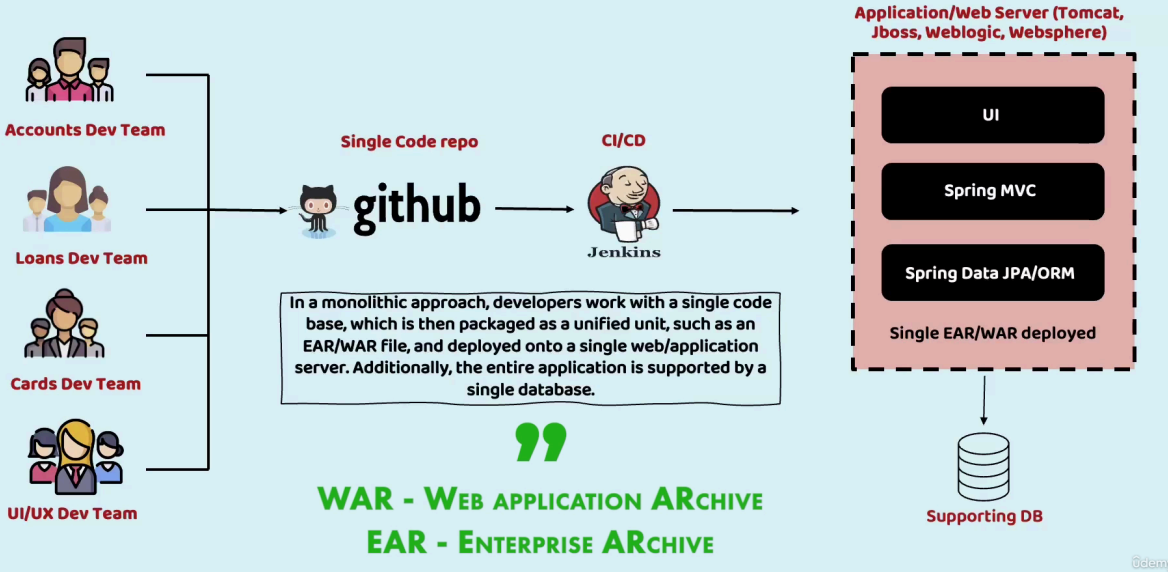
Develop Microservices with Java, Spring Boot, Spring Cloud, Docker, Kubernetes, Helm, Microservices Security

The Monolith

eazy bytes



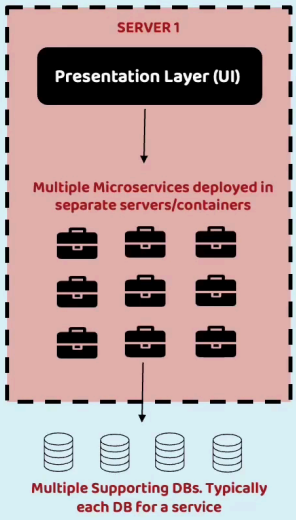
In a monolithic approach, developers work with a single code base, which is then packaged as a unified unit, such as an EAR/WAR file, and deployed onto a single web/application server. Additionally, the entire application is supported by a single database.

WAR - Web application ARchive

EAR - Enterprise ARchive

The GREAT MICROSERVICES

eazy bytes



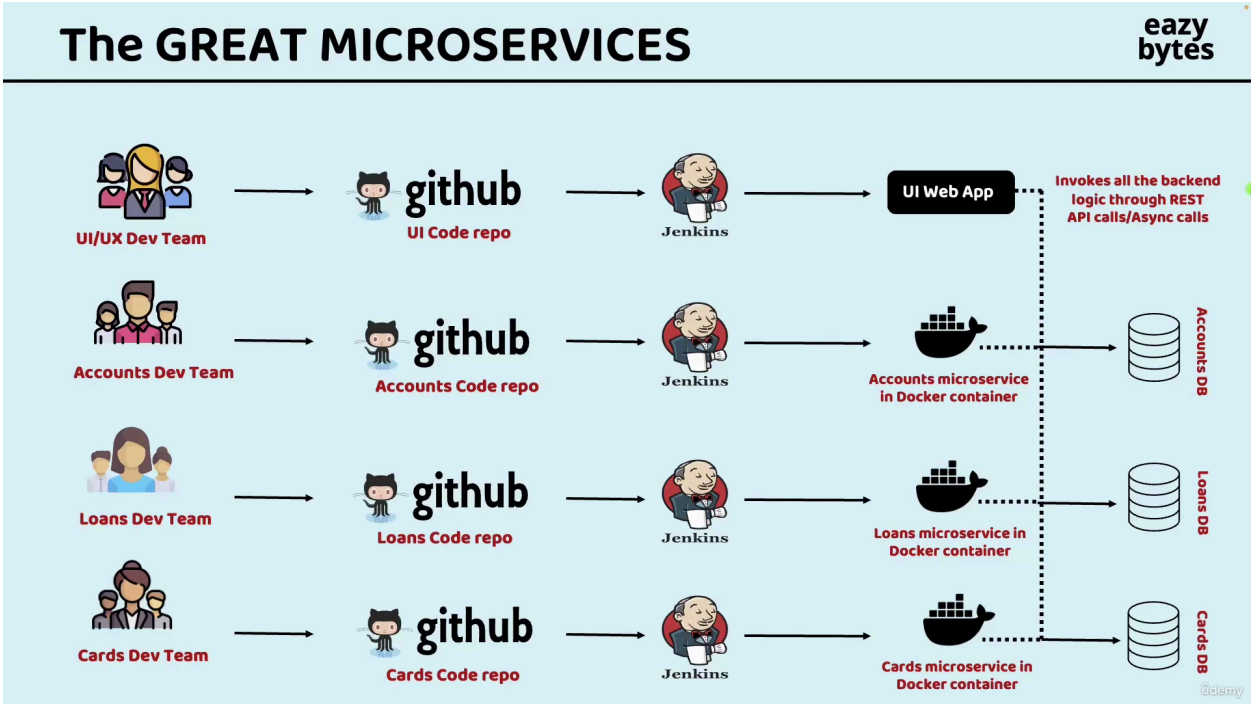
Microservices are independently releasable services that are modeled around a business domain. A service encapsulates functionality and makes it accessible to other services via networks—you construct a more complex system from these building blocks. One microservice might represent Accounts, another Cards, and yet another Loans, but together they might constitute an entire bank system.

Pros

Easy to develop, test, and deploy
Increased agility
Ability to scale horizontally
Parallel development
Modeled Around a Business Domain

Cons

Complexity
Infrastructure overhead
Security concerns



Official spring website : https://spring.io/

Monolithic Application

Monolithic Application is a **single-tier software application** in which **all the components and features** of the system (such as user interface, business logic, and database access) are **developed, built, and deployed together** as **one single unit**.

All the modules of the application are **tightly coupled** (closely connected), meaning they depend on each other.
Usually, the entire application runs on **a single server** and uses **a single shared database**.

When you make any change or update in one module, you have to **rebuild and redeploy the entire application**, even if the change is small.
This architecture is simple to develop for small projects but becomes hard to manage and scale as the application grows larger.

🛒 Example: Online Shopping Application

Imagine you are creating an **Online Shopping App** (like Amazon).

This app has several features:

- 👤 **User Login and Registration**
- 🛍️ **Product Catalog**
- 🛒 **Shopping Cart**
- 💳 **Payment Processing**
- 📦 **Order Tracking**

In a **monolithic architecture**:

- All these features are **developed together** inside **one project**.
- They **share the same codebase** and **use one common database**.
- When you deploy the app, you **deploy it as one single application** (e.g., one .war or .jar file).

So, if you want to update the payment module, you must **rebuild and redeploy the entire application**, even if no other part has changed.

Microservices Application

Microservices Application is a type of software architecture where the **whole application is divided into many small, independent services (modules)**.

Each service is responsible for **a specific function or feature** (like login, payment, order, etc.), and
each one can be **developed, deployed, and scaled separately**.

These services **communicate with each other** using **APIs (usually REST APIs)** or messaging systems.
Each service can also use **its own database** and **its own programming language or technology** if needed.

Because of this independence, if one service fails or needs an update, it **doesn't affect the other services** — making the system **more flexible, scalable, and reliable**.

🛒 Example: Online Shopping Application

Let's take the same **Online Shopping App** example.

In a **Microservices Architecture**, the app is broken down into separate services like:

Service Name	Responsibility	Own Database
👤 User Service	Handles user registration & login	Yes
🛍️ Product Service	Manages product details	Yes
🛒 Cart Service	Manages shopping cart	Yes
💳 Payment Service	Handles payments	Yes
📦 Order Service	Manages orders & tracking	Yes

Each of these services is:

- Developed **independently** by separate teams.
- **Deployed separately** on different servers or containers.
- Can be **updated or scaled** without affecting others.

For example, if many users are shopping at once, you can scale only the **Cart Service** or **Order Service**, instead of scaling the entire application.

—---

Sample Dependencies

Dependency	Description	Environment
Spring Web (WEB)	Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.	All (Dev, Prod)

H2 Database (SQL)	Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2 MB) footprint. Supports embedded and server modes as well as a browser-based console application.	Dev / Test
Spring Data JPA (SQL)	Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.	All (Dev, Prod)
Spring Boot Actuator (OPS)	Supports built-in (or custom) endpoints that let you monitor and manage your application — such as application health, metrics, sessions, etc.	All (Dev, Prod)
Spring Boot DevTools (DEVELOPER TOOLS)	Provides fast application restarts, LiveReload, and configurations for enhanced development experience.	Dev only
Validation (I/O)	Bean Validation with Hibernate Validator.	All (Dev, Prod)
Lombok (DEVELOPER TOOLS)	Java annotation library which helps to reduce boilerplate code.	All

—----------

Spring Boot DevTools

Spring Boot DevTools is primarily designed for **local development only**, **not for production or staging environments**.

Here's a breakdown to make it clear 👇

⚙️ What Spring Boot DevTools Does

Spring DevTools helps **speed up local development** by:

- Automatically **restarting** the application when code changes are detected.
- Enabling **LiveReload**, so your browser refreshes automatically.
- Disabling caching for templates, static files, etc. (so you always see fresh changes).

- Improving developer experience for **rapid iteration**.

🚫 Why Not Use It in Production

- It **adds overhead** (monitors file changes, triggers restarts).
- It **disables caching** (hurts performance).
- It may **expose internal info** if accidentally left enabled.
- Spring Boot **automatically disables DevTools in a packaged (JAR/WAR) production build** — it only activates when running from your IDE or spring-boot:run.

💡 Typical Usage

Environment	Use DevTools?	Notes
Local Development	✅ Yes	Fast reloads and testing changes quickly
Staging / QA	❌ No	Use normal builds for stability
Production	❌ No	Not recommended; automatically disabled

Spring Boot – schema.sql & data.sql Summary

- **Placement:**
  Put the files in src/main/resources/
  - schema.sql → for DDL (e.g., CREATE TABLE, ALTER TABLE)
  - data.sql → for DML (e.g., INSERT, UPDATE)
- **Execution Order:**
  - schema.sql → creates the schema
  - data.sql → inserts initial data

- **Important Rule:**

If you use **schema.sql / data.sql**, disable Hibernate's auto schema generation:

spring.jpa.hibernate.ddl-auto=none

  - ✅ Use **only one** approach to manage your schema.

If you prefer Hibernate to handle schema creation, skip schema.sql and set for example:

spring.jpa.hibernate.ddl-auto=create

  - Hibernate will then generate tables automatically from your JPA entities.

- **Common ddl-auto Options:**

Option	Description
none	Manual schema via SQL files
validate	Only checks schema consistency
update	Updates schema to match entities
create	Recreates schema on startup
create-drop	Creates on startup, drops on shutdown

1. Development Environment

- **Goal:** Fast iteration, flexibility.
- **Approaches:**
  - **Hibernate ddl-auto:**
    - create-drop: Recreates schema on every app start/stop – ideal for testing new features.

        - update: Incrementally updates schema; convenient for local single-developer setups but can cause schema drift.

    - **schema.sql + data.sql:** Less common, ensures all developers work from the same baseline schema and test data.

    - **Migration tools (Flyway/Liquibase):** Used to catch migration issues early, even in dev.

2. Testing Environment

- **Goal:** Consistency, repeatability.

- **Approaches:**

    - **Hibernate ddl-auto (create-drop) with in-memory DBs:** Ensures a fresh database for each test suite; prevents test contamination.

    - **schema.sql + data.sql:** Loads a known schema and dataset before tests for predictable, reproducible results.

    - Spring Boot has built-in support for these in integration/unit tests.

3. Production Environment

- **Goal:** Stability, data integrity, controlled changes.

- **Recommended Approach:**

    - **Migration Tools (Flyway/Liquibase):**

        - Versioned scripts (V1__create_tables.sql, etc.)

        - Track applied migrations and safely apply only new changes.

        - Ensures explicit, reviewable, and reversible schema changes.

- **Not Recommended:**

- **Hibernate ddl-auto**: Too risky – can cause accidental data loss or schema inconsistencies.

- **schema.sql**: All-or-nothing approach; doesn't support incremental updates or version tracking.

✅ **Key Principle:**

- **Dev/Test:** Convenience and speed (ddl-auto, in-memory DBs, SQL scripts).

- **Production:** Safety and predictability (migration tools, versioned scripts).

Springdoc OpenAPI

springdoc-openapi java library helps to automate the generation of API documentation using spring boot projects.
Automatically generates documentation in JSON/YAML and HTML format APIs. This documentation can be completed by comments using swagger-api annotations.
This library supports:

- OpenAPI 3
- Spring-boot v3 (Java 17 & Jakarta EE 9)
- JSR-303, specifically for @NotNull, @Min, @Max, and @Size.
- Swagger-ui
- Scalar
- OAuth 2
- GraalVM native images

Getting Started

For the integration between spring-boot and swagger-ui, add the library to the list of your project dependencies (No additional configuration is needed)

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.8.14</version>
</dependency>
```

This will automatically deploy swagger-ui to a spring-boot application:

- Documentation will be available in HTML format, using the official swagger-ui jars
- The Swagger UI page will then be available at http://server:port/context-path/swagger-ui.html and the OpenAPI description will be available at the following url for json format: http://server:port/context-path/v3/api-docs

    - server: The server name or IP
    - port: The server port
    - context-path: The context path of the application
- Documentation will be available in yaml format as well, on the following path : /v3/api-docs.yaml

Run and Access the Docs

Once you start your Spring Boot app, open:

👉 **Swagger UI:**

http://localhost:8080/swagger-ui.html

👉 **OpenAPI JSON/YAML:**

http://localhost:8080/v3/api-docs
http://localhost:8080/v3/api-docs.yaml

Annotations

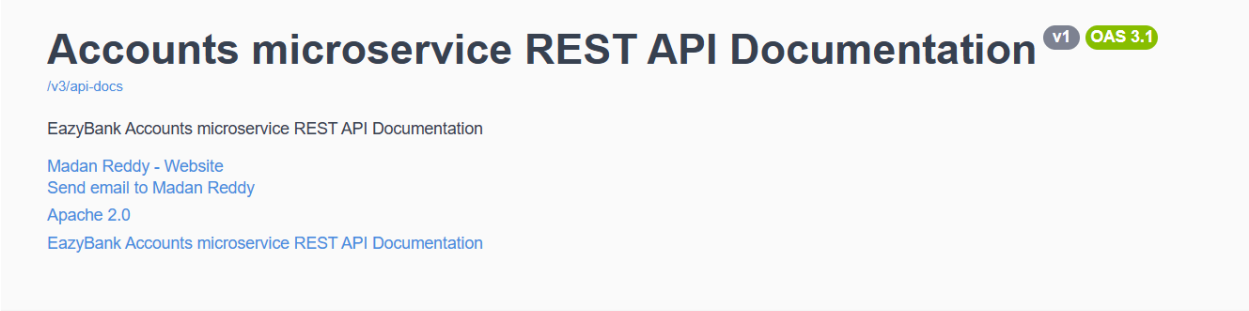
@OpenAPIDefinition

Where: In your AccountsApplication.java (main class)

Purpose:
Provides **global information** about your API — such as title, description, version, contact, license, and external documentation.

```java
@SpringBootApplication
/*@ComponentScans({ @ComponentScan("com.eazybytes.accounts.controller") })
@EnableJpaRepositories("com.eazybytes.accounts.repository")
@EntityScan("com.eazybytes.accounts.model")*/
@EnableJpaAuditing(auditorAwareRef = "auditAwareImpl")
@OpenAPIDefinition(
        info = @Info(
                title = "Accounts microservice REST API Documentation",
                description = "EazyBank Accounts microservice REST API Documentation",
                version = "v1",
                contact = @Contact(
                        name = "Madan Reddy",
                        email = "tutor@eazybytes.com",
                        url = "https://www.eazybytes.com"
                ),
                license = @License(
                        name = "Apache 2.0",
                        url = "https://www.eazybytes.com"
                )
        ),
        externalDocs = @ExternalDocumentation(
                description =  "EazyBank Accounts microservice REST API Documentation",
                url = "https://www.eazybytes.com/swagger-ui.html"
        )
)
public class AccountsApplication {

    public static void main(String[] args) {
        SpringApplication.run(AccountsApplication.class, args);
    }

}
```

💡 Helps with:
Displays API title, author, contact info, license, etc. at the top of your Swagger UI page.



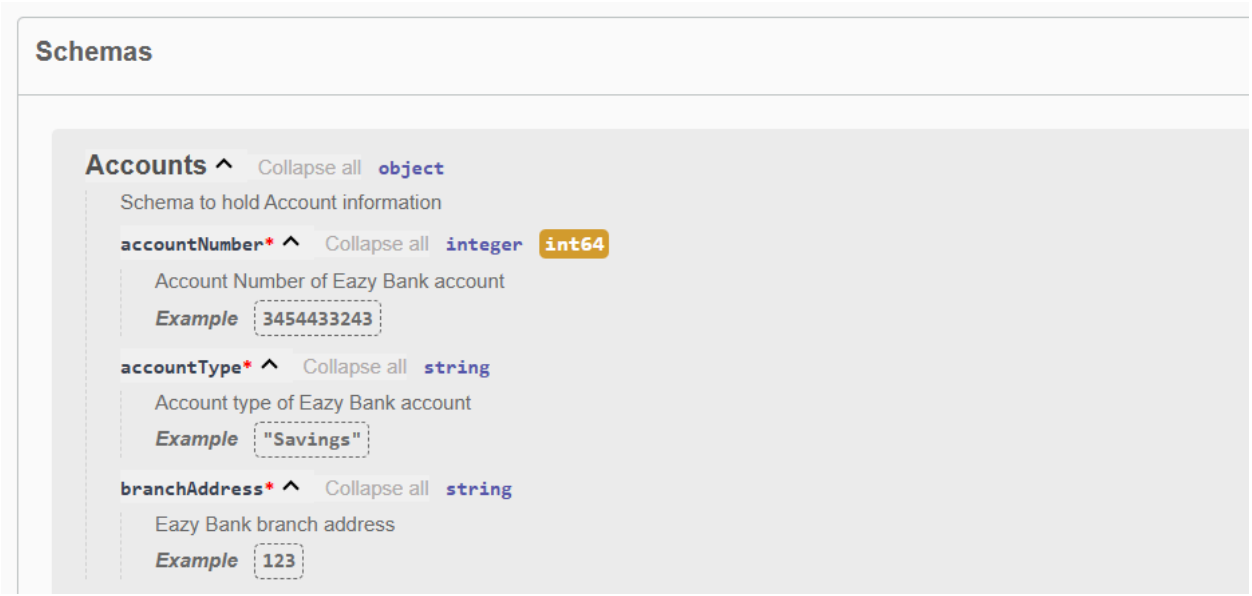
@Schema

Where: On your AccountsDto.java

Purpose:
Describes the structure (model) of your DTO or entity class.
You can annotate both the **class** and **individual fields**.

package com.eazybytes.accounts.dto;

```
import io.swagger.v3.oas.annotations.media.Schema;
import jakarta.validation.constraints.NotEmpty;
import jakarta.validation.constraints.Pattern;
import lombok.Data;

@Data
@Schema(
        name = "Accounts",
        description = "Schema to hold Account information"
)
public class AccountsDto {

    @NotEmpty(message = "AccountNumber can not be a null or empty")
    @Pattern(regexp="(^$|[0-9]{10})",message = "AccountNumber must be 10 digits")
    @Schema(
            description = "Account Number of Eazy Bank account", example = "3454433243"
    )
    private Long accountNumber;

    @NotEmpty(message = "AccountType can not be a null or empty")
    @Schema(
            description = "Account type of Eazy Bank account", example = "Savings"
    )
    private String accountType;

    @NotEmpty(message = "BranchAddress can not be a null or empty")
    @Schema(
            description = "Eazy Bank branch address", example = "123 NewYork"
    )
    private String branchAddress;
}
```



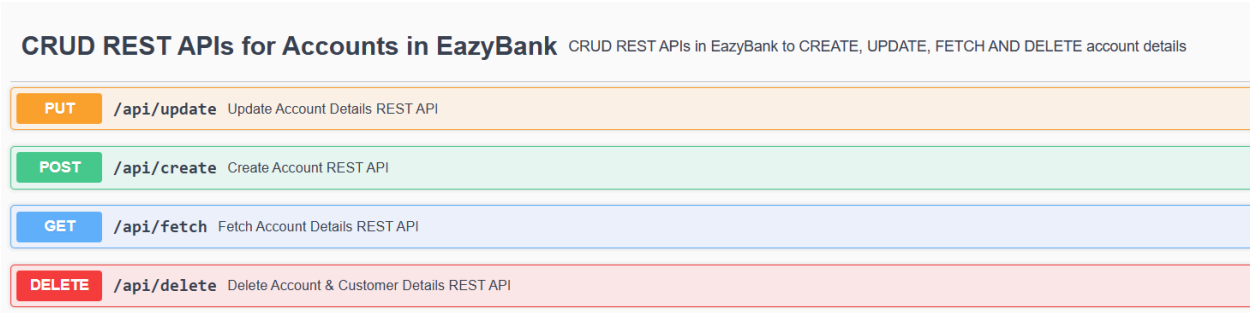
💡 **Helps with:**
In Swagger UI → displays field descriptions and sample values when showing request/response schemas.

@Tag

Where: On top of your controller.

Purpose:
Groups related endpoints into sections for better readability.

```
@Tag(
        name = "CRUD REST APIs for Accounts in EazyBank",
        description = "CRUD REST APIs in EazyBank to CREATE, UPDATE, FETCH AND DELETE account details"
)
@RestController
@RequestMapping(path="/api", produces = {MediaType.APPLICATION_JSON_VALUE})
@AllArgsConstructor
@Validated
public class AccountsController {
```



💡 **Helps with:**
Creates a labeled section in Swagger UI like:

CRUD REST APIs for Accounts in EazyBank

and lists all related endpoints underneath.

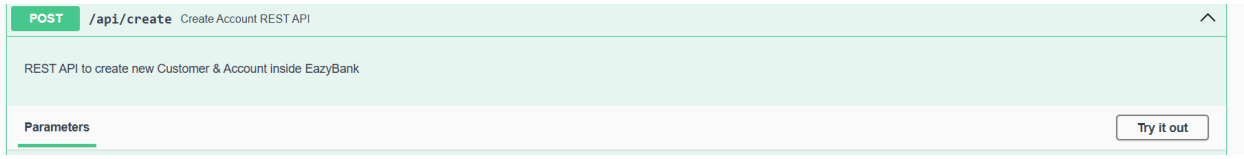
@Operation

Where: Above each controller method.

Purpose:
Describes what a specific endpoint does — its summary and details.

```
@Operation(
        summary = "Create Account REST API",
        description = "REST API to create new Customer &  Account inside EazyBank"
    )
    @ApiResponses({
        @ApiResponse(
            responseCode = "201",
            description = "HTTP Status CREATED"
        ),
        @ApiResponse(
            responseCode = "500",
            description = "HTTP Status Internal Server Error",
            content = @Content(
                schema = @Schema(implementation = ErrorResponseDto.class)
            )
        )
    }
    )
    @PostMapping("/create")
```

```
    public ResponseEntity<ResponseDto> createAccount(@Valid @RequestBody CustomerDto
customerDto) {
        iAccountsService.createAccount(customerDto);
        return ResponseEntity
                .status(HttpStatus.CREATED)
                .body(new ResponseDto(AccountsConstants.STATUS_201,
AccountsConstants.MESSAGE_201));
    }
```



💡 **Helps with:**
In Swagger UI, this text appears next to the API endpoint so users immediately understand its purpose.

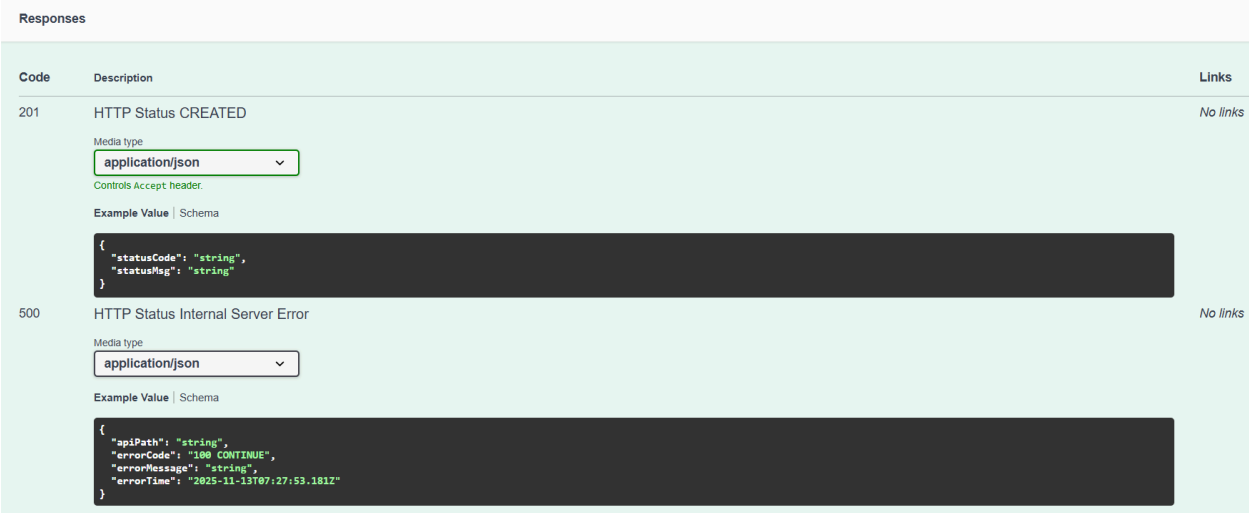
@ApiResponse and @ApiResponses

Where: Above each controller method.

Purpose:
Describes possible HTTP responses from the endpoint (e.g., 200, 201, 500).

