الجمهوريـــة الجـزائريـة الديمـقراطيـة الشـعبية
République Algérienne Démocratique et Populaire
وزارة التـعليـم العـالي و البـحث العلـمي
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE

المدرسة الوطنية العليا للإعلام الآلي
(المعهد الوطني للتكوين في الإعلام الآلي سابقا)
Ecole nationale Supérieure d'Informatique
ex. INI (Institut National de formation en Informatique)

SGBDA Project

2nd year of Higher Education Cycle (2CS)

2023-2024

Information Systems and Software (SIL)

Theme 1:

Database-Centric Business Logic Implementation "Warehouse Management System"

Made by :
- Mazouz Abderrahmane Aimen (SL1)
- Islem Medjahdi (SL1)
- Bilal Arab (SL1)
- Abderrahmène Habouche (SL1)
- Ilyes Benabbes (SL1)

School year : 2023-2024

Contents

Chapter 1

Database-Centric Business Logic Implementation

1.1 Introduction:

In an era defined by data-driven decision-making, the intersection of database management and business logic implementation holds immense significance. As a group of five SIL1's students, we embarked on a journey to explore the depths of Advanced Database Management Systems. Our objective? To bridge the gap between theoretical knowledge and practical application by delving into the realm of SQL programmability.

In this document, we present our findings and insights derived from our project focused on implementing Database-Centric Business Logic. We chose the warehouse scenario as our real-world context, recognizing its pivotal role in various companies such as Amazon and AliExpress. Through this scenario, we aim to showcase the importance of efficient, scalable, and integrated data management solutions in today's technology landscape.

Within the warehouse environment, we navigate through the intricacies of writing business logic at both the database layer and the application layer. Through comparative analysis, we unravel the benefits and challenges of each approach.

1.2 Overview :

We begin our Project Presentation document with this overview section to provide a clear understanding of a warehouse system, its main functionalities, and our database schema representing the WMS: Warehouse Management System.

1.2.1 Warehouse system :

A **warehouse** is a building that keeps products safe before they're packed and shipped to customers. Warehouses are central locations that manage both inbound and outbound products. They are important for any business that sells physical products or works with suppliers from a wholesale marketplace. More space is needed to store products as a business's revenue increases.

Figure 1.1: Warhouse

The building is **divided into zones**, in which each product will be dispatched.The dispatching process is carried out by **the stock management agents.**

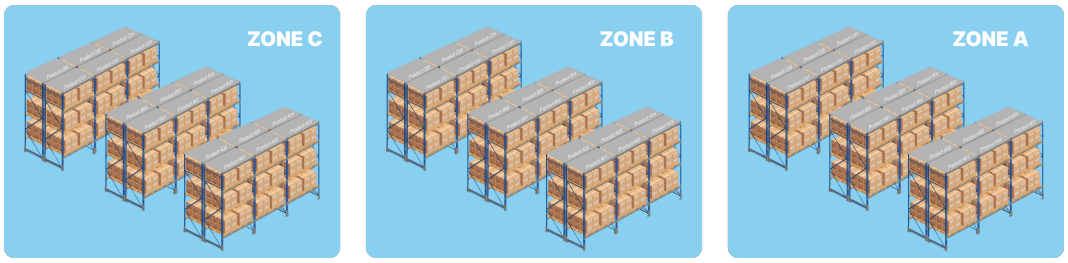

Figure 1.2: Zones

The agent will determine how to distribute the quantity of the product across the zones, **ensuring that no zone contains 2 or more stocks of the same product.**

Each zone is divided into stocks, each designated for a specific product.

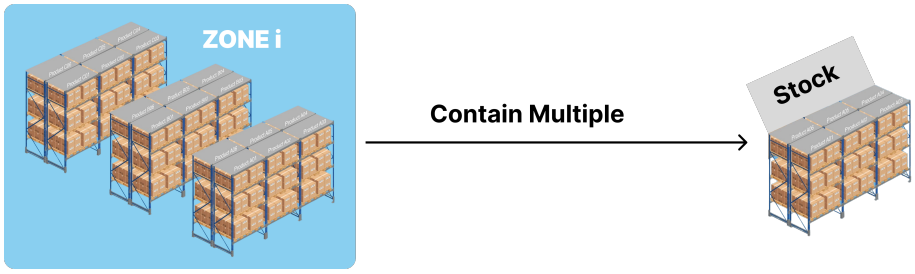

Figure 1.3: Stocks

1.2.2 Main functionalities :

The primary functionalities of our Warehouse Management System, implemented in our project, are **Inbound Requests and Outbound Requests** (This refers to the daily movement of in and out of the warehouse).

A- InBound request :

Handling the reception and storage of incoming products within the warehouse, ensuring efficient inventory management.

Through the following scenario, we gain a clear understanding of this process.

- Scenario:

The provider brings in a new product, 20 Asus laptops, to the warehouse. The agent at the counter creates an account for the provider if it's their first time. Upon receiving the 20 laptops, the agent creates a product entry named "Asus_Laptop" for this provider. Subsequently, this inbound request translates into product movements, where the agent dispatches 5 products to one stock in Zone A, 10 to another stock in Zone B, 3 to a stock in Zone C, and 2 to a stock in Zone D.

The sum of these stocks equals the quantity of this product in the warehouse, i.e., 20 laptops.

