# Docker-Compose

# Services

### **1️⃣ PostgreSQL (Relational Database)**

postgresql:

container\_name: ms\_pg\_sql

image: postgres

environment:

POSTGRES\_USER: postgres

POSTGRES\_PASSWORD: root

PGDATA: /data/postgres

volumes:

- postgres:/data/postgres

ports:

- "5432:5432"

networks:

- microservices-net

restart: unless-stopped

✅ **Purpose:**

* This sets up a **PostgreSQL database**, a widely used relational database system.

✅ **Key Configurations:**

* image: postgres → Uses the official **PostgreSQL** Docker image.
* container\_name: ms\_pg\_sql → The name of the running container.
* environment:  
  + POSTGRES\_USER: postgres → Sets the default database user as **postgres**.
  + POSTGRES\_PASSWORD: root → Sets the password for authentication.
  + PGDATA: /data/postgres → Stores database data in **/data/postgres** instead of the default location.
* volumes:  
  + Stores PostgreSQL data in a persistent volume (postgres:/data/postgres) so that data isn’t lost when the container restarts.
* ports:  
  + 5432:5432 → Maps PostgreSQL's default port **5432** from the container to the host.
* networks:  
  + Connects this container to the custom **microservices-net** network for communication with other services.
* restart: unless-stopped → Restarts automatically unless manually stopped.

### **2️⃣ pgAdmin (PostgreSQL Admin Interface)**

pgadmin:

container\_name: ms\_pgadmin

image: dpage/pgadmin4

environment:

PGADMIN\_DEFAULT\_EMAIL: ${PGADMIN\_DEFAULT\_EMAIL:-pgadmin4@pgadmin.org}

PGADMIN\_DEFAULT\_PASSWORD: ${PGADMIN\_DEFAULT\_PASSWORD:-admin}

PGADMIN\_CONFIG\_SERVER\_MODE: 'False'

volumes:

- pgadmin:/var/lib/pgadmin

ports:

- "5050:80"

networks:

- microservices-net

restart: unless-stopped

✅ **Purpose:**

* pgAdmin is a web-based GUI to manage and interact with **PostgreSQL**.

✅ **Key Configurations:**

* image: dpage/pgadmin4 → Uses the official **pgAdmin 4** image.
* environment:  
  + PGADMIN\_DEFAULT\_EMAIL: ${PGADMIN\_DEFAULT\_EMAIL:-pgadmin4@pgadmin.org} → The default email to log into **pgAdmin**.
  + PGADMIN\_DEFAULT\_PASSWORD: ${PGADMIN\_DEFAULT\_PASSWORD:-admin} → The default password for logging in.
  + PGADMIN\_CONFIG\_SERVER\_MODE: 'False' → Runs pgAdmin in **desktop mode** (not multi-user mode).
* volumes:  
  + pgadmin:/var/lib/pgadmin → Persists configuration and session data.
* ports:  
  + 5050:80 → Maps port **80** inside the container to port **5050** on the host (accessible via http://localhost:5050).
* networks:  
  + Connects pgAdmin to **microservices-net** so it can communicate with PostgreSQL.

### **3️⃣ MongoDB (NoSQL Database)**

mongodb:

image: mongo

container\_name: ms\_mongo\_db

ports:

- 27017:27017

volumes:

- mongo:/data

environment:

- MONGO\_INITDB\_ROOT\_USERNAME=aditya

- MONGO\_INITDB\_ROOT\_PASSWORD=aditya

✅ **Purpose:**

* MongoDB is a NoSQL database used for storing JSON-like documents.

✅ **Key Configurations:**

* image: mongo → Uses the official **MongoDB** image.
* container\_name: ms\_mongo\_db → Sets the container name.
* environment:  
  + MONGO\_INITDB\_ROOT\_USERNAME=aditya → Creates an **admin user** named aditya.
  + MONGO\_INITDB\_ROOT\_PASSWORD=aditya → Sets the password to aditya.
* volumes:  
  + mongo:/data → Stores MongoDB data persistently.
* ports:  
  + 27017:27017 → Maps MongoDB’s default **27017** port from the container to the host.

### 

### 

### **4️⃣ Mongo Express (MongoDB Admin Interface)**

mongo-express:

image: mongo-express

container\_name: mongo\_express

restart: always

ports:

- 8081:8081

environment:

- ME\_CONFIG\_MONGODB\_ADMINUSERNAME=aditya

- ME\_CONFIG\_MONGODB\_ADMINPASSWORD=aditya

- ME\_CONFIG\_MONGODB\_SERVER=mongodb

✅ **Purpose:**

* Mongo Express is a web-based UI to interact with **MongoDB**.