```
@Operation(
        summary = "Create Account REST API",
        description = "REST API to create new Customer &  Account inside EazyBank"
    )
    @ApiResponses({
        @ApiResponse(
                responseCode = "201",
                description = "HTTP Status CREATED"
        ),
        @ApiResponse(
                responseCode = "500",
                description = "HTTP Status Internal Server Error",
                content = @Content(
                        schema = @Schema(implementation = ErrorResponseDto.class)
                )
        )
    }
```

```
    )
    @PostMapping("/create")
    public ResponseEntity<ResponseDto> createAccount(@Valid @RequestBody CustomerDto
customerDto) {
        iAccountsService.createAccount(customerDto);
        return ResponseEntity
                .status(HttpStatus.CREATED)
                .body(new ResponseDto(AccountsConstants.STATUS_201,
AccountsConstants.MESSAGE_201));
    }
```



💡 **Helps with:**
In Swagger UI → users can see:

- All possible HTTP status codes.
- Descriptions for success or error responses.
- Error model structure (like ErrorResponseDto).

@Parameter

Where: On method parameters (like @RequestParam or @PathVariable).

Purpose:
Describes query or path parameters, including validation rules and examples.

In your code:

```
@RequestParam
@Pattern(regexp="(^$|[0-9]{10})", message = "Mobile number must be 10 digits")
String mobileNumber
```

You could optionally add:

```
@Parameter(description = "Customer's 10-digit mobile number", example = "9876543210")
```

💡 **Helps with:**
Shows parameter descriptions, data type, and example input.

more refer:

Spring Boot validation annotations

Add dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Step 2 — Use @Valid in Controller

```
@PostMapping("/create")
public ResponseEntity<ResponseDto> createAccount(@Valid @RequestBody CustomerDto customerDto) {
    ...
}
```

✅ This tells Spring to validate the CustomerDto before using it.

And also

```java
@Validated
public class AccountsController {

    private IAccountsService iAccountsService;
```

```java
)
@PostMapping("/create")
public ResponseEntity<ResponseDto> createAccount(@Valid @RequestBody CustomerDto customerDto) {
    iAccountsService.createAccount(customerDto);
    return ResponseEntity
            .status(HttpStatus.CREATED)
            .body(new ResponseDto(AccountsConstants.STATUS_201, AccountsConstants.MESSAGE_201));
}
```

Step 3 — Annotate DTO Fields

Inside DTOs, use annotations like @NotNull, @Size, @Pattern, etc.

A. Null and Empty Checks

Annotation	Purpose	Example
`@NotNull`	Field cannot be null	`@NotNull(message = "ID cannot be null")`
`@NotEmpty`	Field cannot be null or empty (`""`)	`@NotEmpty(message = "Name cannot be empty")`
`@NotBlank`	Field cannot be null, empty, or whitespace	`@NotBlank(message = "Username cannot be blank")`
`@Null`	Must be null (useful for auto-generated fields)	`@Null(message = "ID must be null during creation")`

🔢 B. Numeric Validations

Annotation	Purpose	Example
`@Min(value)`	Minimum value	`@Min(value = 18, message = "Age must be >= 18")`
`@Max(value)`	Maximum value	`@Max(value = 60, message = "Age must be <= 60")`
`@Positive`	Must be > 0	`@Positive(message = "Amount must be positive")`
`@PositiveOrZero`	Must be ≥ 0	`@PositiveOrZero(message = "Balance must be zero or positive")`
`@Negative`	Must be < 0	`@Negative(message = "Credit should be negative")`
`@Digits`	Restricts integer/fraction digits	`@Digits(integer = 5, fraction = 2)`

🧾 C. String & Pattern Validations

Annotation	Purpose	Example
`@Size(min, max)`	Validates string/collection length	`@Size(min = 3, max = 20, message = "Username must be 3-20 chars")`
`@Pattern(regexp)`	Regex pattern validation	`@Pattern(regexp="(^$
`@Email`	Validates email format	`@Email(message = "Invalid email address")`
`@URL`	Validates URL	`@URL(message = "Invalid website URL")`

📅 D. Date Validations

Annotation	Purpose	Example
`@Past`	Must be in the past	`@Past(message = "DOB must be in the past")`
`@PastOrPresent`	Must be past or today	`@PastOrPresent(message = "Date cannot be future")`
`@Future`	Must be in the future	`@Future(message = "Expiry date must be in the future")`
`@FutureOrPresent`	Must be today or future	`@FutureOrPresent(message = "Booking date must be today or later")`

Example DTO with Validations

```
@Data
@Schema(description = "Customer details schema")
public class CustomerDto {


    @NotNull(message = "Customer ID cannot be null")
    private Long customerId;
    @NotBlank(message = "Customer name is mandatory")
    @Size(min = 3, max = 50, message = "Name must be between 3–50 characters")
    private String name;
    @Email(message = "Email should be valid")
    private String email;
    @Pattern(regexp="(^$|[0-9]{10})", message = "Mobile number must be 10 digits")
    private String mobileNumber;
    @Valid
    private AccountsDto accounts;  // Nested validation
}
```

Exception Handling

Global Exception Handling allows you to handle all exceptions in a **centralized place** rather than writing try-catch in every controller.
This improves **code reusability**, **readability**, and **maintainability**.

extends ResponseEntityExceptionHandler

By extending this Spring class, you can **override built-in exception handlers**, such as:

- MethodArgumentNotValidException → for validation errors from @Valid or @Validated annotations

```java
@ControllerAdvice
public class GlobalExceptionHandler  extends ResponseEntityExceptionHandler {

    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(
            MethodArgumentNotValidException ex, HttpHeaders headers, HttpStatusCode status, WebRequest request) {
        Map<String, String> validationErrors = new HashMap<>();
        List<ObjectError> validationErrorList = ex.getBindingResult().getAllErrors();

        validationErrorList.forEach((error) -> {
            String fieldName = ((FieldError) error).getField();
            String validationMsg = error.getDefaultMessage();
            validationErrors.put(fieldName, validationMsg);
        });
        return new ResponseEntity<>(validationErrors, HttpStatus.BAD_REQUEST);
    }
```

Generic Exception — handleGlobalException

```java
    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponseDto> handleGlobalException(Exception exception,
                                                                   WebRequest webRequest) {
        ErrorResponseDto errorResponseDTO = new ErrorResponseDto(
                webRequest.getDescription(false),
                HttpStatus.INTERNAL_SERVER_ERROR,
                exception.getMessage(),
                LocalDateTime.now()
        );
        return new ResponseEntity<>(errorResponseDTO, HttpStatus.INTERNAL_SERVER_ERROR);
    }
```

Custom Exceptions — ResourceNotFoundException

```java
    @ExceptionHandler(CustomerAlreadyExistsException.class)
    public ResponseEntity<ErrorResponseDto> handleCustomerAlreadyExistsException(CustomerAlreadyExistsException exception,
                                                                                  WebRequest webRequest){
        ErrorResponseDto errorResponseDTO = new ErrorResponseDto(
                webRequest.getDescription(false),
                HttpStatus.BAD_REQUEST,
                exception.getMessage(),
                LocalDateTime.now()
        );
        return new ResponseEntity<>(errorResponseDTO, HttpStatus.BAD_REQUEST);
    }
```

```java
1  package com.eazybytes.accounts.exception;
2
3  import org.springframework.http.HttpStatus;
5
6  @ResponseStatus(value = HttpStatus.BAD_REQUEST)
7  public class CustomerAlreadyExistsException extends RuntimeException {
8
9      public CustomerAlreadyExistsException(String message) {
10         super(message);
11     }
12
13 }
14
```

Sample Project Structure

Recommended Spring Boot Project Structure

pgsql

Copy code

```
com.example.projectname/
|
├── constants/
|   └── AppConstants.java
|
├── controller/
|   └── UserController.java
|
├── dto/
|   └── UserDTO.java
|
├── entity/
|   └── User.java
|
├── exception/
|   ├── GlobalExceptionHandler.java
|   ├── ResourceNotFoundException.java
|   └── CustomException.java
|
├── mapper/
|   └── UserMapper.java
|
├── repo/
|   └── UserRepository.java
|
├── service/
|   ├── UserService.java
|   └── impl/
|       └── UserServiceImpl.java
|
└── ProjectNameApplication.java
```

@MappedSuperclass in Spring Boot / JPA

@MappedSuperclass is a **JPA annotation** used on a **base class** whose fields should be **inherited by entity classes**,
but the base class **itself is not an entity or table**.

🧩 Simple meaning:

It's like saying:

> “This class has some common fields for other entities,
> but don’t create a table for it in the database.”

🧱 **Example**

```
import jakarta.persistence.MappedSuperclass;

import jakarta.persistence.Column;

import java.time.LocalDateTime;

@MappedSuperclass

public abstract class BaseEntity {

    @Column(name = "created_at")

    private LocalDateTime createdAt;

    @Column(name = "updated_at")

    private LocalDateTime updatedAt;

    // getters and setters

}
```

Now, any entity that extends BaseEntity automatically inherits these columns.

🧠 **Usage Example**

```
import jakarta.persistence.Entity;

import jakarta.persistence.Id;

@Entity

public class User extends BaseEntity {

    @Id

    private Long id;
```

```
    private String username;
}
```

✅ The **user** table will have:

id

username

created_at

updated_at

🛑 But there will be **no table for BaseEntity**.

💡 **When to Use**

Use @MappedSuperclass when you have:

- Common columns like createdBy, createdAt, updatedAt, status, etc.
- You want to avoid repeating them in every entity.
- You **don't need a separate table** for the base class.

—----------

when moving from older Spring style (@Autowired) to modern **Spring Boot best practices**.

The old way — using @Autowired

```
@RestController
public class CustomerController {

    @Autowired
```

```
    private CustomerService customerService;

    // endpoints

}
```

✅ Works fine.
❌ But has **some drawbacks**:

- Makes the class **harder to test** (you can't easily pass a mock in constructor).
- Makes the dependency **hidden** — it's not obvious what this class needs.
- Causes problems if you use **final** fields (you can't make them final here).

The new & preferred way — Constructor Injection (with Lombok's @AllArgsConstructor)

```
@RestController
@RequiredArgsConstructor   // or @AllArgsConstructor
@RequestMapping("/customers")
public class CustomerController {
    private CustomerService customerService;
    // endpoints
}
```

✅ Advantages:

1. **No need for @Autowired**
   → Spring automatically injects dependencies into the constructor.
2. **Fields can be final**
   → makes them immutable.

3. **Easier to test**
   → you can manually pass mock objects in unit tests.

4. **Cleaner and safer**
   → promotes dependency immutability and makes it clear what's required.

What happens behind the scenes

When you use @AllArgsConstructor (from Lombok):

```
@AllArgsConstructor
public class CustomerController {
    private final CustomerService customerService;
}
```

➡️ Lombok generates this constructor for you:

```
public CustomerController(CustomerService customerService) {
    this.customerService = customerService;
}
```

➡️ Then Spring automatically injects the bean using **constructor injection** — no need for @Autowired.

—----------

Package for constants/utility

The class is a **constants holder** — it's used to store **static final values** (fixed values that never change) used throughout your Spring Boot application.

It helps you **avoid hardcoding strings and numbers** in multiple places in your code.

Ex

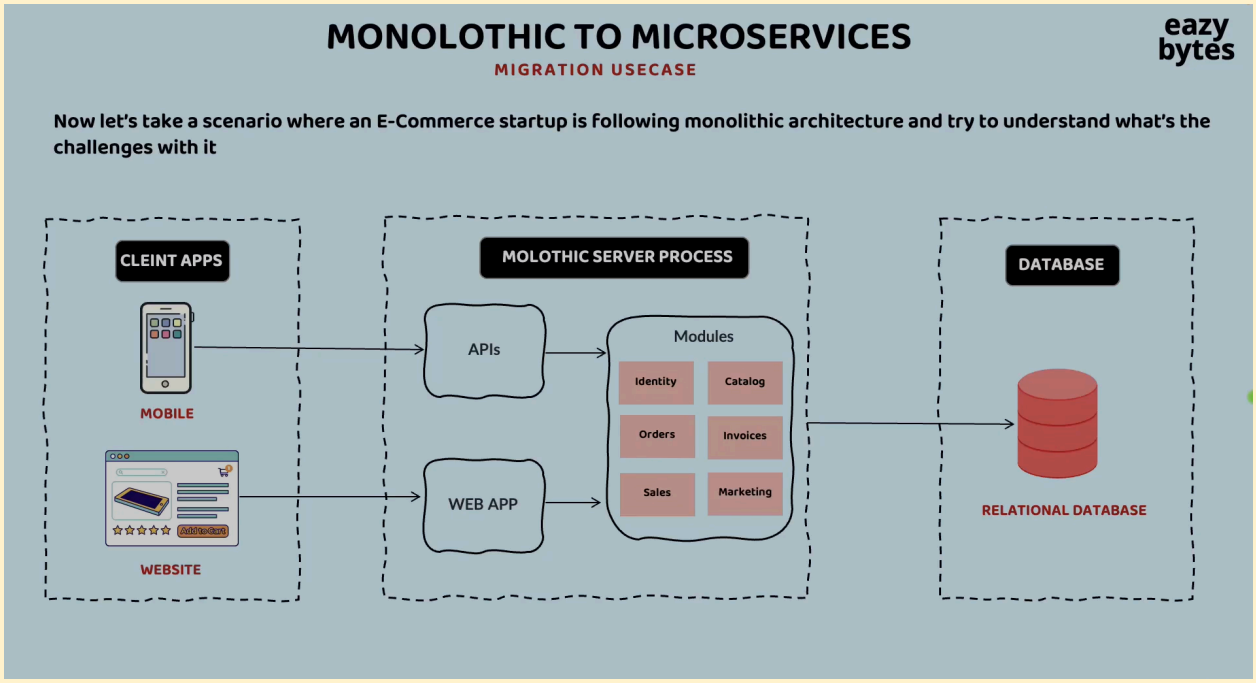
```java
package com.eazybytes.accounts.constants;

public final class AccountsConstants {

    private AccountsConstants() {
        // restrict instantiation
    }

    public static final String  SAVINGS = "Savings";
    public static final String  ADDRESS = "123 Main Street, New York";
    public static final String  STATUS_201 = "201";
    public static final String  MESSAGE_201 = "Account created successfully";
    public static final String  STATUS_200 = "200";
    public static final String  MESSAGE_200 = "Request processed successfully";
    public static final String  STATUS_417 = "417";
    public static final String  MESSAGE_417_UPDATE= "Update operation failed. Please try again or contact Dev team";
    public static final String  MESSAGE_417_DELETE= "Delete operation failed. Please try again or contact Dev team";
    // public static final String  STATUS_500 = "500";
    // public static final String  MESSAGE_500 = "An error occurred. Please try again or contact Dev team";

}
```

The **private constructor** prevents anyone from creating an object:

This is a **design pattern** known as a *"utility class" pattern*.

Monolith → Microservices (Migration)

Monolithic Architecture (Before Migration)



- All modules like:
    - Identity

    - Catalog
    - Orders
    - Invoices
    - Sales
    - Marketing
- run inside **one big server/application**.

- A **single relational database** is used.

- Clients (mobile/web) call a single API layer which connects to this big system.

Advantages (initial days)

✔ Easy to develop
✔ Easy to test
✔ Easy to deploy
✔ Works well with small user load
✔ Good for small teams

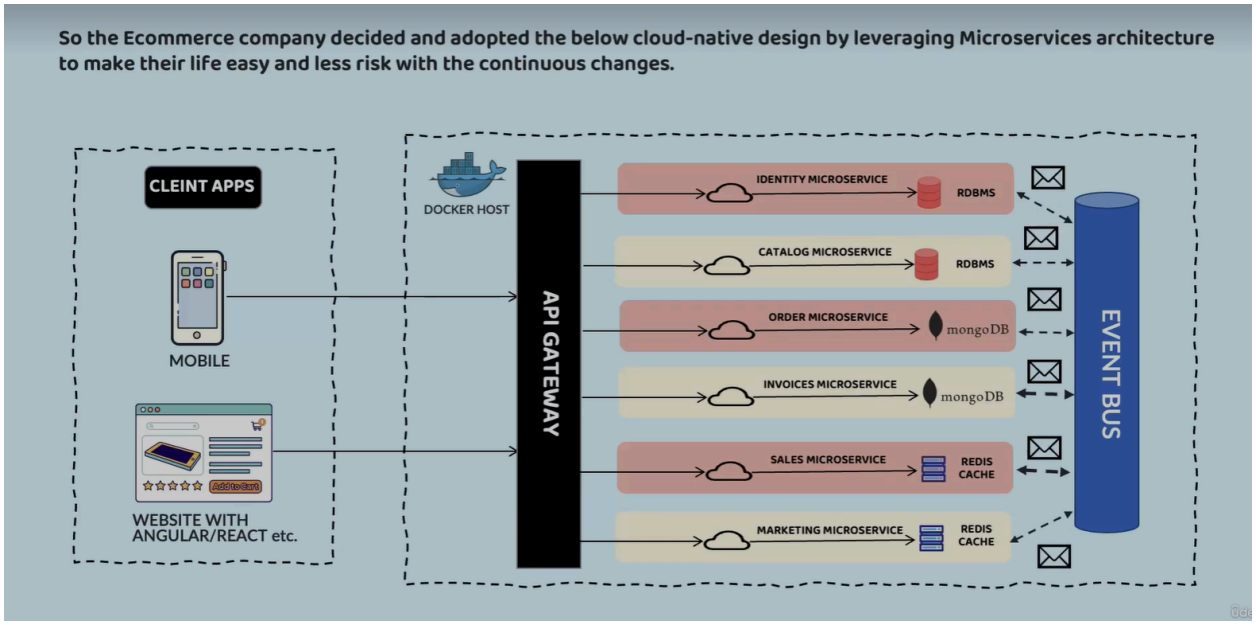
Problems after the app grows (from the second diagram)

Once the system becomes large:

1. **Too much complexity** – one person cannot understand everything.
2. **Fear of changes** – one small change causes side effects.
3. **New feature development becomes slow & costly**.
4. **Deployment becomes risky** – even a small fix needs the whole app to redeploy.
5. **Single point of failure** – if one module breaks, the whole app crashes.
6. **Can't use new tech** – because everything is tightly coupled.
7. **Hard to scale teams** – small teams can't work independently.

This is the typical pain that forces companies to move to microservices.

Migration to Microservices (After Migration)



Key elements of the new design

- **API Gateway**
  – single entry point for all external users
  – routes calls to correct microservice
- **Multiple Microservices**:
    - Identity Service → RDBMS
    - Catalog Service → RDBMS
    - Order Service → MongoDB
    - Invoice Service → MongoDB
    - Sales Service → Redis Cache
    - Marketing Service → Redis Cache

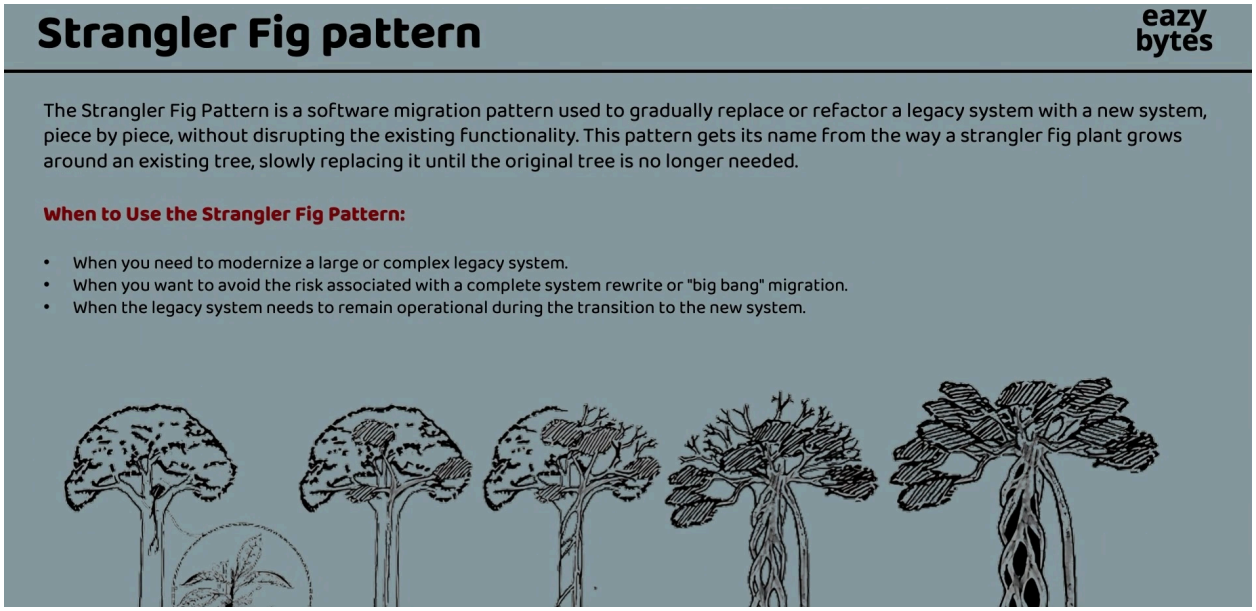
Each microservice:
✔ Has its own database
✔ Runs independently (usually Docker containers)
✔ Can be deployed independently

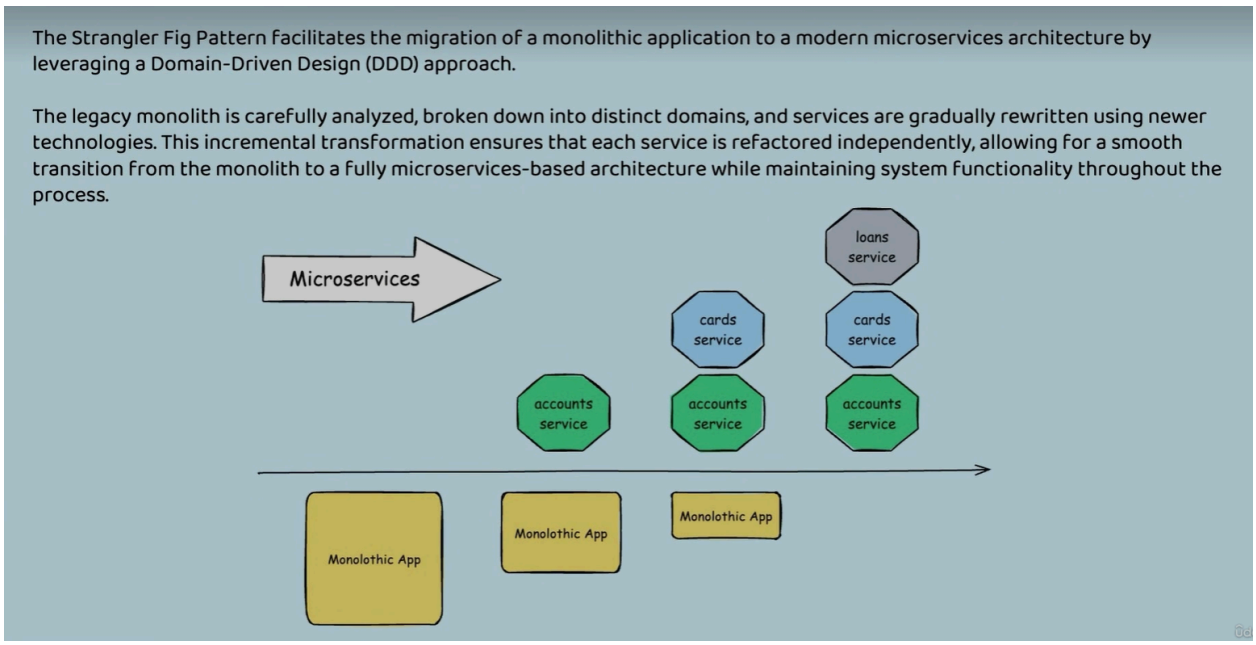
Event Bus

- Used for **async communication**

- Microservices publish/subscribe events
  Example:
    - Order Service publishes "Order Created"
    - Invoice Service listens and generates invoice

Strangler Fig Pattern



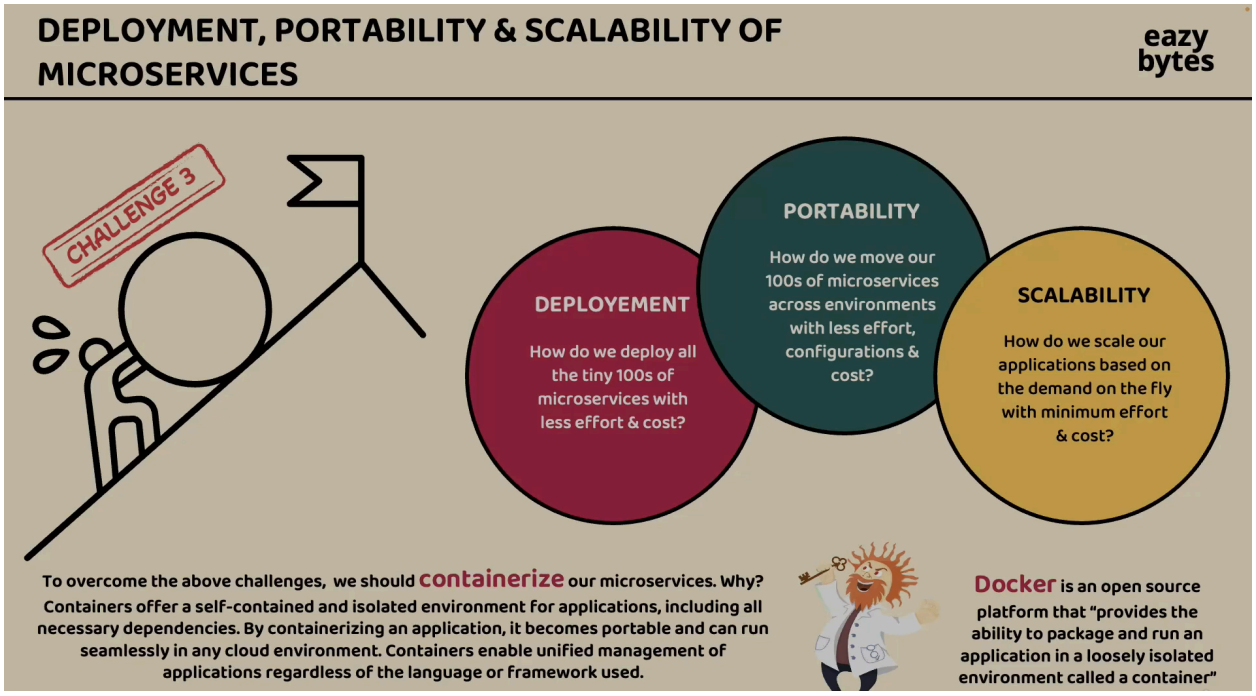


Strangler Fig Pattern migration:

✔ Identification
✔ Transformation
✔ Co-existence
✔ Elimination

Stage	Description
Identification	Select the module to extract
Transformation	Build new microservice version
Co-existence	Old & new systems run together
Elimination	Remove monolith part

Deployment ,portability and scaling of microservices



1. Deployment (Before vs After Containers)

🔴 **Before Containers**

- Each microservice had to be deployed on a **VM or physical server**.
- You needed to install:
    - Java/Node/Python runtime
    - Libraries
    - OS-level dependencies
- Every environment (dev/test/prod) behaved **differently**.
- Deployment was heavy, slow, and error-prone.
- “It works on my machine” issues were very common.

❌ **Example**

To deploy a Payments service:

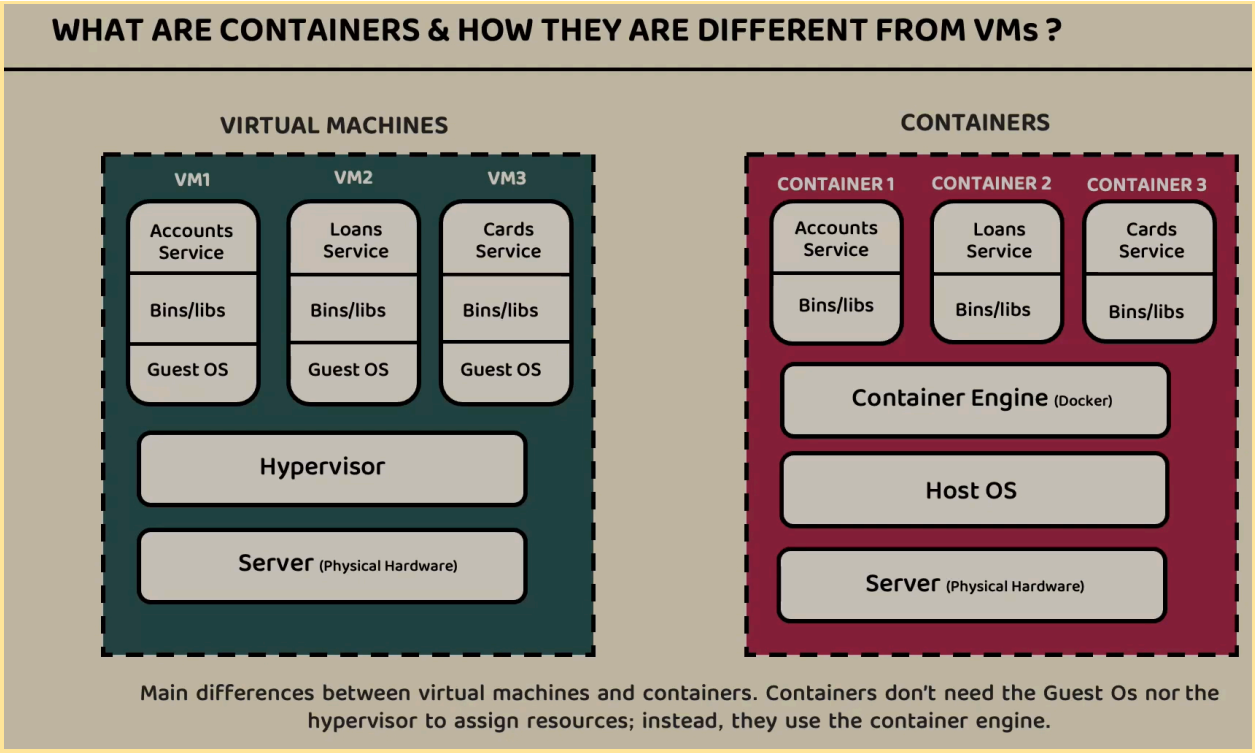
- Install JDK manually on the server
- Install Maven
- Install dependent libraries
- Configure environment variables
- Run JAR manually

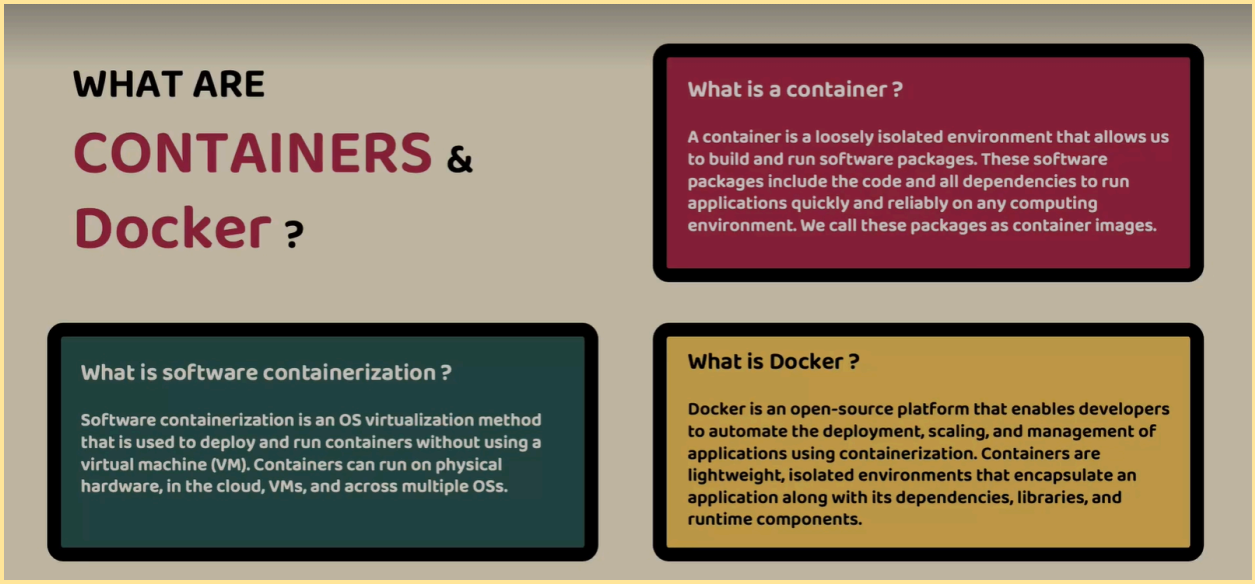
🟢 **After Containers**

- Every microservice is packaged into a **Docker image**.
- Image contains:
    - Application code

    - Runtime (JDK, Node, Python)
    - Libraries
    - OS dependencies

- Same image is used in **dev → test → prod**.
- Fast deployment through Kubernetes, ECS, Docker Swarm.





Container : A container is a lightweight, isolated runtime environment that **packages an application together with its code, runtime, libraries, configuration, and dependencies**, while sharing the host operating system's kernel.

Isolated means the container runs separately with its own files, processes, and network, without affecting or interfering with other containers.

Lightweight means it uses very few resources because it shares the host OS kernel, so it starts fast and needs less memory/CPU.

Key Characteristics

- Does **not include a full OS**, unlike virtual machines
- **Fast** to start and stop
- **Portable**: runs the same on any system that supports containers
- **Isolated**: each container runs independently

✔ In Simple Terms

A container is a **small, isolated environment** that runs an application with all its dependencies bundled inside.

A container is a **small box** that has everything your application needs to run, anywhere.

Containerization : Containerization is the process of packaging an application and all its dependencies into a container image, and running it in isolated environments called containers.

📌 Key Points

- Ensures consistent behavior across dev → test → prod
- Avoids "works on my machine" problems
- Enables microservices architecture
- Used for fast deployment and scaling

✔ Simple Meaning

Containerization is the **method** of putting your app inside a container so it can run everywhere the same way.

Docker : Docker is a containerization platform designed to help developers build, share, and run container applications. It allows us to build, test, and deploy applications quickly.

Docker Engine → runs containers

Dockerfile → defines container blueprint

Docker Hub → stores and shares container images

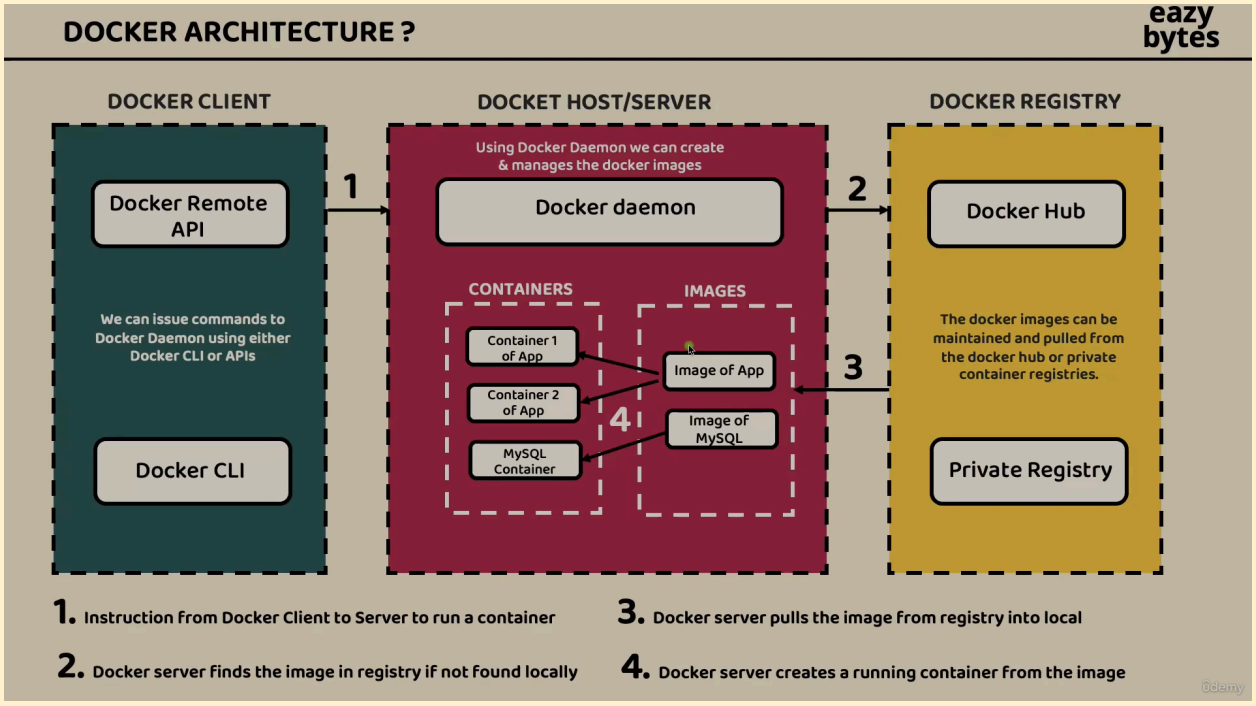
Works with Kubernetes for orchestration

Container: A lightweight, isolated environment that packages an **application** with required dependencies.

Containerization: The process of packaging and running applications inside containers.

Docker: A platform used to build, run, and manage containers.

Docker Architecture



1. Docker Client: This is the primary interface for users to interact with Docker. It provides a command-line interface (CLI)/API that sends commands to the Docker daemon.

When you run commands like:

docker build

docker pull

docker run

2. Docker Host:

This refers to the machine where the Docker daemon runs and where images and containers are stored and executed. It provides the environment for containerized applications.

Docker Daemon (dockerd): Running on the Docker host, the daemon is responsible for managing Docker objects such as images, containers, networks, and volumes. It listens for requests from the client and executes the necessary actions.

Docker Images: Images are read-only templates containing the application code, runtime, libraries, and dependencies needed to run a container. They are built from a Dockerfile and can be shared and versioned.

Docker Containers: A container is a runnable instance of a Docker image. It's an isolated environment where an application and its dependencies run, providing consistency across different environments. Containers can be created, started, stopped, moved, and deleted.

3.Docker Registry: A registry is a centralized repository for storing and distributing Docker images. Docker Hub is a public registry, while private registries can be set up for internal use. Images are pushed to and pulled from registries.

Docker Hub (default)

Amazon ECR

GitHub Container Registry

Azure Container Registry

Google Artifact Registry

Private registry

User → Docker Client → Docker Daemon → (Images/Containers) ↔ Docker Registry

Docker is open source platform

i) **Go to** https://www.docker.com/ and install a desktop for docker (Installs Docker CLI (client) + Docker Engine (server/host) on your local machine).

Docker Desktop is an **application/GUI** for Windows or Mac.

ii) go to https://hub.docker.com/ and create an account.

