**🧑‍🏫 Session 8 - AWS & MobaXterm: Managing Multiple EC2 Instances**

**📝 Objectives:**

* Launch and manage EC2 instances using AWS Console
* Use **MobaXterm** for SSH access with **multi-tab** setup
* Understand **intranet vs internet IP** usage
* Prepare for managing **11 instances** in the future

**✅ Step-by-Step Class Flow**

**🔹 1. Introduction to Multi-instance Setup**

* In this session, we created **3 EC2 instances** as part of the 11-instance architecture needed in future projects.
* Goal: Understand how to manage multiple servers **easily and simultaneously** using tools.

**🔹 2. Tool Setup: MobaXterm**

* Installed and launched **MobaXterm** (Windows-based SSH tool).
* Benefits:
  + Supports **multiple tabs** (one tab per server)
  + Easy SSH configuration
  + Can save sessions with **default usernames**

**🔹 3. MobaXterm Configuration**

1. Go to **Settings > Configuration > SSH**
2. Set **default SSH username** = ec2-user  
   ✅ This skips entering the username every time

**🔹 4. Launch EC2 Instances in AWS**

We created **3 EC2 instances** using the following steps:

* Instance type: t3.micro
* No Key Pair selected (as we simulate passwordless access or will use internal setup)
* Security Group: Custom named **"allow-all"**
  + Inbound Rules:
    - Allow All Traffic (for training/demo purposes only)
* Click **Launch**

**🔹 5. Rename Instances**

After launch, rename the instances as follows:

| **Instance ID** | **New Name** |
| --- | --- |
| i-xxxxxx1 | frontend |
| i-xxxxxx2 | catalogue |
| i-xxxxxx3 | mongodb |

**🔹 6. Public vs Private IP (Concept Recap)**

| **Type** | **Use Case** |
| --- | --- |
| **Private IP** | Used for **intra-server** communication (within AWS VPC) |
| **Public IP** | Used to access from **outside the AWS network** (like your laptop) |

**🔹 7. Connect via MobaXterm**

* Open **MobaXterm**
* For each instance:
  1. Click **"Session" > SSH**
  2. Enter the **Public IP** of each instance
  3. Save and open the session
  4. Change each tab name to:
     + frontend
     + catalogue
     + mongodb
* Now you can work with all 3 servers in one screen using multiple tabs

**🧠 Summary**

| **Task** | **Tool / Feature Used** |
| --- | --- |
| Launch EC2 | AWS Management Console |
| Rename instances | AWS > Instances > Name |
| SSH Access | MobaXterm |
| Multi-tab SSH | MobaXterm tabs |
| Save login username | MobaXterm > SSH Config |
| Private vs Public IP Use | AWS Networking Concept |

**🧑‍🏫 Session 8 – Part 2: Frontend Setup in RoboShop Project using Nginx**

**💻 01 – Frontend Service Overview**

* **Frontend** is responsible for presenting the **web interface** to the user.
* This is **static content**, such as HTML, CSS, and JavaScript.
* To serve static content, we need a **Web Server**.

**🛠️ Web Server Used: Nginx**

* Developer has selected **Nginx** to serve static files.
* Nginx is fast, lightweight, and widely used for **frontend and reverse proxy** purposes.

**📦 Step-by-Step Frontend Setup**

**🔹 Step 1: Check Available Nginx Modules**

dnf module list nginx

**🔹 Step 2: Disable Default Module & Enable Specific Version**

dnf module disable nginx -y

dnf module enable nginx:1.24 -y

**🔹 Step 3: Install Nginx**

dnf install nginx -y

**🔹 Step 4: Enable and Start Nginx Service**

systemctl enable nginx

systemctl start nginx

* At this point, check in browser:  
  http://<Public-IP>  
  You should see the **default Nginx welcome page**.

