explore API endpoints, view request and response models, and perform interactive tests.

---

10. Technologies Used

- **Java**: Main programming language.
- **Spring Boot**: Framework for rapid Java application development.
- **Spring Data JPA**: Abstraction for data access using JPA.
- **Hibernate**: JPA implementation used by Spring Data JPA.
- **PostgreSQL**: Relational database used for persistence.
- **HikariCP**: High-performance JDBC connection pool.
- **Spring Security**: Framework for application security (authentication and authorization).
- **Spring Security OAuth2**: Framework for OAuth2 implementation.
- **JSON Web Tokens (JWT)**: Standard for creating secure access tokens.
- **Spring WebSocket**: WebSocket communication support for real-time notifications.
- **MapStruct**: Library for generating mapper code between Java objects.

- **Lombok**: Library to reduce Java boilerplate code.
- **Jakarta Bean Validation**: API for bean validation.
- **Swagger/OpenAPI**: Tool for RESTful API documentation.
- **Maven**: Dependency management and build tool.
- **Spring ShedLock**: (Potentially) To ensure that scheduled tasks (like cart reminders) are executed only once in distributed environments.

---

11. How to Use the API

This quick guide will help you get the API up and running and test its main endpoints using Swagger UI.

Prerequisites Make sure you have Docker and Docker Compose installed on your machine.

Initialize the Application Clone the repository:

```
git clone https://github.com/gabrielpetry23/ecommerce-api.git
cd ecommerce-api
```

Attention regarding Environment Variables: In the docker-compose.yml file, you will find environment variables for external services (like Google and Email). To run the project locally, please:

- Replace placeholders (e.g., SUA_CLIENT_ID_LOCAL_OU_DUMMY) with your own development/test credentials, or with dummy values if these functionalities are not crucial for the local demonstration.

Clean and Build/Start the Application: To ensure the database is initialized with the latest test data, it is crucial to remove persisted volumes before bringing up the containers. This ensures that the data.sql script is executed whenever the database container is initialized and avoids issues with existing IDs.

```
docker-compose down --volumes
docker-compose up -d --build
```

- The down --volumes command removes previous database data, including the db_data persistent volume.
- The up -d --build command rebuilds the application image (if there are changes in the Dockerfile) and starts all services (database and API) in the background.

Wait for Initialization: Wait a few seconds (or observe the logs with docker-compose logs -f) until the Spring Boot application (ecommerce_api_local) is fully started. You will see messages like Started EcommerceapiApplication in X.X seconds in

the ecommerce_api_local container log.

1. **Access Swagger UI** Once the application is up, you can access the Swagger UI interface to explore and test the endpoints:
    - Swagger UI: http://localhost:8080/swagger-ui/index.html
2. **Test Users and Credentials** The application's database comes pre-populated with some test data, including users, categories, products, and carts, through the data.sql script. You can use the following credentials for authentication:

User	Email	Password	Role
Gabriel Silva	gabriel.user@example.com	gabriel123	USER
Ana Gerente	gerente.admin@example.com	gerente123	MANAGER
Maria Souza	maria.customer@example.com	maria123	USER
João Pereira	joao.customer@example.com	joao123	USER

Important: Since IDs are dynamically generated `gen_random_uuid()) in data.sql every time the database is recreated (`docker-compose down --volumes`), the UUIDs listed above are for reference only and will be different in your environment. You will need to obtain the real IDs via API after login or listing.

5. **Common Test Flow (Example)** Follow these steps to test a basic flow of authentication, product listing, and creating a new cart/order.
    - **Get Access Token (Login):** Authentication can be done in two ways:
        - **Via /auth/login Endpoint (Recommended for API usage):**
            - In Swagger UI, expand the AuthController endpoint.
            - Select /auth/login [POST].

