SESSION - 11

**✅ Why MySQL Accepts Remote Connections by Default (in your setup)?**

In **MySQL 8.x** (especially on RHEL-based systems):

* By default, **MySQL listens on all interfaces (0.0.0.0)**, but the **user permissions** control which host can connect.
* So even if it accepts remote connections, **users (like root)** must be granted access **from a specific host or IP**.
* That’s why **remote access is allowed**, but it’s **restricted at the user level** for security.

✅ So, **MySQL is listening for remote connections**, but it still needs **proper authentication and grants** for access.

**🔒 Why Username and Password Are Mandatory in MySQL (but not in MongoDB by default)?**

**🧩 MySQL (Relational DB):**

* MySQL **enforces strong authentication** for **every user**, including root and application users.
* Without setting a password, **you can’t even log in**, especially for newer versions like 8.x.
* **Security is mandatory** by default — a username & password are **always required**.

**🍃 MongoDB (NoSQL DB):**

* By **default**, MongoDB allows **unauthenticated access** locally (when auth is disabled).
* You must **manually enable authentication** in MongoDB’s config file (mongod.conf) to enforce usernames and passwords.
* This is often done later when security hardening is needed (especially in dev/test).

📌 That’s why in MongoDB, you might connect without a password if authentication is not enabled.

**🧠 Summary:**

| **Feature** | **MySQL** | **MongoDB** |
| --- | --- | --- |
| Auth Required | ✅ Yes (mandatory) | ❌ No (by default) |
| Default Bind Address | 0.0.0.0 (may vary) | 127.0.0.1 (localhost only) |
| Security Level | High (passwords enforced) | Low by default (unless enabled) |
| User Grants Needed | ✅ Yes | ✅ Optional if auth enabled |

**✅ MySQL Installation & Configuration (MySQL 8.x) for RoboShop**

**🔹 1. Install MySQL Server**

dnf install mysql-server -y

This command installs **MySQL 8.x** from the default AppStream repositories.

**🔹 2. Enable & Start MySQL Service**

systemctl enable mysqld

systemctl start mysqld

* enable: ensures MySQL starts on system boot.
* start: starts MySQL service immediately.

**🔹 3. Set Root Password**

MySQL 8 enforces password authentication. So we set the root password using:

mysql\_secure\_installation --set-root-pass RoboShop@1

* This sets the root password to **RoboShop@1**.
* It also secures your MySQL by disabling anonymous access and test databases.

**🔸 (Optional but Recommended)**

**✅ 4. Login to MySQL for the First Time**

mysql -u root -p

Then enter password: RoboShop@1

**🔹 5. Allow Remote Connections (if needed)**

By default, MySQL **listens on all interfaces**, but access is **controlled by user permissions**.

✅ Ensure firewall rules and security groups (on cloud) allow access on port **3306**.

**✅ Check if MySQL is Running**

systemctl status mysqld

**✅ Login to MySQL**

mysql -u root -p

Enter the password when prompted (RoboShop@1 if you followed previous steps).

**✅ Basic MySQL Commands**

Once you're inside the MySQL shell (mysql> prompt), use these:

**🔹 1. Show all databases**

SHOW DATABASES;

**🔹 2. Use a specific database**

USE shipping;

This switches to the shipping database.

**🔹 3. Show all tables in the current database**

SHOW TABLES;

**🔹 4. Describe the structure of a table**

DESCRIBE tablename;

**🚚 Shipping Component Overview:**

* **Language**: Java
* **Source Code Files**: .java
* **Build Tool**: Maven
* **Build Configuration File**: pom.xml
* **Dependencies**: Cart (Node.js), MySQL (Database)

**🧰 What is Maven?**

**Maven** is a **build automation tool** for Java projects. It manages:

* Project compilation
* Dependency management (libraries you need)
* Building .jar or .war files
* Testing & packaging

The configuration of everything is done in the **pom.xml** file (Project Object Model).

**📘 What is pom.xml?**

This is the **main configuration file** in Maven projects. It tells Maven:

* What project this is
* What dependencies (external libraries) to download
* How to compile/build/test the code

**🔍 Inside pom.xml: What are these?**

**✅ 1. Group ID**

* This is like your **company or organization name**.
* It’s a reverse domain style name (e.g., com.roboshop)
* Helps to uniquely identify your project.

**✅ 2. Artifact ID**

* This is the **name of your project or module**.
* Example: shipping or cart.