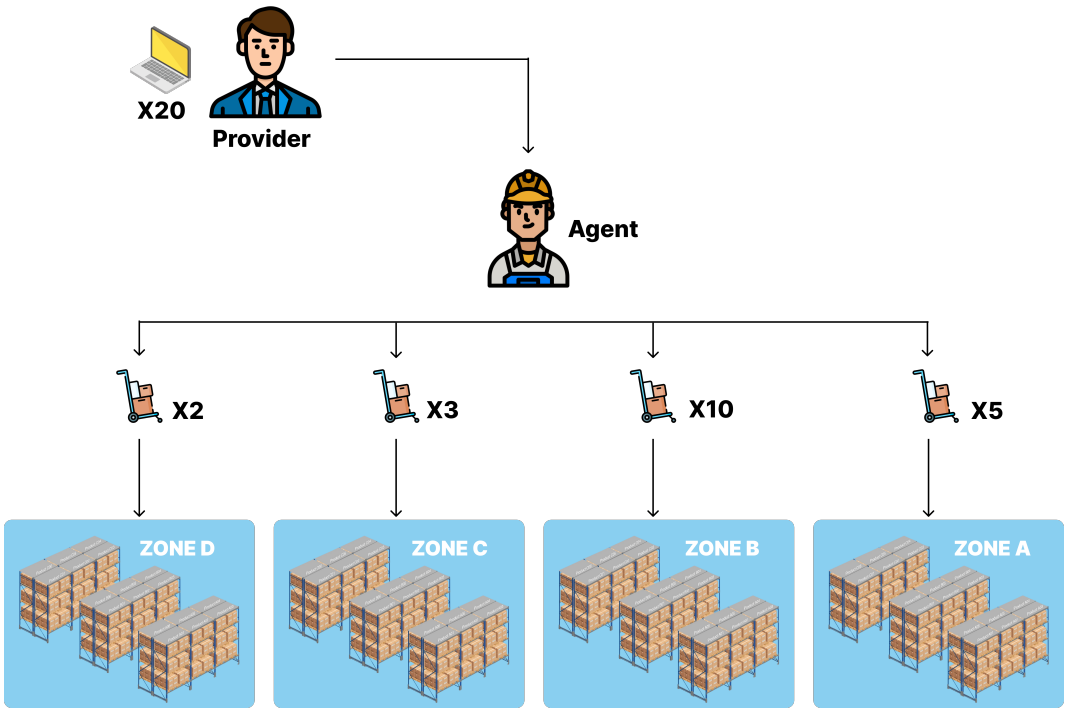

Figure 1.4: InBound Request

B- OutBound request :

Handling the retrieval of products from the warehouse,The logic of outbound processing starts by selecting the stock with the highest quantity, ensuring optimal utilization of available inventory.

Through the following scenario, we gain a clear understanding of this process.

- Scenario:

The provider demands 11 laptops. this Outbound request translates into product movements,The agent begins by selecting the stock with the highest quantity, discovering 10 laptops in Zone B. After retrieval, the agent moves to Zone A, where 5 laptops are found. One laptop is taken, leaving 4.

In total, 11 laptops were successfully retrieved, leaving the remaining quantity as 9, distributed as follows: 4 in Zone A, 3 in Zone C, and 2 in Zone D.

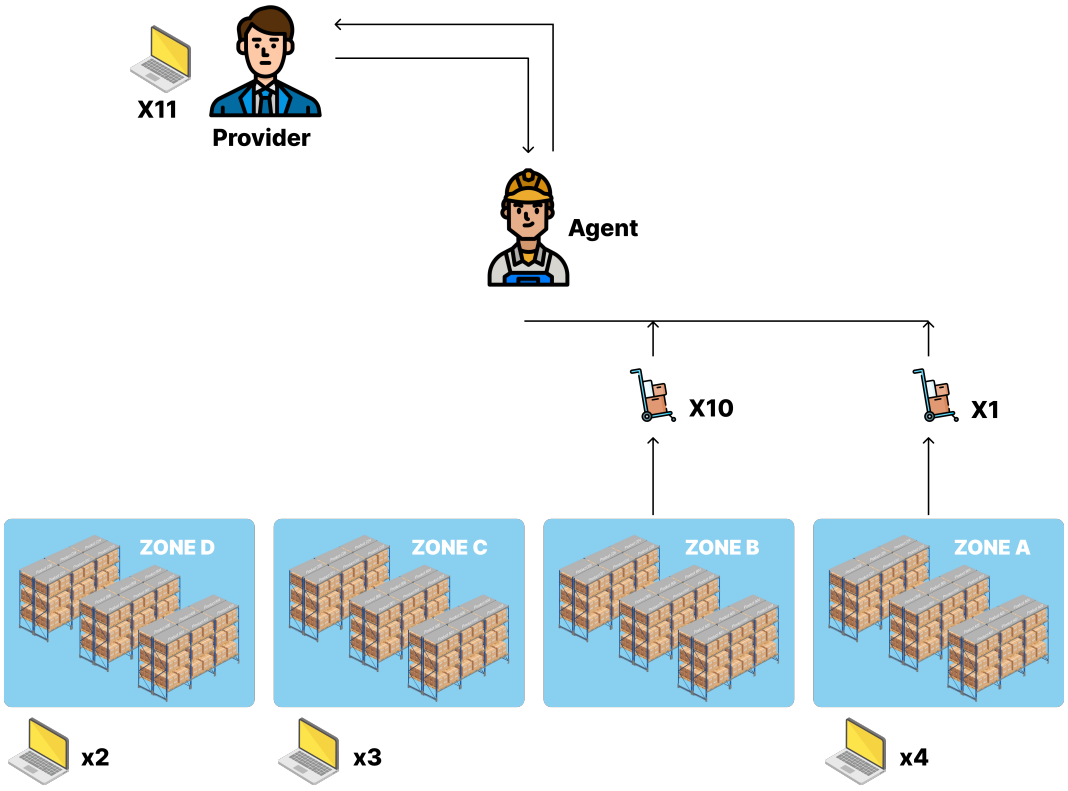

Figure 1.5: OutBound Request

1.2.3 Database Schema :

To simulate the WMS (Warehouse Management System), we propose the following database schema:

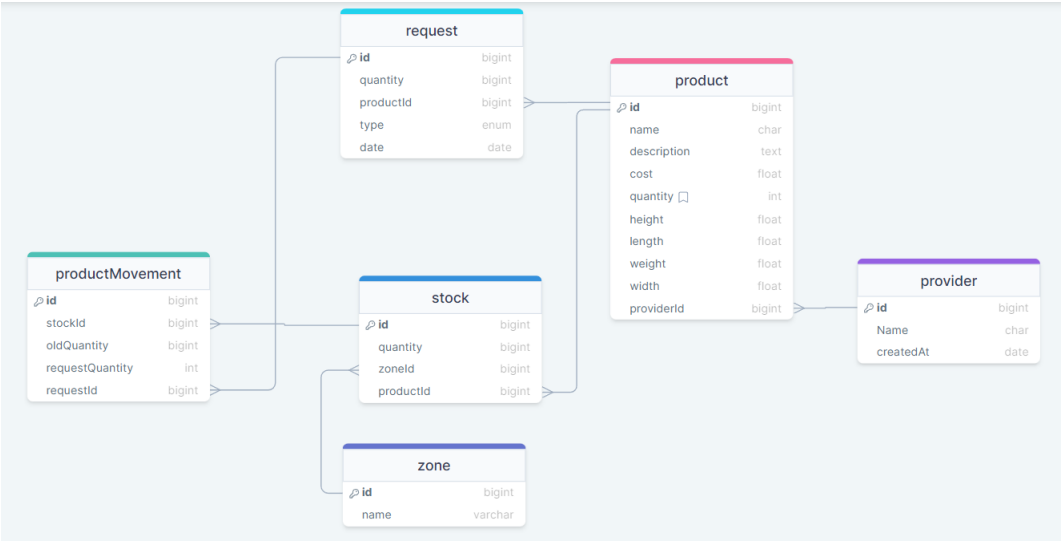

Figure 1.6: Database Schema

The request has two types: Inbound and Outbound, which translate into multiple product movements, including storing the product in a stock within a zone.