```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>docker version
Client:
 Version:           28.5.2
 API version:       1.51
 Go version:        go1.25.3
 Git commit:        ecc6942
 Built:             Wed Nov  5 14:45:58 2025
 OS/Arch:           windows/amd64
 Context:           desktop-linux

Server: Docker Desktop 4.51.0 (210443)
 Engine:
  Version:          28.5.2
  API version:      1.51 (minimum version 1.24)
  Go version:       go1.25.3
  Git commit:       89c5e8f
  Built:            Wed Nov  5 14:43:25 2025
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          v1.7.29
  GitCommit:        442cb34bda9a6a0fed82a2ca7cade05c5c749582
 runc:
  Version:          1.3.3
  GitCommit:        v1.3.3-0-gd842d771
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0
```

Docker CLI (Client)

- The command-line tool you use to interact with Docker.
- Example: commands like docker run, docker build, docker push.
- Installed on your local machine (part of Docker Desktop).

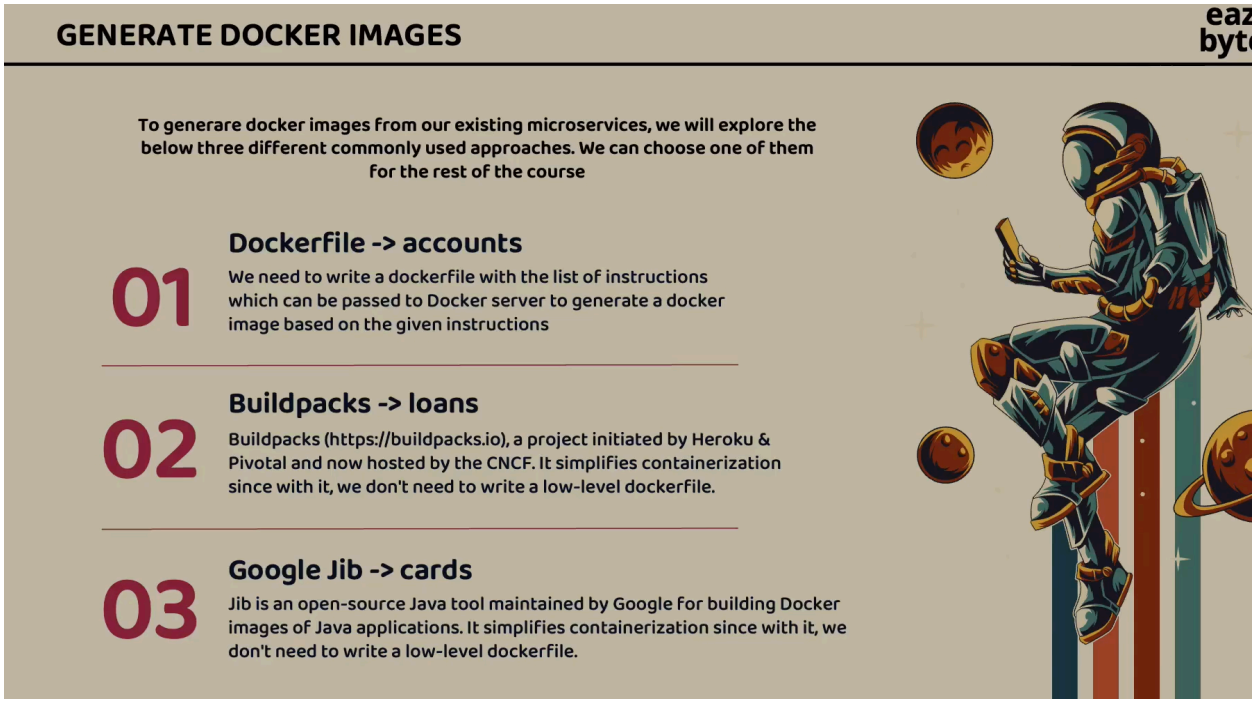
Docker Host (Server / Engine / Daemon)

- The environment where **containers actually run**.
- Manages images, containers, volumes, and networks.
- On Windows/Mac, Docker Desktop internally runs a **Linux VM** as the Docker Host.
- When we run the docker version, the **Server** info we see corresponds to this Docker Host.

Docker Registry

- Central repository for Docker images.
- Example: **Docker Hub** (hub.docker.com).

Generate Docker images



Spring Boot 2.x requires a minimum of Java 8

Spring Boot 3.x requires:

✔ Java 17 or higher

✔ Maven 3.6.3 or higher

Basic things before generating docker images

1. packaging **jar/war**
   Check pom.xml — Packaging Type

   Open your pom.xml and make sure the packaging is:

   <packaging>jar</packaging>

   ✔ If <packaging> is missing → Maven will **not** generate a JAR.
   ✔ If packaging is set to war, Maven will build a **WAR file**, not JAR.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 h
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.4.1</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.eazybytes</groupId>
    <artifactId>accounts</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>accounts</name>
    <packaging>jar</packaging>
    <description>Microservice for Accounts</description>
    <url/>
    <licenses>
        <license/>
```

2. Add Spring Boot Plugin (if missing)

Spring Boot requires this plugin to create a runnable JAR:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

Without this, the JAR may not start properly.

3. Open cmd terminal **inside the folder where pom.xml exists**:

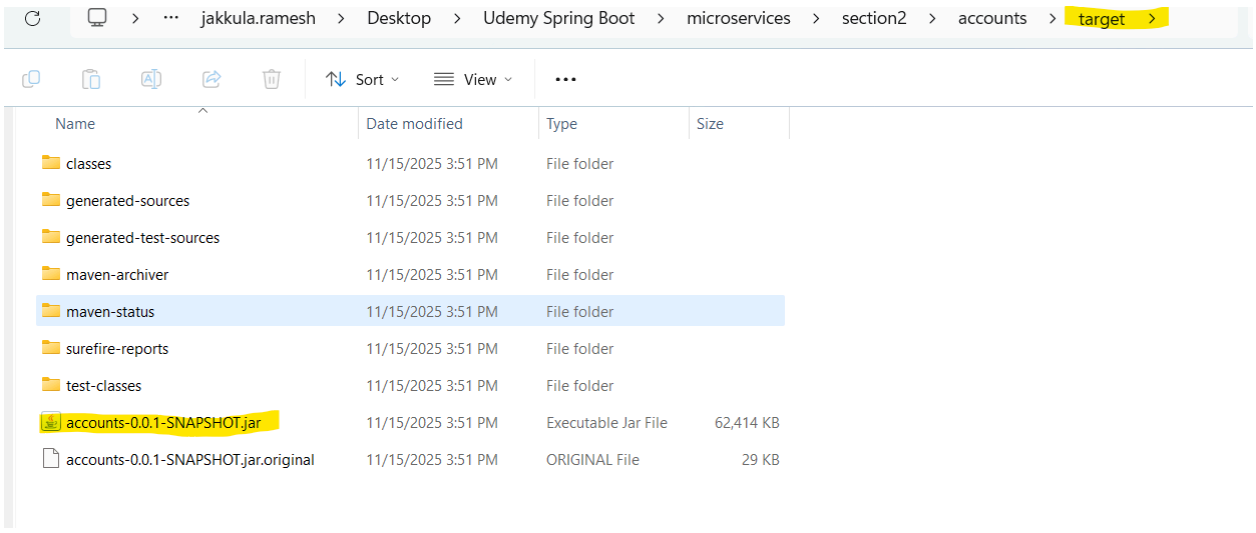
```
mvn clean install
```

```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>mvn clean install
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< com.eazybytes:accounts >----------------------
[INFO] Building accounts 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-clean-plugin:3.4.0:clean (default-clean) @ accounts ---
[INFO] Deleting C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts\target
[INFO]
[INFO] --- maven-resources-plugin:3.3.1:resources (default-resources) @ accounts ---
[INFO] Copying 1 resource from src\main\resources to target\classes
[INFO] Copying 1 resource from src\main\resources to target\classes
[INFO]
[INFO] --- maven-compiler-plugin:3.13.0:compile (default-compile) @ accounts ---
[INFO] Recompiling the module because of changed source code.
[INFO] Compiling 20 source files with javac [debug parameters release 17] to target\classes
[INFO]
[INFO] --- maven-resources-plugin:3.3.1:testResources (default-testResources) @ accounts ---
[INFO] skip non existing resourceDirectory C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts\src
[INFO]
[INFO] --- maven-compiler-plugin:3.13.0:testCompile (default-testCompile) @ accounts ---
[INFO] Recompiling the module because of changed dependency.
[INFO] Compiling 1 source file with javac [debug parameters release 17] to target\test-classes
[INFO]
[INFO] --- maven-surefire-plugin:3.5.2:test (default-test) @ accounts ---
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/surefire/surefire-api/3.5.2/surefire-api-3.5.2.j
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/surefire/maven-surefire-common/3.5.2/maven-surefi
```

```
[INFO] --- maven-jar-plugin:3.4.2:jar (default-jar) @ accounts ---
[INFO] Building jar: C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts\target\accounts-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:3.4.1:repackage (repackage) @ accounts ---
Downloading from central: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-buildpack-platform/3.4.1/spring-boot-buildpack-pl
```

The **JAR name** comes from Only **artifactId + version** are used in your pom.xml:

```
<groupId>com.eazybytes</groupId>
<artifactId>accounts</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>accounts</name>
<packaging>jar</packaging>
<description>Microservice for Accounts</description>
```



4. We can spring boot run in two ways using maven command or using java command

i) Maven command to run Spring Boot

mvn spring-boot:run

This directly starts your application without creating or running the JAR manually.

```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>mvn spring-boot:run
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------< com.eazybytes:accounts >------------------------
[INFO] Building accounts 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] >>> spring-boot-maven-plugin:3.4.1:run (default-cli) > test-compile @ accounts >>>
[INFO]
[INFO] --- maven-resources-plugin:3.3.1:resources (default-resources) @ accounts ---
[INFO] Copying 1 resource from src\main\resources to target\classes
[INFO] Copying 1 resource from src\main\resources to target\classes
```

```
2025-11-15T16:00:18.129+05:30  WARN 24252 --- [  restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Th
erefore, database queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2025-11-15T16:00:18.926+05:30  INFO 24252 --- [  restartedMain] o.s.b.a.h2.H2ConsoleAutoConfiguration    : H2 console available at '/h2-console'. Database a
vailable at 'jdbc:h2:mem:testdb'
2025-11-15T16:00:19.009+05:30  INFO 24252 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
2025-11-15T16:00:19.024+05:30  INFO 24252 --- [  restartedMain] o.s.b.a.e.web.EndpointLinksResolver      : Exposing 1 endpoint beneath base path '/actuator'
2025-11-15T16:00:19.160+05:30  INFO 24252 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context p
ath '/'
2025-11-15T16:00:19.179+05:30  INFO 24252 --- [  restartedMain] c.e.accounts.AccountsApplication         : Started AccountsApplication in 10.129 seconds (pr
ocess running for 10.791)
```

to stop the application in cmd use “ctrl + c”

ii) using the “.jar” file

run:

java -jar target/yourappname-version.jar

Example:

java -jar target/accounts-service-0.0.1-SNAPSHOT.jar

```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>java -jar target\accounts-0.0.1-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/

 :: Spring Boot ::                (v3.4.1)
```

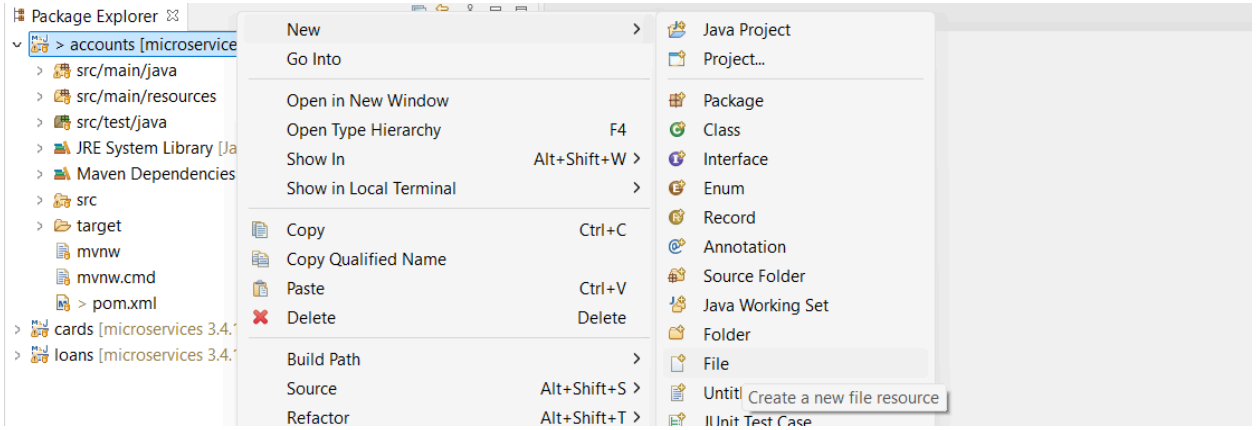
```
2025-11-15T16:00:18.129+05:30  WARN 24252 --- [  restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Th
erefore, database queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2025-11-15T16:00:18.926+05:30  INFO 24252 --- [  restartedMain] o.s.b.a.h2.H2ConsoleAutoConfiguration    : H2 console available at '/h2-console'. Database a
vailable at 'jdbc:h2:mem:testdb'
2025-11-15T16:00:19.009+05:30  INFO 24252 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
2025-11-15T16:00:19.024+05:30  INFO 24252 --- [  restartedMain] o.s.b.a.e.web.EndpointLinksResolver      : Exposing 1 endpoint beneath base path '/actuator'
2025-11-15T16:00:19.160+05:30  INFO 24252 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context p
ath '/'
2025-11-15T16:00:19.179+05:30  INFO 24252 --- [  restartedMain] c.e.accounts.AccountsApplication         : Started AccountsApplication in 10.129 seconds (pr
ocess running for 10.791)
```

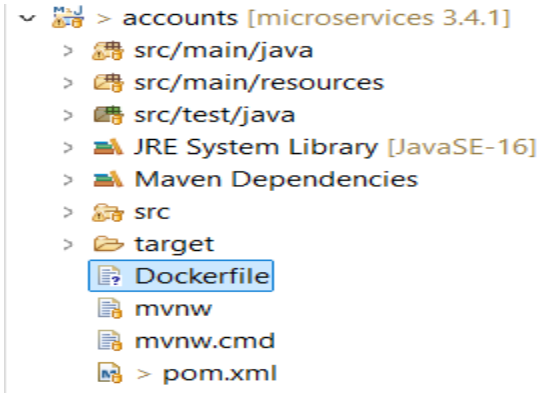
iii) In any IDE directly.

1) using Dockerfile

i) Create a file named Dockerfile (with no extension) in the project root by right-clicking on the project and selecting “New → File”.

The file name must be Dockerfile (exactly this name, with no extension), because Docker automatically looks for a file named Dockerfile while building the image





ii) Add the instructions in the Dockerfile, which are used to build your Docker image.

Dockerfile

```
# Start with a base image containing Java runtime

FROM openjdk:17-jdk-slim   # (image name: tag name)

# One line space, then add information about who maintains this Docker image

LABEL "org.opencontainers.image.authors"="jram224.com"

# Add the application's JAR to the image
# Tells Docker to take the same .jar from the target folder and put it into the Docker image

COPY target/accounts-0.0.1-SNAPSHOT.jar accounts-0.0.1-SNAPSHOT.jar

# Execute the application


ENTRYPOINT ["java", "-jar", "accounts-0.0.1-SNAPSHOT.jar"]

```

FROM openjdk:17-jdk-slim

- First requirement to run any Java application is a system with JRE/JDK.
- We instruct Docker to use an official Java 17 runtime as the base.

LABEL "org.opencontainers.image.authors"="jram224.com"

- Maintainer information: who is responsible for this Docker image.

COPY target/accounts-0.0.1-SNAPSHOT.jar accounts-0.0.1-SNAPSHOT.jar

- Add the compiled application JAR into the Docker image.
- Docker will copy the .jar from the target folder of your project.

ENTRYPOINT ["java", "-jar", "accounts-0.0.1-SNAPSHOT.jar"]

- Tells Docker how to run the application when the container starts.
- Executes the Spring Boot JAR inside the container automatically.

1. Open **Command Prompt** (cmd) **from your project folder** where pom.xml is located.

   Make sure this is also the folder containing your Dockerfile.

2. Run the following command to build your Docker image:

docker build . -t <dockerhub-username>/<image-name>:<tag>

- **<dockerhub-username>** → your Docker Hub username.
- **<image-name>** → the name you want to give the image.
- **<tag>** → version of the image (e.g., 1.0, latest).
- **.** → indicates the current directory as the build context.

Example:

docker build -t jram224/accounts:1.0 .

3. Docker will read the Dockerfile in the current folder and build the image.

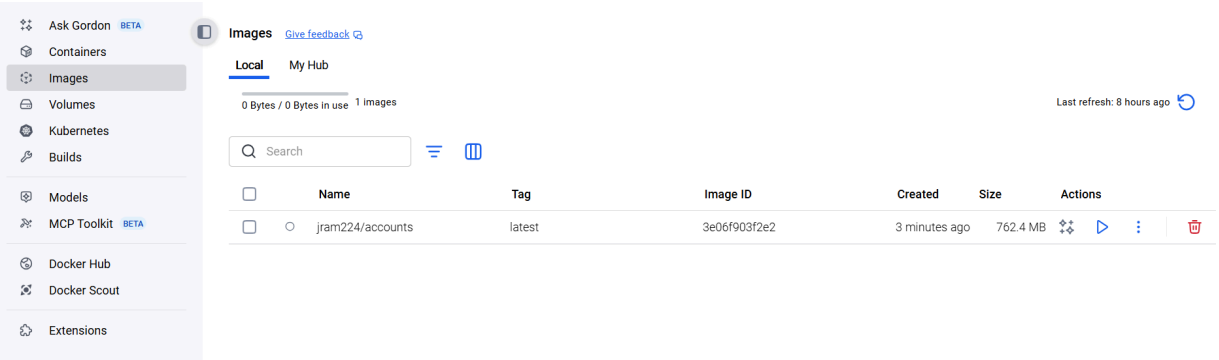
```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>docker build . -t jram224/accounts:latest
[+] Building 54.1s (7/7) FINISHED                                                                                  docker:desktop-linux
 => [internal] load build definition from Dockerfile                                                                              0.0s
 => => transferring dockerfile: 585B                                                                                              0.0s
 => [internal] load metadata for docker.io/library/openjdk:17.0.1-jdk-slim                                                        2.2s
 => [internal] load .dockerignore                                                                                                 0.0s
 => => transferring context: 2B                                                                                                   0.0s
 => [internal] load build context                                                                                                19.4s
 => => transferring context: 63.93MB                                                                                             19.4s
 => [1/2] FROM docker.io/library/openjdk:17.0.1-jdk-slim@sha256:fc5fa503124ba7021bbf8cb3718bf08791590d0aa2295c7cc551de65f9919290  47.9s
 => => resolve docker.io/library/openjdk:17.0.1-jdk-slim@sha256:fc5fa503124ba7021bbf8cb3718bf08791590d0aa2295c7cc551de65f9919290   0.0s
 => => sha256:a2abf6c4d29d43a4bf9fbb769f524d0fb36a2edab49819c1bf3e76f409f953ea 31.36MB / 31.36MB                                 12.8s
 => => sha256:2bbde5250315969db657b55bd8b2f5507fb659c0cf7f135edc84b684ffeab44a 1.58MB / 1.58MB                                    0.9s
 => => sha256:1343f138b677c0b1457cc7cb6310108df5388665281e0962273fb3492e52b86d 187.55MB / 187.55MB                               44.4s
 => => extracting sha256:a2abf6c4d29d43a4bf9fbb769f524d0fb36a2edab49819c1bf3e76f409f953ea                                         3.2s
 => => extracting sha256:2bbde5250315969db657b55bd8b2f5507fb659c0cf7f135edc84b684ffeab44a                                         0.3s
 => => extracting sha256:1343f138b677c0b1457cc7cb6310108df5388665281e0962273fb3492e52b86d                                         3.3s
 => [2/2] COPY target/accounts-0.0.1-SNAPSHOT.jar accounts-0.0.1-SNAPSHOT.jar                                                     0.9s
 => exporting to image                                                                                                            2.9s
 => => exporting layers                                                                                                           2.3s
 => => exporting manifest sha256:6e899f478c7381f3ca7ba73df37eac22f1476070db3c2f9738bb6b24c650e6ec                                 0.0s
 => => exporting config sha256:b43fe4172b9d3aad8104d642e2da94bcb787d310a17131541241eb808fdfcb5d                                   0.0s
 => => exporting attestation manifest sha256:53483db3ee3ffd8510eb482eab124c7f9ab981e7b04754480a2aaaa9373af496                     0.0s
 => => exporting manifest list sha256:3e06f903f2e26811035ef1c2b308a779af2dee887ea5ab0902005dad0232b6b2                            0.0s
 => => naming to docker.io/jram224/accounts:latest                                                                                0.0s
 => => unpacking to docker.io/jram224/accounts:latest                                                                             0.5s
```

3. After the build completes, you can verify the image using:
   You should also be able to list it using the command line:

   docker images

```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>docker images
REPOSITORY          TAG       IMAGE ID       CREATED              SIZE
jram224/accounts    latest    3e06f903f2e2   About a minute ago   762MB
```

After building the image, you can **see it in Docker Desktop** under the **Images** tab.



4. If you want to **inspect Docker images**

Inspect detailed image info

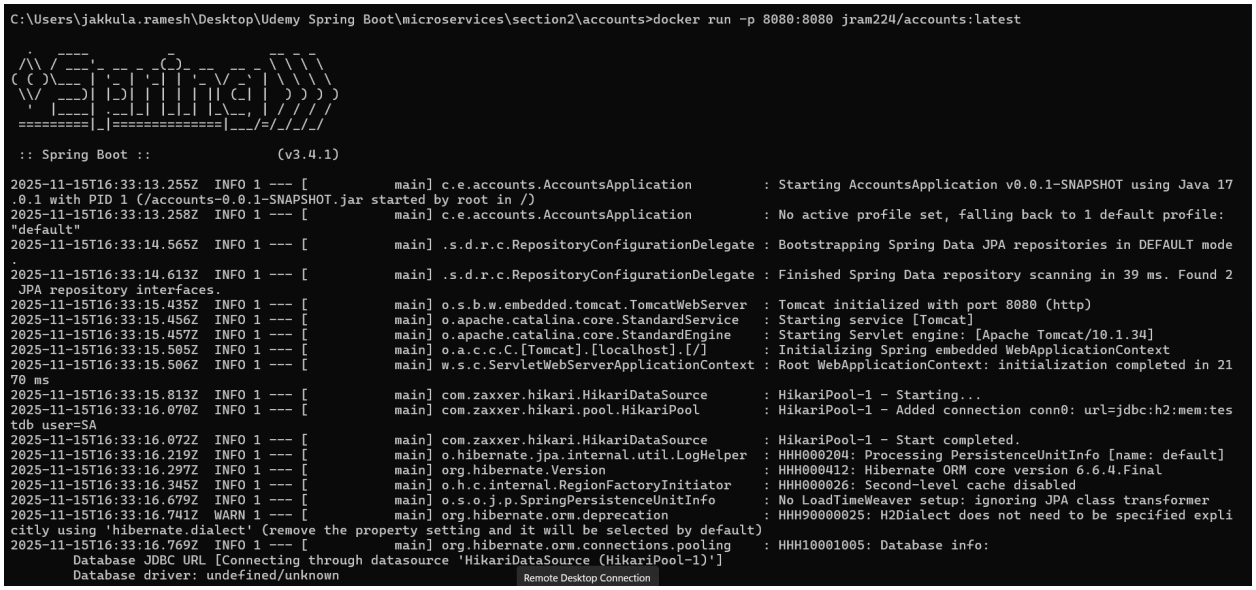
docker image inspect <image-id>

Shows JSON output with layers, environment variables, entrypoint, labels, and more.

```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>docker images
REPOSITORY         TAG       IMAGE ID       CREATED              SIZE
jram224/accounts   latest    3e06f903f2e2   About a minute ago   762MB

C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>docker inspect image 3e06f
[
    {
        "Id": "sha256:3e06f903f2e26811035ef1c2b308a779af2dee887ea5ab0902005dad0232b6b2",
        "RepoTags": [
            "jram224/accounts:latest"
        ],
        "RepoDigests": [
            "jram224/accounts@sha256:3e06f903f2e26811035ef1c2b308a779af2dee887ea5ab0902005dad0232b6b2"
        ],
        "Parent": "",
        "Comment": "buildkit.dockerfile.v0",
        "Created": "2025-11-15T16:06:01.901063866Z",
        "DockerVersion": "",
        "Author": "",
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 277988098,
        "GraphDriver": {
            "Data": null,
            "Name": "overlayfs"
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:2edcec3590a4ec7f40cf0743c15d78fb39d8326bc029073b41ef9727da6c851f",
                "sha256:afda989d53ee25f18b7810206b39580df30518043d1f34f1d514f665ba8026f4",
```

5. Run Docker Container Using the Image

docker run -p <host-port>:<container-port> <image-name>:<tag>



Step-by-Step Explanation

1. **docker run** → tells Docker to start a new container from an image.
2. **-p <host-port>:<container-port>** → maps a port from your container to your local machine:
    - When you run a Docker container, it has its **own internal network** separate from your computer.
    - Your Spring Boot app inside the container usually runs on a port defined in the application, e.g., **8080**.
    - **<container-port>** → the port your application runs **inside the container** (Spring Boot app port).
    - **<host-port>** → the port on your **local machine** you want to use to access the app.
3. **<image-name>:<tag>** → specifies which image to use.
    - Must match the image name you built (or pulled).