**✅ 3. Version**

* This is the **version of your application**.
* Example: 1.0.0, 1.0-SNAPSHOT

**🛠 What happens when you run mvn package?**

* Maven reads pom.xml
* Downloads all required libraries (dependencies)
* Compiles the .java files
* Packages the final .jar or .war file inside target/ directory

🔹 **Java**

* **Source Code**: Written in .java files.
* **Needs Compilation:** Must be **compiled** using tools like javac or **Maven**, which generates .class files.
* **Bytecode**: The .class files contain **bytecode**, which is not directly understandable by the operating system but is executed by the **Java** **Virtual Machine (JVM).**
* **Execution Flow:**  
  Java Source Code (.java) ➡️ Bytecode (.class) ➡️ JVM ➡️ Native Machine Execution

**✅ Java vs Node.js vs Python — Execution Flow & Compilation**

| **Feature** | **Java** | **Node.js / Python** |
| --- | --- | --- |
| **Source File** | .java | .js for Node.js, .py for Python |
| **Compilation Needed?** | ✅ Yes (via javac) | ❌ No (interpreted at runtime) |
| **Output** | .class (bytecode) | No compilation output |
| **Runtime Engine** | JVM (Java Virtual Machine) | Node.js runtime / Python interpreter |
| **Speed of Execution** | ✅ Faster (compiled + optimized) | Slower (interpreted line-by-line) |

**🔸 Java Compilation Flow:**

1. **Write Code**: ShippingService.java
2. **Compile**: javac ShippingService.java → Outputs ShippingService.class (Bytecode)
3. **Run**: java ShippingService (JVM executes the bytecode)

* ✅ **Bytecode** is platform-independent and optimized.
* JVM translates bytecode into **machine-level instructions** based on OS and hardware.

**🔸 Node.js & Python Flow:**

1. **Write Code**: app.js or app.py
2. **Run Directly**:
   * Node.js → node app.js
   * Python → python3 app.py
3. Code is interpreted **line-by-line**, so no compilation step.

**🚀 Why Java is Faster?**

* Java is **compiled → bytecode**, so it runs faster than pure interpreted code.
* JVM has **Just-In-Time (JIT)** compiler which further speeds up execution.
* Java is ideal for **large, high-performance applications**, like backend services.

**🧠 Summary:**

* ✅ **Java needs compilation**, Node.js and Python do not.
* ✅ Java produces **bytecode** understood by JVM → faster execution.
* ✅ Node.js & Python are interpreted → slower.
* ✅ Installing Maven also installs **Java** automatically on Linux.
* ✅ Java is more **strict**, **robust**, and **faster** for large-scale backend systems.

**🚚 Shipping Component Configuration (Java + Maven)**

**Purpose:**  
Shipping service is responsible for calculating the shipping cost based on the distance between the source and destination.

**Tech Stack:**

* **Language:** Java
* **Build Tool:** Maven (also installs Java as a dependency)
* **Source Code:** .java files
* **Build File:** pom.xml (contains dependency metadata)

**⚙️ Java & Maven Installation**

Maven is used to package Java applications, and it also installs Java (≥1.8) automatically.

dnf install maven -y

**👤 Application User Setup**

We use a non-root system user for running the application.

useradd --system --home /app --shell /sbin/nologin --comment "roboshop system user" roboshop

**📁 App Directory Setup**

mkdir /app

Download and extract application code:

curl -L -o /tmp/shipping.zip https://roboshop-artifacts.s3.amazonaws.com/shipping-v3.zip

cd /app

unzip /tmp/shipping.zip

**📦 Build the Application (Maven)**

cd /app

mvn clean package

mv target/shipping-1.0.jar shipping.jar

**🛠️ SystemD Service Setup**

Create systemd service file:

# /etc/systemd/system/shipping.service

[Unit]

Description=Shipping Service

[Service]

User=roboshop

Environment=CART\_ENDPOINT=<CART-SERVER-IPADDRESS>:8080

Environment=DB\_HOST=<MYSQL-SERVER-IPADDRESS>

ExecStart=/bin/java -jar /app/shipping.jar

SyslogIdentifier=shipping

[Install]

WantedBy=multi-user.target

Enable and start the service:

systemctl daemon-reload

systemctl enable shipping

systemctl start shipping

**🧩 Load MySQL Schema**

Install MySQL client:

dnf install mysql -y

Load schema files:

# Load database schema

mysql -h <MYSQL-IP> -uroot -pRoboShop@1 < /app/db/schema.sql

# Create application user for Shipping

mysql -h <MYSQL-IP> -uroot -pRoboShop@1 < /app/db/app-user.sql

# Load master data (country-city-distance info)

mysql -h <MYSQL-IP> -uroot -pRoboShop@1 < /app/db/master-data.sql

**🔄 Restart Shipping Service**

systemctl restart shipping

**✅ mvn clean package – Maven Command Explained Clearly**

Maven is a **build tool** used in Java projects to handle compilation, packaging, dependency management, testing, etc.

