EBD: Database Specification Component

> Project vision.

A4: Conceptual Data Model

> This artifact includes 2 elements:

> 1. A UML class diagram, that outlines the main entities within the Porto's Bookshelf platform, showing how they are connected, their attributes, their attribute types, and the number of connections allowed between them.
> 2. Some additional Business Rules.

1. Class diagram

> UML class diagram containing the classes, associations, multiplicity and roles.
> For each class, the attributes, associations and constraints are included in the class diagram.

UML_FINAL.drawio

2. Additional Business Rules

> Business rules can be included in the UML diagram as UML notes or in a table in this section.

Identifier	Name	Description
BR07	Positive Review Rating	The rating given in a review must be a positive integer between 1 and 5.
BR08	Restricted Review Editing	A user can only edit their review within 7 days of posting it.
BR09	Limit on Wishlist Items	A user can have a maximum of 15 items on their wishlist at any given time.
BR10	Shipping Address Requirement	A valid shipping address must be associated with the user’s profile to complete any order.

A5: Relational Schema, validation and schema refinement

This artifact outlines the relational schema design for the Porto’s Bookshelf system, focusing on structuring and validating data storage. The primary goals include defining relation schemas with attributes, keys, and integrity constraints to ensure data consistency and accuracy. Functional dependencies are analyzed, and each schema is normalized to BCNF to prevent redundancy and maintain data integrity. Additionally, compact notation is used for clarity in presenting relations, attributes, and domains. Where necessary, schema refinement is applied to bring tables into BCNF, ensuring a streamlined and efficient database structure.

1. Relational Schema

> The Relational Schema includes the relation schemas, attributes, domains, primary keys, foreign keys and other integrity rules: UNIQUE, DEFAULT, NOT NULL, CHECK.
> Relation schemas are specified in the compact notation:

Relation reference	Relation Compact Notation
R01	users(id, username **UK NN**, email **UK NN**, password **NN**, fullname **NN**)
R02	administrator(id_admin -> users)
R03	authenticatedUser(id_authenticatedUser -> users)
R04	product(id, id_category -> category, id_author -> author, title **NN**, price **NN CK** price >= 0, stock **NN CK** stock >= 0, description **NN**)
R05	reviewProduct(id, id_product -> product, id_authenticatedUser -> authenticatedUser, comment **NN**, date **NN**, rating **NN CK** rating >=1 AND <= 5)
R06	reviewAuthor(id, id_author -> author, id_authenticatedUser -> authenticatedUser, comment **NN**, date **NN**, rating **NN CK** rating >=1 AND <= 5)
R07	paymentInfo(id_paymentinfo -> authenticatedUser, shippingAddress **NN**, nif **UK NN CK** LENGTH(nif) == 9, cardNumber **UK NN CK** LENGTH(cardNumber) == 16)
R08	order(id, id_authenticatedUser -> authenticatedUser, shippingAdress **NN** ,totalPrice **NN CK** totalPrice > 0 , orderDate **NN DF** Today, arrivalDate **NN CK** arrivalDate > orderDate, orderStatus **NN**)
R09	shoppingCart(id, id_authenticatedUser -> authenticatedUser, id_product -> product, quantity **NN CK** quantity >= 0)
R10	wishlist(id, id_authenticatedUser -> authenticatedUser, id_product -> product)
R11	category(id, categoryName **UK NN**)
R12	author(id, name **NN**, biography **NN**)
R13	notification(id, text **NN**)
R14	paymentApprovedNotification(id, id_order -> order, id_notification -> notification)
R15	changeOrderStatusNotification(id, id_order -> order, id_notification -> notification)
R16	productOnWishlistNotification(id, id_wishlist -> wishlist, id_notification -> notification)
R17	productOnCartPriceChangedNotification(id, id_shoppingCart -> shoppingCart, id_notification -> notification)

Legend:

- **UK** = UNIQUE KEY
- **NN** = NOT NULL
- **DF** = DEFAULT
- **CK** = CHECK

2. Domains

> The specification of additional domains can also be made in a compact form, using the notation:

Domain Name	**Domain Specification**
Today	DATE DEFAULT CURRENT_DATE
OrderStatus	ENUM ('Processing', 'Shipped', 'In Transit', 'Delivered')

3. Schema validation

> To validate the Relational Schema obtained from the Conceptual Model, all functional dependencies are identified and the normalization of all relation schemas is accomplished. Should it be necessary, in case the scheme is not in the Boyce–Codd Normal Form (BCNF), the relational schema is refined using normalization.