✅ **Key Configurations:**

* image: mongo-express → Uses the official **Mongo Express** image.
* container\_name: mongo\_express → The name of the container.
* environment:  
  + ME\_CONFIG\_MONGODB\_ADMINUSERNAME=aditya → Connects as the **admin user**.
  + ME\_CONFIG\_MONGODB\_ADMINPASSWORD=aditya → Uses the same password as MongoDB.
  + ME\_CONFIG\_MONGODB\_SERVER=mongodb → Connects to the **MongoDB** container.
* ports:  
  + 8081:8081 → Exposes **Mongo Express** on http://localhost:8081.
* restart: always → Ensures it automatically restarts if it crashes.

### **5️⃣ MailDev (Email Testing Service)**

mail-dev:

container\_name: ms-mail-dev

image: maildev/maildev

ports:

- 1080:1080

- 1025:1025

✅ **Purpose:**

* MailDev is a tool to **test and debug emails** sent by the microservices.

✅ **Key Configurations:**

* image: maildev/maildev → Uses the MailDev image.
* container\_name: ms-mail-dev → Sets the container name.
* ports:  
  + 1080:1080 → Opens MailDev UI at **http://localhost:1080**.
  + 1025:1025 → The SMTP server for catching test emails.

## **Networks**

networks:

microservices-net:

driver: bridge

✅ **Purpose:**

* Defines a **Docker network** called microservices-net.
* driver: bridge → Uses the **bridge** network mode (default for containers).
* **Why is it needed?**
  + Allows containers (PostgreSQL, MongoDB, pgAdmin, Mongo Express) to communicate **securely** without exposing services to the public.

## **Volumes**

volumes:

postgres:

pgadmin:

mongo:

✅ **Purpose:**

* Volumes **store data persistently** so that databases do not lose data when containers restart.
* postgres: → Stores PostgreSQL data.
* pgadmin: → Stores pgAdmin settings and history.
* mongo: → Stores MongoDB data.

## **Summary**

| **Service** | **Purpose** |
| --- | --- |
| postgresql | Relational database (PostgreSQL) |
| pgadmin | Web UI for managing PostgreSQL |
| mongodb | NoSQL database (MongoDB) |
| mongo-express | Web UI for managing MongoDB |
| mail-dev | Local SMTP server for email testing |

✅ **Why is this setup required?**

* **PostgreSQL & MongoDB** → Supports both relational and NoSQL databases.
* **pgAdmin & Mongo Express** → Easy GUI access to databases.
* **MailDev** → Helps debug emails in a microservices app.
* **Networks & Volumes** → Ensures seamless communication and data persistence.

# Configuration Server (config-server) Setup

Your config-server is a Spring Cloud Config Server, which is used for centralized configuration management in microservices architecture. It enables multiple microservices to fetch their configurations from a single location instead of maintaining separate configuration files for each service.

### 

### **Breakdown of Your app.prop (application.properties) File**

spring.application.name=config-server

* **Defines the name of the application** as config-server.
* This is useful when multiple microservices interact, as services can refer to it by name.

server.port=8888

* This **sets the port** on which the config server runs. The default port for a Spring Boot application is 8080, but here it's set to 8888.

spring.profiles.active=native

* Specifies that the **active profile is native**, which means configuration files will be read from the local file system or classpath instead of an external Git repository.
* Other possible values:  
  + git → Fetch configs from a remote Git repository.
  + vault → Fetch configs securely from HashiCorp Vault.
  + jdbc → Fetch from a database.

spring.cloud.config.server.native.search-locations=classpath:/configurations

* This **tells the config server where to look for configuration files**.
* In this case, it will search in classpath:/configurations, which means it expects configuration files to be present inside the resources/configurations directory of the project.

### 

### 

### **Why is This Needed?**

1. **Centralized Configuration Management**: Instead of maintaining separate configuration files for each microservice, we store them in one place (Config Server).
2. **Dynamic Configuration Updates**: If a service configuration changes, we don’t need to restart the microservices. They can dynamically fetch the latest configuration.
3. **Environment-Specific Configurations**: The config server can provide different configurations based on environments (e.g., dev, prod).
4. **Security**: Configurations (such as database credentials) can be stored securely and accessed centrally.