Running Docker Container in Detached Mode

By default, docker run runs the container in the **foreground**, showing all logs in your terminal.

While it's running, you **cannot run any other command** in the same terminal.

To avoid this, we use **detached mode** so the container runs in the background.

just add -d to your normal docker run command

docker run -d -p <host-port>:<container-port> <image-name>:<tag>

```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>docker run -d -p 8080:8080 jram224/accounts:latest
9a3c912e16928e8798d685152b78c4873dd777988a3d1159cb22a9e706a8cfec

C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>
```

We can check running containers with:

docker ps

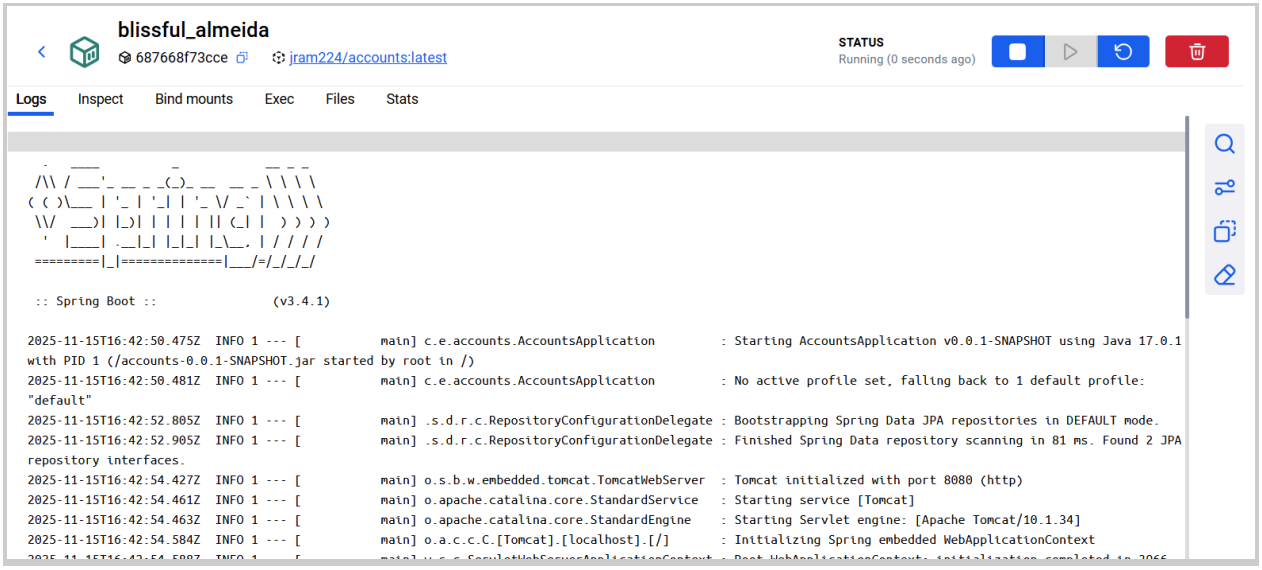
```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>docker ps
CONTAINER ID   IMAGE                    COMMAND                  CREATED         STATUS         PORTS                                       NAMES
9a3c912e1692   jram224/accounts:latest   "java -jar accounts-…"   3 minutes ago   Up 3 minutes   0.0.0.0:8080->8080/tcp, [::]:8080->8080/tcp   affectionate_
euler

C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>
```

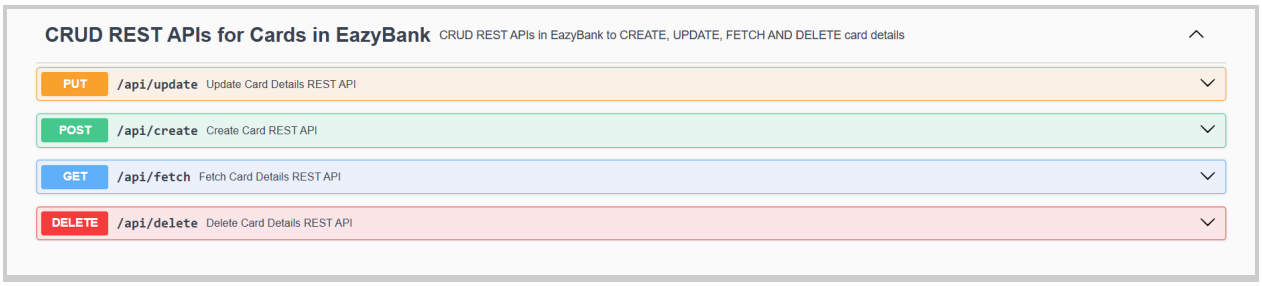
And stop them with:

docker stop <container_id>

```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>docker stop 9a3c912e1692
9a3c912e1692

C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES

C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>
```

We can also run Docker containers using **Docker Desktop**, without using the command line.





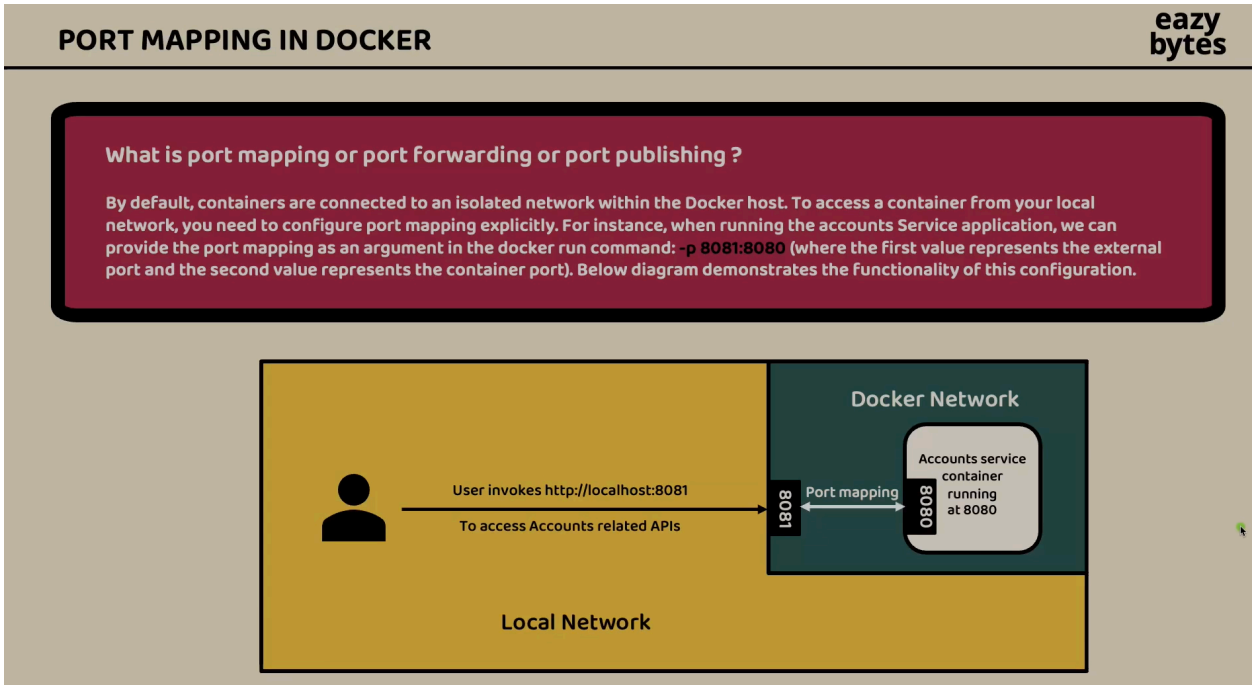
docker run → always creates a **new container** from the image.

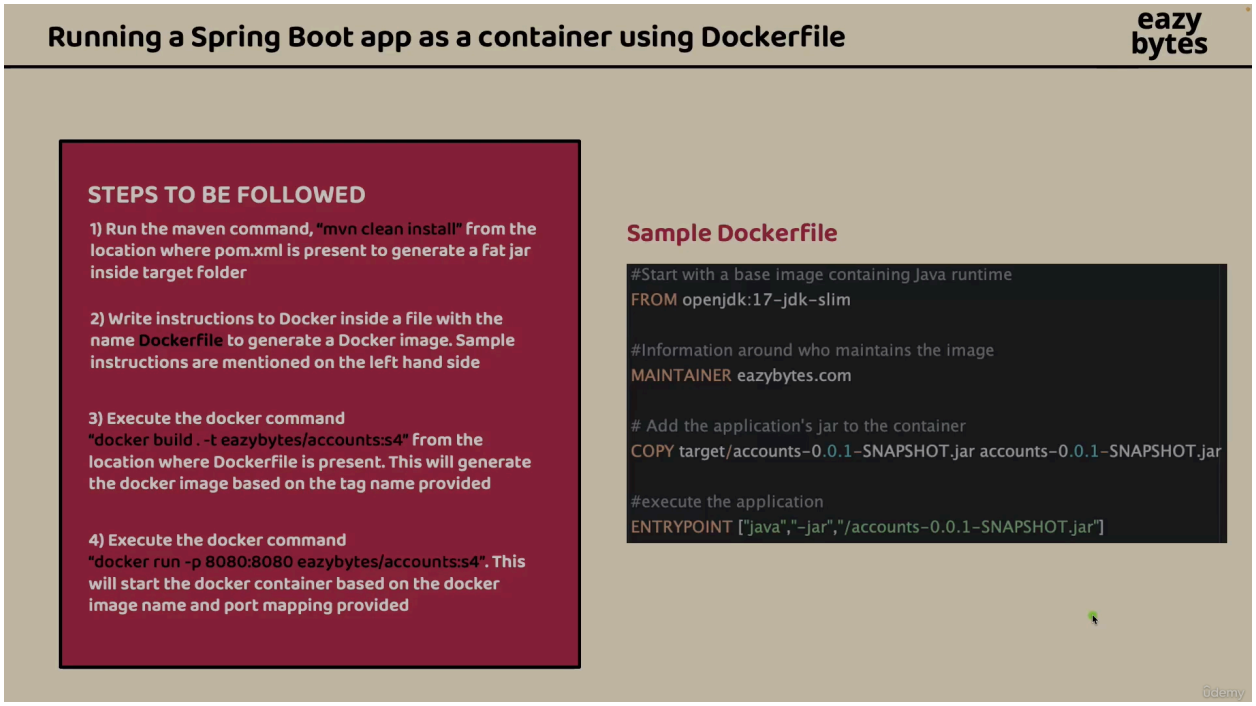
docker ps -a → lists all containers (running or stopped).

docker start <container_id> → starts a **previously created container** without creating a new one.

```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>docker ps -a
CONTAINER ID   IMAGE                     COMMAND                  CREATED          STATUS                      PORTS     NAMES
687668f73cce   jram224/accounts:latest   "java -jar accounts-…"   5 minutes ago    Exited (143) 5 minutes ago            blissful_almeida
9a3c912e1692   jram224/accounts:latest   "java -jar accounts-…"   10 minutes ago   Exited (143) 6 minutes ago            affectionate_euler
bdebfa29690f   jram224/accounts:latest   "java -jar accounts-…"   15 minutes ago   Exited (130) 11 minutes ago           ecstatic_cori

C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>docker start 687668f73cce
687668f73cce

C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>
```

Finally





Challenges with Dockerfile Approach

Steep Learning Curve

- Developers need to **learn all Dockerfile instructions** (FROM, COPY, RUN, ENTRYPOINT, etc.) to create optimized images.
- Also need to know **Docker concepts** like layers, caching, port mapping, volumes, and environment variables.

Maintenance & Updates

- Dockerfile may need frequent updates for:
    - Base image versions (openjdk:17-slim → newer version)
    - Application dependencies
    - Optimizations to reduce image size or improve build speed

Not Developer-Focused

- Developers often want to **focus on writing code**, not on DevOps tasks.
- Writing Dockerfiles requires effort outside their core skills.

Error-prone

- Small mistakes (like wrong COPY path, missing dependencies, incorrect ENTRYPOINT) can break builds.
- Need to test images repeatedly.

Alternatives

To reduce the burden on developers, we have **tools that automate image creation**:

Buildpacks **or** Google Jib(only for Java applications)

2) using Buildpacks

Official site : https://buildpacks.io/ , https://paketo.io/

Buildpacks automatically **convert your application into a Docker image** without writing a Dockerfile. Spring Boot Maven Plugin (spring-boot-maven-plugin) has Buildpacks support built-in since Spring Boot 2.3+ so don't need to install any separate Buildpacks plugin.

i) Added the **image name** under the Spring Boot plugin configuration in pom.xml.

The <image> configuration should be **inside spring-boot-maven-plugin**

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
            <image>
            <name>jram224/${project.artifactId}:latest</name>
            </image>
            </configuration>
        </plugin>
```

ii) Open a cmd terminal **where your pom.xml file is located** as Maven needs to read the pom.xml from your application folder. Make sure this pom.xml has the spring-boot-maven-plugin configured with the <image> tag, like your latest version.

Run the Buildpack Docker image command

```
mvn spring-boot:build-image
```

- Maven will automatically use **Paketo Buildpacks** to generate the Docker image.
- It will detect your **Java version** (from Maven compiler plugin / Spring Boot version) and build the image.

```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\loans>mvn spring-boot:build-image
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------< com.eazybytes:loans >-------------------------
[INFO] Building loans 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] >>> spring-boot-maven-plugin:3.4.1:build-image (default-cli) > package @ loans >>>
[INFO]
[INFO] --- maven-resources-plugin:3.3.1:resources (default-resources) @ loans ---
[INFO] Copying 1 resource from src\main\resources to target\classes
[INFO] Copying 1 resource from src\main\resources to target\classes
[INFO]
[INFO] --- maven-compiler-plugin:3.13.0:compile (default-compile) @ loans ---
[INFO] Recompiling the module because of added or removed source files.
```

```
[INFO] Successfully built image 'docker.io/jram224/loans:latest'
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  03:28 min
[INFO] Finished at: 2025-11-16T12:27:22+05:30
[INFO] ------------------------------------------------------------------------
```

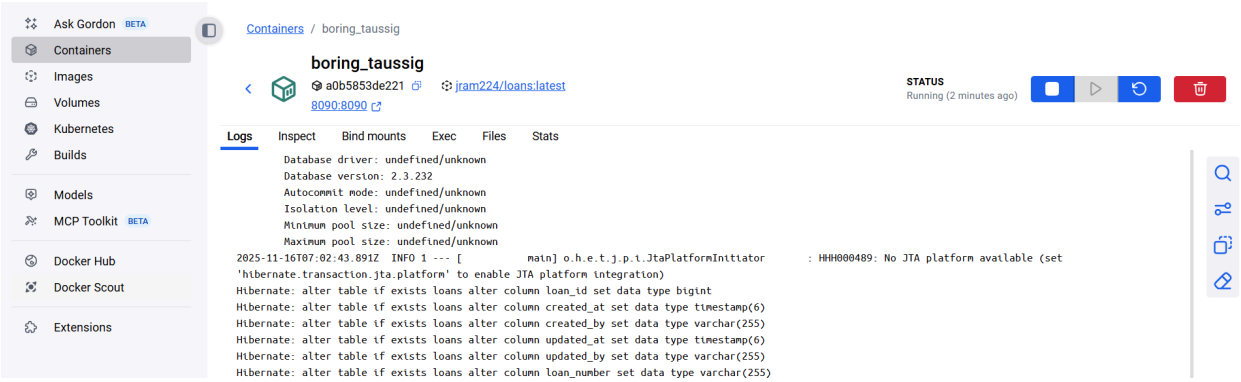
iii) Run Docker Container Using the Image

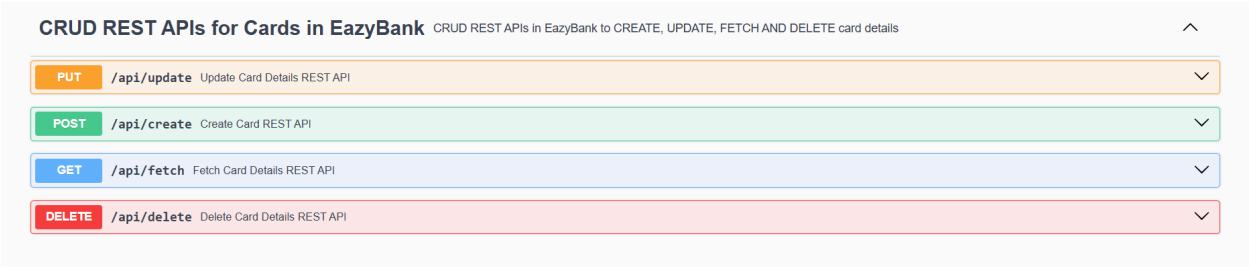
docker run -d -p <host-port>:<container-port> <image-name>:<tag>

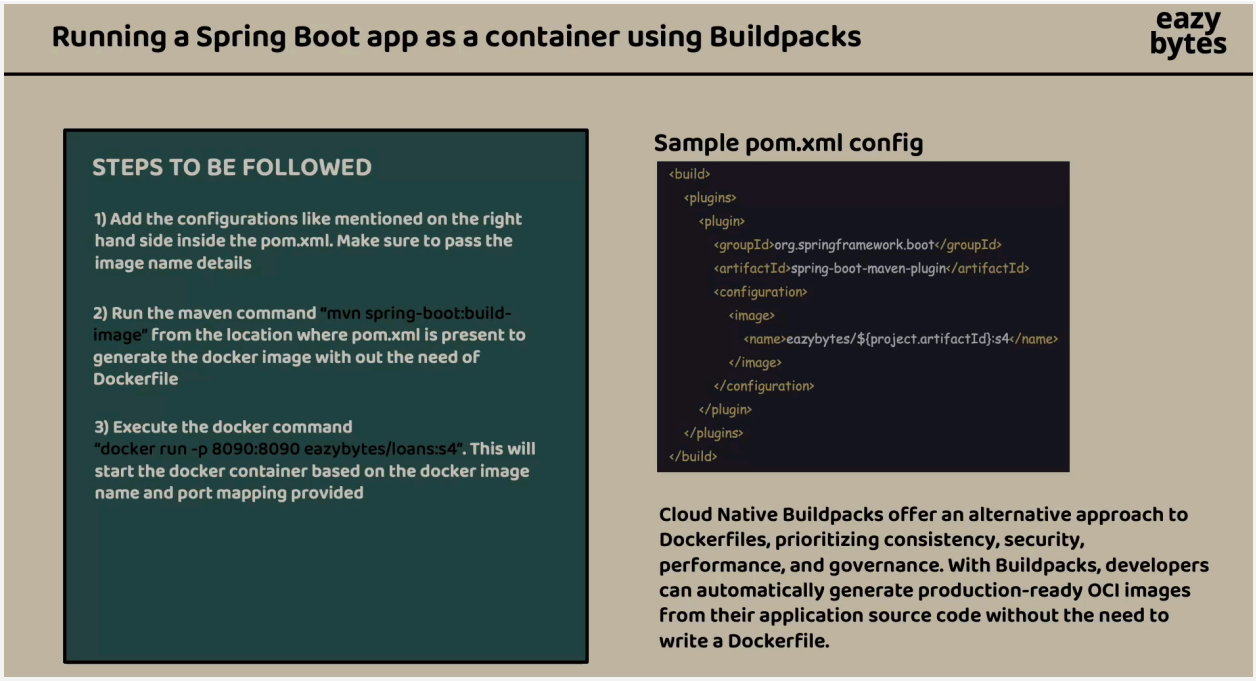
The “45 years ago” date shows up because the base image used by Buildpacks doesn’t have real timestamps, so Docker defaults to the Unix epoch (1970).

```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\loans>docker images
REPOSITORY                                TAG       IMAGE ID       CREATED         SIZE
jram224/accounts                          latest    3e06f903f2e2   15 hours ago    762MB
paketobuildpacks/run-jammy-tiny           latest    e108bbfe02f8   3 days ago      38.7MB
jram224/loans                             latest    0fc4b2a76bd4   45 years ago    561MB
paketobuildpacks/builder-jammy-java-tiny  latest    4c0d99dc3034   45 years ago    1.13GB

C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\loans>docker run -d -p 8090:8090 jram224/loans:latest
a0b5853de22195bb05137764a6fbf5009ad11d359c87446193380671f93626c1

C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\loans>
```

		Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
☐	○	blissful_almeida	687668f73cce	jram224/accounts:latest		0%	14 hours ago	
☐	●	boring_taussig	a0b5853de221	jram224/loans:latest	8090:8090	0.13%	1 minute ago	







2) using Google Jib

URL: https://github.com/GoogleContainerTools/jib , https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin#quickstart

i) Make sure the jib-maven-plugin is inside your <plugins> section of pom.xml.

```
            <plugin>
                <groupId>com.google.cloud.tools</groupId>
                <artifactId>jib-maven-plugin</artifactId>
                <version>3.4.6</version>
                <configuration>
                    <to>
                        <image>jram224/${project.artifactId}:latest</image>
                    </to>
                </configuration>
            </plugin>
```

ii) Open a cmd terminal **where your pom.xml file is located** as Maven needs to read the pom.xml from your application folder.

Run the Buildpack Docker image command

```
mvn compile jib:dockerBuild
```

jib:dockerBuild builds the Docker image **directly into your local Docker environment**.

No Dockerfile is needed.

Jib automatically detects your Java version from the project.

The resulting image will have your configured <image> name and <tag>.

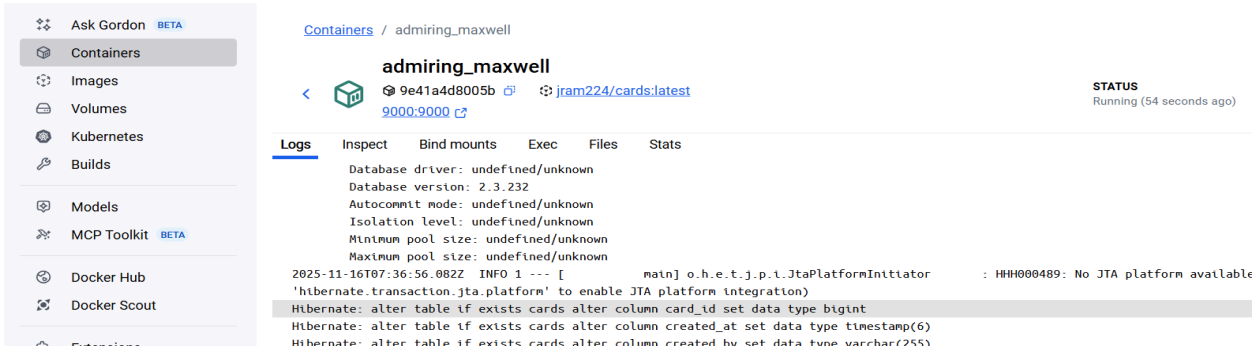
```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\cards>mvn compile jib:dockerBuild
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------< com.eazybytes:cards >-------------------------
[INFO] Building cards 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-resources-plugin:3.3.1:resources (default-resources) @ cards ---
[INFO] Copying 1 resource from src\main\resources to target\classes
[INFO] Copying 1 resource from src\main\resources to target\classes
[INFO]
```

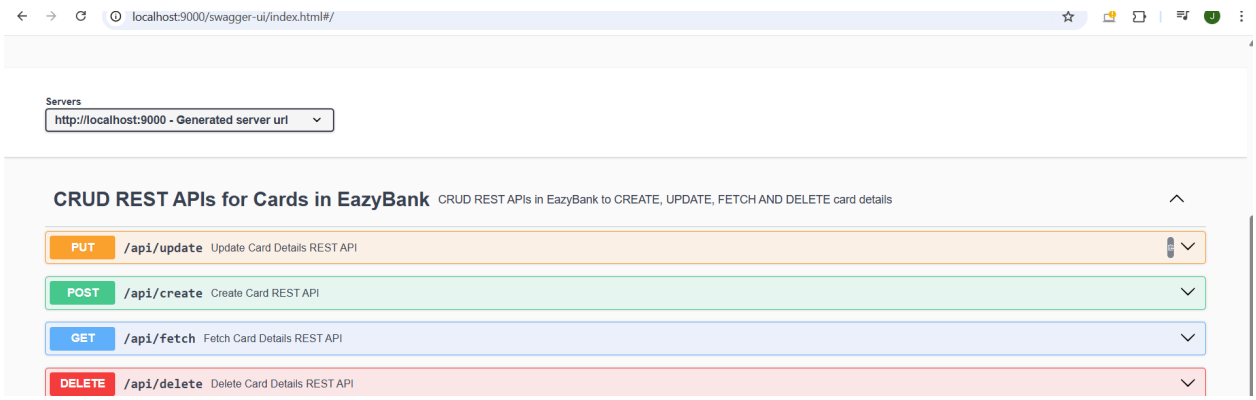
```
[INFO] Using credentials from Docker config (C:\Users\jakkula.ramesh\.docker\config.json) for eclipse-temurin:17-jre
[INFO] Using base image with digest: sha256:75ab7d1b4b18483e9245342cbee253b558952c1def5c1c18956196330a01683e
[INFO]
[INFO] Container entrypoint set to [java, -cp, @/app/jib-classpath-file, com.eazybytes.cards.CardsApplication]
[INFO]
[INFO] Built image to Docker daemon as jram224/cards
[INFO] Executing tasks:
[INFO] [==============================] 100.0% complete
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:31 min
[INFO] Finished at: 2025-11-16T13:04:44+05:30
[INFO] ------------------------------------------------------------------------
```

iii) Run Docker Container Using the Image

```
docker run -d -p <host-port>:<container-port> <image-name>:<tag>
```

```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\cards>docker images
REPOSITORY                                TAG       IMAGE ID       CREATED         SIZE
jram224/accounts                          latest    3e06f903f2e2   16 hours ago    762MB
paketobuildpacks/run-jammy-tiny           latest    e108bbfe02f8   3 days ago      38.7MB
jram224/loans                             latest    0fc4b2a76bd4   45 years ago    561MB
paketobuildpacks/builder-jammy-java-tiny  latest    4c0d99dc3034   45 years ago    1.13GB
jram224/cards                             latest    c3bc1b14dfd1   55 years ago    493MB

C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\cards>docker run -d -p 9000:9000 jram224/cards:latest
9e41a4d8005b6baaafff8dd69cc59f0082717ff8d9a426d01a11d7d26a706152
```





Running a Spring Boot app as a container using Google Jib

eazy bytes

STEPS TO BE FOLLOWED

1) Add the configurations like mentioned on the right hand side inside the pom.xml. Make sure to pass the image name details

2) Run the maven command "mvn compile jib:dockerBuild" From the location where pom.xml is present to generate the docker image with out the need of Dockerfile

3) Execute the docker command "docker run -p 9000:9000 eazybytes/cards:s4". This will start the docker container based on the docker image name and port mapping provided

Sample pom.xml config

```
<build>
  <plugins>
    <plugin>
      <groupId>com.google.cloud.tools</groupId>
      <artifactId>jib-maven-plugin</artifactId>
      <version>3.3.2</version>
      <configuration>
        <to>
          <image>eazybytes/${project.artifactId}:s4</image>
        </to>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Google Jib offer an alternative approach to Dockerfiles, prioritizing consistency, security, performance, and governance. With Jib, developers can automatically generate production-ready OCI images from their application source code without the need to write a Dockerfile and even local Docker setup.

Comparison of the Dockerfile, Buildpacks, and Jib:

https://buildpacks.io/features/

Comparison

	Cloud Native Buildpacks	Dockerfile	source-to-image (s2i)	Jib	ko
Advanced Caching	Yes	No	Yes	No	Yes
Auto-detection	Yes	No	Yes	Yes	Yes
Bill-of-Materials	Yes	No	No	No	Yes
Modular / Pluggable	Yes	No	No	N/A †	N/A †
Multi-language	Yes	Yes	Yes	No	No
Multi-process	Yes	No	No	No	No
Minimal app image	Yes	Yes ♦	Yes ‡	Yes	Yes
Rebasing	Yes	No	No	No	No
Reproducibility	Yes	No	No	Yes	Yes
Reusability	Yes	No	Yes	N/A †	N/A †
Integrations	○ Azure ○ CircleCI ○ GitLab ○ Google ○ Heroku ○ Spring Boot ○ Tekton ○ ...	○ Amazon ECS ○ Azure ○ CircleCI ○ GitLab ○ Google ○ Tekton ○ ...	○ OpenShift	○ Gradle ○ Maven	○ Terraform ○ GoReleaser ○ Skaffold ○ Carvel kbld ○ Tilt ○ ...
Governance	CNCF	Docker	Red Hat	Google	CNCF
Best for Building...	○ Applications	○ Applications ○ Base images ○ OS Images	○ Applications	○ Applications	○ Applications

So far, we have generated Docker images and stored them only in our local machine.
But storing images locally does not make sense for real projects — because applications must be deployed in multiple environments like Dev, QA, UAT, and Production.

To make the image available to all environments, **we need to push our Docker images to a remote registry** (public or private).
Once the image is stored in a remote registry, **any environment can pull the same image whenever needed**, ensuring consistency.