**🔹 Step 5: Remove Default Content**

rm -rf /usr/share/nginx/html/\*

**🔹 Step 6: Download RoboShop Frontend**

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

**🔹 Step 7: Extract Frontend Content**

cd /usr/share/nginx/html

unzip /tmp/frontend.zip

* Now reload the browser.  
  You should see the **RoboShop UI page**.

**🔄 Nginx Reverse Proxy Configuration**

* Open the Nginx config file:

vim /etc/nginx/nginx.conf

* Delete everything and paste the configuration provided by the developer.

⚠️ **Do not share or edit the existing config unless instructed. Only paste the new developer-provided config.**

**🔁 Replace Proxy URLs**

In the configuration:

proxy\_pass http://localhost:8080/;

Replace localhost with the **Private IP** of the backend services (catalogue, user, cart, etc.)

This enables **Nginx to act as a reverse proxy** – directing API calls to the backend components.

**🔄 Restart Nginx to Apply Configuration**

systemctl restart nginx

**📁 Important Directories**

| **Directory Path** | **Description** |
| --- | --- |
| /usr/share/nginx/html/ | Web root directory where static files like index.html are kept |
| /etc/nginx/ | Nginx main configuration directory |
| /etc/nginx/nginx.conf | Main Nginx configuration file |
| /etc/nginx/conf.d/ | Extra custom site configurations |
| /var/log/nginx/ | Location for access and error logs |

🧑‍🏫 **Session 8 – Part 3: Setting Up MongoDB in RoboShop**

**SQL (Structured Query Language)**

* **Type**: Relational Database
* **Data Structure**: Tables (rows and columns)
* **Examples**: MySQL, PostgreSQL, Oracle, Microsoft SQL Server
* **Schema**: Fixed schema (predefined structure)
* **Query Language**: Uses SQL for querying data
* **Best For**: Structured data with clear relationships (e.g., financial data, HR systems)
* **ACID Compliance**: Strong support for transactions (Atomicity, Consistency, Isolation, Durability)

**NoSQL (Not Only SQL)**

* **Type**: Non-relational or distributed database
* **Data Structure**: Varies – Key-Value, Document, Column, Graph
* **Examples**: MongoDB (document), Redis (key-value), Cassandra (column), Neo4j (graph)
* **Schema**: Dynamic schema (flexible and scalable)
* **Query Language**: Varies; often uses custom APIs or JSON-based queries
* **Best For**: Unstructured or semi-structured data, big data, real-time applications
* **Scalability**: Highly scalable, good for distributed systems

**What is MongoDB?**

**MongoDB** is a **NoSQL, document-oriented database** used for storing large amounts of **unstructured or semi-structured data**. Instead of tables and rows (as in SQL databases), MongoDB stores data in **JSON-like documents** called **BSON** (Binary JSON).

**🔑 Key Features of MongoDB:**

* **Document-based**: Stores data in flexible, schema-less documents.
* **No fixed schema**: Each document can have different fields and structures.
* **High performance**: Fast read/write operations.
* **Scalable**: Supports horizontal scaling using **sharding**.
* **Rich queries**: Allows querying with filters, aggregations, and indexes.
* **Open-source**: Free and maintained by MongoDB Inc.
* Schema-less (flexible structure)
* Very fast for **big data applications**
* Used in projects where **large-scale unstructured or semi-structured data** is stored

**📚 Common Use Cases:**

* Real-time analytics
* Content management systems
* E-commerce applications
* Mobile apps
* IoT and sensor data storage

**Technical Architect / Software Architect**

* **Main Responsibility**: Designs the system architecture.
* **Decides**: High-level choices like backend frameworks, databases, infrastructure, and how components interact.
* **Why**: Ensures scalability, security, and maintainability.

**🔹 What is a repo?**

* A **repo** (repository) is a source that provides packages/software.
* Official repos (like CentOS) contain system packages.
* **MongoDB is third-party**, so we need to add its **own repo**.