# Discovery Service (discovery-service) Setup

Your discovery-service is a **Spring Cloud Netflix Eureka Server**, which is used for **service discovery** in a microservices architecture. It allows microservices to register themselves and discover other services dynamically.

## **Breakdown of Your app.prop (application.properties) File in discovery-service**

spring.application.name=discovery-service

* Sets the **application name** as discovery-service, which is useful when other services need to communicate with it.

spring.config.import=optional:configserver:http://localhost:8888

* **Imports configuration from the Spring Cloud Config Server running on http://localhost:8888**.
* The optional: part ensures that if the config server is unavailable, the service can still start with local configurations.

## **Breakdown of discovery-service.properties Inside the Config Server**

Your config-server has a file discovery-service.properties, which contains the actual configuration for the **Eureka server**.

server.port=8761

* **Sets the Eureka Server to run on port 8761**, which is the default port for Eureka.

eureka.instance.hostname=localhost

* Defines the **hostname** for Eureka as localhost.
* If deployed on multiple machines, this could be an IP address or domain name.

eureka.client.registerWithEureka=false

* **Prevents the Eureka Server from registering itself as a client**.
* A Eureka **server** does not need to register itself, only microservices should register.

eureka.client.fetchRegistry=false

* **Prevents the Eureka Server from fetching other registered services**.
* Since this is the central server, it does not need to look up other services.

eureka.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:${server.port}/eureka/

* **Defines the URL where microservices should register themselves**.

It expands to:  
http://localhost:8761/eureka/

* Other microservices will use this URL to **register** and **discover** each other.

## **Why is This Needed?**

1. **Service Discovery**: Allows microservices to register dynamically instead of hardcoding their URLs.
2. **Load Balancing**: Multiple instances of a service can be registered, and clients can discover the least busy instance.
3. **Fault Tolerance**: If one instance goes down, other instances are still available.
4. **Centralized Registry**: Keeps track of all active services and their locations.

## **How It Works?**

1. The **Eureka server (discovery-service) starts on port 8761**.

Other **microservices register themselves** using:  
  
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/

1. When a **microservice needs another service**, it **queries Eureka** to get the latest available instance.
2. This enables **dynamic discovery** of services instead of using hardcoded URLs.

# Customer Service (customer-service) Setup

Your customer-service is a **microservice** that interacts with MongoDB and fetches its configuration from the **Spring Cloud Config Server**.

## **Breakdown of app.prop (application.properties) in customer-service**

spring.application.name=customer-service

* Sets the **application name** as customer-service, which is useful when registering with Eureka and fetching configs.

spring.config.import=optional:configserver:http://localhost:8888

* **Fetches configuration from the Spring Cloud Config Server (http://localhost:8888)**.
* The optional: part ensures that the service can still start even if the config server is unavailable.

## **Breakdown of customer-service.properties Inside the Config Server**

Your **Config Server** has a file customer-service.properties, which contains database-related configurations.

server.port=8090

* Runs the **customer-service** on port 8090.

spring.data.mongodb.username=aditya

spring.data.mongodb.password=aditya

* Sets the **MongoDB username and password**.
* The credentials aditya/aditya are used for authentication.

spring.data.mongodb.host=localhost

spring.data.mongodb.port=27017

* **Specifies the MongoDB host and port**.
* The MongoDB container is running at localhost:27017.

spring.data.mongodb.database=customer

* Sets the **database name** as customer.
* This means all customer-related data will be stored in the customer database.

spring.data.mongodb.authentication-database=admin

* **Defines the authentication database** (admin), where MongoDB credentials are verified before allowing access.

## **Why is This Needed?**

1. **Configuration Management**: Centralizes MongoDB connection settings via the **Config Server**.
2. **Security**: Credentials are managed in a single place, avoiding exposure in multiple microservices.
3. **Microservice Isolation**: Each service (customer-service) gets its own dedicated database (customer).
4. **Scalability**: MongoDB allows flexible data storage, making it ideal for customer data management.

## **How It Works?**

1. **Customer Service Starts on Port 8090**.
2. It **fetches database configuration** from the Config Server (http://localhost:8888).
3. It **connects to MongoDB** running on localhost:27017 with database customer.
4. **Other services (like order-service) can communicate with customer-service** to retrieve customer data.

**Using Docker Exec (Inside the MongoDB Container)**

If MongoDB is running inside Docker, access it using:

docker exec -it ms\_mongo\_db mongosh -u aditya -p aditya --authenticationDatabase admin

Then, run:

use customer

show collections

db.customer.find().pretty()