TABLE R01	**users**
Keys	{ id }, { email }, {username}, {fullname}
Functional Dependencies:	
FD0101	id → {email, username, password, fullname}
FD0102	email → {id, username, password, fullname}
FD0103	username → {id, email, password, fullname}
FD0104	fullname → {id, email, password, username}
NORMAL FORM	BCNF

TABLE R02	**administrator**
Keys	{ id_admin }
Functional Dependencies:	
FD0201	id_admin → {}
NORMAL FORM	BCNF

TABLE R03	**authenticatedUser**
Keys	{ id_authenticatedUser }
Functional Dependencies:	
FD0301	id_authenticatedUser → { }
NORMAL FORM	BCNF

TABLE R04	**product**
Keys	{ id }
Functional Dependencies:	

TABLE R04	**product**
FD0401	id → {id_category, id_author, title, authorName, price, stock, description}
NORMAL FORM	BCNF

TABLE R05	**reviewProduct**
Keys	{ id_product, id_authenticatedUser }
Functional Dependencies:	
FD0501	{id_product, id_authenticatedUser} → {comment, date, rating}
NORMAL FORM	BCNF

TABLE R06	**reviewAuthor**
Keys	{ id_author, id_authenticatedUser }
Functional Dependencies:	
FD0601	{id_author, id_authenticatedUser} → {comment, date, rating}
NORMAL FORM	BCNF

TABLE R07	**paymentInfo**
Keys	{ id_paymentinfo }
Functional Dependencies:	
FD701	id_paymentInfo → {id_authenticatedUser, shippingAdress, nif, cardNumber}
NORMAL FORM	BCNF

TABLE R08	**order**
Keys	{id}
Functional Dependencies:	
FD801	id → {id_authenticatedUser, totalPrice, orderDate, arrivalDate, orderStatus, shippingAdress}
NORMAL FORM	BCNF

TABLE R09	**shoppingCart**
Keys	{id_product, id_authenticatedUser}
Functional Dependencies:	
FD901	{id_product, id_authenticatedUser} → {quantity}

TABLE R10	**wishlist**
Keys	{ id }, { id_authenticatedUser, id_product }
Functional Dependencies:	

TABLE R10	**wishlist**
FD1001	id → { id_authenticatedUser, id_product }

NORMAL FORM BCNF

TABLE R11	**category**
Keys	{ id }, { categoryName }
Functional Dependencies:	
FD1101	id → { categoryName }
FD1102	categoryName → { id }

NORMAL FORM BCNF

TABLE R12	**author**
Keys	{ id }
Functional Dependencies:	
FD1201	id → { name, biography }

NORMAL FORM BCNF

TABLE R13	**notification**
Keys	{ id }
Functional Dependencies:	
FD1301	id → { id_user, message, date }

NORMAL FORM BCNF

TABLE R14	**paymentApprovedNotification**
Keys	{ id }
Functional Dependencies:	
FD1401	id → { id_order, id_notification }

NORMAL FORM BCNF

TABLE R15	**changeOrderStatusNotification**
Keys	{ id }
Functional Dependencies:	
FD1501	id → { id_order, id_notification }

NORMAL FORM BCNF

TABLE R16	**productOnWishlistNotification**
Keys	{ id }

TABLE R16	**productOnWishlistNotification**
Functional Dependencies:	
FD1601	id → { id_wishlist, id_notification }
NORMAL FORM	BCNF

TABLE R17	**productOnCartPriceChangedNotification**
Keys	{ id }
Functional Dependencies:	
FD1701	id → { id_shoppingCart, id_notification }
NORMAL FORM	BCNF

> If necessary, description of the changes necessary to convert the schema to BCNF.
> Justification of the BCNF.

A6: Indexes, triggers, transactions and database population

The following sections provide detailed information about various database components, including workload estimates, performance indices, full-text search indices, triggers, transactions, and SQL code for schema creation and population

1. Database Workload

> A study of the predicted system load (database load). Estimate of tuples at each relation.