**📁 Repo location:**

/etc/yum.repos.d/

**✅ Why We Add the MongoDB Repo to the Repos File in Linux:**

When you install MongoDB on a Linux system (like Ubuntu or CentOS), you **add the MongoDB repository (repo)** to your system’s list of software sources so that your **package manager (like apt or yum) knows where to find the official MongoDB packages**.

**🔍 What Happens When You Add the MongoDB Repo:**

1. **Makes MongoDB Packages Discoverable**  
   Your package manager (e.g., apt) doesn't know where to find MongoDB by default. Adding the MongoDB repo tells it:  
   *“Here’s the official MongoDB source for your system version.”*
2. **Ensures Latest and Official Versions**  
   You get **verified and up-to-date packages** directly from MongoDB Inc., not outdated versions in default Linux repos.

**📝 Why MongoDB and What This Document Means**

A **developer** has chosen to use **MongoDB**, a **NoSQL** database.

**📘 Example Document:**

{

"\_id": "6638fa12345abc",

"name": "Alice",

"age": 30,

"skills": ["Java", "DevOps"],

"address": {

"city": "Hyderabad",

"zip": "500001"

}

}

**🔍 Explanation:**

* This is how MongoDB stores data — in **JSON-like documents**.
* Flexible structure: Different documents can have different fields.
* Easy to store **nested** or **array** data like skills and address.
* \_id is a unique identifier for each document (like primary key in RDBMS).

**🛠️ Step-by-Step MongoDB Setup Explained**

We are setting up a **MongoDB** server on an EC2 instance in **AWS**, as per the developer's requirement.

**🧩 Step 1: Get Version Info From Developer**

"Versions of the DB Software you will get context from the developer"

* You shouldn't assume which version to install.
* The **developer decides the version** that fits their application’s compatibility.
* Here, it's **MongoDB 7.x**

**📁 Step 2: Create MongoDB Repo File**

vim /etc/yum.repos.d/mongo.repo

Add:

[mongodb-org-7.0]

name=MongoDB Repository

baseurl=https://repo.mongodb.org/yum/redhat/9/mongodb-org/7.0/x86\_64/

enabled=1

gpgcheck=0

**🧠 Why this step?**

* **yum** or **dnf** fetches packages from repositories.
* CentOS/RHEL doesn’t come with MongoDB repo by default.
* This repo is **from MongoDB’s official website** (3rd party)
* We must add this to get the **correct version**.
* Files under /etc/yum.repos.d/ are used by dnf/yum to find where to download packages from.

ℹ️ **/etc** = Extra Configuration → stores configuration files for services.

* gpgcheck=0 disables key verification (can be enabled for secure installs).

**💽 Step 3: Install MongoDB**

dnf install mongodb-org -y

**🧠 Why this step?**

* Installs MongoDB **binaries** and **required services** (mongod, mongos, etc.)
* -y auto-confirms prompts during installation.

**⚙️ Step 4: Start and Enable MongoDB**

systemctl enable mongod

systemctl start mongod

**🧠 Why this step?**

* enable: Starts MongoDB automatically **on boot**.
* start: Starts the **mongod** service (MongoDB server daemon).

At this stage, **only local applications can connect** to MongoDB.  
External servers (like frontend/catalogue) **will get connection refused** until you change bindIp to 0.0.0.0.

To check port:

**netstat -lntp**

You’ll see output like this:

tcp 0 0 127.0.0.1:27017 0.0.0.0:\* LISTEN <PID>/mongod

**🔒 Step 5: Allow Remote Access**

**Problem:**

By default, MongoDB listens on:

127.0.0.1:27017

👉 This means **only the local server** can talk to the DB.  
External servers (like **frontend** or **catalogue**) **cannot connect**.

**🧾 Step 6: Modify Configuration : Allow External Connections**

vim /etc/mongod.conf

Change:

bindIp: 127.0.0.1

To:

bindIp: 0.0.0.0

**🧠 Why this step?**

* 0.0.0.0 = Accept connections from **any IP address**.
* This tells MongoDB to listen on **all network interfaces** (including private IPs from other EC2s).
* This is needed if your database server is **separate** from your application server.