Examples of remote image repositories:

- **Docker Hub** (Public)
- **GitHub Container Registry (GHCR)**
- **AWS ECR (Elastic Container Registry)**
- **Azure Container Registry (ACR)**
- **Google Container Registry / Artifact Registry**
- **Private/self-hosted registries** like Harbor, JFrog Artifactory, Nexus

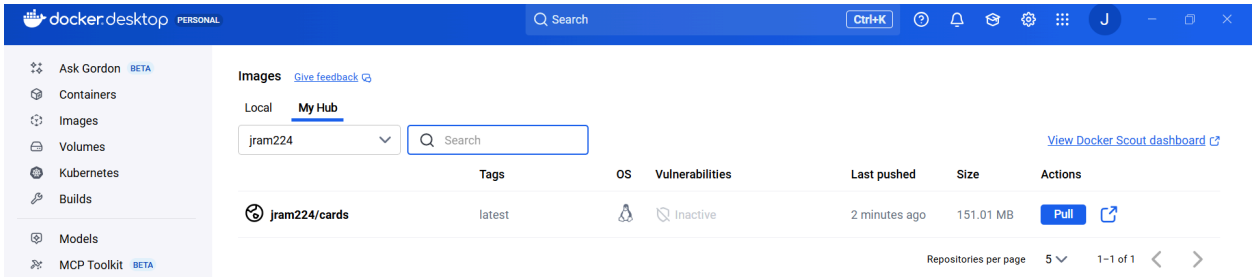
How to Push Docker Images to Docker Hub

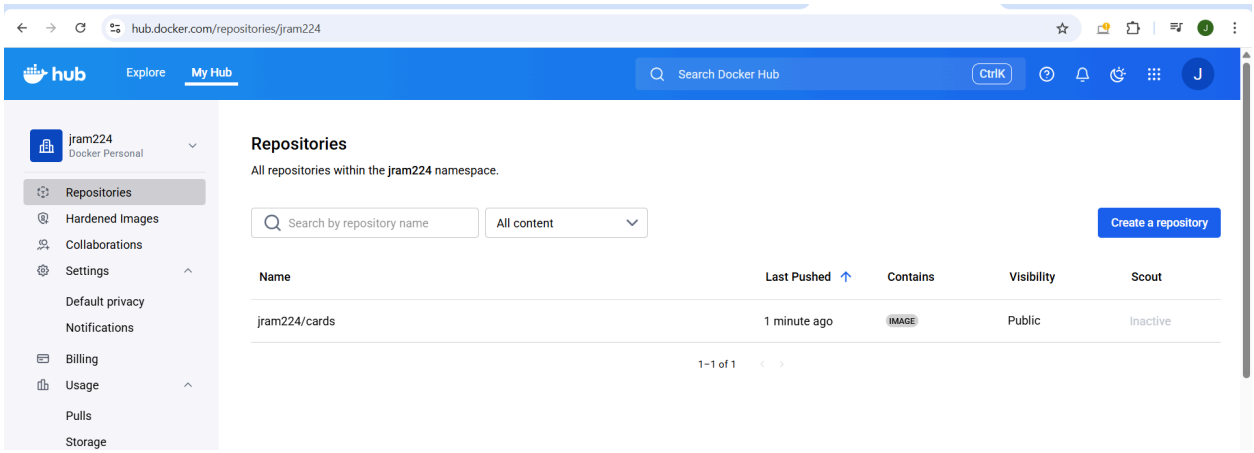
Since you are already **logged in to Docker Desktop** on your local machine, Because of this existing authentication, **you can push images to Docker Hub without entering your username and password again**.

docker image push docker.io/<imagename>:<tag>

```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts> docker ps -a
CONTAINER ID   IMAGE                     COMMAND                  CREATED        STATUS                      PORTS     NAMES
9e41a4d8005b   jram224/cards:latest      "java -cp @/app/jib-…"   20 hours ago   Exited (143) 20 hours ago             admiring_maxwell
a0b5853de221   jram224/loans:latest      "/cnb/process/web"       21 hours ago   Exited (143) 20 hours ago             boring_taussig
687668f73cce   jram224/accounts:latest   "java -jar accounts-…"   35 hours ago   Exited (143) 35 hours ago             blissful_almeida

C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>docker image push docker.io/jram224/cards:latest
The push refers to repository [docker.io/jram224/cards]
2581bc3ff3b6: Pushed
ec1e0321681c: Pushed
d9c6d7e52592: Pushed
b315b75540ca: Pushed
20043066d3d5: Pushed
469f7f46f06b: Pushed
a12c659f8ac1: Pushed
2abdf4167e30: Pushed
fd4969aa9957: Pushed
latest: digest: sha256:c3bc1b14dfd1324cbbef6ca568e19ac9f5567b942b30485cb01c93cb41caf14f size: 1729

C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>
```

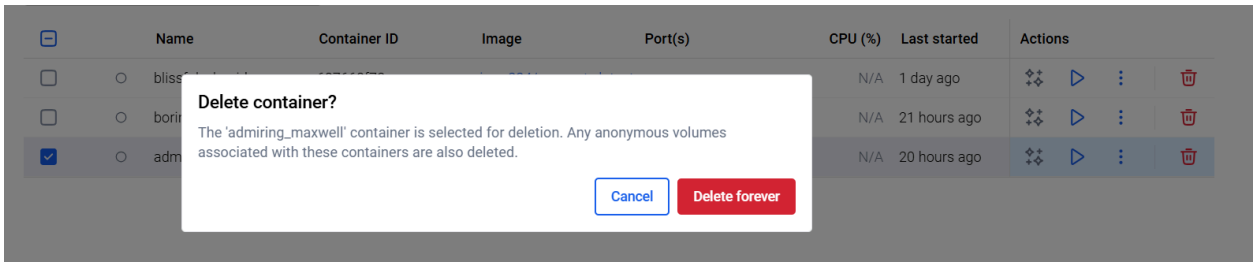
We can see either docker hub site or Hub option in docker desktop.

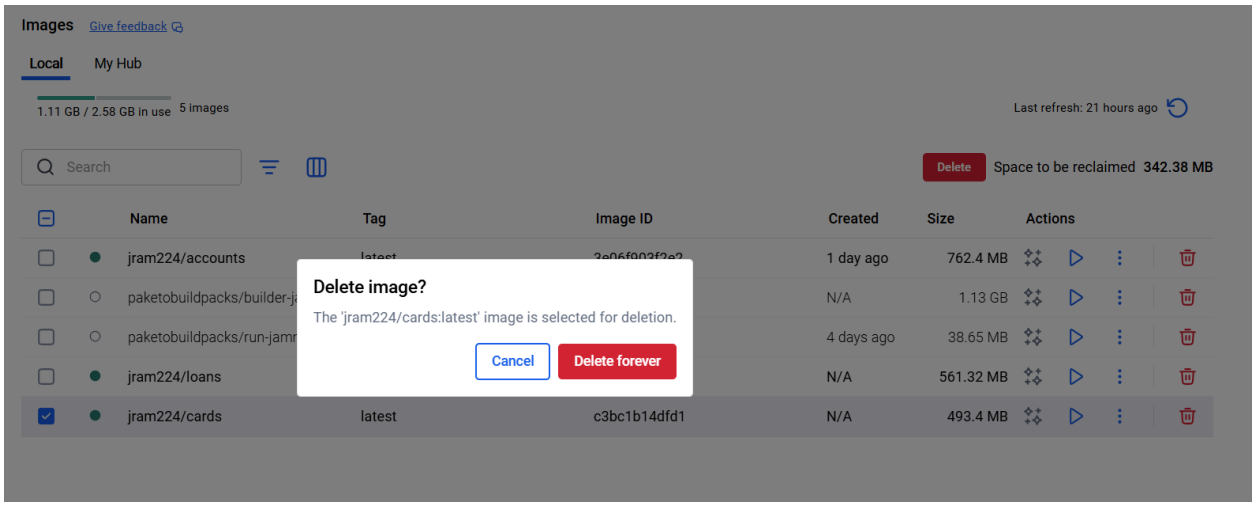




We can pull Docker images from a remote Docker registry (Docker Hub, ECR, ACR, etc.)also.

Let me test , i will delete cards docker images from local and will pull it from remote repo.






```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts> docker ps -a
CONTAINER ID   IMAGE                     COMMAND                  CREATED        STATUS                      PORTS     NAMES
a0b5853de221   jram224/loans:latest      "/cnb/process/web"       21 hours ago   Exited (143) 21 hours ago             boring_taussig
687668f73cce   jram224/accounts:latest   "java -jar accounts-…"   35 hours ago   Exited (143) 35 hours ago             blissful_almeida

C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>
```

Pull
docker pull <imagename>:<tag>

```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>docker pull jram224/cards:latest
latest: Pulling from jram224/cards
Digest: sha256:c3bc1b14dfd1324cbbef6ca568e19ac9f5567b942b30485cb01c93cb41caf14f
Status: Image is up to date for jram224/cards:latest
docker.io/jram224/cards:latest
```

```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>docker images
REPOSITORY                                TAG       IMAGE ID       CREATED        SIZE
jram224/accounts                          latest    3e06f903f2e2   36 hours ago   762MB
paketobuildpacks/run-jammy-tiny           latest    e108bbfe02f8   4 days ago     38.7MB
jram224/loans                             latest    0fc4b2a76bd4   45 years ago   561MB
paketobuildpacks/builder-jammy-java-tiny  latest    4c0d99dc3034   45 years ago   1.13GB
jram224/cards                             latest    c3bc1b14dfd1   55 years ago   493MB
```

After we build Docker images (either locally or by pulling them from a remote registry like Docker Hub), we normally run the container using:

docker run <options> <image-name>

But this approach has **major disadvantages**:

❌ 1. Running multiple applications becomes painful

If you want to run many services (e.g., accounts, loans, cards microservices),
you must run:

docker run ...

docker run ...

docker run ...

for each application manually.

❌ 2. Running multiple instances is even more time-consuming

If you want **multiple instances** of the same application for load balancing (e.g., 3 replicas):

docker run ...

docker run ...

docker run ...

You must repeat it manually → very slow and error-prone.

❌ 3. Managing networks, volumes, and environment variables is difficult

You must manually provide:

- ports
- volumes
- networks
- env variables (--env)
- dependencies (--link)

This becomes messy.

Solution: Use Docker Compose

To overcome all these limitations, Docker provides **Docker Compose**.

Docker Compose

site:https://docs.docker.com/compose/
Docker Compose is a tool for defining and running multi-container applications.

Compose simplifies the control of your entire application stack, making it easy to manage services, networks, and volumes in a single YAML configuration file. Then, with a single command, you create and start all the services from your configuration file.

Compose works in all environments - production, staging, development, testing, as well as CI workflows. It also has commands for managing the whole lifecycle of your application:

- Start, stop, and rebuild services
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service

Without Compose

Run 6 commands manually:

docker run service1

docker run service2

docker run service3

docker run instance1

docker run instance2

With Compose

Run **one** command:

docker compose up

If you want, I can also show:

Complete **docker-compose.yml** for your Spring Boot microservices.

1️⃣ **Start all services**

docker compose up

Runs all containers defined in docker-compose.yml.

To run in the background:

docker compose up -d

2️⃣ **Stop and remove containers, networks, volumes created by compose**

docker compose down

Equivalent to:

- stop containers
- remove containers
- remove networks

3️⃣ **Start existing containers (without recreating)**

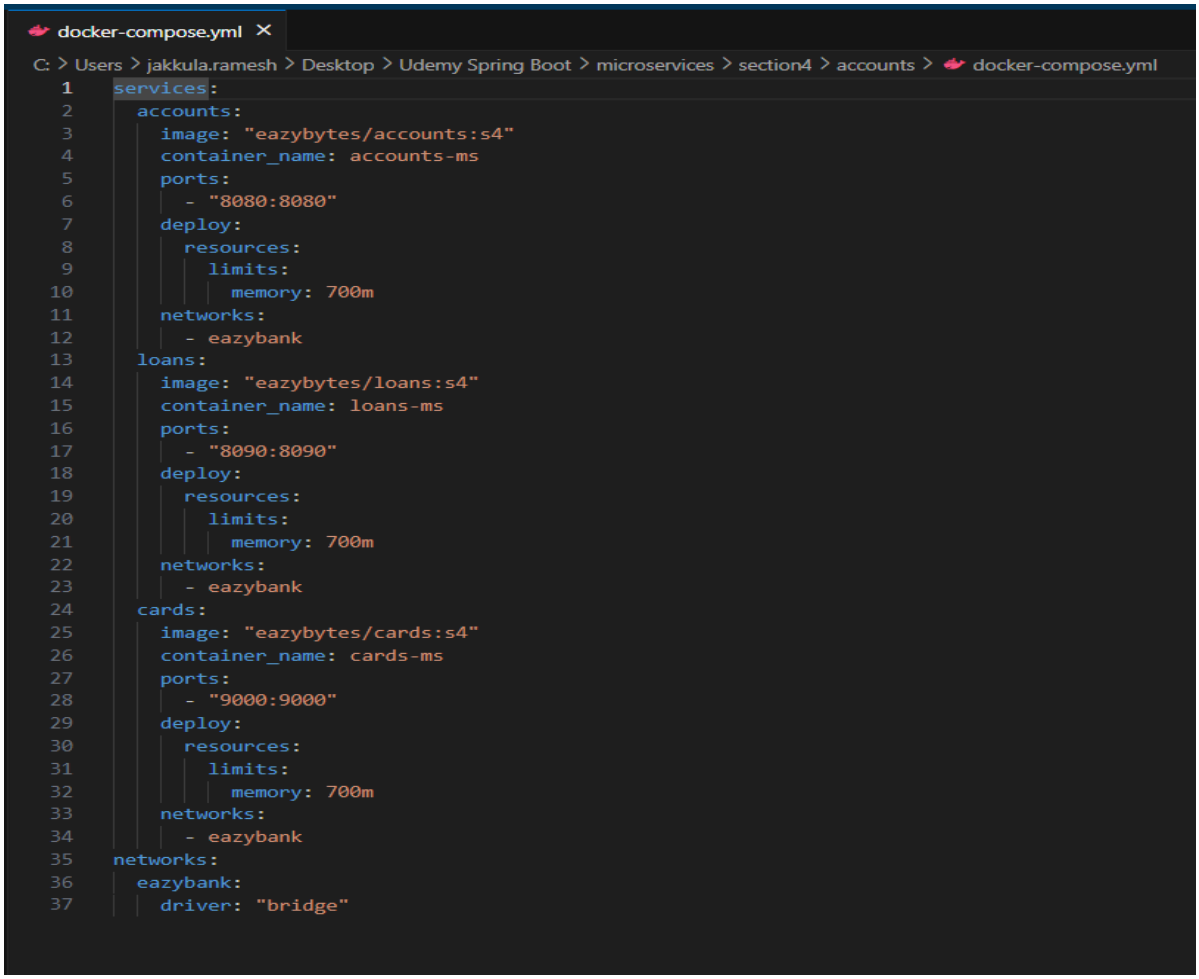
docker compose start

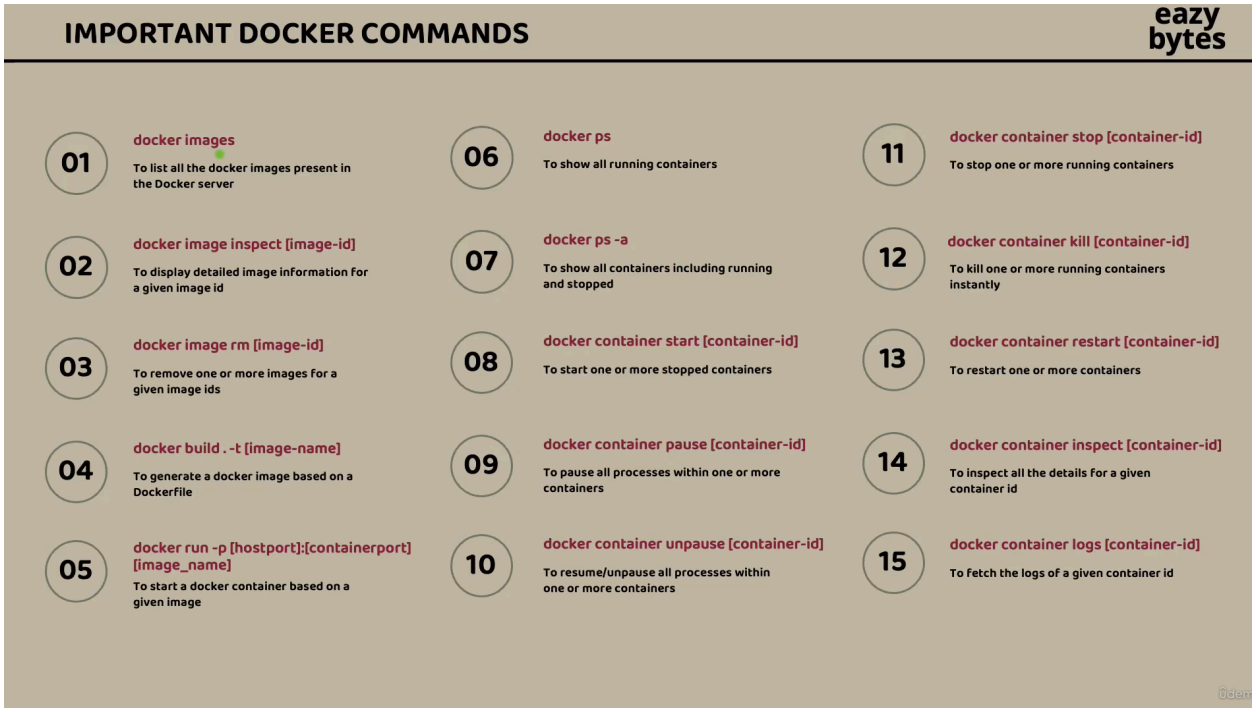
Start containers that are already created but stopped.

4️⃣ Stop running containers (but do not remove them)

docker compose stop

Stops services but does *not* remove containers.



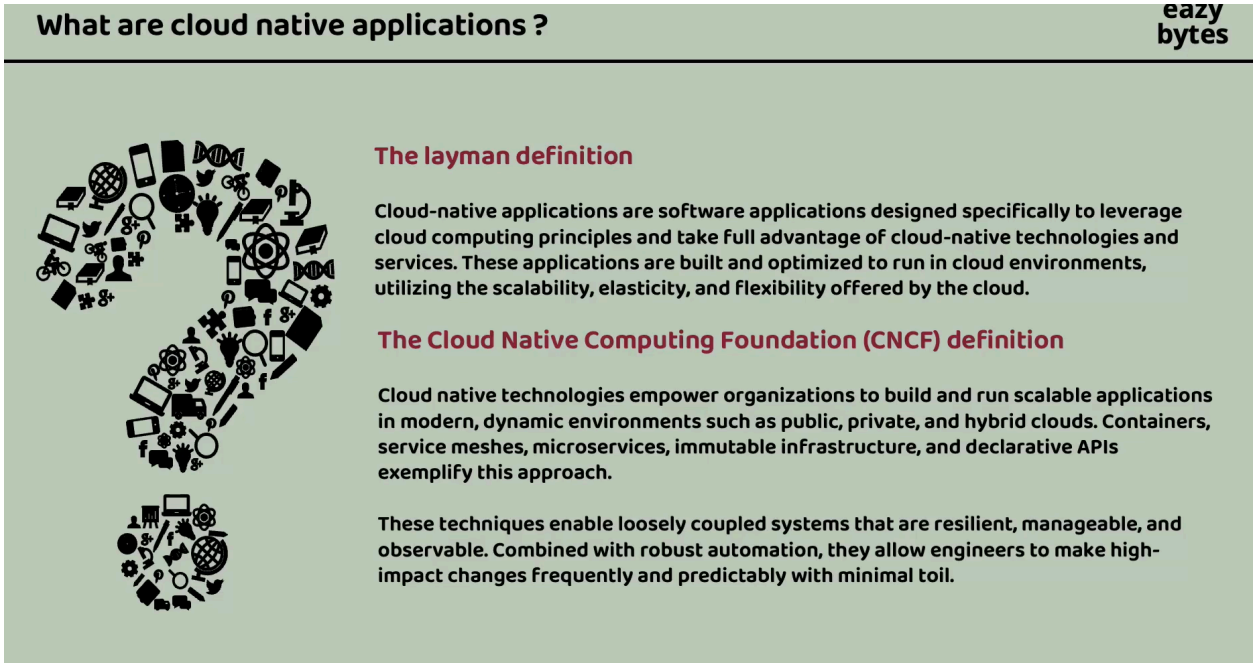


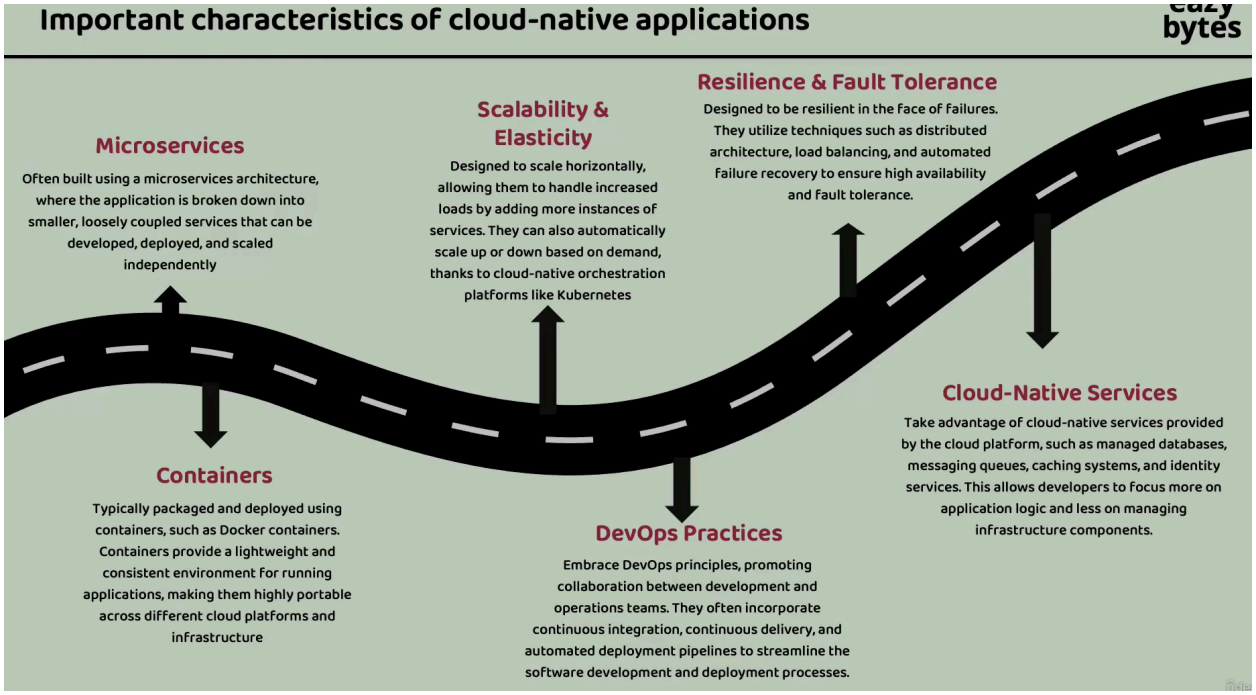
Cloud-Native Application

Cloud-native applications are **modern software applications designed and built specifically to run in cloud environments**.

A **cloud-native application** is an application built to **run, scale, and recover automatically in the cloud**, using technologies like **containers, microservices, DevOps, and Kubernetes** to achieve high scalability, resilience, and agility (Speed + Flexibility).

They are broken down into small, independent services that can be developed, deployed, and scaled individually without impacting other parts of the application, leading to faster updates and greater flexibility.





Key Characteristics of Cloud-Native Applications

1 Microservices Architecture

- Application is broken into small, independent services.
- Each service can be developed, deployed, and scaled separately.

2️⃣ Containers (Docker)

- Each microservice runs inside a lightweight container.
- Ensures consistency across environments (dev → test → prod).

3️⃣ Orchestration (Kubernetes)

- Manages container deployment, scaling, and recovery automatically.

4️⃣ DevOps + CI/CD

- Automated build, testing, deployment pipelines.
- Enables frequent and reliable releases.

5️⃣ Scalability & Elasticity

- Applications automatically scale up during high load and scale down during low load.

6️⃣ Resilience & Fault Tolerance

- If one microservice fails, the system continues running.
- Self-healing, rollback, and auto-recovery.

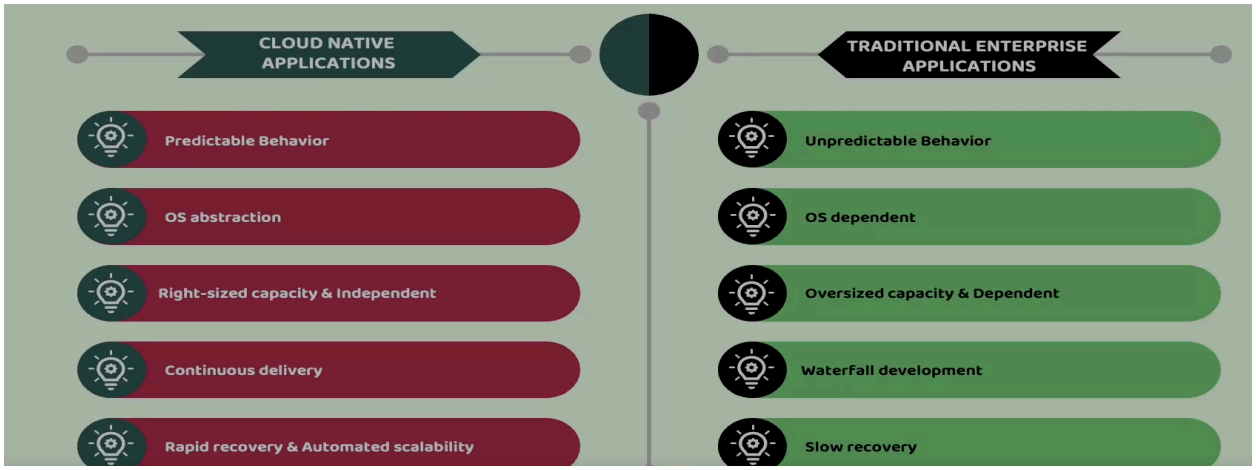
7️⃣ API-Driven

- Services communicate using APIs (REST, gRPC, GraphQL).

8️⃣ Cloud Managed Services

- Uses cloud-native databases, caches, queues (e.g., AWS DynamoDB, Azure Cosmos DB, GCP Pub/Sub).

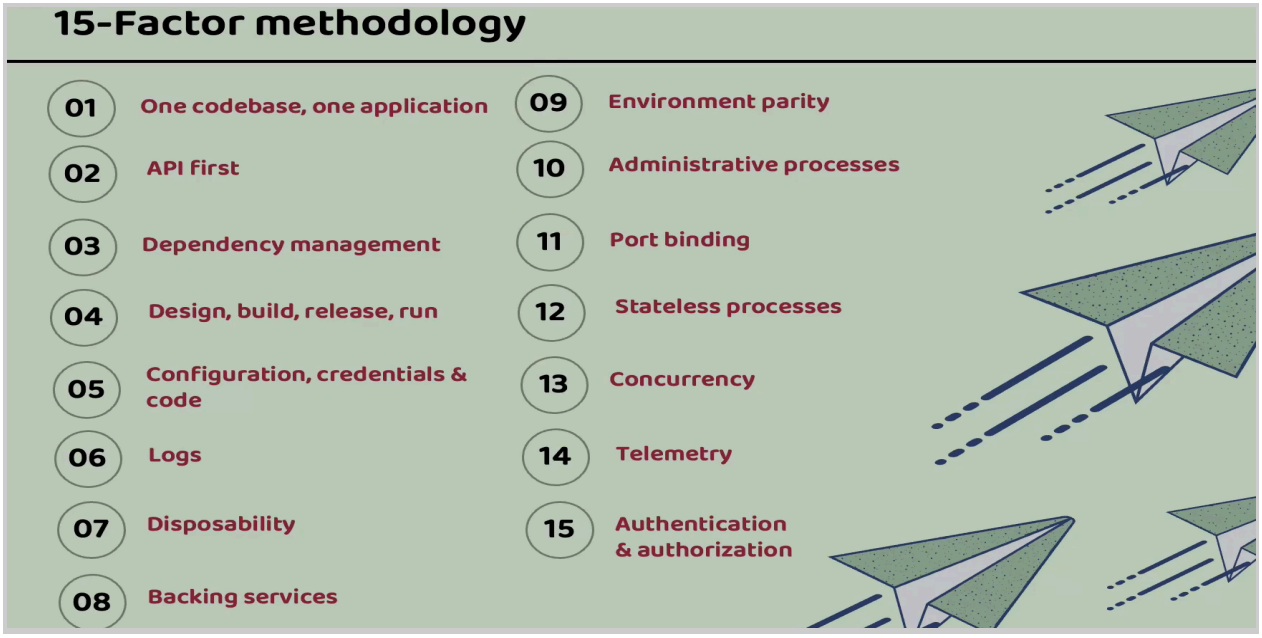
Cloud-Native vs Traditional Applications



📌 Cloud-Native vs Traditional Applications

Feature	Cloud-Native	Traditional (Monolithic)
Architecture	Microservices	One big application
Deployment	Containers (Docker)	VMs or servers
Scaling	Automatic, per service	Manual, whole app
Resilience	Self-healing	Single point of failure
Release Cycle	Fast, frequent	Slow, risky
Cloud Compatibility	Built for cloud	Adapted to cloud

Core Development Principles of Cloud-Native Applications

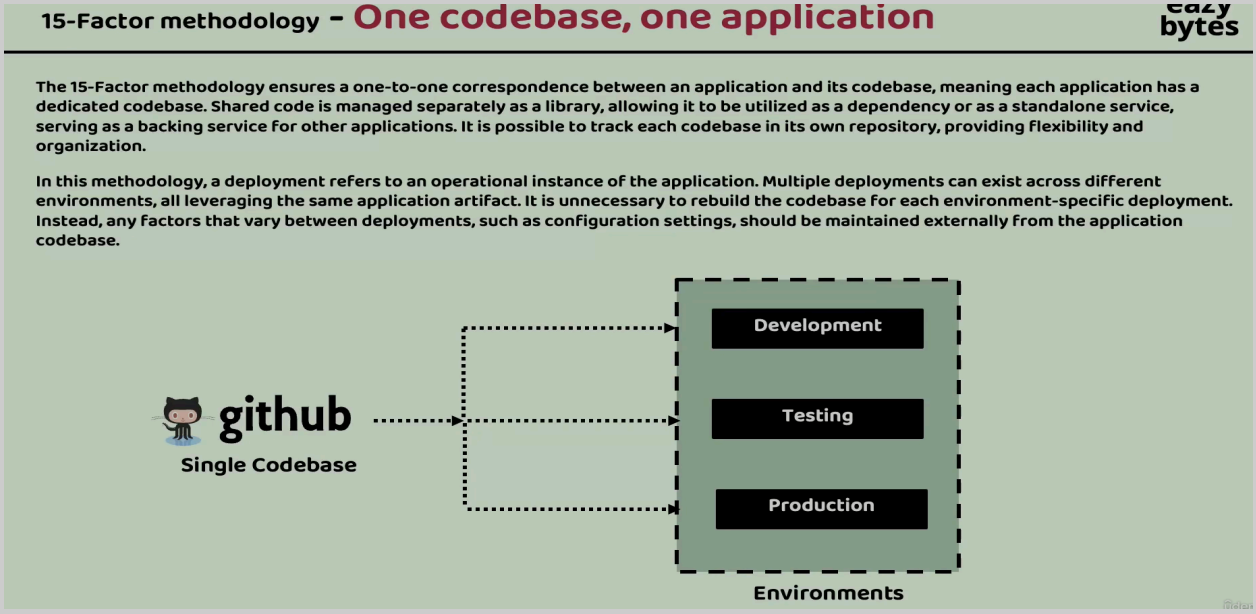


1️⃣ One Codebase, One Application

Each application must have **one single codebase**, tracked in version control (like GitHub).
Multiple deployments (Dev, Test, Prod) must all come from the **same codebase**.