Database Workload

Relation	**Relation Name**	**Order of Magnitude**	**Estimated Growth**
R01	user	10 k (tens of thousands)	10 (tens) / day
R02	author	1 k (thousands)	1 / day
R03	category	10 (hundreds)	1 / day
R04	order	1 k	10 / day
R05	product	1 k	1 / day
R06	review author	1 k	10 / day
R07	review product	1 k	10 / day
R08	wish_list	10 k	10 / day

2. Proposed Indices

2.1. Performance Indices

> Indices proposed to improve performance of the identified queries.

Performance Indices

Index Relation	product
Index Attribute	id
Index Type	B-tree
Cardinality	Non-Unique
Clustering	No
Justification	The `product` table is fundamental to the application, and having a primary key index on `id` ensures fast access to individual product records. This index is unique and is suitable for clustering.
SQL Code	`CREATE INDEX IDX01_product ON product (id);`

Index Relation	product
Index Attribute	id_category
Index Type	B-tree
Cardinality	Non-Unique
Clustering	No
Justification	The `id_category` index improves performance when filtering products by category. Although the cardinality is non-unique, the B-tree index allows for efficient range queries. Clustering is not needed as multiple products can belong to the same category.
SQL Code	`CREATE INDEX IDX04_product_category ON product (id_category);`

Index Relation	product
Index Attribute	id_author
Index Type	B-tree
Cardinality	Non-Unique
Clustering	No

Justification	The `id_author` index speeds up lookups for products by their authors, which is a common operation in the application. The B-tree index is suitable for this purpose and allows for efficient searches. Clustering is not required as multiple products can have the same author.
SQL Code	`CREATE INDEX IDX05_product_author ON product (id_author);`

2.2. Full-text Search Indices

> The system being developed must provide full-text search features supported by PostgreSQL. Thus, it is necessary to specify the fields where full-text search will be available and the associated setup, namely all necessary configurations, indexes definitions and other relevant details.

Index	**FTS01**
Index Relation	product
Field(s)	product_name, description
Weighting	2, 1
Justification	Enables efficient searching for products by name and description, with greater relevance given to the name.
SQL Code	

```sql
-- Add column to product to store computed ts_vectors.
ALTER TABLE product
ADD COLUMN tsvectors TSVECTOR;

-- Create a function to automatically update ts_vectors.
CREATE FUNCTION product_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
         setweight(to_tsvector('english', NEW.product_name), 'A') ||
         setweight(to_tsvector('english', NEW.description), 'B')
        );
 END IF;
 IF TG_OP = 'UPDATE' THEN
         IF (NEW.product_name <> OLD.product_name OR NEW.description <>
OLD.description) THEN
           NEW.tsvectors = (
             setweight(to_tsvector('english', NEW.product_name), 'A') ||
             setweight(to_tsvector('english', NEW.description), 'B')
           );
         END IF;
 END IF;
 RETURN NEW;
END $$
```

```
LANGUAGE plpgsql;

-- Create a trigger before insert or update on product.
CREATE TRIGGER product_search_update
 BEFORE INSERT OR UPDATE ON product
 FOR EACH ROW
 EXECUTE PROCEDURE product_search_update();

-- Finally, create a GIN index for ts_vectors.
CREATE INDEX search_idx ON product USING GIN (tsvectors);
```

Index	**FTS02**
Index Relation	review_product
Field(s)	review_text
Weighting	1
Justification	Facilitates the search for product reviews, allowing users to find relevant opinions.
SQL Code	

```
-- Add column to review_product to store computed ts_vectors.
ALTER TABLE review_product
ADD COLUMN tsvectors TSVECTOR;

-- Create a function to automatically update ts_vectors.
CREATE FUNCTION review_product_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
         setweight(to_tsvector('english', NEW.review_text), 'A')
        );
 END IF;
 IF TG_OP = 'UPDATE' THEN
         IF (NEW.review_text <> OLD.review_text) THEN
           NEW.tsvectors = (
             setweight(to_tsvector('english', NEW.review_text), 'A')
           );
         END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;

-- Create a trigger before insert or update on review_product.
CREATE TRIGGER review_product_search_update
 BEFORE INSERT OR UPDATE ON review_product
 FOR EACH ROW
 EXECUTE PROCEDURE review_product_search_update();
```

```
-- Finally, create a GIN index for ts_vectors.
CREATE INDEX search_review_idx ON review_product USING GIN (tsvectors);
```