Click "Try it out". In the "Request body" field, enter user credentials (e.g., gabriel.user@example.com and gabriel123):

```
{
  "email": "gabriel.user@example.com",
```

```
  "password": "gabriel123"
}
```

        - Click "Execute".
        - Copy the accessToken from the response.
    - **Via Swagger UI "Authorize" Button (Simplified login for Swagger tests):**
        - At the top of the Swagger UI page, click the "Authorize" button.
        - A pop-up window will appear. Select the bearerAuth (OAuth2, implicit) option.
        - In the "Username" field, enter the user's email (e.g., gabriel.user@example.com).
        - In the "Password" field, enter the user's password (e.g., gabriel123).
        - Click "Authorize" and then "Close". Swagger UI will automatically make the login request and store the token for subsequent requests.
- **Authorize Requests in Swagger (if you used the /auth/login endpoint):**
    - At the top of the Swagger UI page, click the "Authorize" button.
    - Paste the accessToken you copied into the Value field for bearerAuth (OAuth2, implicit).
    - Click "Authorize" and then "Close". Your requests will now be authenticated.
- **Access Logged-in User Profile:**
    - Expand the UserController endpoint.
    - Select /users/me [GET].
    - Click "Try it out" and then "Execute".
    - You should see the logged-in user's details, including the id (UUID). Save this ID! It will be needed for other operations.
- **List Products and Get IDs:**
    - Expand the ProductController endpoint.
    - Select /products [GET].
    - Click "Try it out" and "Execute".
    - Analyze the returned list of products. Copy the IDs of products you want to add to the cart/order (e.g., the ID of a Gaming Laptop XYZ).
- **Create a New Address (if needed for a new order):**
    - Expand the AddressController endpoint.
    - Select /addresses [POST].
    - Click "Try it out".

In the "Request body", enter the address details using the userId you obtained in step 3.

```
{
  "userId": "UUID_OF_YOUR_USER_HERE",
  "street": "Rua Exemplo",
  "number": "456",
  "complement": "Apto 202",
  "city": "Cidade Teste",
  "state": "TS",
  "zipCode": "99999-999",
  "country": "Brasil"
}
```

        - Click "Execute". Copy the returned address ID.
    - **Create a Cart and Add Items:**

        - Expand the CartController endpoint.
        - Select /carts [POST] to create a new cart for the logged-in user (using the userId from step 3). The cart ID will be returned.

Select /cart-items [POST] to add items to the cart. In the "Request body", use the cartId you just obtained and the productIds you copied in step 4.

```
{
  "cartId": "UUID_OF_YOUR_CART_HERE",
  "productId": "UUID_OF_THE_PRODUCT_HERE",
  "quantity": 1
}
```

        - Repeat to add more products, if desired.
    - **Finalize an Order:**

        - Expand the OrderController endpoint.
        - Select /orders [POST].
        - Click "Try it out".

In the "Request body", fill in the details, using the IDs you obtained (or use existing reference IDs from data.sql for testing). Example using cartId:

```
{
  "userId": "UUID_OF_YOUR_USER_HERE",
  "status": "PENDING",
  "deliveryAddressId": "UUID_OF_THE_USER_ADDRESS_HERE",
  "paymentMethodId": "UUID_OF_THE_USER_PAYMENT_METHOD_HERE",
  "cartId": "UUID_OF_THE_USER_CART_HERE"
```

```
}
```

Example building orderItems directly (without cartId):

```
{
  "userId": "UUID_OF_YOUR_USER_HERE",
  "status": "PENDING",
  "deliveryAddressId": "UUID_OF_THE_USER_ADDRESS_HERE",
  "paymentMethodId": "UUID_OF_THE_USER_PAYMENT_METHOD_HERE",
  "orderItems": [
    {
      "productId": "UUID_OF_THE_GAMING_LAPTOP_HERE",
      "quantity": 1,
      "price": 7500.00
    },
    {
      "productId": "UUID_OF_THE_HEADPHONES_HERE",
      "quantity": 2,
      "price": 250.00
    }
  ]
}
```

        - Click "Execute". The created order ID will be returned.
    - **Verify Existing Orders (data.sql):**
        - Select /orders/{orderId} [GET] and use the ID of one of the test users (e.g., Gabriel, Maria, João) to see the orders associated with them. Since order IDs in data.sql are also dynamic now, you will need to:
            - Log in as gerente.admin@example.com (gerente123).
            - Access /orders [GET] to list all orders (the MANAGER role has permission).
            - Identify Gabriel, Maria, and João's orders by their respective userIds and statuses.
    - **Real-time Notifications:** To receive real-time notifications via WebSocket, connect to the /ws (or /app, depending on your STOMP configuration) WebSocket endpoint and subscribe to relevant topics.
        - Example: /topic/user-notifications/{userId} for user-specific notifications.