We are **temporarily opening access to everyone**. In future sessions, we’ll **restrict it only to the catalogue server** for security.

**🔁 Step 7: Restart MongoDB**

systemctl restart mongod

**🧠 Why this step?**

* Needed to **apply the configuration changes**.
* Otherwise, MongoDB will still be listening only on 127.0.0.1.

To verify listening port:

netstat -lntp

✅ You will see an output like:

tcp 0 0 0.0.0.0:27017 0.0.0.0:\* LISTEN <PID>/mongod

✅ **Now MongoDB is ready to accept connections** from other servers like:

* catalogue
* user
* cart, etc.

**✅ Summary:**

| **Step** | **Purpose** |
| --- | --- |
| Add Repo | So OS knows where to get MongoDB |
| Install MongoDB | To get MongoDB server and tools |
| Enable & Start | To make MongoDB run and auto-start on reboot |
| Change Bind IP | To allow remote servers to access MongoDB |
| Restart Service | To apply new configuration |
| Use netstat | Verify DB is accessible on the network |

**🛠️ Step-by-Step Catalogue Setup Explained**

**🧩 1. Project Context & Purpose**

* **Catalogue** microservice is part of the **roboshop** application.
* It is responsible for displaying the list of items/products on the app.
* Backend is built using **Node.js**, and it stores its data in **MongoDB**.

**🔧 2. Node.js Installation (Why & How)**

**➤ Why?**

* Application is built using **Node.js > v20**.
* System’s default Node.js version is 16, so we need to install version 20 manually.

**➤ Steps:**

dnf module list nodejs # See available Node.js versions

dnf module disable nodejs -y # Disable the default 16 version

dnf module enable nodejs:20 -y # Enable Node.js version 20

dnf install nodejs -y # Install Node.js v20

**✅ System User**

A **system user** is a special type of user account created **for running system-level processes or software**, not for logging into the system like a regular user.

**🔹 Key Characteristics:**

* **UID < 1000** (on most systems)
* Often has **no login shell** (/usr/sbin/nologin or /bin/false)
* No home directory (or a fake one like /nonexistent)
* Created by Linux itself or package maintainers

**🔸 Examples:**

* root – Superuser
* daemon, bin, sys, nobody – Various legacy or utility users
* mysql, mongodb, nginx, postfix – Users created by software packages

**✅ Service User (a type of system user)**

A **service user** is a **dedicated system user** created specifically for **running a particular service or daemon** (like a database or web server).

**🔹 Why use a separate service user?**

* 🔒 **Security isolation**: Limits damage if that service is compromised.
* 🔄 **Process ownership**: Helps track and manage which service is using which resources.

**🔸 Examples:**

* nginx → NGINX web server
* mysql → MySQL database
* mongodb → MongoDB server
* www-data → Used by Apache/NGINX in Debian systems

These users are **not meant for human login**.

**Why You Should NOT Run Services as root or a Regular User on a Linux Server:**

1. **Security Risk**:

If a service running as root is hacked, the attacker gets full control of the server.

1. **Least Privilege Principle**:

Services should only have the minimum access they need to reduce risk.

1. **System Damage Potential**:

A small bug in a root-run service can crash or corrupt the whole system.

1. **Better Isolation**:  
   Dedicated service users (like nginx, mysql) isolate services from each other and from the system.

Using separate users for each service keeps them safely separated from each other and the system.

1. **Cleaner Logs & Audits**:

Service-specific users make it easier to track which service did what.

1. **Avoid Conflicts with Login Users**:

Regular users have personal settings that could interfere with background services.

1. **Best Practice**:

Always use a special, non-login user for running each service securely.