Index	**FTS03**
Index Relation	review_author
Field(s)	review_text
Weighting	1
Justification	Enables searching for reviews written by authors, assisting in filtering opinions.
SQL Code	

```
-- Add column to review_author to store computed ts_vectors.
ALTER TABLE review_author
ADD COLUMN tsvectors TSVECTOR;

-- Create a function to automatically update ts_vectors.
CREATE FUNCTION review_author_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
         setweight(to_tsvector('english', NEW.review_text), 'A')
        );
 END IF;
 IF TG_OP = 'UPDATE' THEN
         IF (NEW.review_text <> OLD.review_text) THEN
           NEW.tsvectors = (
             setweight(to_tsvector('english', NEW.review_text), 'A')
           );
         END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;

-- Create a trigger before insert or update on review_author.
CREATE TRIGGER review_author_search_update
 BEFORE INSERT OR UPDATE ON review_author
 FOR EACH ROW
 EXECUTE PROCEDURE review_author_search_update();

-- Finally, create a GIN index for ts_vectors.
CREATE INDEX search_author_review_idx ON review_author USING GIN (tsvectors);
```

3. Triggers

> User-defined functions and trigger procedures that add control structures to the SQL language or perform complex computations, are identified and described to be trusted by the database server.

> Every kind of function (SQL functions, Stored procedures, Trigger procedures) can take base types, composite types, or combinations of these as arguments (parameters). In addition, every kind of function can return a base type or a composite type. Functions can also be defined to return sets of base or composite values.

Trigger	**TRIGGER01**
Description	An Authenticated User can only review a product which he has previously bought.

```
DROP FUNCTION IF EXISTS check_product_purchase() CASCADE;
CREATE FUNCTION check_product_purchase() RETURNS TRIGGER AS
$BODY$
BEGIN
    IF NOT EXISTS (
        SELECT 1
        FROM "orderProduct" op
        JOIN "order" o ON o.id = op.id_order
        WHERE o.id_authenticated_user = NEW.id_authenticated_user
          AND op.id_product = NEW.id_product
    ) THEN
        RAISE EXCEPTION 'You can only review a product that you have bought';
    END IF;
    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS check_product_purchase ON "review_product";
CREATE TRIGGER check_product_purchase
BEFORE INSERT ON "review_product"
FOR EACH ROW
EXECUTE PROCEDURE check_product_purchase();
```

Trigger	**TRIGGER02**
Description	An Authenticated User cannot buy more than the available stock of a product.

```
DROP FUNCTION IF EXISTS limit_product_purchase() CASCADE;
CREATE FUNCTION limit_product_purchase() RETURNS TRIGGER AS
$BODY$
BEGIN
    IF (SELECT stock FROM "product" WHERE id = NEW.id_product) < NEW.quantity THEN
        RAISE EXCEPTION 'The solicited amount exceeds the product''s stock';
    END IF;
    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;
```

```
DROP TRIGGER IF EXISTS limit_product_purchase ON orderProduct;
CREATE TRIGGER limit_product_purchase
BEFORE INSERT ON "orderProduct"
FOR EACH ROW
EXECUTE PROCEDURE limit_product_purchase();
```

Trigger	**TRIGGER03**
Description	An Authenticated User can only review a product once.

```
DROP FUNCTION IF EXISTS limit_one_review_per_product() CASCADE;
CREATE FUNCTION limit_one_review_per_product() RETURNS TRIGGER AS
$BODY$
BEGIN
    IF EXISTS (
        SELECT 1
        FROM "review_product"
        WHERE id_authenticated_user = NEW.id_authenticated_user
          AND id_product = NEW.id_product
    ) THEN
        RAISE EXCEPTION 'You already reviewed this product.';
    END IF;
    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS limit_one_review_per_product ON review_product;
CREATE TRIGGER limit_one_review_per_product
BEFORE INSERT ON "review_product"
FOR EACH ROW
EXECUTE PROCEDURE limit_one_review_per_product();
```