Stop the Application To stop the containers and free up resources (keeping the database data, unless you add --volumes):

```
docker-compose down
```

---

12. Business Rules

Users and Profiles

- **Mandatory Authentication for Protected Resources**: Most API endpoints require the user to be authenticated and have a valid JWT (JSON Web Token) in the request header.
- **Role-Based Access Control**: The system defines three user roles (profiles):
    - **USER**: Can manage their own profile (addresses, payment methods), create carts and orders, add product reviews, and view their notifications.
    - **MANAGER**: Has all USER permissions and, in addition, can manage products (create, list, update, delete), categories, and coupons. Can also list all orders and clients, and update order status.
    - **ADMIN**: Has unrestricted access to all system functionalities, including complete user management, OAuth client management, and dashboard metrics access.
- **Encrypted Passwords**: User passwords are stored in the database using bcrypt, ensuring they are not recoverable in plain text.
- **Email Verification (Implicit)**: Although there is no explicit email confirmation endpoint in the provided list, the existence of the email_verification_tokens table suggests that, at some point in the user creation flow, an email verification process is expected to validate the authenticity of the user's email address.

Products and Categories

- **Category Uniqueness**: Each category must have a unique name.
- **Product Availability**: Products have a stock that must be checked before being added to the cart or an order. The API must ensure that the purchased quantity does not exceed the available stock.
- **Mandatory Price and Stock**: Every product must have a price and stock defined.
- **Product Images**: Products can have multiple images, and one of them can be marked as primary (is_main).

Shopping Carts

- **One Cart Per User**: Each authenticated user can have only one active shopping cart.
- **Adding Items**: Only products with available stock can be added to the cart. The quantity must be a positive integer.
- **Updating Items**: It is possible to adjust the quantity of an item in the cart. Stock validation also applies here.
- **Emptying Cart**: The cart can be completely emptied, or individual items can be removed.

- **Abandoned Cart Reminders**: A scheduled service (CartReminderService) periodically checks for carts that have not been updated for more than 24 hours (ABANDONED_HOURS) and have not received a reminder in the last 48 hours (REMINDER_COOLDOWN_HOURS). For these carts, a notification is sent to the user.

Orders

- **Order Creation from Cart**: An order can only be created if the user's cart is not empty.
- **Address and Payment Validation**: The delivery address and payment method selected for the order must belong to the user creating the order.
- **Price Consistency**: The price of items in the order is captured at the time of order creation and does not change if the original product price is modified later.
- **Order Status**: Orders follow a predefined status flow (PENDING, PAID, IN_PREPARATION, IN_DELIVERY, DELIVERED, CANCELLED). Invalid status transitions must be prevented (e.g., cannot go from DELIVERED to PENDING).
- **Tracking Generation**: When an order's status changes to PAID, a tracking code and simulated details are automatically generated and associated with the order.
- **Coupon Application**: A coupon can be applied to an order during its creation, and the order total will be recalculated based on the coupon's discount.
- **Cart Emptying**: After a successful order creation, the associated user's cart is emptied.
- **Order Notifications**: Notifications are sent to the user on important order lifecycle events, such as creation and status updates.

Coupons

- **Unique Code**: Each coupon must have a unique code.
- **Discount Types**: Coupons can offer a discount_amount (fixed value) or discount_percentage (percentage).
- **Validity**: Coupons have a validity date (valid_until) and an is_active status. A coupon is only valid if it is active and within its validity period.
- **New Coupon Notification**: When a new active coupon is created, all users with the USER profile receive a notification about the coupon.

Notifications

- **Notifications Per User**: Notifications are linked to specific users.
- **Read Status**: Notifications can be marked as read (read_at), individually or in bulk.
- **Real-time Notifications**: Notifications are sent in real-time to users via WebSockets.
- **Persistence**: All sent notifications are persisted in the database.

Emails

- **Email Queue for Asynchronous Processing**: The application uses a queue (email_queue) to manage email sending asynchronously, preventing the email sending operation from blocking the application's main flow.