**👤 3. System User Creation (Why roboshop user?)**

**➤ Why?**

* It's a best practice **not to run applications as root** user (security risk).
* Instead, we use a **system user** (also called service or daemon user), which:
  + Cannot log in (no shell access)
  + Is only used to run the service

**➤ Command:**

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

Then confirm:

cat /etc/passwd | grep roboshop

It **creates a system user** named roboshop with:

* A **home directory** at /app
* A **shell set to /sbin/nologin**, meaning the user cannot log in interactively, This is commonly used for system users, preventing login attempts and ensuring the user can only run background services.
* A **description** that indicates it's a system user for the "roboshop" service
* This user will be used for running background services related to the "roboshop" application.

**4. App Directory Setup and Code Deployment**

**➤ Steps:**

mkdir /app # Create directory for app code

curl -o /tmp/catalogue.zip https://... # Download code

cd /app

unzip /tmp/catalogue.zip # Unzip code into /app

**Dependencies and Libraries:**

Dependencies:

Dependencies are **external code packages** or **tools** that your application needs to function. They provide extra functionality that is not part of the core language.

Instead of writing everything from scratch, you use **existing, pre-written code** that solves specific problems (like working with a database, creating a web server, etc.).

Libraries:

A library is a collection of **pre-written code** (functions, classes, and methods) that helps you **perform common tasks** without reinventing the wheel.

Libraries are **included in your project** as dependencies. They provide you with ready-made solutions for common problems.

**Build Tools**

**What are Build Tools?**

Build tools help automate and streamline the process of preparing your code to run. They manage tasks such as:

* **Compiling code** (converting high-level language to machine code)
* **Downloading dependencies** (like libraries and packages)
* **Managing versions** of your dependencies
* **Running tests**, etc.

**Compilers**

A **compiler** is a tool that **converts human-readable code** (source code written in a programming language) into machine-readable code (binary files that the computer can execute).

* **What does it do?**
  + It **checks the syntax** of your code to make sure it follows the rules of the language.
  + It converts the code into a form that the computer can execute.
* You use a build tool like **npm** (JavaScript), **Maven** (Java), or **pip** (Python) to **automatically download** these libraries.

**What is package.json?**

package.json is a **JSON configuration file** located at the root of every Node.js project.  
It tells Node.js:

* What the project is
* What dependencies it needs
* What scripts to run
* Metadata like version, author, and license

**📦 5. Install Application Dependencies**

**➤ Why?**

* App uses open-source Node.js libraries, mentioned in package.json.
* These are needed for the app to work (like frameworks, DB drivers, etc.).

**➤ Command:**

npm install

* This will:
  + Read package.json
  + Download all dependencies from the internet
  + Create a node\_modules/ directory
  + Create or update package-lock.json (used to lock versions)

**Summary**

* package.json is the **blueprint** of your Node.js project.
* It keeps track of **dependencies**, **scripts**, and **project metadata**.
* It enables **easy project setup, builds, and automation**.

**Why is node\_modules/ Created?**

* 1. **Reads package.json**:  
     It checks which packages are listed under "dependencies" and "devDependencies".
  2. **Downloads the packages**:  
     It downloads those libraries from the npm registry (online).
  3. **Creates node\_modules/ folder** (if not already present):  
     It places all the downloaded libraries and their dependencies here.
  4. **Creates package-lock.json**:  
     This file locks the exact versions of all installed packages, to keep your environment consistent.

**What is package-lock.json?**

It’s a file that **remembers exactly** which versions of libraries were installed when you ran:

npm install

**🛠 Why do we need it?**

Let’s say your app needs **Express**.

* You write in package.json:  
  "express": "^4.17.1"
* That means:  
  "Install Express version 4.17.1 or any newer version that still works."

🔁 But if you and your friend run npm install at different times, **npm might install different versions** of Express.

🚫 That can lead to:

* Bugs
* Code that works on your machine but not on theirs

✅ So, **package-lock.json saves the exact versions** that were installed — not just for Express, but for every library Express depends on too.

**🛠️ 6. Set Up systemd Service for Catalogue**

**➤ Why?**

* So we can manage the app using systemctl (start, stop, enable on boot).
* Systemd gives logs, auto-restart, and is standard in Linux.