Trigger	**TRIGGER04**
Description	When a product is bought, its stock is reduced by the amount purchased.

```
DROP FUNCTION IF EXISTS reduce_stock_after_purchase() CASCADE;
CREATE FUNCTION reduce_stock_after_purchase() RETURNS TRIGGER AS
$BODY$
DECLARE
    order_product RECORD;
BEGIN
    FOR order_product IN
        SELECT id_product, quantity
        FROM "orderProduct"
        WHERE id_order = NEW.id
    LOOP
        UPDATE "product"
        SET stock = stock - order_product.quantity
        WHERE id = order_product.id_product;
    END LOOP;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS reduce_stock_after_purchase ON "order";
CREATE TRIGGER reduce_stock_after_purchase
AFTER INSERT ON "order"
FOR EACH ROW
EXECUTE PROCEDURE reduce_stock_after_purchase();
```

Trigger	**TRIGGER05**
Description	When an Authenticated User buys a product that is present in his wishlist, it is removed from there.

```
DROP FUNCTION IF EXISTS remove_from_wishlist_after_purchase() CASCADE;
CREATE FUNCTION remove_from_wishlist_after_purchase() RETURNS TRIGGER AS
$BODY$
BEGIN
    DELETE FROM "wishlistProduct"
    WHERE id_product IN (
        SELECT id_product
        FROM "orderProduct"
        WHERE id_order = NEW.id
    ) AND id_wishlist = (
        SELECT id FROM "wishlist"
        WHERE id_authenticated_user = NEW.id_authenticated_user
    );

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS remove_from_wishlist_after_purchase ON "order";
CREATE TRIGGER remove_from_wishlist_after_purchase
AFTER INSERT ON "order"
FOR EACH ROW
EXECUTE PROCEDURE remove_from_wishlist_after_purchase();
```

Trigger	**TRIGGER06**
Description	A User cannot add a product more than once to his wishlist

```sql
DROP FUNCTION IF EXISTS prevent_duplicate_wishlist_items() CASCADE;
CREATE FUNCTION prevent_duplicate_wishlist_items() RETURNS TRIGGER AS
$BODY$
BEGIN
    IF EXISTS (
        SELECT 1
        FROM "wishlistProduct" wp
        JOIN "wishlist" w ON w.id = wp.id_wishlist
        WHERE w.id_authenticated_user = NEW.id_authenticated_user
          AND wp.id_product = NEW.id_product
    ) THEN
        RAISE EXCEPTION 'This product is already on your wishlist';
    END IF;
    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS prevent_duplicate_wishlist_items ON "wishlistProduct";
CREATE TRIGGER prevent_duplicate_wishlist_items
BEFORE INSERT ON "wishlistProduct"
FOR EACH ROW
EXECUTE PROCEDURE prevent_duplicate_wishlist_items();
```

Trigger	**TRIGGER07 - paymentApprovedNotification**
Description	Trigger for creating a notification when a payment is approved, linking the notification to the order.

```sql
DROP FUNCTION IF EXISTS create_payment_approved_notification() CASCADE;
CREATE FUNCTION create_payment_approved_notification() RETURNS TRIGGER AS $$
BEGIN
    -- Insert a notification for the payment approval
    INSERT INTO notification (id_authenticated_user, text)
    VALUES (NEW.id_authenticated_user, 'Your payment has been approved.');

    -- Insert the payment approved notification
    INSERT INTO paymentApprovedNotification (id_order, id_notification)
    VALUES (NEW.id_order, currval('notification_id_seq'));

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS trigger_payment_approved_notification ON
paymentApprovedNotification;
CREATE TRIGGER trigger_payment_approved_notification
AFTER INSERT ON paymentApprovedNotification
FOR EACH ROW
EXECUTE PROCEDURE create_payment_approved_notification();
```

Trigger	TRIGGER08 - changeOrderStatusNotification
Description	Trigger for creating a notification when the order status changes.

```
DROP FUNCTION IF EXISTS create_order_status_change_notification() CASCADE;
CREATE FUNCTION create_order_status_change_notification() RETURNS TRIGGER AS $$
BEGIN
    -- Insert a notification for the order status change
    INSERT INTO notification (id_authenticated_user, text)
    VALUES (NEW.id_authenticated_user, 'The status of your order has changed.');

    -- Insert the order status change notification
    INSERT INTO changeOrderStatusNotification (id_order, id_notification)
    VALUES (NEW.id_order, currval('notification_id_seq'));

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS trigger_order_status_change_notification ON order;
CREATE TRIGGER trigger_order_status_change_notification
AFTER UPDATE ON order
FOR EACH ROW
EXECUTE PROCEDURE create_order_status_change_notification();
```