Key Points:

- One codebase → many deployments
- No separate repos for each environment
- Config changes should be externalized, not kept in code
- Shared code should be packaged as libraries or services, not duplicated



2️⃣ API First

APIs are designed **before** writing the actual implementation.
This ensures clear contracts between services.

Key Points:

- Follows distributed system design
- Teams work independently using API contracts
- Prevents integration issues
- API changes won't break other services if the contract remains stable
- Helps in building reusable and testable services

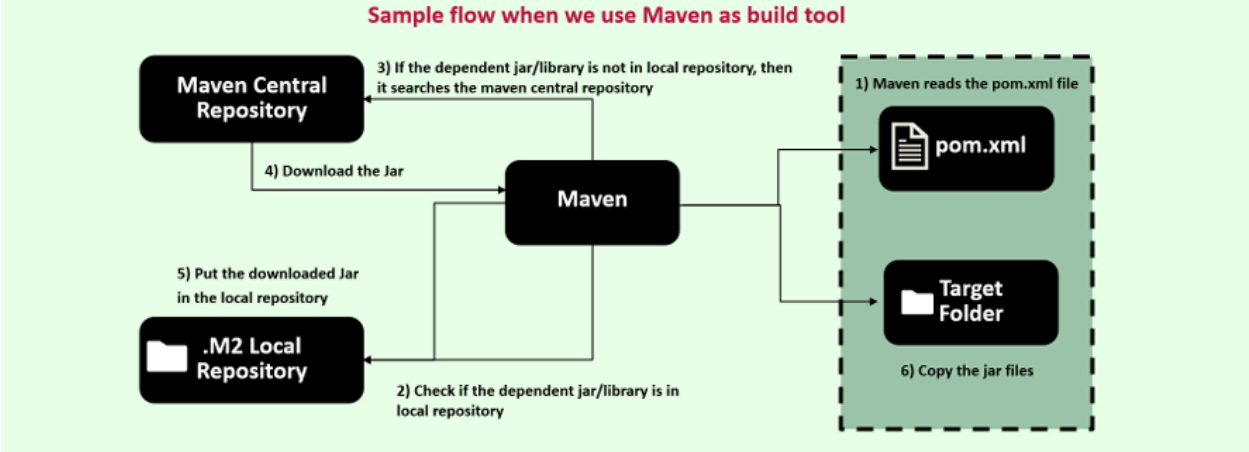
3️⃣ Dependency Management

All dependencies must be **declared explicitly** in a manifest (like pom.xml).

Key Points:

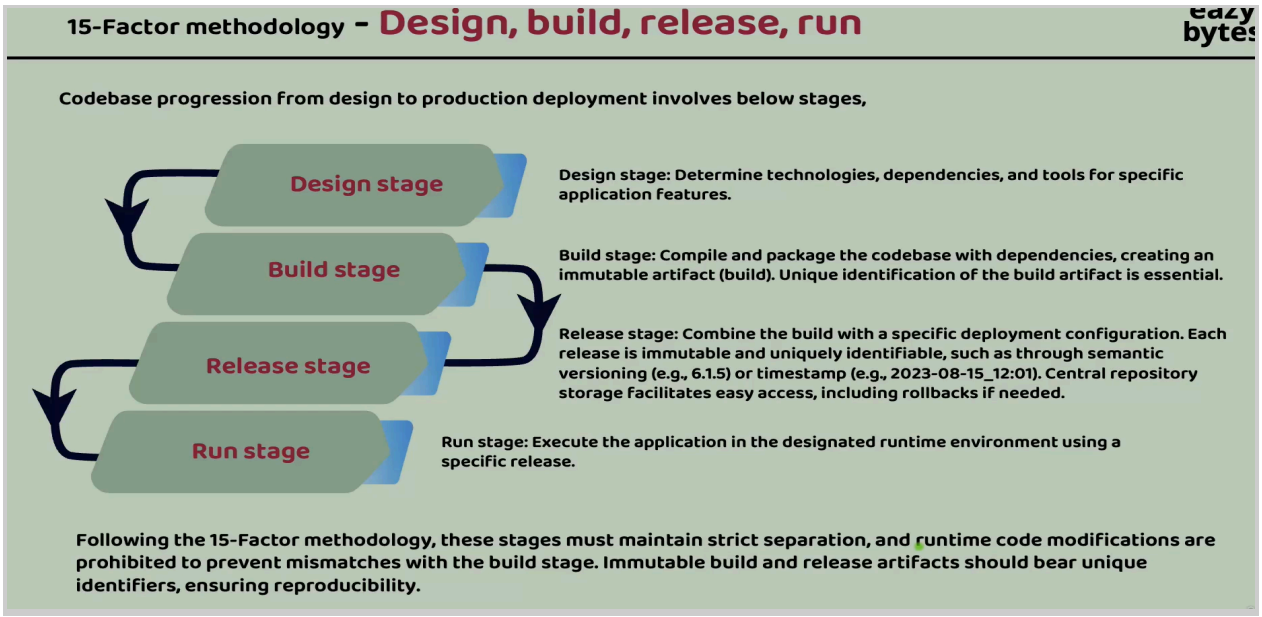
- No implicit/hidden dependencies
- Use dependency managers (Maven, Gradle, npm, etc.)
- Dependency manager will:
    - Check local repo
    - If missing → download from central repo
    - Store in cache
- Ensures consistency and reproducibility

Java Example Flow (Maven):



4 Design, Build, Release, Run

Application lifecycle is split into **four strict stages**, and they should not mix.



Key Points:

- No code changes allowed at runtime
- Build & Release must be immutable
- Release must be uniquely identifiable (version or timestamp)
- Ensures reproducibility and easier rollbacks

5 Configuration, Credentials & Code (Separation of Concerns)

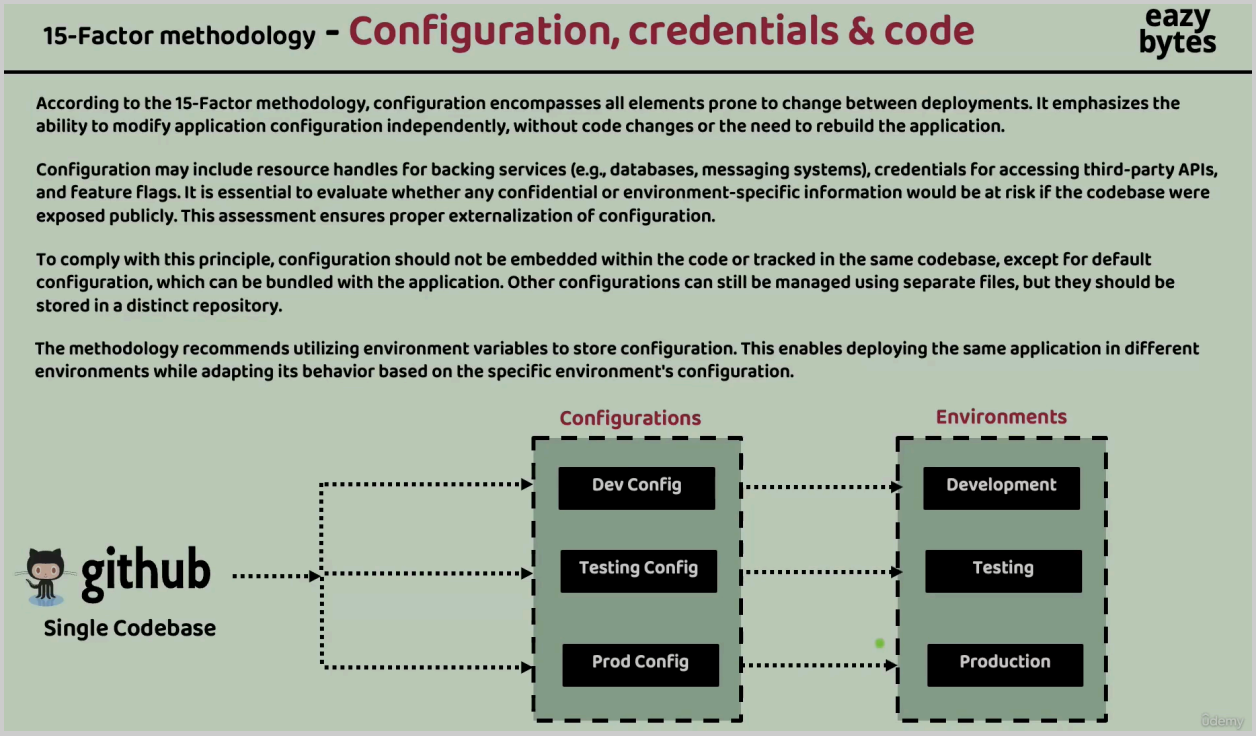
Configuration must be **externalized**, not hardcoded inside the application.

Examples of configs to externalize:

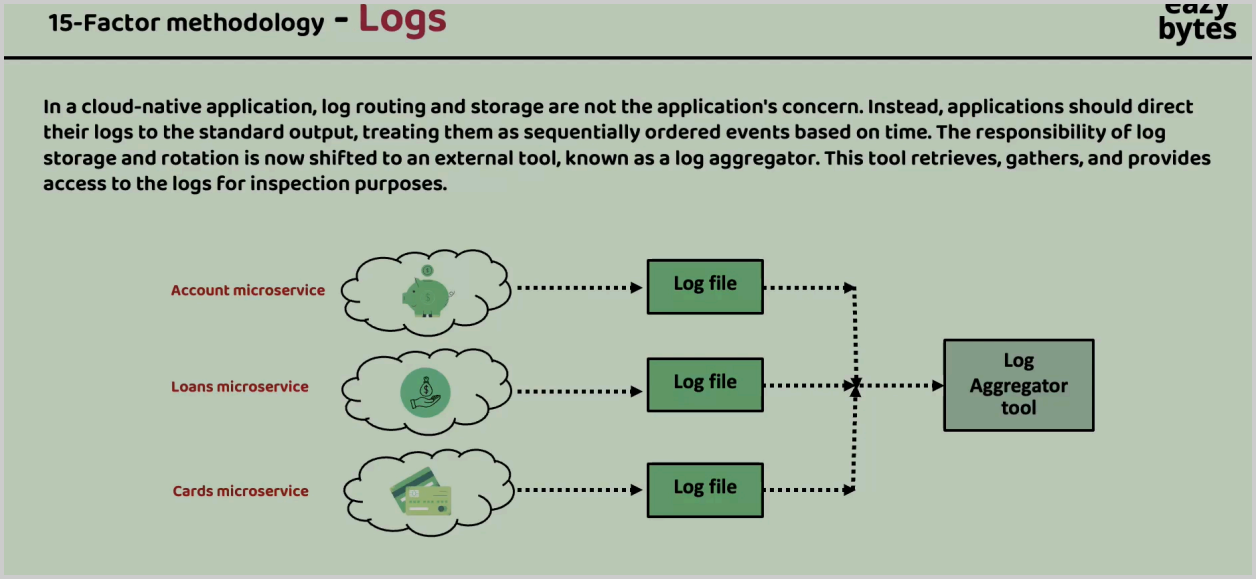
- Database URLs
- Credentials / API keys
- Environment variables
- Feature flags

Why?

- Same code runs in Dev/Test/Prod
- No rebuild needed for different environments
- More secure (no passwords inside code)



6 Logs – Treat Logs as Event Streams



7 Disposability – Fast Startup & Graceful Shutdown

Applications must be:

- Quick to start
- Safe to stop at any time
- Able to handle unexpected restarts

Why?
Cloud-native environments like Kubernetes constantly start/stop containers.

Practices:

- Use graceful shutdown hooks
- Keep startup lightweight
- Don’t store state inside app
- Ensure long-running operations can restart safely

8 Backing Services – Treat Services as Attached Resources

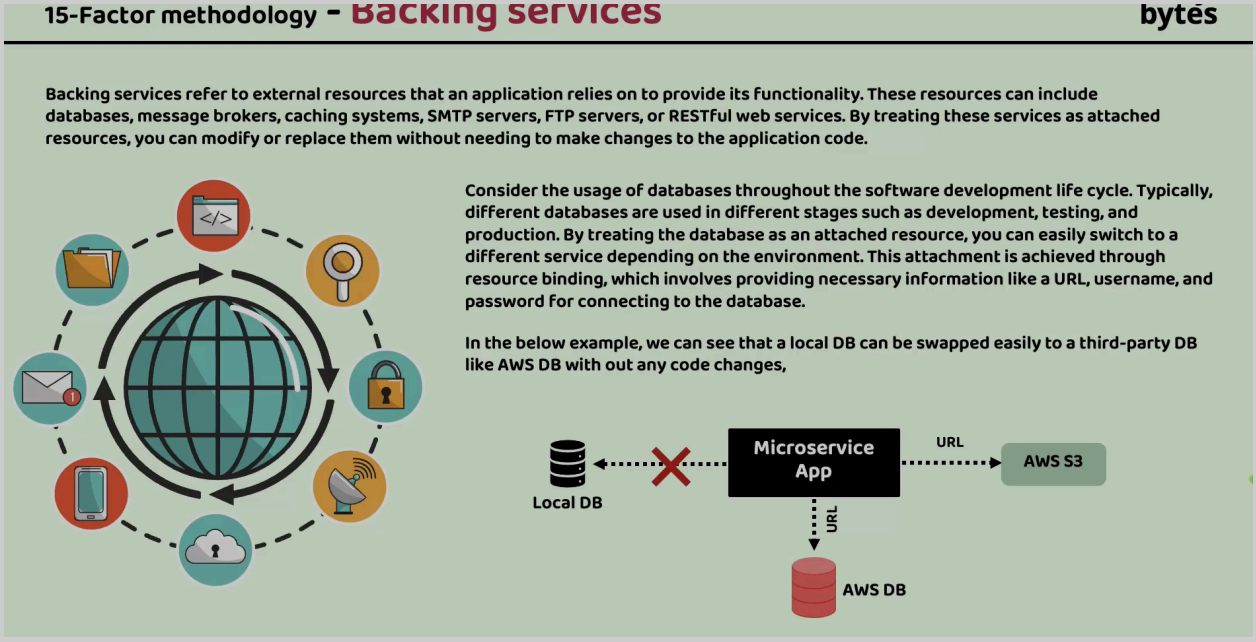
Resources like:

- DB
- Cache
- Message queue
- Storage
- Third-party APIs

must be considered **external services** that can be attached/detached without code changes.

Your app only changes **config**, never **code**, when switching between:

- Local DB → Cloud DB
- Test Redis → Production Redis



9 Environment Parity

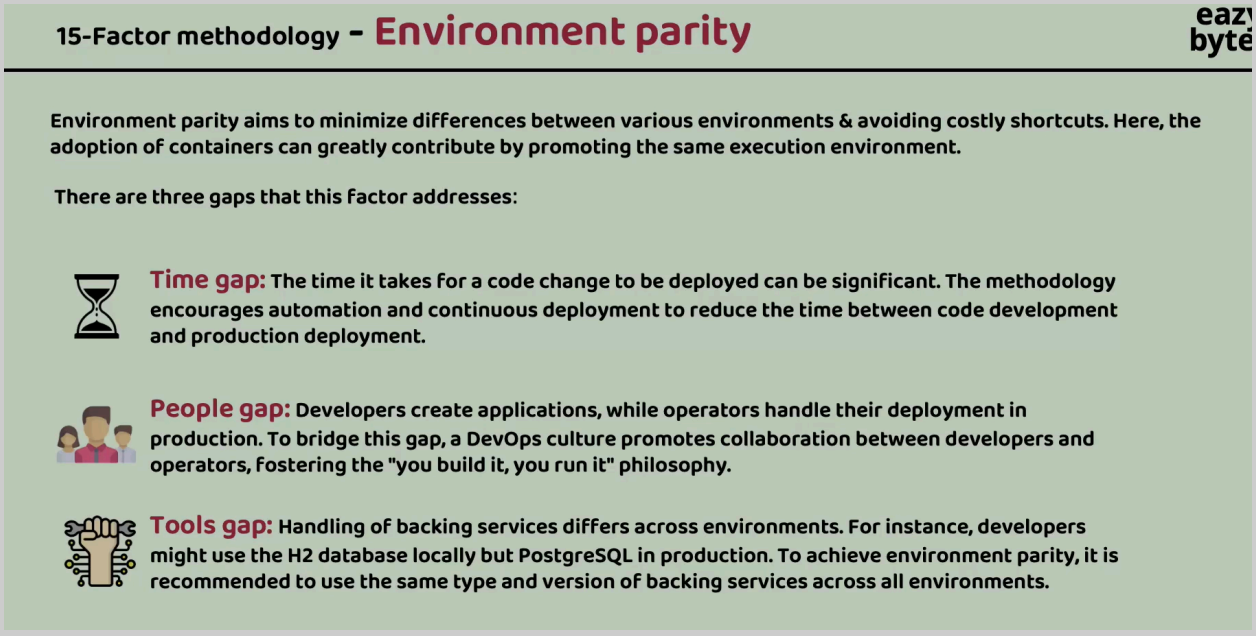
Keep Dev, Test, QA, and Prod environments **as similar as possible**.

Why?

- Fewer bugs caused by environment differences
- Easier debugging
- Predictable deployments

Achieved by:

- Containers
- IaC (Terraform)
- Same build artifacts used everywhere



🔟 Admin Processes

Administrative tasks must run as **one-off processes**, not part of the app.

Examples:

- Database migration scripts
- Cron jobs
- Data cleanup
- Debug commands

These should run:

- As standalone jobs
- Separate from the main app container

1️⃣1️⃣ Port Binding

Apps should **self-contain** their web server.

Instead of relying on external servers like:

- Apache
- Nginx
- Tomcat (traditional WAR)

Cloud-native apps:

- Bind to a port internally
- Expose it to the platform

Example:
Spring Boot app runs via embedded Tomcat on port 8080.

1️⃣2️⃣ Stateless Processes

Do **not store session or state inside the application instance**. State must be stored in:

- Database
- Redis
- Message queues
- Object storage

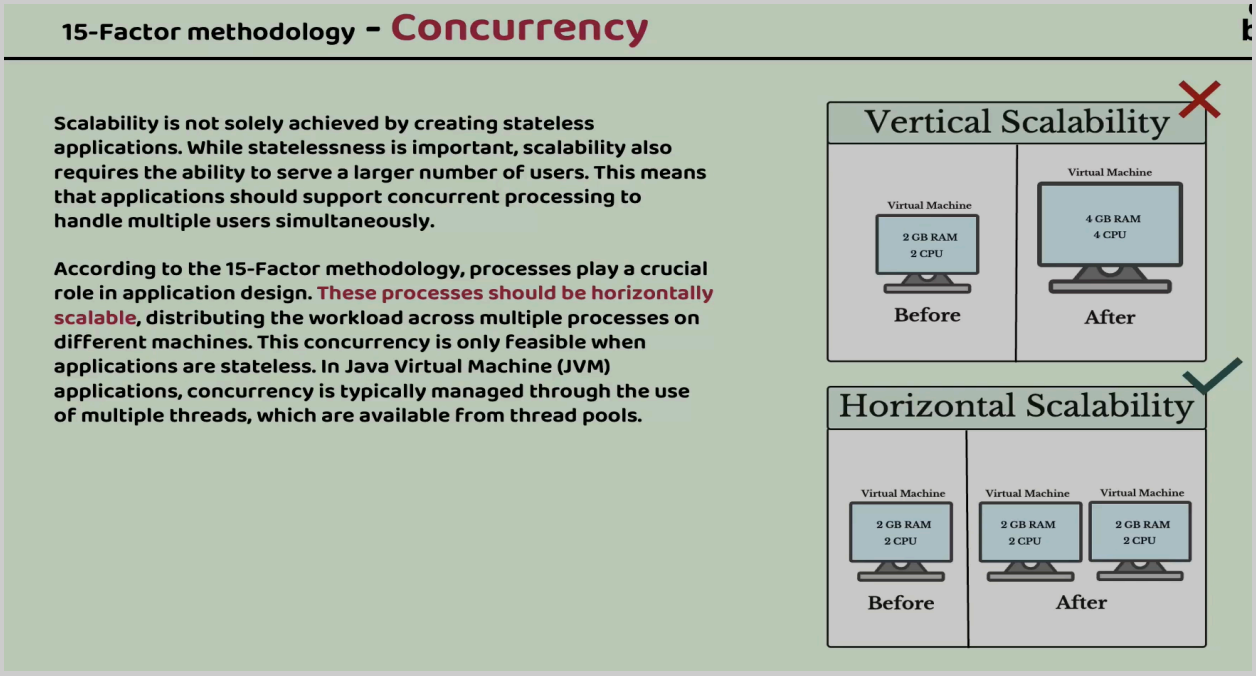
Reason:
Cloud-native platforms frequently kill and recreate containers. If your state is inside the container → it's lost.

1️⃣3️⃣ Concurrency – Scale Out via Processes

Scale by **running many small instances**, not by making one big server.

Cloud-native apps scale **horizontally**.

Example: 5 containers → 10 containers → 20 containers depending on load.



1️⃣4️⃣ Telemetry

Applications must expose:

- Metrics

- Traces
- Health checks

Why?
To allow monitoring tools like:

- Prometheus
- Grafana
- ELK
- Jaeger

to give system visibility.

Telemetry enables:

- Auto-scaling
- Alerting
- Diagnosis
- Performance insights

1️⃣5️⃣ Authentication & Authorization (Secure by Default)

Every cloud-native app must:

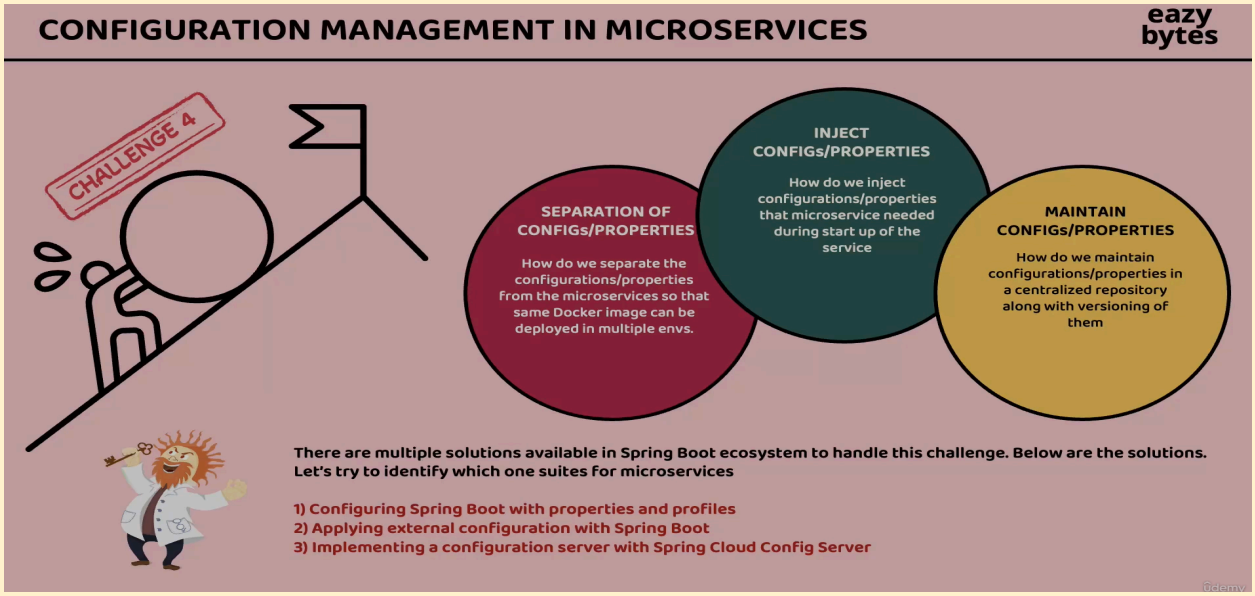
- Authenticate users
- Authorize access
- Secure data in transit and at rest

Common methods:

- OAuth2
- JWT
- OpenID

—----------

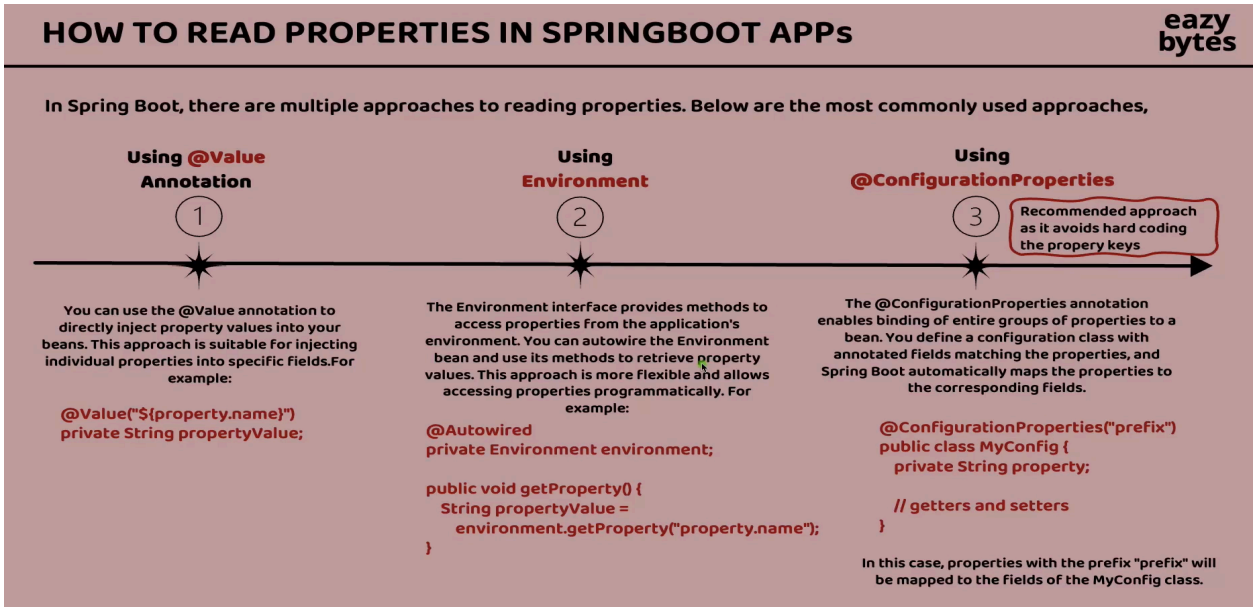
Configuration Management in Microservices



Microservices need **flexible, environment-specific configurations** so that the **same code** can run in multiple environments (dev, QA, stage, prod).

Spring Boot provides many ways to **externalize configuration**, from simple property files to advanced centralized configuration systems. Ex : **application.properties / application.yml, Spring Profiles ,Command-Line Arguments, Environment Variables,External Java Properties File** and **Spring Cloud Config Server.**

Read properties in spring boot:



Basic ways:

Using @value : which @Value → reads single values

Values which we need inject from application.propeties or yml

```yaml
build:
  version: "1.0"
```

```java
@Validated
public class AccountsController {

    private final IAccountsService iAccountsService;  5 usages
    public AccountsController(IAccountsService iAccountsService){  no usages
        this.iAccountsService =iAccountsService ;
    }
    @Value("${build.version}")  no usages
    private String buildVersion;
```

```java
@GetMapping("/build-version")  no usages
public ResponseEntity<String> getBuildVersion(){
        return ResponseEntity.status(HttpStatus.OK).body(buildVersion);
}
```

Code	Details
200	**Response body** 1

Not Suitable for Multiple or Grouped Properties

@Value is good for **one or two properties**, but when you have 5, 10, or many properties, you get:

@Value("${app.name}") String name;

@Value("${app.version}") String version;

@Value("${app.build}") String build;

@Value("${app.author}") String author;

Your class becomes messy and unmaintainable.

Using Environment Interface:

We use this instead of directly accessing values from application.properties or application.yml when needed. Also, some sensitive information like passwords should not be stored in these files because they can be exposed. In such cases, we can inject values through **environment variables** and read them using the Environment interface. It also reads single values.

```
@Autowired  no usages
private Environment environment;
```

I want to get the java/maven version from environment from local pc

```
@GetMapping("Environment_Interface")  no usages
public ResponseEntity<String> environmentInterface(){
    return ResponseEntity.status(HttpStatus.OK).body(environment.getProperty("MAVEN_HOME"));
}
}
```

We can see we got the location

```
C:\Program Files\Maven\apache-maven-3.6.3
```

Ex 2: for know java version

```
C:\Program Files\Java\jdk-17
```

Using @ConfigurationProperties Annotation:

We use this approach when we want to **cleanly manage multiple or grouped configuration values inside a single POJO class**, instead of using many @Value annotations. It is type-safe, structured, and ideal for microservices.