**➤ File: /etc/systemd/system/catalogue.service**

[Unit]

Description = Catalogue Service

[Service]

User=roboshop

Environment=MONGO=true

Environment=MONGO\_URL="mongodb://<MONGODB-IP>:27017/catalogue"

ExecStart=/bin/node /app/server.js

SyslogIdentifier=catalogue

[Install]

WantedBy=multi-user.target

**Replace <MONGODB-IP> with the private IP or DNS name of MongoDB server**.

We create **DNS (Domain Name System) records** to make websites and services reachable by **human-friendly names** like example.com instead of hard-to-remember IP addresses like 192.0.2.1

1. **Create a Record in Route 53**
2. **Update your systemd unit to use that name**
3. **Reload and restart the service**

**1. Create the Route 53 Record**

1. **Log in** to the AWS Console and open **Route 53 → Hosted zones**.
2. **Select** your domain’s Hosted Zone (e.g. roboshop.internal).
3. **Click** **“Create record”** and fill in:
   * **Record name**: e.g. mongodb
   * **Record type**: **A – IPv4 address** (or **CNAME** if you point at another DNS name)
   * **Value**: the **private IP** of your MongoDB server (e.g. 10.0.5.42)
   * **TTL**: e.g. 60 seconds
4. **Save** the record.

You’ll now be able to resolve mongodb.roboshop.internal (or whatever sub-domain you chose) to your MongoDB host.

**2. Update /etc/systemd/system/catalogue.service**

Replace the raw IP in Environment=MONGO\_URL with your new DNS name:

**➤ Reload and Start Service:**

systemctl daemon-reload

systemctl enable catalogue

systemctl start catalogue

**Now your Catalogue service will always connect via the DNS name you manage in Route 53**

**✅ Check if DNS is updated:**

* **Run:**
* **nslookup mongodb.roboshop.internal**
* **If successful, it shows:**
  + **DNS server used**
  + **IP address resolved (e.g. 10.0.5.42)**
* **If not working:**
  + **Make sure the Route 53 record exists**
  + **Check that the server has correct DNS settings**

**7.Steps to Install MongoDB Client (mongosh) in RHEL 9 / CentOS 9:**

To install MongoDB tools like mongosh, your system needs to know where to find them. That's why we add the MongoDB repo.

sudo vim /etc/yum.repos.d/mongo.repo

# Paste the given repo config to install mongodb-client

**Install mongosh** (MongoDB shell):

dnf install mongodb-mongosh -y

**Load Master Data:**

mongosh --host MONGODB-SERVER-IP </app/db/master-data.js

**💡 What it does:**

* **mongosh**: Starts the MongoDB shell (MongoDB's command-line client).
* **--host MONGODB-SERVER-IPADDRESS**: Connects to a remote MongoDB server using its IP address.
* **</app/db/master-data.js**: Feeds the JavaScript file (master-data.js) into the shell to run its MongoDB commands automatically (< is input, > is output).

1. **Frontend and DNS Integration**

**🧾 Steps to Do This in Route 53:**

1. **Go to AWS Console → Route 53 → Hosted Zones**
2. **Select your domain (e.g. roboshop.internal)**
3. **Click “Create record”**
4. **Fill i**
   * **Record name: catalogue**
   * **Record type: A (IPv4 address)**
   * **Value: IP address of your Catalogue service (e.g. 10.0.5.42)**
5. **Click Create record**

**To make Nginx forward frontend requests to the Catalogue service, you need to edit the Nginx config file:**

**📝 File to edit:**

**/etc/nginx/nginx.conf**

**🔁 What to do:**

**Inside the Nginx config, you’ll find a section like this:**

location /catalogue/ {

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

}

✅ Replace catalogue.roboshop.internal with your actual DNS name (from Route 53), and make sure the port matches the one your catalogue app is running on.

**💾 After editing:**

Run:

sudo nginx -t # to test for errors

sudo systemctl reload nginx