Trigger	TRIGGER09 - productOnWishlistNotification
Description	Trigger for creating a notification when a product is added to the wishlist.

```
DROP FUNCTION IF EXISTS create_product_on_wishlist_notification() CASCADE;
CREATE FUNCTION create_product_on_wishlist_notification() RETURNS TRIGGER AS $$
BEGIN
    -- Insert a notification for the product added to the wishlist
    INSERT INTO notification (id_authenticated_user, text)
    VALUES (NEW.id_authenticated_user, 'A product has been added to your
wishlist.');

    -- Insert the product added to wishlist notification
    INSERT INTO productWishlistAvailableNotification (id_wishlist,
id_notification)
    VALUES (NEW.id_wishlist, currval('notification_id_seq'));

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS trigger_product_on_wishlist_notification ON
wishlistProduct;
CREATE TRIGGER trigger_product_on_wishlist_notification
```

```
AFTER INSERT ON wishlistProduct
FOR EACH ROW
EXECUTE PROCEDURE create_product_on_wishlist_notification();
```

4. Transactions

> Transactions needed to assure the integrity of the data.

TRAN01 - Insert New Product

Transaction	**TRAN01 - Insert New Product**
Description	When a new product is added, both product details and initial stock levels must be recorded to maintain inventory accuracy. If the insertion fails, a rollback prevents incomplete data entry.
Justification	Ensures data integrity by rolling back the insert if theres an error, maintaining inventory consistency.
Isolation Level	REPEATABLE READ

```
BEGIN TRANSACTION;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

-- Insert product details with initial stock
INSERT INTO product (product_name, description, id_category, id_author, price,
stock, image)
VALUES ($product_name, $description, $category_id, $author_id, $price,
$initial_stock, $image);

COMMIT;
```

TRAN02 - Update Product Details with Inventory Check

Transaction	**TRAN02 - Update Product Details with Inventory Check**
Description	Updates product details while ensuring the product has enough stock available before any changes. If stock check fails, the update is rolled back.
Justification	Prevents products with insufficient stock from being updated, ensuring accurate inventory.
Isolation Level	SERIALIZABLE

```
BEGIN TRANSACTION;

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
-- Check if there is enough stock before updating the product
IF (SELECT stock FROM product WHERE id = $product_id) >= $required_quantity THEN
    -- Update product details
    UPDATE product
    SET description = $new_description, price = $new_price
    WHERE id = $product_id;
ELSE
    ROLLBACK;
END IF;

COMMIT;
```

Annex A. SQL Code

> The database scripts are included in this annex to the EBD component.
>
> The database creation script and the population script should be presented as separate elements. The creation script includes the code necessary to build (and rebuild) the database. The population script includes an amount of tuples suitable for testing and with plausible values for the fields of the database.
>
> The complete code of each script must be included in the group's git repository and links added here.