We'll start by detailing the implementation of our WMS: Warehouse Management System at the database layer, then we'll move on to discussing the implementation at the application layer.

1.3 Writing the business Logic in the Database layer (Implementation in PL/SQL SQL API):

In this section, we will delve into the implementation of our WMS at the database layer, utilizing PL/SQL and SQL API. We'll demonstrate how we leverage the concepts learned in the Advanced Database Management System (SGBDA) module to integrate business logic.

We'll illustrate this implementation through the scenarios outlined in the overview section.

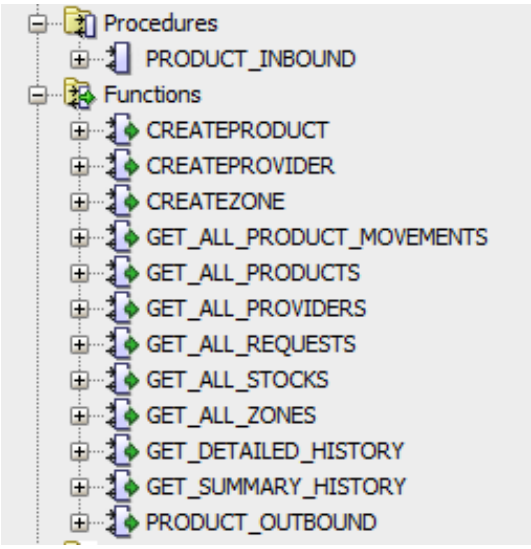

Figure 1.7: Database stored Procedures & Functions

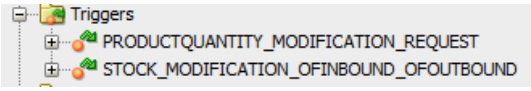

Figure 1.8: Database stored Triggers

1.3.1 Inbound request stored procedure:

```
--- PRODUCT INBOUND
CREATE OR REPLACE PROCEDURE product_inbound(
    productId_input IN NUMBER,
    zonesQuantities IN zone_quantity_list
)
IS
    total_quantity NUMBER := 0;
    requestId NUMBER;
    temporary_stock_id NUMBER;
    stock_quantity NUMBER;
    product_exists NUMBER;
    zone_exists NUMBER;
    stock_exists NUMBER :=0;
    --------------------
     temp_stock_id stock.id%TYPE; -- Variable to hold the stock ID
    temp_stock_quantity stock.quantity%TYPE; -- Variable to hold the
   stock quantity
    temp_stock_zone_id stock.zoneId%TYPE; -- Variable to hold the zone ID
     of the stock
BEGIN
    -- Verify if the product exists with the provided ID
    SELECT COUNT(*) INTO product_exists FROM product WHERE id =
   productId_input;
    IF product_exists = 0 THEN
```

```
22          RAISE_APPLICATION_ERROR(-20001, 'Product with ID ' ||
    productId_input || ' does not exist');
23      END IF;
24      -- Calculate total quantity
25      FOR i IN 1..zonesQuantities.COUNT LOOP
26          total_quantity := total_quantity + zonesQuantities(i).quantity;
27      END LOOP;
28      -- Create a new request entry for inbound products
29      INSERT INTO request (quantity, productId, requestType, requestDate)
30      VALUES (total_quantity, productId_input, 'in', SYSDATE)
31      RETURNING id INTO requestId; -- Retrieve the requestId
32      -- Iterate over the array of zone quantities
33      FOR i IN 1..zonesQuantities.COUNT LOOP
34          -- Accessing zone ID and quantity from the nested table
35          DECLARE
36              zone_id NUMBER;
37              quantity NUMBER;
38          BEGIN
39              -- Extracting zone ID and quantity from the nested table
40              zone_id := zonesQuantities(i).zone_id; -- Zone ID
41              quantity := zonesQuantities(i).quantity; -- Quantity
42
43              -- Verify if the zone exists
44          SELECT COUNT(*) INTO zone_exists FROM zone WHERE id = zone_id;
45              IF zone_exists = 0 THEN
46                  RAISE_APPLICATION_ERROR(-20002, 'Zone with ID ' ||
    zone_id || ' does not exist');
47              END IF;
48
49              -- Verify if stock exists for the product and zone
50                          DBMS_OUTPUT.PUT_LINE('zone exists ' || zone_id );
51                          DBMS_OUTPUT.PUT_LINE('prodcut exists ' ||
    productId_input );
52          SELECT COUNT(*) INTO stock_exists FROM stock WHERE (stock.
    productid = productId_input AND stock.zoneid= zone_id);
53              -- Log the result of the stock existence check
54          IF stock_exists > 0 THEN
55              DBMS_OUTPUT.PUT_LINE('Stock exists ' || stock_exists );
56          ELSE
57              DBMS_OUTPUT.PUT_LINE('Stock does not exist ' ||  stock_exists
    );
58          END IF;
59              --------------
60              IF stock_exists = 0 THEN
61                  -- Insert new stock entry if it doesn't exist
62                  INSERT INTO stock (quantity, zoneId, productId)
63                  VALUES (0, zone_id, productId_input);
64              END IF;
65              -- Retrieve the stock ID and quantity
66              SELECT id, quantity INTO temporary_stock_id, stock_quantity
67              FROM stock
```

```
            WHERE productId = productId_input
            AND zoneId = zone_id;
            -- Insert into productMovement
            INSERT INTO productMovement (stockId, oldQuantity,
    requestQuantity, requestId)
            VALUES (temporary_stock_id, stock_quantity, quantity,
    requestId);
        END;
    END LOOP;

    -- Commit the transaction
    COMMIT;
    -- Logging for debugging
    DBMS_OUTPUT.PUT_LINE('Transaction committed successfully');

EXCEPTION
    WHEN OTHERS THEN
        -- Handle exceptions
        DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
        ROLLBACK;
        -- Logging for debugging
        DBMS_OUTPUT.PUT_LINE('Transaction rolled back');
END;
```

1.3.2 Outbound request stored function:

Note : More details of the implementation are included in the code deliverable.