Step 1. Add Properties in application.yml

We should organize properties using a prefix, so Spring can map them to a Java class.

```yaml
accounts:
  message: "Welcome to EazyBank accounts related local APIs "
  contactDetails:
    name: "John Doe - Developer"
    email: "john@eazybank.com"
  onCallSupport:
    - (555) 555-1234
    - (555) 523-1345
```

Step 2. Create POJO Using property values

When using @ConfigurationProperties, we always define a **prefix** in the annotation, and the **same prefix must exist in application.yml**.

This tells Spring Boot *where exactly to read the values from*.

```java
import java.util.List;
import java.util.Map;

@ConfigurationProperties(prefix = "accounts")  no usages
public record AccountsContactInfoDto(String message, Map<String, String> contactDetails, List<String> onCallSupport) {
}
```

Note: If all values are set in application.yml and you don't plan to modify them programmatically, **setters are optional**. Only **getters** are sufficient for reading the values.

Here we used **Java record** class , instead of usual way of approach like creating getter and other

A **record** is a special kind of class in Java (introduced in Java 16) designed to be **immutable data carriers**.

Key Points

1. **Immutable** by default → fields are final

2. **No setters** → you cannot change values after creation
3. **Automatic getters** → called **accessor methods**, with the **same name as the field**
4. **Auto-generated**: equals(), hashCode(), toString()
5. Perfect for **DTOs or POJOs** that just carry data

Step 3: Enable the properties using @EnableConfigurationProperties in the main class.

You must tell Spring to load this class. we must enable it in the main class using @EnableConfigurationProperties(pojo/dto.class) so that Spring can load and bind the properties from application.yml.

```
@EnableConfigurationProperties(AccountsContactInfoDto.class)
public class AccountsApplication {

    public static void main(String[] args) { SpringApplication.run(AccountsApplication.class, args); }

}
```

Step 4. Use It in Controller or Service

```
@Autowired  no usages
private AccountsContactInfoDto accountsContactInfoDto;
```

```
@GetMapping("account_details")  no usages
public ResponseEntity<AccountsContactInfoDto> getAccountDetails() {
    return ResponseEntity.status(HttpStatus.OK).body(accountsContactInfoDto);
}
```

200

Response body


```
{
  "message": "Welcome to EazyBank accounts related local APIs ",
  "contactDetails": {
    "name": "John Doe - Developer",
    "email": "john@eazybank.com"
  },
  "onCallSupport": [
    "(555) 555-1234",
    "(555) 523-1345"
  ]
}
```


Ex2:

```yaml
app:
  name: Accounts Service
  version: 1.0.0
  build: 2025
  author: Ram
```

```java
@ConfigurationProperties(prefix = "app")  no usages
@Data
public class AppConfigProperties {
    private String name;  no usages
    private String version;  no usages
    private String build;  no usages
    private String author;  no usages

}
```

```java
@EnableConfigurationProperties(AppConfigProperties.class)
public class AccountsApplication {

    public static void main(String[] args) { SpringApplication.run(AccountsApplication.class, args); }

}
```

```java
    @GetMapping("account_details")  no usages
    public ResponseEntity<AppConfigProperties> getAppDetails() {
        return ResponseEntity.status(HttpStatus.OK).body(appConfigProperties);
    }
}
```

200

Response body


```
{
  "name": "Accounts Service",
  "version": "1.0.0",
  "build": "2025",
  "author": "Ram"
}
```


When you have different environments (e.g., **dev, test, prod**) and want different configuration values for each, you **cannot rely on a single application.yml**.

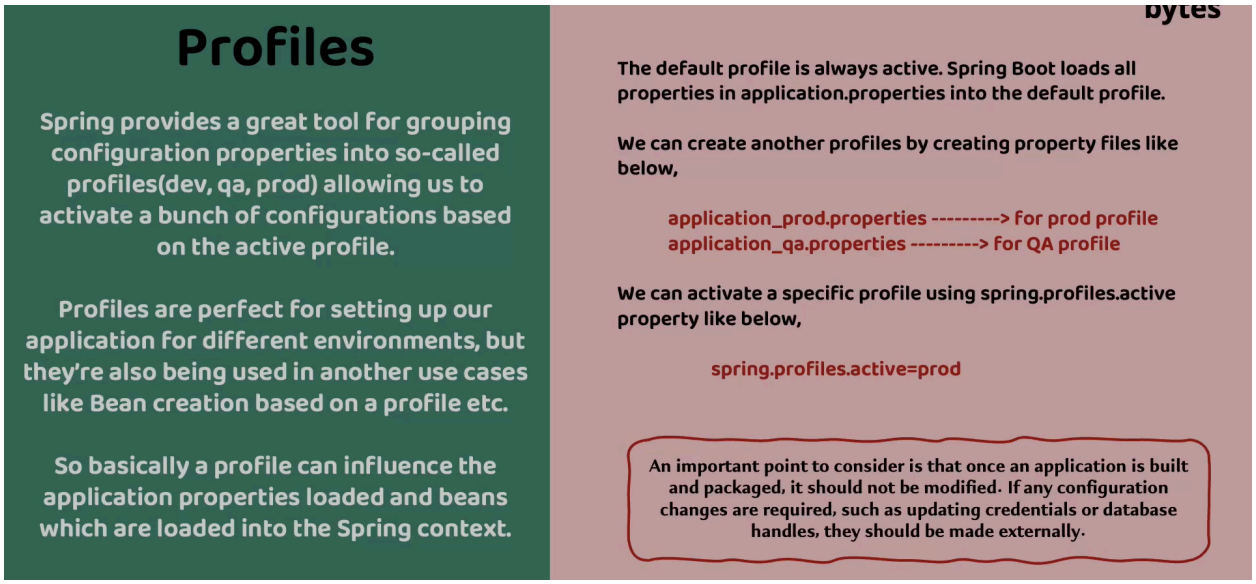
this is where **Spring Profiles** come into play

Spring Profiles

"A Spring Profile is a **way to define and configure different configuration values for different environments, such as dev, test, staging, and prod**.

Only the configuration for the active **profile is loaded at runtime**."

Each environment can have its own application.properties or application.yml file (e.g., application-dev.yml, application-prod.yml) containing environment-specific values."



Example

We have microservice with different configurations for different environments.

In a microservice, each environment (dev, qa, prod) can have its own configuration. Spring Boot automatically loads profile-specific configuration files **only when they follow the correct naming pattern**, such as application-qa.yml, application-prod.yml, or application-dev.yml.

It **does not** load files that use underscores, such as application_qa.yml.

application.yml

```yaml
build:
  version: "3.0"
app:
  name: Accounts Dev Service
  version: 1.0.0
  build: 2025
  author: RamDev
```

application_qa

```yaml
build:
  version: "2.0"
app:
  name: Accounts QA Service
  version: 1.0.0
  build: 2025
  author: RamQA
```

Application_prod

```yaml
build:
  version: "1.0"
app:
  name: Accounts Prod Service
  version: 1.0.0
  build: 2025
  author: RamProd
```

Inside each profile-specific file, you use:

spring.config.activate.on-profile: qa/prod/stg

This tells Spring Boot: **“Load this file only when the QA profile is active.”**

spring.config.activate.on-profile

```yaml
17  config:
18    activate:
19      on-profile: "prod"
```

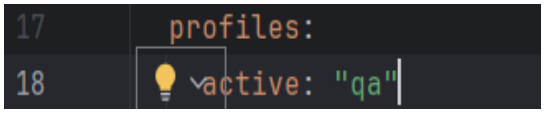
```yaml
17  config:
18    activate:
19      on-profile: "qa"
```

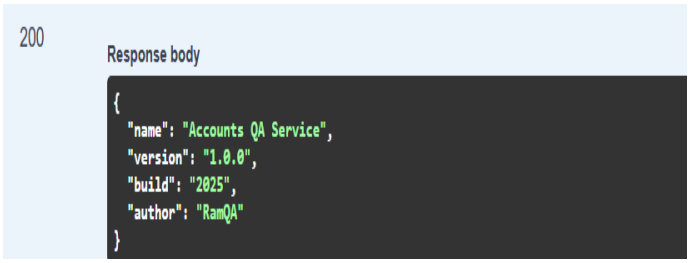
Then we should activate the environment (**spring.profiles.active=dev/prod/qa**) in the default file application.properties/application.yml file.

Spring always loads the default file application.properties/application.yml FIRST, then profile-specific file. The profile file overrides matching values from the default file.

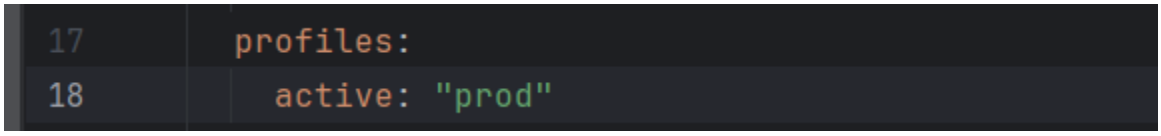
Because of this built-in mechanism, **you do NOT need to manually import or load profile files**

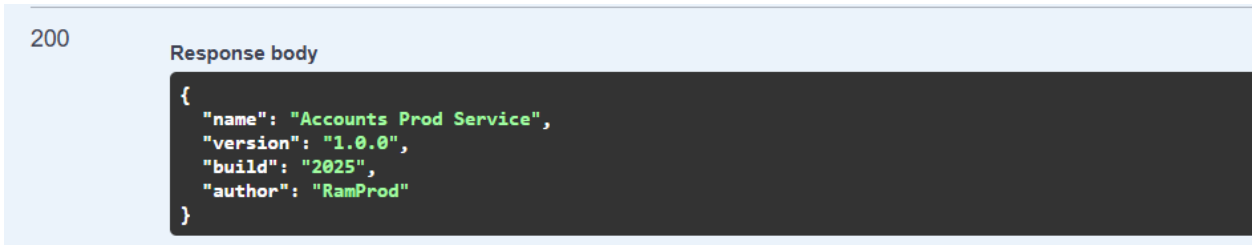
Then it loads the file that matches the active profile:





Or





However, this approach is **not immutable**, because every time we need to run the application in a specific environment, we must manually change spring.profiles.active in the default file (application.yml) and then rebuild or regenerate Docker images.

To avoid this problem, we use dynamic configuration methods that allow us to set the environment **at startup**, without modifying the code or rebuilding the Docker image.

Spring Boot allows overriding properties at startup using:

1. **Command-line arguments**

2. **JVM system properties**
3. **Environment variables**

These methods give you **immutable infrastructure** — the same JAR or Docker image runs in all environments, and only configuration changes.

Externalize the configuration using Command-line arguments

Command-line arguments (highest precedence)

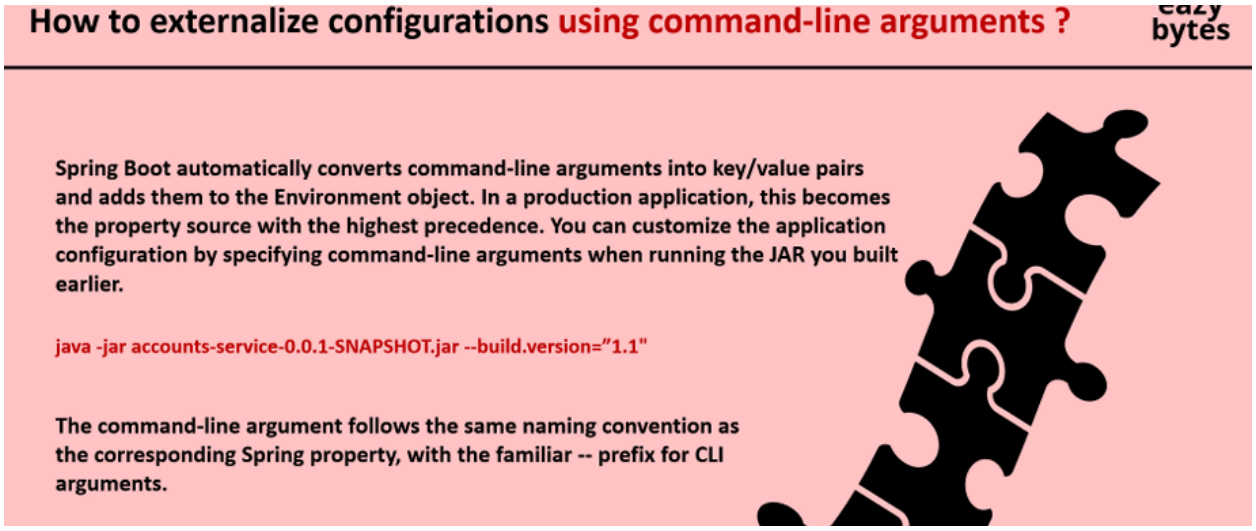
java -jar app.jar --key=value

Notes

- Use the same property name as in Spring configuration.
- CLI arguments (--key=value) override:
    - profile-specific files
    - JVM system properties
    - application.yml

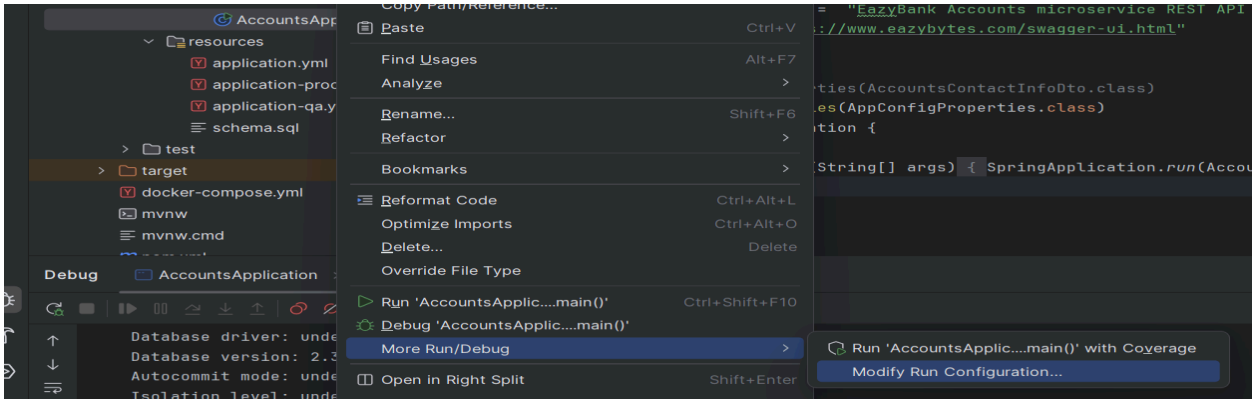
Note: Each property starts with --, Arguments are separated by spaces

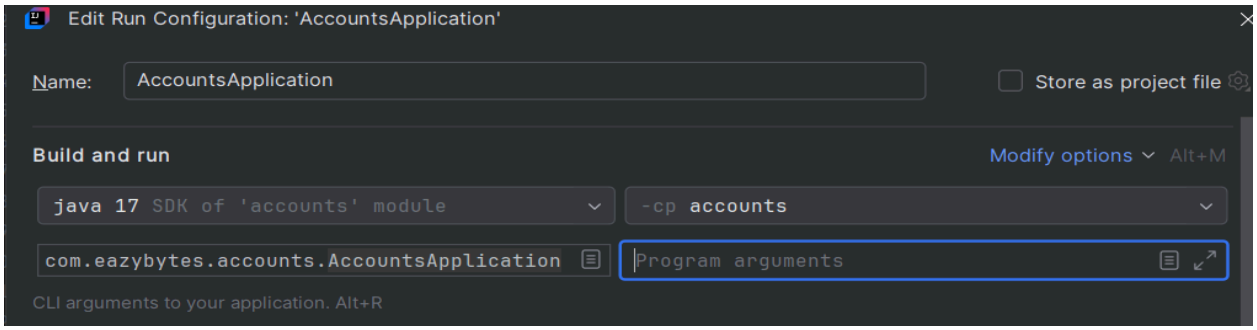
java -jar app.jar --spring.profiles.active=qa --server.port=8085



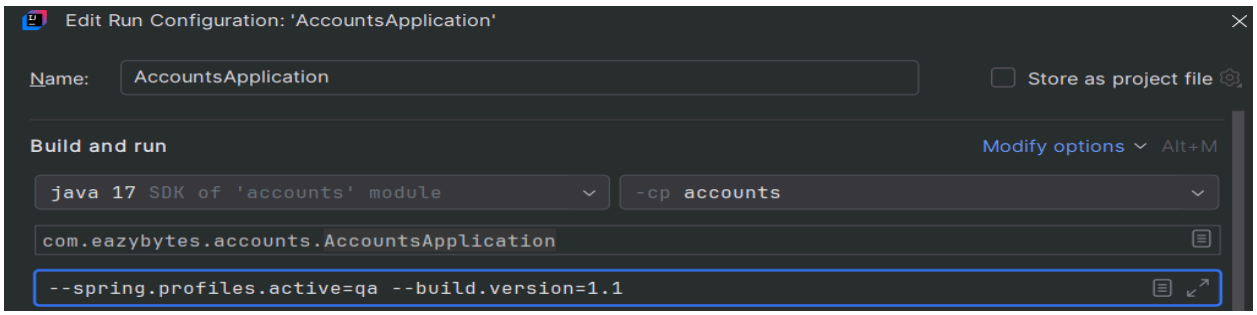
We can **also provide command-line arguments directly in the IDE**'s run configuration, **apart from just from the command line.**"

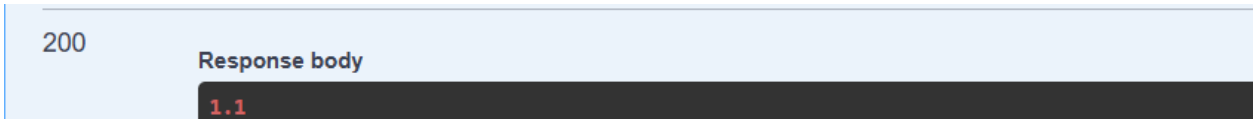
Go to the main class, right-click on it, and select *Modify Run Configuration*."

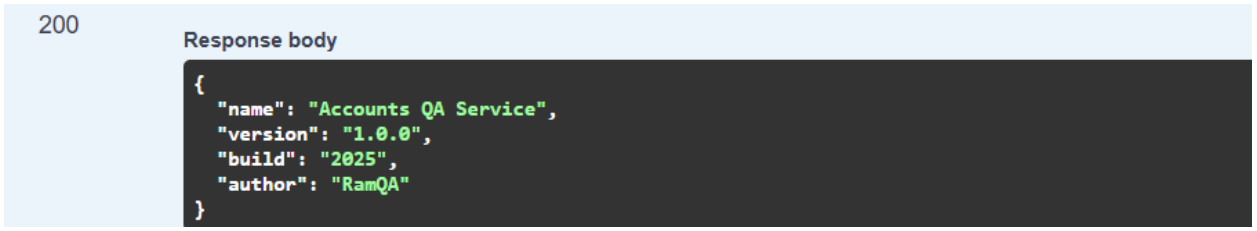




Provide in program arguments , I just providing two Arguments are separated by spaces







Externalize the configuration using JVM System properties

JVM system properties

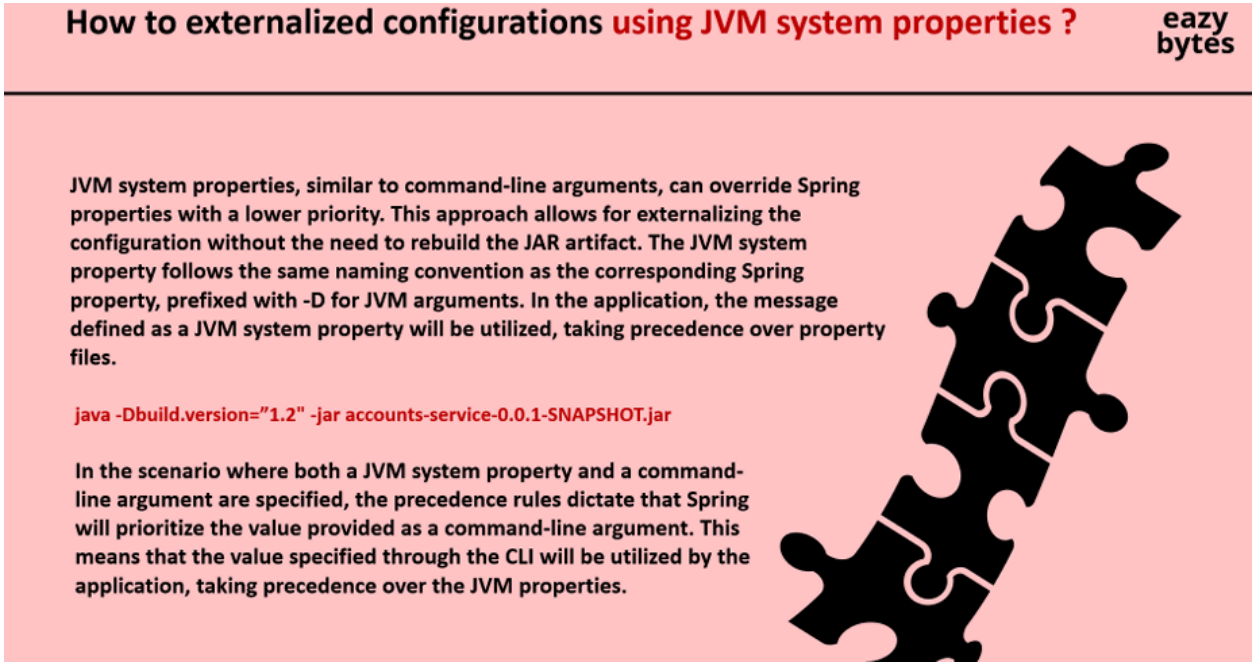
java -Dkey=value -jar app.jar

Notes

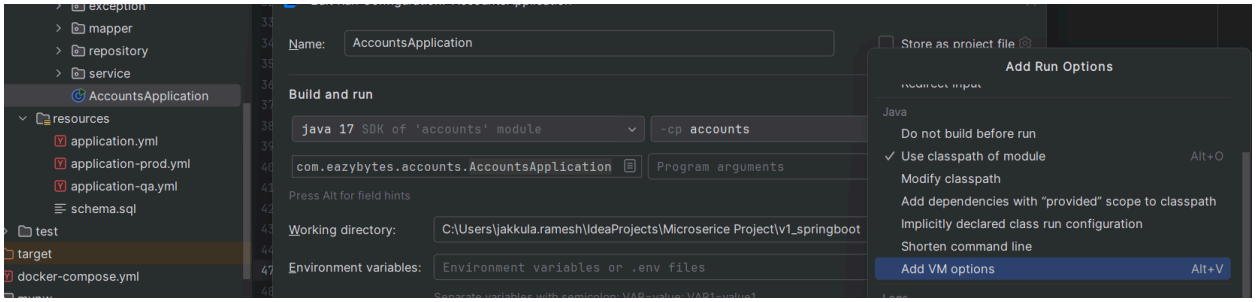
- System properties use the -D prefix.
- Follow the same naming as Spring keys.
- If both a system property **and** a command-line argument are provided, the command-line argument wins.

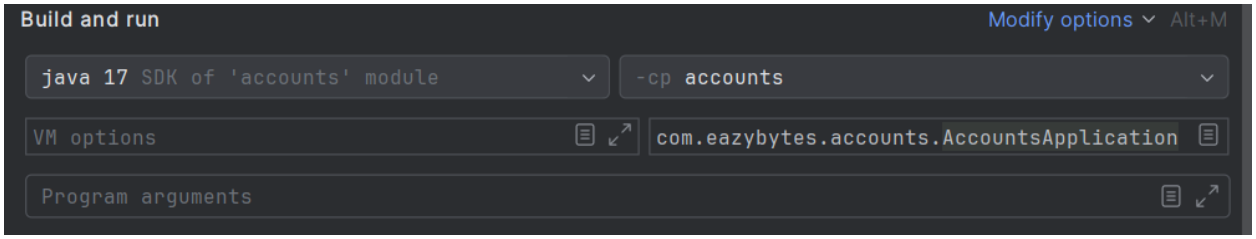
Note: Each property starts with -D, and they are separated by **spaces**

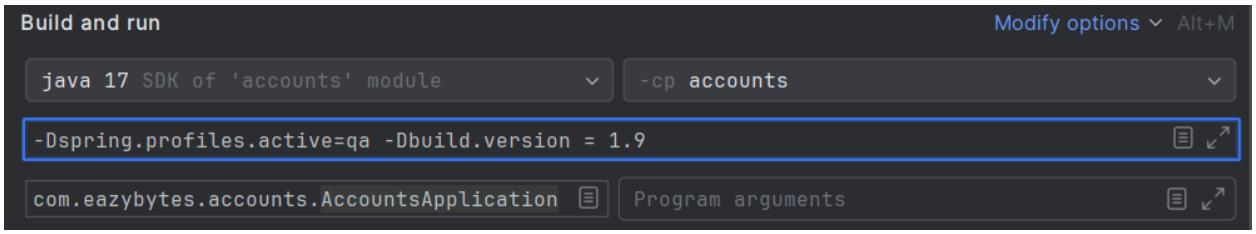
java -Dspring.profiles.active=prod -Dserver.port=9090 -jar app.jar

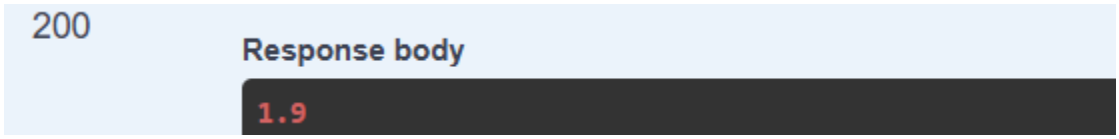


We can **also provide JVM System variables directly in the IDE**'s run configuration. Click on modify option , will get VM option.









Externalize the configuration using environment variables

Environment variables

KEY=value java -jar app.jar

- **Example (Windows CMD):**

```
cmd

set SERVER_PORT=7070
set SPRING_PROFILES_ACTIVE=dev
java -jar app.jar
```

✔ They work **only for that CMD window**.

✔ No need to add them in Windows System Environment Variables.

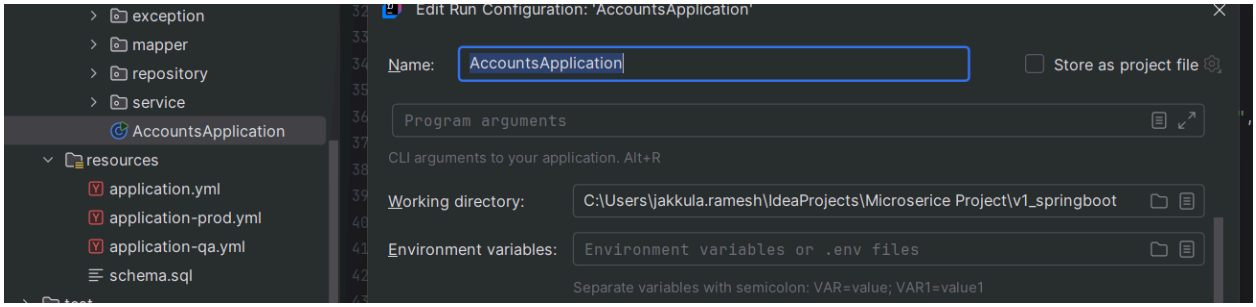
✘ Once you close CMD, those values disappear.