**📦 The full command:**

mvn clean package

This command is telling Maven to:

**1. 🧹 clean**

This **cleans** the previous build:

* Deletes the target/ directory (where compiled files and .jar files are placed).
* Removes old compiled files to **ensure a fresh build**.

🧠 **Why important?**

Avoids issues due to leftover files from old builds. Ensures you build from scratch.

**2. 📦 package**

This step:

* **Compiles** .java source files into **.class** bytecode files.
* **Packages** them into a .jar (Java Archive) file, typically stored in target/ directory.

🧠 **Result:** You get a deployable .jar file like:

target/shipping-1.0.jar

This .jar file is what you run using:

java -jar shipping-1.0.jar

**🎯 Summary in Simple Words:**

| **Command** | **Purpose** |
| --- | --- |
| mvn clean | Deletes old compiled files (cleans target folder) |
| mvn package | Compiles code + packages into a .jar file |
| mvn clean package | Cleans old files and then builds a fresh .jar file |
| mvn compile | Compile java source code and create bytecode in .class format in target folder. |

**✅ What is happening here?**

You're using the mysql client to **connect to a remote MySQL server**, and you're **executing SQL scripts** stored in files (.sql) to prepare the database for your **Shipping service** in the RoboShop project.

**🔍 Line-by-Line Explanation:**

**🔹 1. Load database schema**

mysql -h <MYSQL-IP> -uroot -pRoboShop@1 < /app/db/schema.sql

**👉 What's happening here?**

* mysql – MySQL client tool to connect to the database.
* -h <MYSQL-IP> – Connect to a MySQL server using its IP address.
* -u root – Log in as the MySQL root user (admin).
* -pRoboShop@1 – Provide the root password (RoboShop@1) directly.
* < /app/db/schema.sql – Redirect the contents of the schema.sql file and **execute all the SQL commands inside it**.

**🎯 What does schema.sql contain?**

It usually includes:

* CREATE DATABASE roboshop;
* USE roboshop;
* CREATE TABLE ... — to create necessary tables and relationships.

✅ This sets up the **structure** of your database (tables, columns, etc.).

**🔹 2. Create application user for Shipping**

mysql -h <MYSQL-IP> -uroot -pRoboShop@1 < /app/db/app-user.sql

**👉 What's happening here?**

Same as before, but executing the app-user.sql file.

**🎯 What does app-user.sql usually contain?**

It usually creates a user for the **Shipping application** to connect to MySQL:

CREATE USER 'shipping'@'%' IDENTIFIED BY 'shipping123';

GRANT ALL PRIVILEGES ON roboshop.\* TO 'shipping'@'%';

✅ This ensures the Shipping app has its own user with access to the roboshop database.

**🔹 3. Load master data**

mysql -h <MYSQL-IP> -uroot -pRoboShop@1 < /app/db/master-data.sql

**👉 What's happening here?**

Executing the third SQL file master-data.sql.

**🎯 What does master-data.sql usually contain?**

It contains INSERT statements like:

INSERT INTO countries (id, name) VALUES (1, 'India');

INSERT INTO cities (id, country\_id, name, distance) VALUES (1, 1, 'Hyderabad', 10);

✅ This loads **actual data** (called *master data*) like:

* Country names
* City names
* Distance values used by Shipping app to calculate shipping charges

**📌 Summary:**

| **Command** | **Purpose** |
| --- | --- |
| schema.sql | Creates the tables and database structure |
| app-user.sql | Creates a dedicated MySQL user for Shipping service |
| master-data.sql | Inserts static data like countries, cities, and distances |

All three commands are run as the **MySQL root user**, to fully configure the database needed for the **Shipping microservice**.