1.3.3 Trigger For In and Out Bound:

```
CREATE OR REPLACE TRIGGER stock_modification_ofInbound_ofOutbound
AFTER INSERT ON productMovement
FOR EACH ROW
DECLARE
    v_request_type request.requestType%TYPE;
BEGIN
    -- Retrieve the request type for the new product movement
    SELECT r.requestType INTO v_request_type
    FROM request r
    WHERE r.id = :NEW.requestId;

    -- Check if the operation is an INSERT
    IF INSERTING THEN
        -- Check if the request type is "in"
        IF v_request_type = 'in' THEN
```

```
            -- Add the quantity to the stock
            UPDATE stock
            SET quantity = quantity + :NEW.requestQuantity
            WHERE id = :NEW.stockId;
        -- Check if the request type is "out"
        ELSIF v_request_type = 'out' THEN
            -- Subtract the quantity from the stock
            UPDATE stock
            SET quantity = quantity - :NEW.requestQuantity
            WHERE id = :NEW.stockId;
        END IF;
    END IF;
END;
```

```
CREATE OR REPLACE TRIGGER productQuantity_modification_Request
AFTER INSERT ON request
FOR EACH ROW
BEGIN
    -- Check if the request type is "in"
    IF :NEW.requestType = 'in' THEN
        -- Add the quantity to the product
        UPDATE product
        SET quantity = quantity + :NEW.quantity
        WHERE id = :NEW.productId;
    -- Check if the request type is "out"
    ELSIF :NEW.requestType = 'out' THEN
        -- Subtract the quantity from the product
        UPDATE product
        SET quantity = quantity - :NEW.quantity
        WHERE id = :NEW.productId;
    END IF;
END;
```

1.4 Writing the Business logic in the Application layer:

In this section, we'll delve into the implementation of the Inbound module within our Warehouse Management System (WMS) at the **application layer** using Node.js. By adopting Node.js, we tap into the power of JavaScript and its expansive ecosystem of frameworks and libraries. This strategic decision allows us to encapsulate business logic beyond the confines of the database layer, thereby augmenting flexibility, scalability, and maintainability.

```
import OracleDB from "oracledb";
import { database } from "../database/connection";

```

```
export const createInboundRequestAppLicationLayer = async (
  productId: string,
  body: {
    zonesQuantities: { zone_id: number; quantity: number }[];
  }
): Promise<{
  executionTime: number;
}> => {
  const t1 = performance.now();

  const connection = await database.getConnection();

  try {
    let total_quantity = 0;
    let requestId;
    let temporary_stock_id;
    let stock_quantity;
    let product_exists;
    let zone_exists: number | undefined;
    let stock_exists: number | undefined = 0;

    // Verify if the product exists with the provided ID
    const productCheck = await connection.execute<[number]>(
      `SELECT COUNT(*) AS product_exists FROM product WHERE id = :
   productId`,
      { productId: productId }
    );
    product_exists = productCheck.rows?.[0][0];

    if (product_exists === 0) {
      throw new Error(`Product with ID ${productId} does not exist`);
    }

    // Calculate total quantity
    for (let i = 0; i < body.zonesQuantities.length; i++) {
      total_quantity += body.zonesQuantities[i].quantity;
    }

    console.log({
      total_quantity,
    });

    // Create a new request entry for inbound products
    const requestInsert = await connection.execute<{
      requestId: number[];
    }>(
      `INSERT INTO request (quantity, productId, requestType, requestDate
   )
       VALUES (:quantity, :productId, 'in', SYSDATE)
       RETURNING id INTO :requestId`,
      {
```

```
      quantity: total_quantity,
      productId: productId,
      requestId: { dir: OracleDB.BIND_OUT, type: OracleDB.NUMBER },
    },
    { autoCommit: false }
  );

  requestId = requestInsert.outBinds?.requestId[0];

  // Iterate over the array of zone quantities
  for (let i = 0; i < body.zonesQuantities.length; i++) {
    const zone_id = body.zonesQuantities[i].zone_id;
    const quantity = body.zonesQuantities[i].quantity;

    // Verify if the zone exists
    const zoneCheck = await connection.execute<[number]>(
      `SELECT COUNT(*) AS zone_exists FROM zone WHERE id = :zoneId`,
      {
        zoneId: zone_id,
      }
    );
    zone_exists = zoneCheck.rows?.[0][0];

    if (zone_exists === 0) {
      throw new Error(`Zone with ID ${zone_id} does not exist`);
    }

    // Verify if stock exists for the product and zone
    const stockCheck = await connection.execute<[number]>(
      `SELECT COUNT(*) AS stock_exists FROM stock WHERE productid = :
productId AND zoneid = :zoneId`,
      { productId: productId, zoneId: zone_id }
    );
    stock_exists = stockCheck.rows?.[0][0];

    if (stock_exists === 0) {
      // Insert new stock entry if it doesn't exist
      await connection.execute(
        `INSERT INTO stock (quantity, zoneId, productId) VALUES (0, :
zoneId, :productId)`,
        { zoneId: zone_id, productId: productId }
      );
    }

    // Retrieve the stock ID and quantity
    const stockInfo = await connection.execute<[number, number]>(
      `SELECT id, quantity FROM stock WHERE productId = :productId AND
zoneId = :zoneId`,
      { productId: productId, zoneId: zone_id }
    );
    temporary_stock_id = stockInfo.rows?.[0][0];
```

```
      stock_quantity = stockInfo.rows?.[0][1];

      // Insert into productMovement
      await connection.execute(
        `INSERT INTO productMovement (stockId, oldQuantity,
    requestQuantity, requestId)
         VALUES (:stockId, :oldQuantity, :requestQuantity, :requestId)`,
        {
          stockId: temporary_stock_id,
          oldQuantity: stock_quantity,
          requestQuantity: quantity,
          requestId: requestId,
        }
      );
    }

    await connection.commit();

    const t2 = performance.now();

    return { executionTime: t2 - t1 };
  } catch (e) {
    await connection.rollback();
    throw e;
  } finally {
    await database.closeConnection(connection);
  }
};
```

1.5 Comparison between 3 and 4 :

Through this comparison, we aim to present the pros and cons of embedding business logic within the database layer versus housing it within the application layer, with the goal of identifying the optimal choice for different software development scenarios.