A.1. Database schema

```


DROP TABLE IF EXISTS "paymentInfo" CASCADE;
DROP TABLE IF EXISTS "productOnCartPriceChangeNotification" CASCADE;
DROP TABLE IF EXISTS "changeOrderStatusNotification" CASCADE;
DROP TABLE IF EXISTS "productWishlistAvailableNotification" CASCADE;
DROP TABLE IF EXISTS "paymentApprovedNotification" CASCADE;
DROP TABLE IF EXISTS "notification" CASCADE;
DROP TABLE IF EXISTS "orderProduct" CASCADE;
DROP TABLE IF EXISTS "order" CASCADE;
DROP TABLE IF EXISTS "shoppingCartProduct" CASCADE;
DROP TABLE IF EXISTS "shoppingCart" CASCADE;
DROP TABLE IF EXISTS "review_author" CASCADE;
DROP TABLE IF EXISTS "review_product" CASCADE;
DROP TABLE IF EXISTS "wishlistProduct" CASCADE;
DROP TABLE IF EXISTS "wishlist" CASCADE;
DROP TABLE IF EXISTS "product" CASCADE;
DROP TABLE IF EXISTS "authenticated_user" CASCADE;
DROP TABLE IF EXISTS "author" CASCADE;
DROP TABLE IF EXISTS "category" CASCADE;
DROP TABLE IF EXISTS "admin" CASCADE;
DROP TABLE IF EXISTS "users" CASCADE;
CREATE TABLE "users" (
    id BIGSERIAL PRIMARY KEY,
    username VARCHAR(20) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL UNIQUE,
    fullName VARCHAR(255)
);
CREATE TABLE "admin" (
    id BIGINT PRIMARY KEY REFERENCES "users"(id) ON DELETE CASCADE
);
CREATE TABLE "category" (
    id BIGSERIAL PRIMARY KEY,
    categoryName VARCHAR(255) NOT NULL UNIQUE
);
CREATE TABLE "author" (
    id BIGSERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    biography TEXT NOT NULL,
    rating INTEGER CHECK (
        rating >= 0
        AND rating <= 5
    )
);
CREATE TABLE "authenticated_user" (
    id BIGINT PRIMARY KEY REFERENCES "users"(id) ON DELETE CASCADE
);
CREATE TABLE "product" (
    id BIGSERIAL PRIMARY KEY,
    id_category BIGINT REFERENCES "category"(id) ON DELETE CASCADE NOT NULL,
    id_author BIGINT REFERENCES "author"(id) ON DELETE CASCADE NOT NULL,
    title VARCHAR(255) NOT NULL,
    description TEXT NOT NULL,
    price FLOAT NOT NULL CHECK (price >= 0),
    stock INTEGER NOT NULL CHECK (stock >= 0),
    image BYTEA
);
--should have trigger to check size of wishlistProduct
CREATE TABLE "wishlist" (
    id BIGSERIAL PRIMARY KEY,
    id_authenticated_user BIGINT REFERENCES "authenticated_user"(id) ON DELETE
CASCADE NOT NULL UNIQUE
);
CREATE TABLE "wishlistProduct" (
    id BIGSERIAL PRIMARY KEY,
    id_wishlist BIGINT REFERENCES "wishlist"(id) ON DELETE CASCADE NOT NULL,
    id_product BIGINT REFERENCES "product"(id) ON DELETE CASCADE NOT NULL,
    UNIQUE (id_wishlist, id_product)
);
CREATE TABLE "review_product" (
    id BIGSERIAL PRIMARY KEY,
    id_product BIGINT REFERENCES "product"(id) ON DELETE CASCADE NOT NULL,
    id_authenticated_user BIGINT REFERENCES "authenticated_user"(id) ON DELETE
CASCADE NOT NULL,
    date DATE NOT NULL,
    comment TEXT NOT NULL,
    rating INTEGER CHECK (
        rating >= 1
        AND rating <= 5
    ) NOT NULL
```

```
);
CREATE TABLE "review_author" (
    id BIGSERIAL PRIMARY KEY,
    id_author BIGINT REFERENCES "author"(id) ON DELETE CASCADE NOT NULL,
    id_authenticated_user BIGINT REFERENCES "authenticated_user"(id) ON DELETE
CASCADE NOT NULL,
    date DATE NOT NULL,
    comment TEXT NOT NULL,
    rating INTEGER CHECK (
        rating >= 1
        AND rating <= 5
    ) NOT NULL
);
CREATE TABLE "shoppingCart" (
    id BIGSERIAL PRIMARY KEY,
    id_authenticated_user BIGINT REFERENCES "authenticated_user"(id) ON DELETE
CASCADE NOT NULL UNIQUE
);
CREATE TABLE "shoppingCartProduct" (
    id BIGSERIAL PRIMARY KEY,
    id_shoppingCart BIGINT REFERENCES "shoppingCart"(id) ON DELETE CASCADE NOT
NULL,
    id_product BIGINT REFERENCES "product"(id) ON DELETE CASCADE NOT NULL
);
CREATE TABLE "order" (
    id BIGSERIAL PRIMARY KEY,
    id_authenticated_user BIGINT REFERENCES "authenticated_user"(id) ON DELETE
CASCADE NOT NULL,
    orderDate DATE NOT NULL,
    arrivalDate DATE NOT NULL,
    shippedAddress VARCHAR(255) NOT NULL,
    totalPrice FLOAT NOT NULL CHECK (totalPrice >= 0),
    orderStatus VARCHAR(255) NOT NULL,
    CHECK (arrivalDate >= orderDate),
    CHECK(
        orderStatus IN (
            'processing',
            'shipped',
            'delivered',
            'in transit'
        )
    )
);
CREATE TABLE "orderProduct"(
    id BIGSERIAL PRIMARY KEY,
    id_order BIGINT REFERENCES "order"(id) ON DELETE CASCADE NOT NULL,
    id_product BIGINT REFERENCES "product"(id) ON DELETE CASCADE NOT NULL
);
CREATE TABLE "notification"(
    id BIGSERIAL PRIMARY KEY,
    id_authenticated_user BIGINT REFERENCES "authenticated_user"(id) ON DELETE
CASCADE NOT NULL,
    text VARCHAR(255) NOT NULL
);
```

```
CREATE TABLE "paymentApprovedNotification"(
    id BIGSERIAL PRIMARY KEY,
    id_notification BIGINT REFERENCES "notification"(id) ON DELETE CASCADE NOT
NULL,
    id_order BIGINT REFERENCES "order"(id) ON DELETE CASCADE NOT NULL
);
CREATE TABLE "productWishlistAvailableNotification"(
    id BIGSERIAL PRIMARY KEY,
    id_wishlist BIGINT REFERENCES "wishlist"(id) ON DELETE CASCADE NOT NULL,
    id_notification BIGINT REFERENCES "notification"(id) ON DELETE CASCADE NOT
NULL
);
CREATE TABLE "changeOrderStatusNotification"(
    id BIGSERIAL PRIMARY KEY,
    id_order BIGINT REFERENCES "order"(id) ON DELETE CASCADE NOT NULL,
    id_notification BIGINT REFERENCES "notification"(id) ON DELETE CASCADE NOT
NULL
);
CREATE TABLE "productOnCartPriceChangeNotification"(
    id BIGSERIAL PRIMARY KEY,
    id_shoppingCart BIGINT REFERENCES "shoppingCart"(id) ON DELETE CASCADE NOT
NULL,
    id_notification BIGINT REFERENCES "notification"(id) ON DELETE CASCADE NOT
NULL
);
CREATE TABLE "paymentInfo" (
    id BIGSERIAL PRIMARY KEY,
    id_authenticated_user BIGINT REFERENCES "authenticated_user"(id) ON DELETE
CASCADE NOT NULL UNIQUE,
    shippedAddress VARCHAR(255) NOT NULL,
    nif VARCHAR(9) NOT NULL UNIQUE,
    cardNumber VARCHAR(16) NOT NULL UNIQUE
);
```