- **User Email Verification**: Newly created users must undergo an email verification process (using email_verification_tokens) to confirm the authenticity of their email address.
- **Email Sending Resilience**: The system is designed to include retry mechanisms and error handling for email sending, ensuring that even temporary failures do not prevent the delivery of important communications.

Dashboard

- **Aggregated Metrics**: The dashboard provides sales metrics, such as the number of orders per month, best-selling products, and total sales value.
- **Restricted Access**: Dashboard metrics are accessible only by users with ADMIN or MANAGER profiles.

---

13. Integration Tests

The project has a comprehensive suite of integration tests for all controllers, ensuring that API endpoints function correctly in conjunction with the service and security layers. These tests use MockMvc to simulate HTTP requests and verify response behavior, including status codes, headers, and JSON body.

The main technologies and libraries used in integration tests are:

- **JUnit 5**: Framework for writing and executing tests.
- **Spring Boot Test**: Provides support for integration tests in Spring Boot applications.
- **@AutoConfigureMockMvc**: Automatically configures MockMvc for controller tests.
- **MockMvc**: Object for simulating HTTP requests to controllers.
- **ObjectMapper**: Used to serialize and deserialize Java objects to/from JSON.
- **@MockitoBean**: Allows mocking and injecting Spring beans, isolating the controller layer and controlling service behavior.
- **Spring Security Test**: Offers utilities for simulating authentication (e.g., with JWT) and CSRF protection in tests.

---

14. Contribution

Interested in improving this project? Follow the guidelines below to facilitate the collaboration process and ensure code quality. Your contribution is very welcome!

How to Contribute

1. **Fork the Repository:**

- Create a fork of the main repository to your GitHub account.

Clone your Fork:

```
git clone https://github.com/YOUR_USERNAME/ecommerce-api.git
cd ecommerce-api
```

2. **Create a New Branch:**
   - Create a branch for your feature or fix. Use a descriptive name (e.g., feature/feature-name, bugfix/login-fix).
     git checkout -b feature/my-new-feature
3. **Develop Your Changes:**
   - Implement your changes, following the project's coding conventions.
   - Ensure that all changes are working correctly and that existing tests (if any) still pass. If applicable, add new tests for your feature/fix.
4. **Commit Your Changes:**
   - Write clear and concise commit messages, explaining what was done.

```
git add .
```

```
git commit -m "feat: Adds endpoint for coupon management"
```

- **Commit Convention Suggestion** (optional, but professional):
  - feat: (for new features)
  - fix: (for bug fixes)
  - docs: (for documentation changes)
  - style: (for formatting, no code change)
  - refactor: (for code refactoring)
  - test: (for adding or correcting tests)
  - chore: (for maintenance tasks, such as dependency updates)

Push Your Changes to Your Fork:

```
git push origin feature/my-new-feature
```

5. **Open a Pull Request (PR):**
   - Go to the original repository on GitHub and create a Pull Request from your branch to the main branch (or master, depending on the project's main branch).
   - Provide a detailed description of your changes, explaining the problem it solves and how the solution works.
   - If the PR resolves a specific issue, mention the corresponding issue (e.g., Closes #123).

Code Conventions and Standards

- **Java**: Follow standard Java coding conventions and SOLID principles.
- **Spring Boot**: Use Spring Boot best practices, such as dependency injection and appropriate annotations.
- **Formatting**: Maintain consistent formatting with existing code (it may be useful to configure a Prettier or linter in the IDE for this).
- **Naming**: Use clear and meaningful variable, method, and class names.
- **Tests**: Prioritize writing unit and integration tests to ensure the robustness of functionalities.

Communication

- In case of doubts or discussions about new functionalities, feel free to open an Issue in the repository before starting development.

---

15. Next Steps and Improvements

- Adding unit and integration tests to increase application reliability.
- Implementing more advanced pagination and filtering in other endpoints.
- Improvements in error handling and more detailed response messages.
- Implementing more comprehensive logs for application monitoring.
- Considering the use of a caching system to improve performance.
- Implementing user email verification.
- Resilience mechanisms for email sending, such as retries.
- Possible integration with a real payment gateway service.