By examining factors such as performance, flexibility, and security, we can better understand the trade-offs associated with each approach and determine which approach suits specific use cases best.

Let's explore the advantages and considerations of each approach in more detail :

Aspect	Database Layer Implementation	Application Layer Implementation
Advantages	**- Performance:** Since the business logic executes closer to the data, it can often be more efficient in terms of performance, especially for complex queries and data manipulations. **- Data Integrity:** Business rules enforced at the database layer can ensure data integrity and consistency, preventing invalid or inconsistent data from being stored. **- Security:** By encapsulating business logic within stored procedures or triggers, access to sensitive data can be controlled more effectively, enhancing security.	**- Flexibility:** Implementing business logic in the application layer allows for greater flexibility and agility in adapting to changing business requirements. **- Portability:** Business logic implemented in the application layer is typically more portable across different database platforms, reducing vendor lock-in. **- Ease of Maintenance:** Business logic implemented in code can be easier to maintain and debug, especially with modern development tools and version control systems.
Disadvantages	**- Vendor Lock-in: :** Implementing business logic within the database layer can tie the application to a specific database vendor, limiting portability. **- Maintenance Complexity:** Complex business logic implemented in stored procedures can be difficult to maintain and debug, especially as the application evolves. **- Scalability:** Database resources might become a bottleneck as the application scales, especially if business logic heavily relies on database resources.	**- Performance Overhead:** Business logic executed in the application layer may incur additional overhead, particularly for operations involving large datasets or complex computations. **Data Integrity Risks:** Without proper enforcement mechanisms, there's a risk of data integrity issues if business logic is not consistently applied across all layers of the application. **Security Concerns:** Business logic implemented in the application layer may be more susceptible to security vulnerabilities if not properly secured, potentially exposing sensitive data.

Aspect	Database Layer Implementation	Application Layer Implementation
Use Cases :	Ideal for applications where performance and data integrity are paramount, especially for transaction-heavy systems such as banking or e-commerce platforms.	Suitable for applications that require flexibility, portability, and ease of maintenance, such as web applications, mobile apps, and microservices architectures.

Our analysis indicated that executing the inbound request functionality within the Database layer resulted in significantly better average execution times.

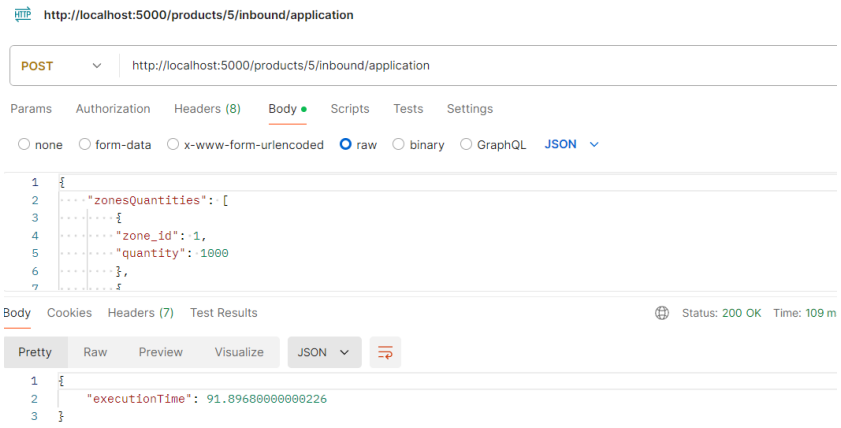

Figure 1.9: Application Layer

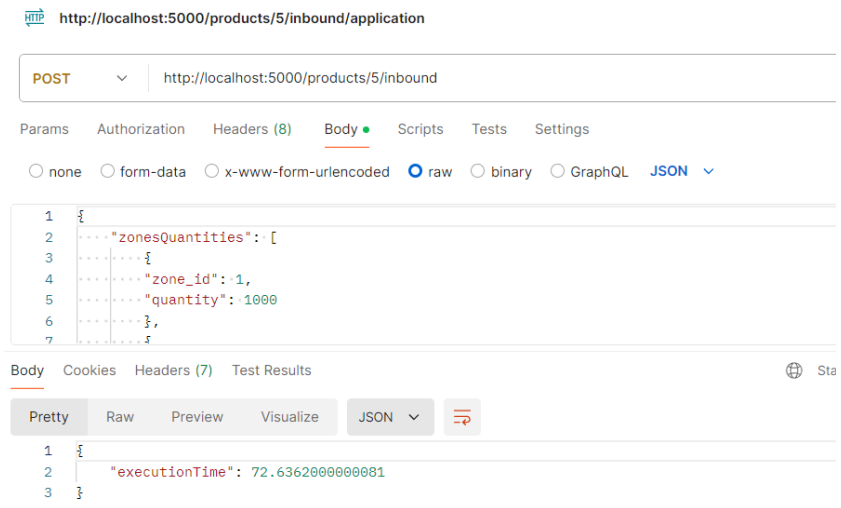

Figure 1.10: Database Layer

This finding is consistent with the typical advantage of database-layer implementations, where operations heavily reliant on data manipulation tend to perform more efficiently due to the close proximity of execution to the data itself.

1.6 SQL API :

An SQL API provides a structured interface that allows applications to interact with a database using SQL commands. This enables efficient data manipulation and retrieval, ensuring that operations such as querying, updating, and managing data are performed seamlessly.

In our project, we have created various endpoints in our SQL API to facilitate interactions with the database. Below are some examples illustrating these endpoints.

In our case, we have created this endpoints in our SQL API.