A.2. Database population

```
INSERT INTO "users" (username, password, email, fullName)
VALUES (
        'user1',
        'password1',
        'user1@example.com',
        'User One'
    ),
    (
        'user2',
        'password2',
        'user2@example.com',
        'User Two'
    ),
    (
        'admin1',
```

```sql
            'adminpassword',
            'admin@example.com',
            'Admin User'
        );
INSERT INTO "admin" (id)
VALUES (3);
INSERT INTO "authenticated_user" (id)
VALUES (1),
    (2);
INSERT INTO "category" (categoryName)
VALUES ('Horror'),
    ('Comics'),
    ('Educational');
INSERT INTO "author" (name, biography, rating)
VALUES ('Author One', 'Biography of Author One', 4),
    ('Author Two', 'Biography of Author Two', 5);
INSERT INTO "product" (
        id_category,
        id_author,
        title,
        description,
        price,
        stock,
        image
    )
VALUES (
        1,
        1,
        'Book Title 1',
        'Description of Book 1',
        19.99,
        100,
        NULL
    ),
    (
        2,
        2,
        'Book Title 2',
        'Description of Electronics Item 1',
        299.99,
        50,
        NULL
    ),
    (
        3,
        1,
        'Book Title 3',
        'Description of Clothing Item 1',
        39.99,
        200,
        NULL
    );
```

Revision history

Changes made to the first submission:

GROUP2462, 05/11/2024

- António Ferreira, up202108834@up.pt
- João Afonso Santos, up202108805@up.pt
- José Ferreira, up202108836@up.pt
- Felix Miranda, up202401479@up.pt