**✅ Why are we running MySQL commands for a Java-based Shipping service?**

You're running these MySQL commands **not because the Shipping service is written in Java**, but because:

🔸 **The Shipping service needs a database to store and read data.**

In this case, the **Shipping service is built in Java**, but it **stores its data in a MySQL database**. So, regardless of the language used (Java, Node.js, Python, etc.), if the application depends on a database, the database must be:

1. **Configured**
2. **Have tables (schema)**
3. **Have necessary data (like cities, distances)**
4. **Allow the app to connect using a user & password**

**🚚 Shipping Service Dependency Chain**

| **Component** | **Dependency** |
| --- | --- |
| Shipping (Java) | MySQL (stores city & distance data) |
| Shipping (Java) | Cart (for items being shipped) |

Your Shipping app calculates **shipping charges based on city-to-city distances**, which are stored in the **MySQL database**. Therefore:

* schema.sql → Creates the tables the app will use.
* app-user.sql → Creates a MySQL user so the app can log in securely.
* master-data.sql → Inserts the data your Java app needs to perform calculations.

**🔁 So why Java + MySQL?**

It’s a common architecture:

* Java: handles business logic
* MySQL: stores data like orders, addresses, distances
* The app connects to the database over network using host, username, password

**📌 Final Summary**

You're running these MySQL commands **not because of Java**, but because:

* The **Shipping microservice needs a database** (MySQL).
* The database must be **prepared** before the Java app can use it.
* These .sql files are provided by the developer to make the database **ready for use**.

**🚛 Example Scenario: Shipping Service in an E-commerce App**

Imagine you are running an online shopping site like **Amazon**. A customer orders a product. Now you need to calculate **how much to charge for shipping**.

Your **Java-based Shipping Service** is responsible for:

* Finding **where the product is shipping from** (e.g., Hyderabad)
* Finding **where the customer lives** (e.g., Mumbai)
* Checking the **distance between those two cities**
* Calculating shipping cost

But how does your app know the distance between cities?

➡️ That information is stored in a **MySQL database**.

**📦 Step-by-step with Example**

**🧱 1. Create Tables for Data**

You run:

mysql -h <MYSQL-IP> -uroot -pRoboShop@1 < /app/db/schema.sql

This SQL file contains commands like:

CREATE TABLE distance (

from\_city VARCHAR(100),

to\_city VARCHAR(100),

distance\_km INT

);

🧠 Now the MySQL database has a table to **store distances between cities**.

**👤 2. Create App User**

You run:

mysql -h <MYSQL-IP> -uroot -pRoboShop@1 < /app/db/app-user.sql

This SQL file might contain:

CREATE USER 'shipping\_user'@'%' IDENTIFIED BY 'shipping123';

GRANT ALL PRIVILEGES ON shipping.\* TO 'shipping\_user'@'%';

🧠 Now your Java app can connect to the MySQL DB using:

username: shipping\_user

password: shipping123

**🌍 3. Load Master Data**

You run:

mysql -h <MYSQL-IP> -uroot -pRoboShop@1 < /app/db/master-data.sql

This file may insert data like:

INSERT INTO distance VALUES ('Hyderabad', 'Mumbai', 700);

INSERT INTO distance VALUES ('Delhi', 'Chennai', 2100);

🧠 Now your database has **real distances** between cities.

**🚀 Now Java App Can Work!**

When a customer from Mumbai orders a product from Hyderabad:

* The Java service connects to MySQL using shipping\_user
* It runs a query like:

SELECT distance\_km FROM distance WHERE from\_city='Hyderabad' AND to\_city='Mumbai';

* Gets 700 KM
* Calculates ₹ Shipping Fee = 700 x ₹2 = ₹1400 (just an example)

**✅ Summary**

| **What You Did** | **Why You Did It** |
| --- | --- |
| Load schema.sql | To create necessary tables |
| Load app-user.sql | To create a DB user your Java app can use |
| Load master-data.sql | To insert important city-distance data |
| Java app connects to MySQL | To calculate shipping cost based on distance |

**Maven Lifecycle Phases (Key Ones)**

Here are the most common phases (in order):

| **Phase** | **Description** |
| --- | --- |
| validate | Checks if the project is correct and all necessary info is available |
| compile | Compiles the source code of the project |
| test | Runs unit tests using a testing framework (e.g., JUnit) |
| package | Packages the compiled code into a JAR, WAR, or other distributable format |
| verify | Runs any checks to verify the package is valid |
| install | Installs the package into the local Maven repository |
| deploy | Copies the final package to a remote repository for sharing with others |

To create **Route 53 records for Shipping and MySQL**, and apply **required changes**, follow this step-by-step guide. I'll explain **how to do it**, and **what needs to be changed in the service configuration**.