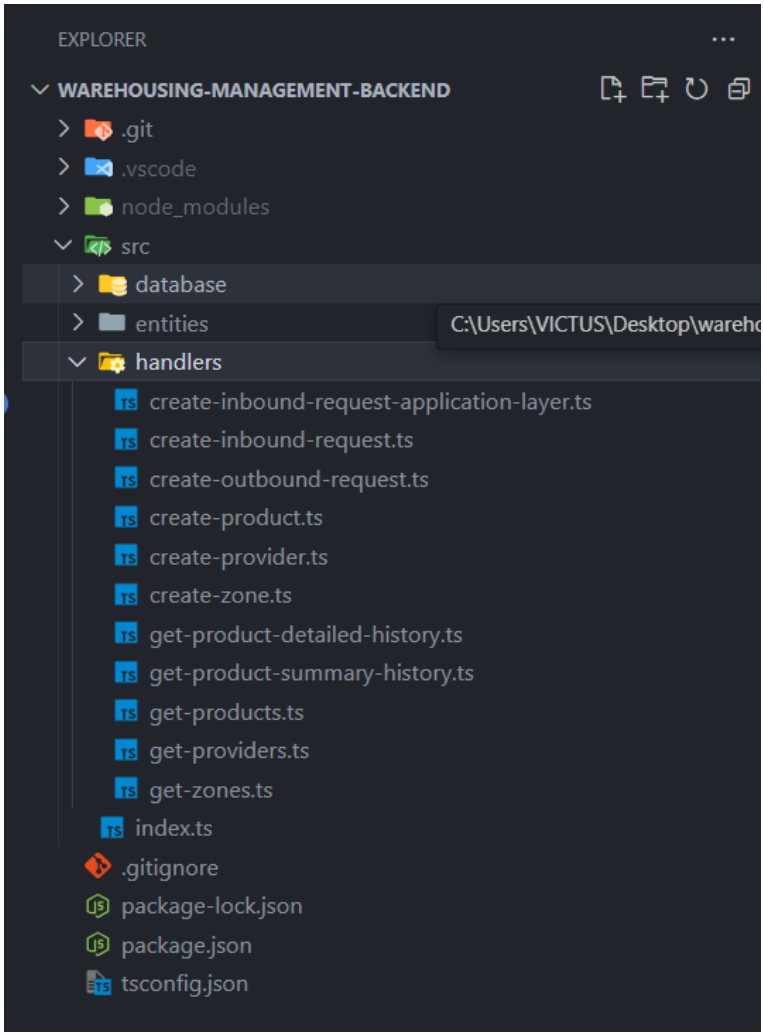

Figure 1.11: Various endpoints in our SQL API for WMS

We demonstrate how to call a stored function named ***"product outbound"*** using our SQL API.

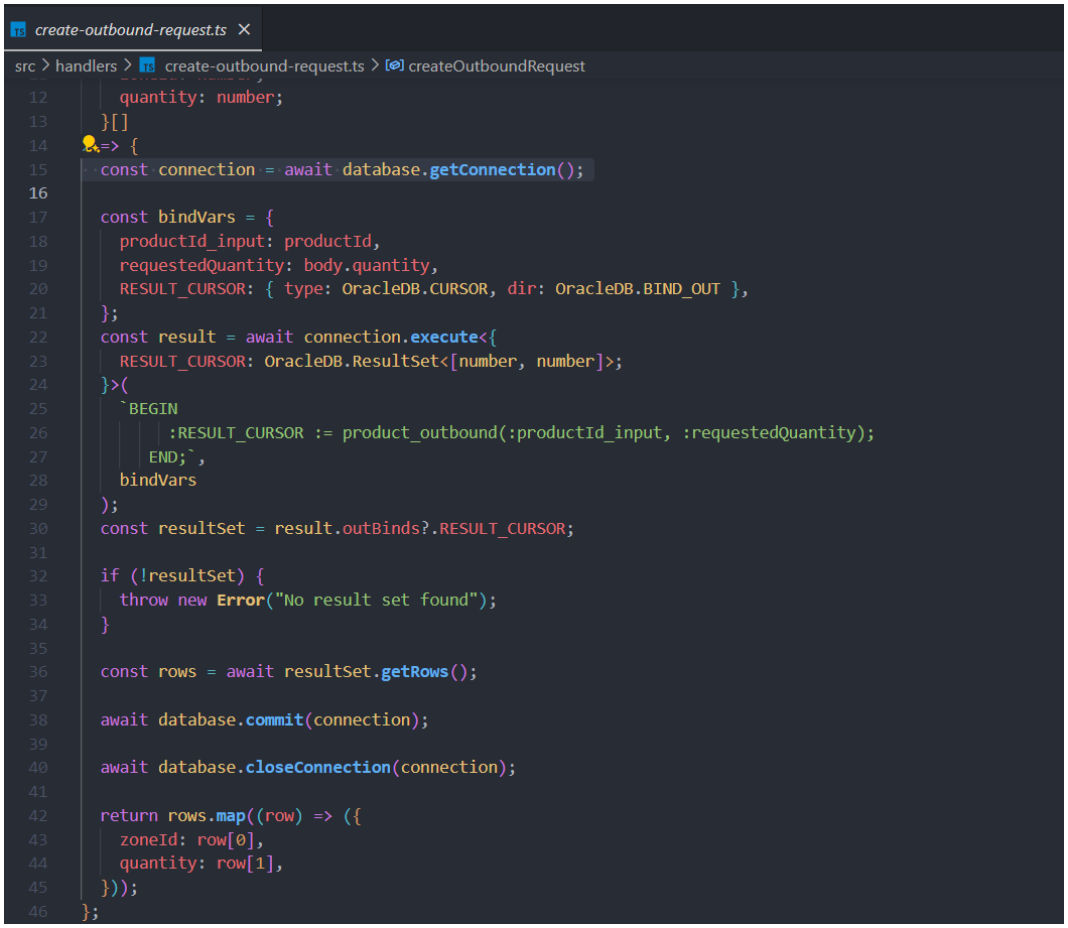

Figure 1.12: Calling the Stored Function for outbound Requests from our SQL API

1.7 Conclusion:

In this project, we delved into the implementation of Database-Centric Business Logic within a (WMS), examining both the Database layer and application layer approaches. Our comparative analysis highlighted the unique strengths and weaknesses of each method.

Implementing business logic at the database layer, particularly using PL/SQL, demonstrated significant advantages in terms of performance, data integrity, and security. By executing logic closer to the data, operations were more efficient, especially for complex queries and data manipulations. Additionally, enforcing business rules at the database layer ensured consistent data integrity and enhanced security by restricting direct data access.

Conversely, the application layer approach, implemented with Node.js, offered greater flexibility, portability, and ease of maintenance. This method allowed for rapid adaptation to changing business requirements and reduced dependency on specific database vendors. The application layer also benefited from modern development tools and practices, making code maintenance and debugging more straightforward. Despite these advantages, the application layer implementation may incur performance overhead and pose challenges related to data integrity and security.

Ultimately, the choice between database layer and application layer implementations should be guided by the specific requirements of the project, considering factors such as performance, flexibility, security, and maintenance needs. Our analysis revealed that database layer implementation yielded better performance metrics for transaction-heavy operations, while the application layer provided a more versatile and maintainable solution.

1.8 Bibliography

- Warehouse Process Flow: The Warehouse Management Process
- Warehouse Management Guide | How to Manage a Warehouse
- Business Logic: Database or Application Layer StackOverFlow
- Oracle Corporation. (2022). Oracle Database PL/SQL Language Reference. Oracle Help Center.
- Oracle Corporation. (2022). SQL API. Oracle Help Center.
- What is SQL API?
- Introduction to the SQL API
- Next.JS Framework documentation (For the frontend)
- Node.JS Framework documentation (For the backend)

1.9 Appendix :

In this section, we'll complement our discussion with some illustrative screenshots from our frontend interface. These screenshots offer a visual insight into the user experience and functionality of our Warehouse Management System (WMS) application.