**✅ 1. Route 53 Setup for Shipping and MySQL**

Assuming your domain is: roboshop.internal  
We will create these DNS records:

| **Component** | **Private DNS Name** | **Points to (IP address)** |
| --- | --- | --- |
| MySQL | mysql.roboshop.internal | <MySQL-Private-IP> |
| Shipping | shipping.roboshop.internal | <Shipping-Private-IP> |

**Steps in AWS Route 53:**

1. **Go to Route 53 → Hosted Zones**
2. Select your hosted zone: roboshop.internal
3. Click **Create Record**

**➤ For MySQL**

* **Record Name**: mysql
* **Type**: A
* **Value**: <MySQL EC2 Private IP>

**➤ For Shipping**

* **Record Name**: shipping
* **Type**: A
* **Value**: <Shipping EC2 Private IP>

**✅ 2. Update Configuration Files with DNS Names**

**📌 A. Update Shipping Service systemd file:**

Open:

vim /etc/systemd/system/shipping.service

Update these lines:

Environment=CART\_ENDPOINT=cart.roboshop.internal:8080

Environment=DB\_HOST=mysql.roboshop.internal

Instead of using IP addresses, now you’re using **DNS names**.

Then reload and restart the service:

systemctl daemon-reload

systemctl restart shipping

**✅ 3. Update Frontend Config**

In the **frontend** component, the shipping service API must use the DNS name.

Open:

vim /etc/nginx/default.d/roboshop.conf

Look for a block like this (if you're using shipping endpoint):

location /shipping/ {

proxy\_pass http://shipping.roboshop.internal:8080/;

}

Make sure it points to:  
http://shipping.roboshop.internal:8080/

Then restart nginx:

systemctl restart nginx

**✅ Final Checklist**

| **Step** | **Component** | **Description** |
| --- | --- | --- |
| ✅ | Route 53 | Created A-records for mysql and shipping |
| ✅ | Shipping Service | Used DNS names in /etc/systemd/system/shipping.service |
| ✅ | Frontend | Used shipping.roboshop.internal in nginx config |
| ✅ | Restarted services | systemctl daemon-reload && systemctl restart shipping, nginx |

**🎯 Shipping Depends on:**

**✅ 1. Cart Service**

**✅ 2. MySQL Database**

**✅ 1. Why Shipping Needs Cart Service**

**📦 Example Scenario:**

Imagine a customer adds items to their **shopping cart** (TV, headphones, etc.).

Now the customer **proceeds to checkout** and selects **delivery**.

👉 At this point, the **Shipping service needs to know**:

* **What items are in the cart**
* **How many**
* **What is the total weight (to calculate shipping cost)**

**🔁 So, what does Shipping do?**

✅ It sends a request to the **Cart service (via HTTP API)**:

GET http://cart.roboshop.internal:8080/cart/12345

12345 = Cart ID or User ID

✅ Cart responds with:

{

"items": [

{ "name": "TV", "quantity": 1, "weight": 10 },

{ "name": "Headphones", "quantity": 2, "weight": 1 }

]

}

✅ Now, the **Shipping service** calculates:

* Total weight: 10 + (2 \* 1) = 12 kg
* Shipping cost: Let's say ₹10/kg = ₹120

**✅ 2. Why Shipping Needs MySQL**

The **Shipping service needs a database** to:

* Store past shipping records
* Store delivery addresses
* Store city-to-city distance & rates (loaded via master-data.sql)
* Store delivery tracking info
* Store users' shipping preferences

**Example:**

Shipping calculates the delivery cost from **Hyderabad to Mumbai** based on city-distance data from **MySQL**.

So MySQL might have:

SELECT distance\_km FROM city\_distance WHERE from\_city='Hyderabad' AND to\_city='Mumbai';

And store shipping records like:

INSERT INTO shipping\_records (user\_id, address, items, shipping\_cost)

VALUES ('12345', 'Mumbai, MH', '[TV, Headphones]', 120);

**🧠 Summary (Simple Table)**

| **Dependency** | **Why Needed** | **Real Example** |
| --- | --- | --- |
| Cart | To get list & weight of items | /cart/12345 → Item data for cost calc |
| MySQL | To store shipping records & location info | City distances, shipping history, etc. |