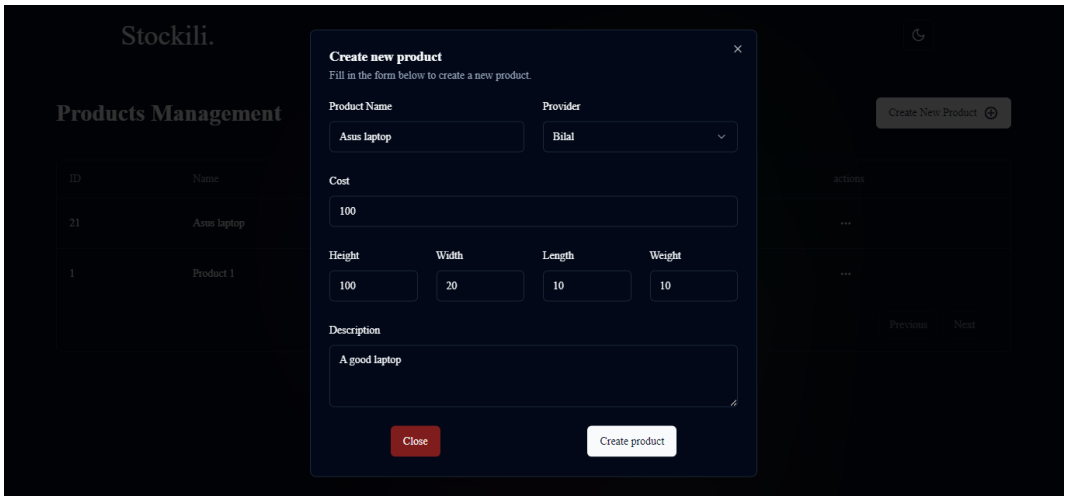

Figure 1.13: Create a new product

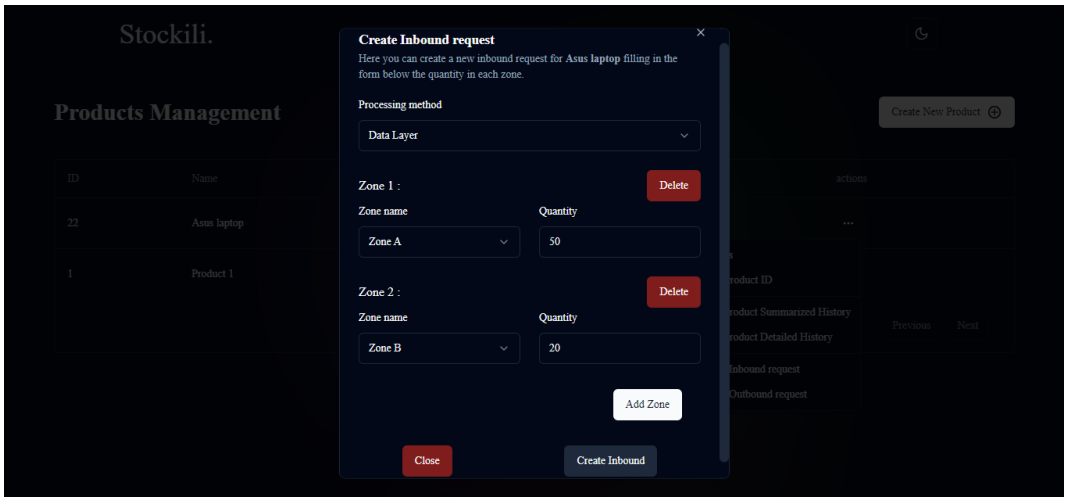

Figure 1.14: Create a new inbound request

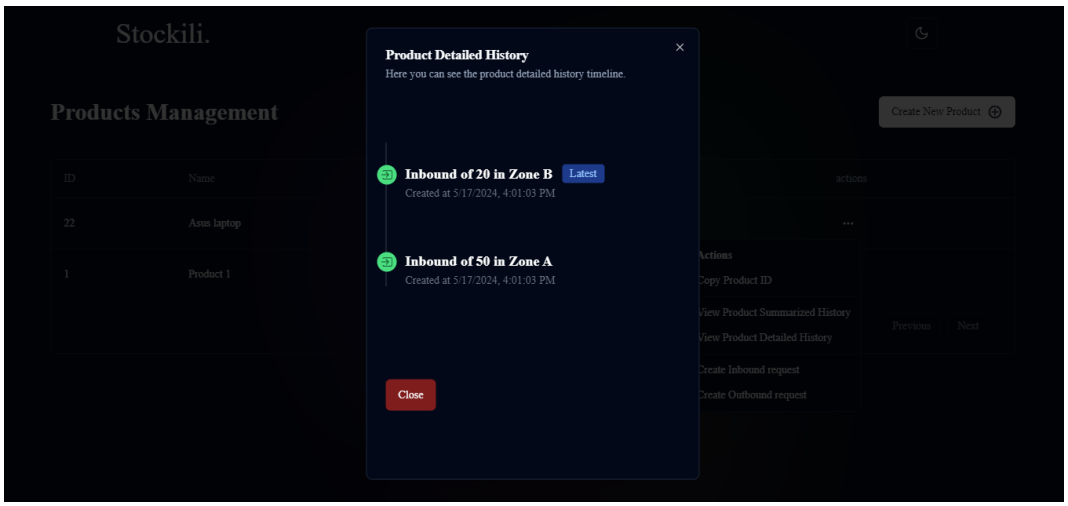

Figure 1.15: Product details history after inbound request

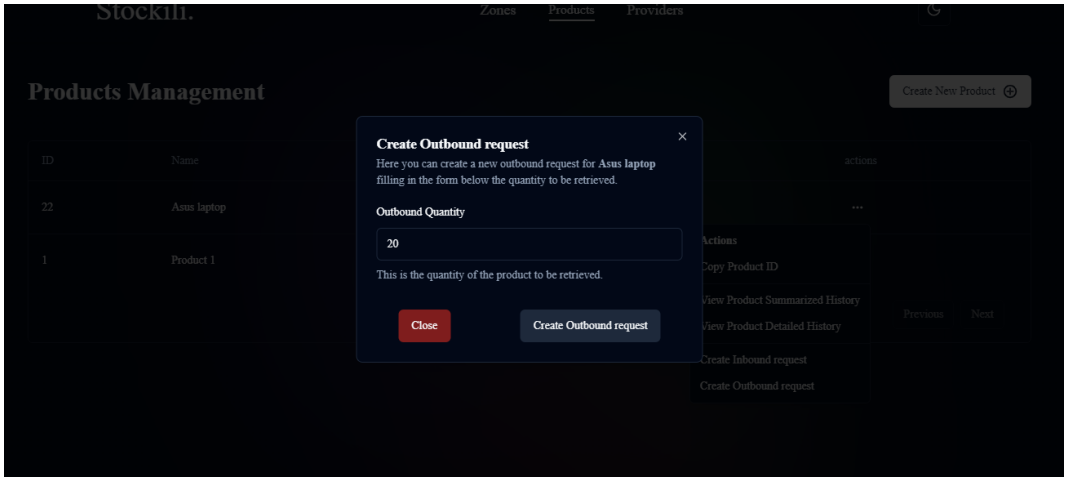

Figure 1.16: Create a new outbound request

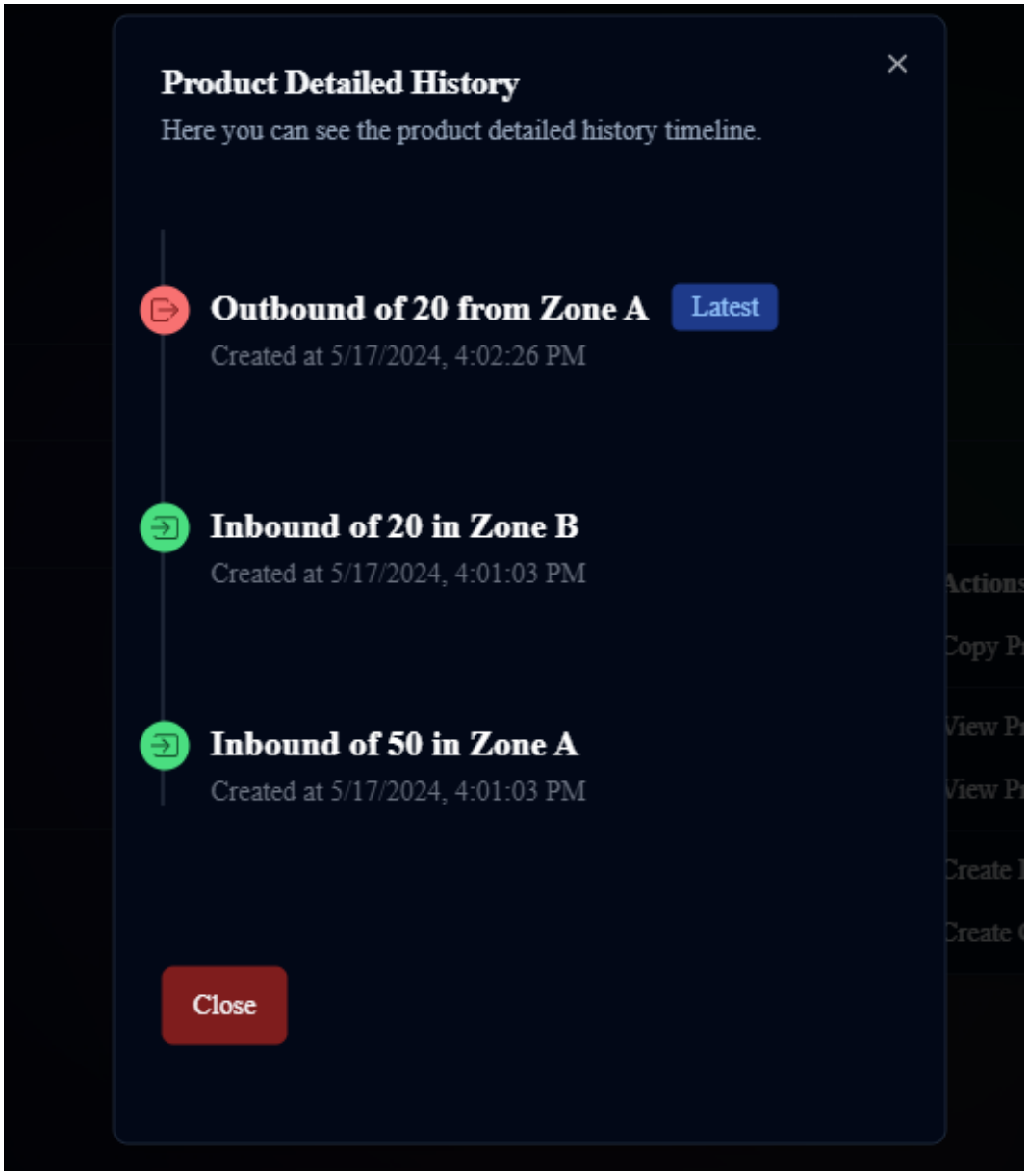

Figure 1.17: Product details history after outbound request