To map an environment variable to a Spring property:

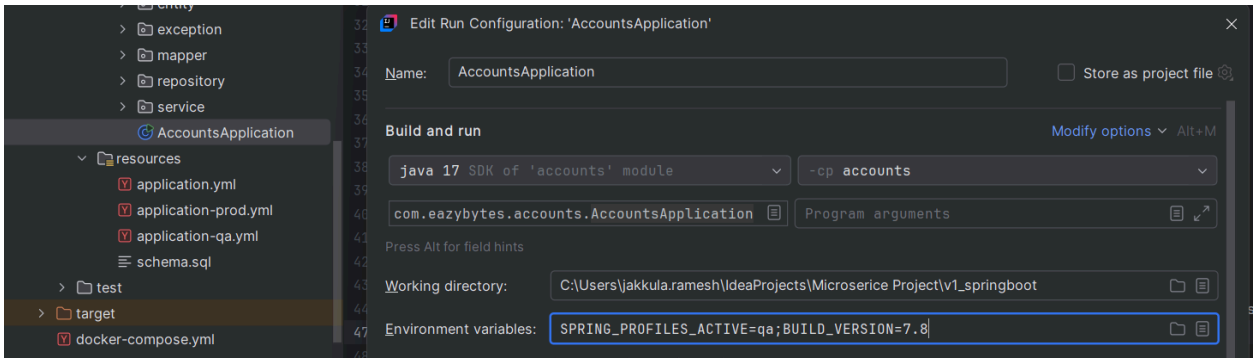
- Convert all letters to **uppercase**
- Replace dots (.) or dashes (-) with **underscores** (_)

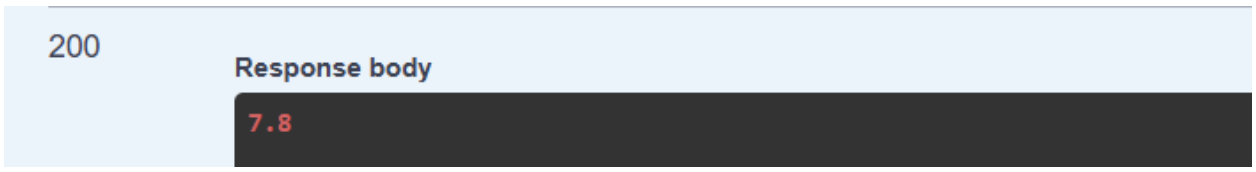
Example: BUILD_VERSION → recognized as build.version

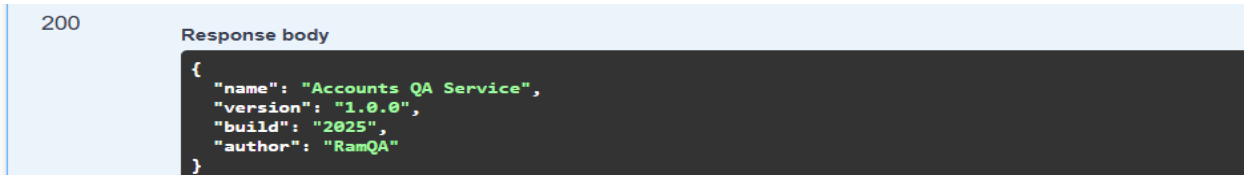
We can **also provide environment variables directly in the IDE**'s run configuration, **apart from just from the environment variables.**"

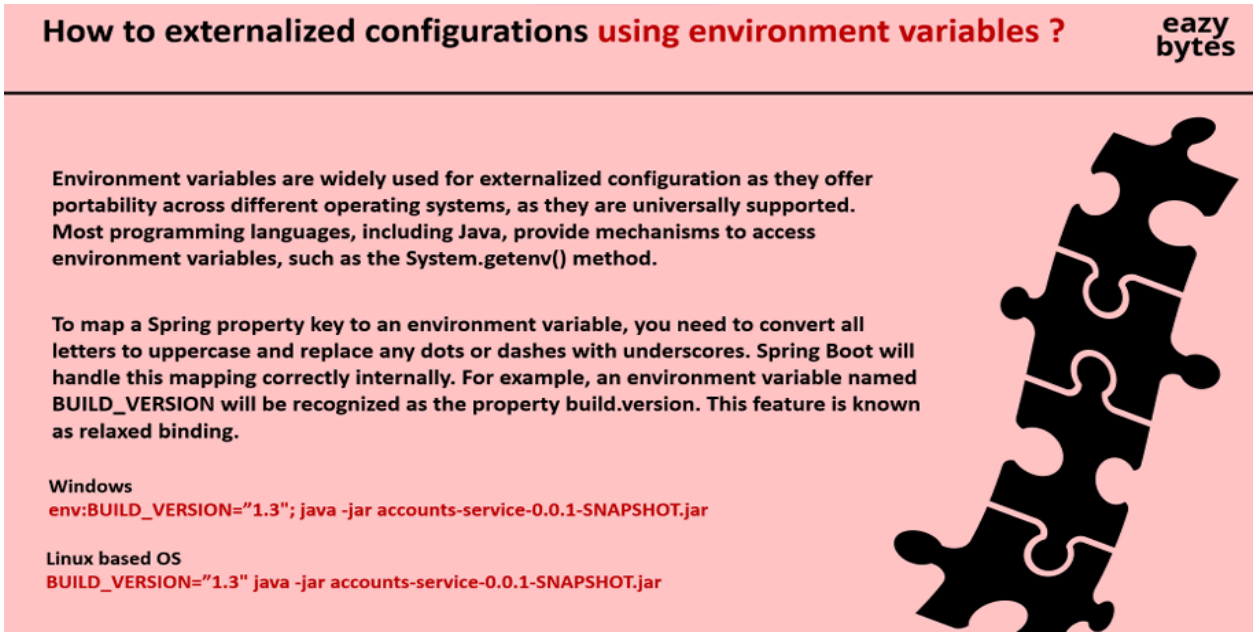


They are not separated by ; .







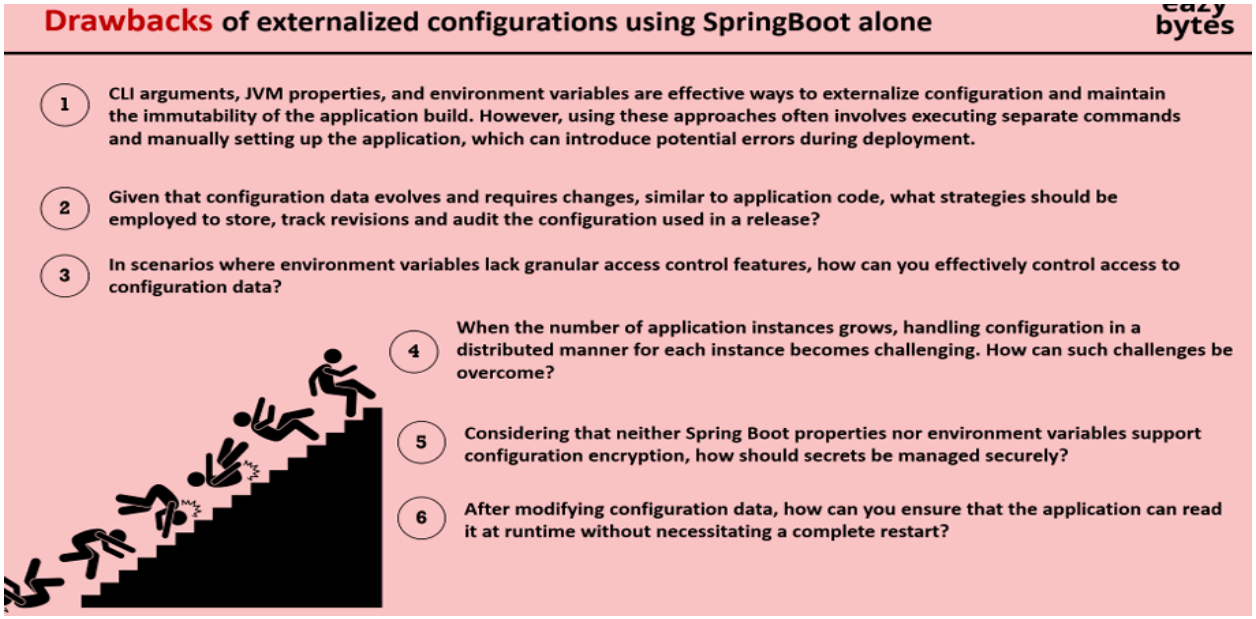


Most-Useful Priority

Priority	Source	Example	Notes
1 (Highest)	Command-line arguments	`--server.port=9090`	Overrides everything
2	System Properties	`-Dserver.port=9090`	Next highest
3	Environment Variables	`SERVER_PORT=9090`	Works for OS-level
4	Profile-specific files	`application-dev.properties`	Only when profile is active

While Spring Boot lets us override configuration at startup using command-line arguments, JVM properties, or environment variables, it still falls short for real microservice environments. There is **no centralized configuration**, **no version control**, **no secret encryption**, and **no runtime refresh**. Managing configs across many microservices becomes difficult.

This is why **Spring Cloud Config** is used— it provides **centralized, versioned, secure, and refreshable configuration** for all environments, enabling truly immutable deployments.

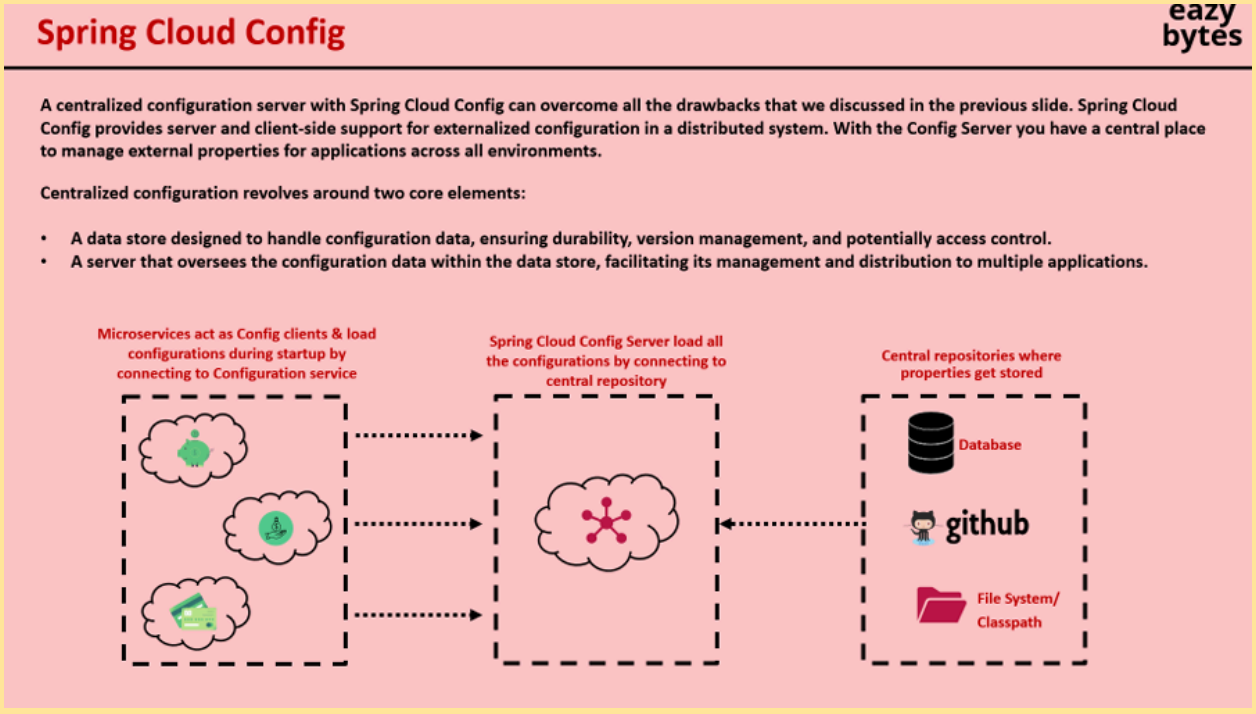


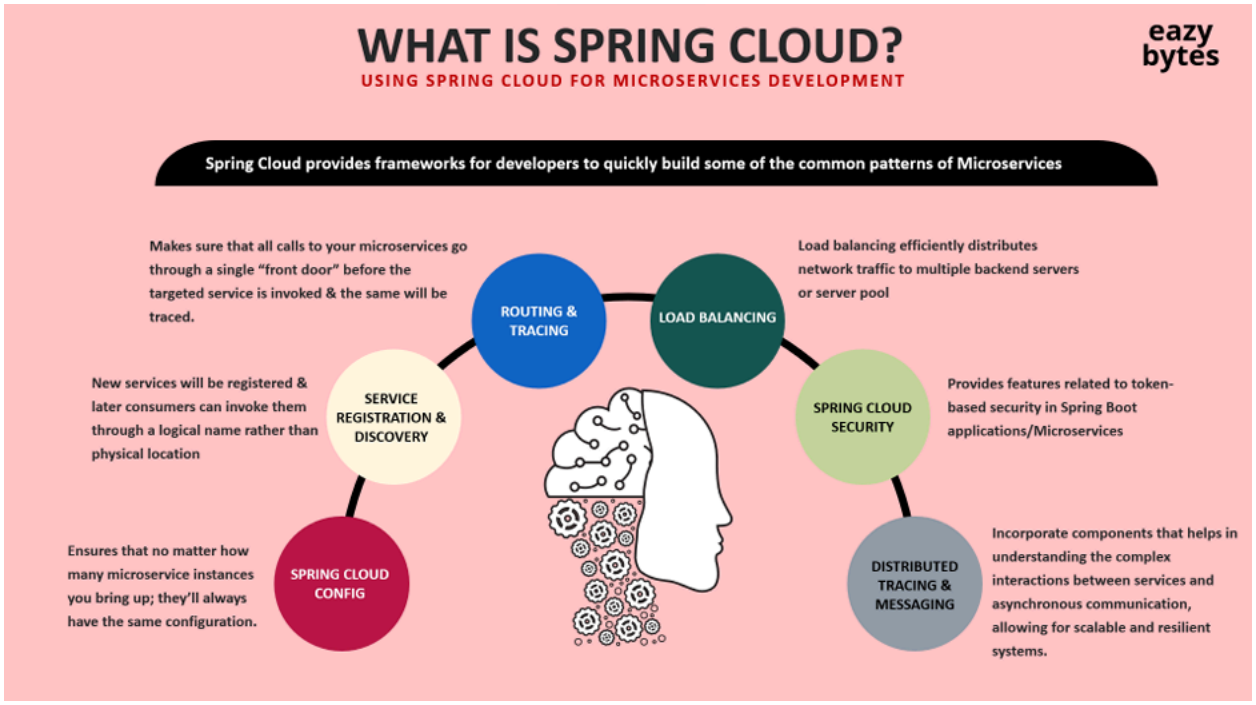
Spring Cloud Config

Spring Cloud Config is a **centralized configuration** management solution for applications, especially **designed for microservices architectures**. It allows you to **store, manage, and provide configuration properties for multiple applications and environments(dev/prod/stg/qa) from a central place**.
https://spring.io/projects/spring-cloud-config

Key Points:

- **Centralized storage** – all configs are in one place (Git repository ,Filesystem,Database,Classpath,Cloud storage or other).
- **Environment-specific** – supports different settings for dev, staging, qa, prod, etc.
- **Version-controlled** – you can track changes and roll back if needed.
- **Secure and refreshable** – secrets can be encrypted, and services can refresh configs at runtime without restarting.
- **Works for many microservices** – multiple microservices can fetch their configuration from the same central Config Server, ensuring consistency and ease of management.





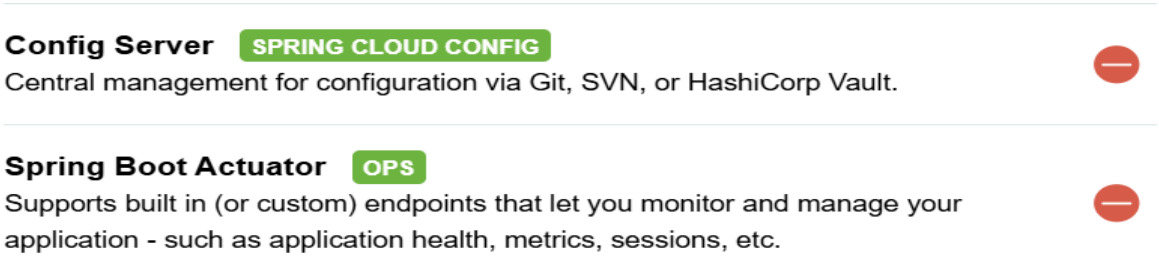
Spring Boot and Spring Cloud follow different release cycles, so their versions do **not** match automatically. Always use a **compatible combination** based on the official compatibility matrix, otherwise the application will show errors like:

Spring Boot [3.x.x] is not compatible with this Spring Cloud release train

More info : https://spring.io/projects/spring-cloud-config#support

Spring Cloud Config Server Setup

1. **Create a Spring Boot Project**
    - Use Spring Initializr or your IDE.
    - Add dependencies:
        - spring-cloud-config-server
        - spring-boot-starter-actuator



2. **Enable Config Server**
    - Annotate the main class with @EnableConfigServer.

```
@SpringBootApplication
@EnableConfigServer
public class SpringcloudconfigserverApplication {

    public static void main(String[] args) { SpringApplication.run(SpringcloudconfigserverApplication.class, a
```

3. **Configure Server Properties in config server**
    - In application.yml or application.properties:
        - Set the server port (e.g., 8888).

```
server:
  port: "8071"
spring:
  application:
    name: "springcloudconfigserver"
```

Depending on the requirement we can add more actuator things and others.

Point to your configuration repository (Git, filesystem, etc.):

Ex 1: Reading Configurations from the Classpath location of spring cloud config

- Create folder (ex: config) a inside Config Server project → src/main/resources

- **Add all .yml / .properties files of your microservices** (e.g., accounts, orders, payment, etc.) **inside the config folder** of the Spring Cloud Config Server.

  The name of the .yml / .properties files in the Config Server's classpath must match the spring.application.name of the microservice.
  So, if your microservice has:

  spring.application.name: accounts

  Ex: accounts-qa.yml

  accounts-prod.yml

  Only keep the configuration details that you want to externalize (i.e., values that will change across environments like **dev, test, prod**).

- Enable native profile in **Config Server**

  spring.profiles.active=native

  This tells the Config Server to read configuration **from the local file system or classpath** instead of Git.

  Tell Config Server to read from classpath

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

  This specifies the **exact folder inside the classpath** (src/main/resources/config) where all configuration files are stored.

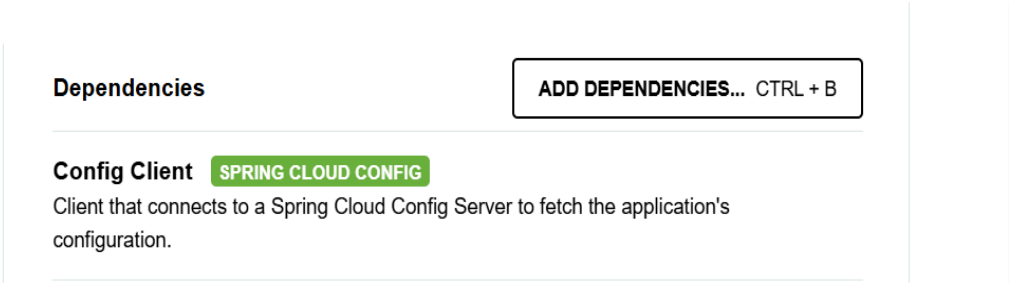
```yaml
server:
  port: "8071"
spring:
  application:
    name: "springcloudconfigserver"
  profiles:
    active: native
  cloud:
    config:
      server:
        native:
          search-locations: "classpath:/config"

```

- **Start Config Server**

  When the Config Server starts, it **automatically reads all configuration files from the classpath /config folder** and makes them available to client microservices.

- **Client microservice setup**

  Add Config Client dependency + set Config Server URL inside microservices.




```yaml
spring:
  application:
    name: "accounts"
  profiles:
    active: "prod"
  config:
    import: "optional:configserver:http://localhost:8071/"
```

spring.config.import=optional:configserver:http://localhost:8071

This tells the microservice to **fetch its configuration from the Config Server** running at port **8071** during application startup.

optional: means do not fail the application startup if the Config Server is not reachable or not running.

spring.profiles.active=prod

Your microservice will **activate only the prod profile** and therefore it will **fetch only the configuration files related to the prod environment** from the Config Server.

- **Start microservices and test**

```
restartedMain] c.e.accounts.AccountsApplication         : The following 1 profile is active: "prod"
restartedMain] o.s.c.c.c.ConfigServerConfigDataLoader   : Fetching config from server at : http://localhost:8071/
restartedMain] o.s.c.c.c.ConfigServerConfigDataLoader   : Located environment: name=accounts, profiles=[default], label=null, version=null, state=null
restartedMain] o.s.c.c.c.ConfigServerConfigDataLoader   : Fetching config from server at : http://localhost:8071/
restartedMain] o.s.c.c.c.ConfigServerConfigDataLoader   : Located environment: name=accounts, profiles=[prod], label=null, version=null, state=null
```

We can see output which uses configuration details of the selected environment (ex: prod).

200

Response body

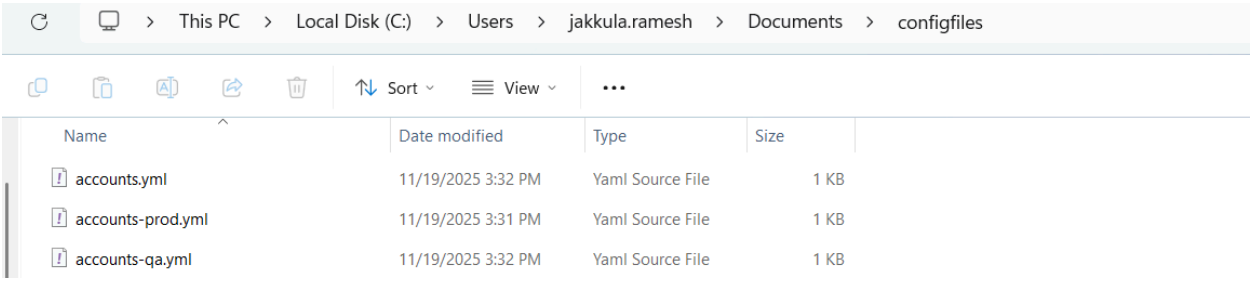
```
{
  "name": "Accounts Prod Service",
  "version": "1.0.0",
  "build": "2025",
  "author": "RamProd"
}
```

The classpath approach, where config files are kept inside src/main/resources/config, is suitable only for small or demo projects because **it is not secure—anyone who has access to the Config Server's codebase can view all configuration details**. It also **requires rebuilding and redeploying the Config Server whenever configuration changes are needed**.

To overcome these limitations, we use a file system location, which keeps configuration external, more secure, and fully separated from the Config Server application.

Ex 2: Reading Configurations from the System location

- Create a folder on your local machine to store all configuration files for different environments (dev, qa, prod, etc.). Place the .yml or .properties files for each microservice inside this folder.

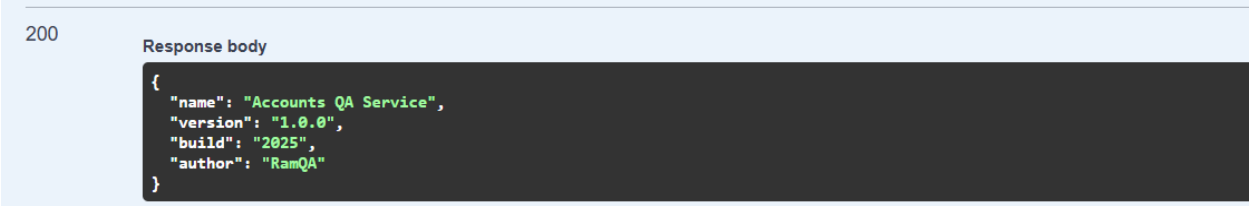


- Configure Spring Cloud Config Server to point to this system folder path using: spring.cloud.config.server.native.search-locations=file:///path/to/config-folder/

springcloudconfigserver\...\application.yml

```yaml
server:
  port: "8071"
spring:
  application:
    name: "springcloudconfigserver"
  profiles:
    active: native
  cloud:
    config:
      server:
        native:
          search-locations: "file:///C://Users//jakkula.ramesh//Documents//configfiles"
```

- The Config Server will read all configuration files directly from this external system location. This approach keeps configuration external, secure, and independent from the Config Server application.



The other two methods — **classpath** and **local file system** — are not ideal for real projects because they lack **tracking, security, reliability**, and require manual maintenance.

Using a **Git repository** is the most recommended way for Spring Cloud Config because it provides proper **version control, security, centralized storage, collaboration, auditing**, and **environment-wise configuration management**.

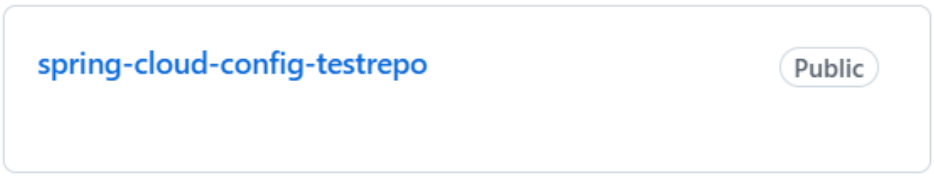
Ex 3: Reading Configurations from the GitHub Repository

GitHub (or any Git) repository is the **most recommended and industry-standard way** for Spring Cloud Config.

Using a **Git repository** is the most recommended way for Spring Cloud Config because it provides proper **version control, security, centralized storage, collaboration, auditing**, and **environment-wise configuration management**.

- Create a new GitHub repository

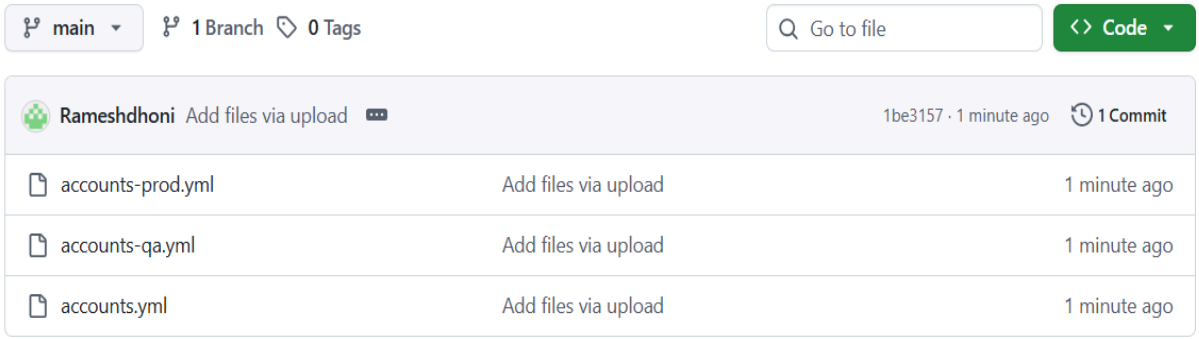
  Example: **spring-cloud-config-repo**



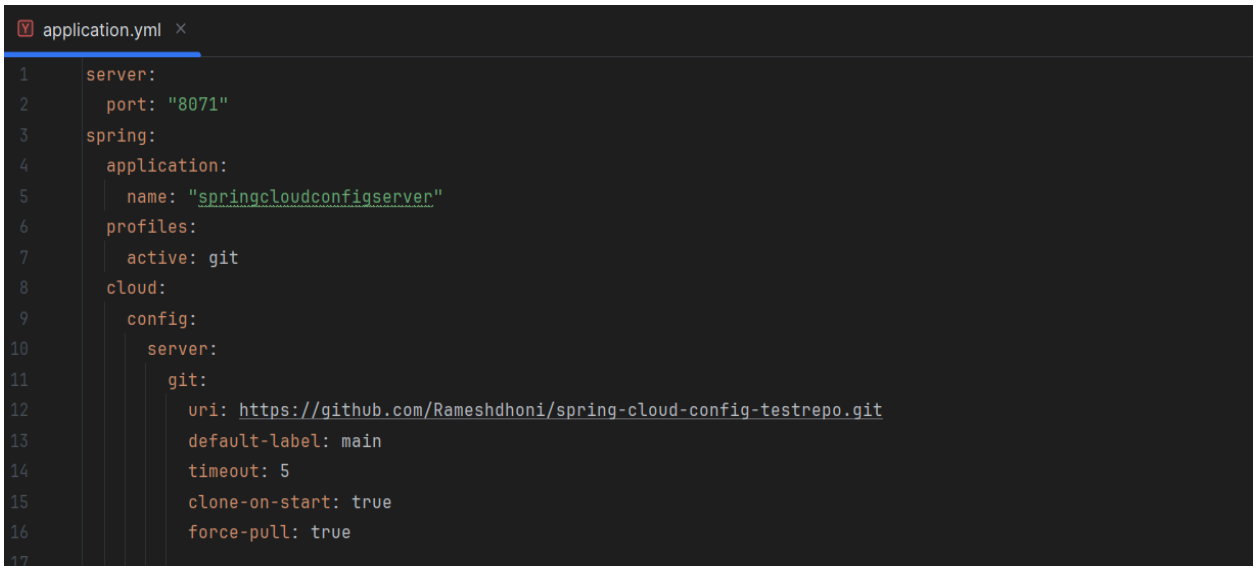
- Add your configuration files into the repo

  You can add files in **either of these ways**:

  - Upload directly using GitHub UI
  - Add files locally → commit → push to GitHub



- Configure Spring Cloud Config Server to use that GitHub repo



i) spring.profiles.active=git

This activates the **git profile** in the Config Server.
It tells Spring Cloud Config Server to **read configuration from a Git repository** instead of native filesystem.

```yaml
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/xxxx/your-repo.git
          default-label: main
          timeout: 5
          clone-on-start: true
          force-pull: true
```

Or

```properties
spring.cloud.config.server.git.uri=https://github.com/xxxx/your-repo.git
spring.cloud.config.server.git.default-label=main
spring.cloud.config.server.git.timeout=5
spring.cloud.config.server.git.clone-on-start=true
spring.cloud.config.server.git.force-pull=true
```

uri**:** This is the Git repository URL where all configuration files are stored.

default-label: Specifies the **default branch** in Git. Most modern Git repos use **main**, but you can use **master**, **dev**, etc. If the client does not request a specific branch, Config Server uses this branch.

timeout: Sets a **network timeout (in seconds)** for connecting to the Git repository.

clone-on-start: Tells Config Server to **clone the Git repository immediately when the server starts**.

force-pull: Forces the Config Server to **pull the latest changes** from Git even if local files were modified. Ensures your Config Server always serves the **latest configuration** from GitHub.

- Start the Config Server and microservices

200

Response body


```
{
  "name": "Accounts QA Service",
  "version": "1.0.0",
  "build": "2025",
  "author": "RamQA"
}
```


For more info official doc:
https://docs.spring.io/spring-cloud-config/reference/server/environment-repository.html

We can **encrypt and decrypt sensitive configuration values** such as:

- passwords
- API keys
- database credentials
- access tokens
  secrets

Spring Cloud Config provides **built-in encryption and decryption** support using the **Spring Cloud Config Server + Spring Security Crypto library**.

1. Config Server must be configured with a symmetric or asymmetric key

You can use:

- **Symmetric key** (shared secret)
- **Asymmetric key** (RSA key pair – more secure)

Example (symmetric key in Config Server):

```
encrypt:
  key: mysecretkey123
```

2. Encrypt values using Config Server endpoints

You can send plain values to:

POST http://localhost:8888/encrypt

It returns encrypted text like:

AQBsd908asda87asdasd...

3. Store encrypted values in GitHub repo

In your Git repo config file:

```
db:
  password: "{cipher}AQBsd908asda87asdasd..."
```

The prefix **{cipher}** tells Config Server that this value is encrypted.

4. Microservices receive decrypted values automatically

Clients will get the **plain decrypted value**, not the encrypted one.

So the microservice will see:

```
db.password = realpassword123
```

even though the Git repo stores:

```
{cipher}EncryptedStringHere
```

Refreshing the configuration at runtime

➤ Without any refresh mechanism

Whenever you change configuration in the **Config Server** (ex: Git-backed), normally:

- You must **restart the Config Server** to pull latest Git changes
- You must **restart each microservice** to re-read the configuration

This is the default behavior, and it's **not suitable for production**.

Way 1: Manual Refresh using /actuator/refresh

- Add **Actuator dependency** in **each microservice**

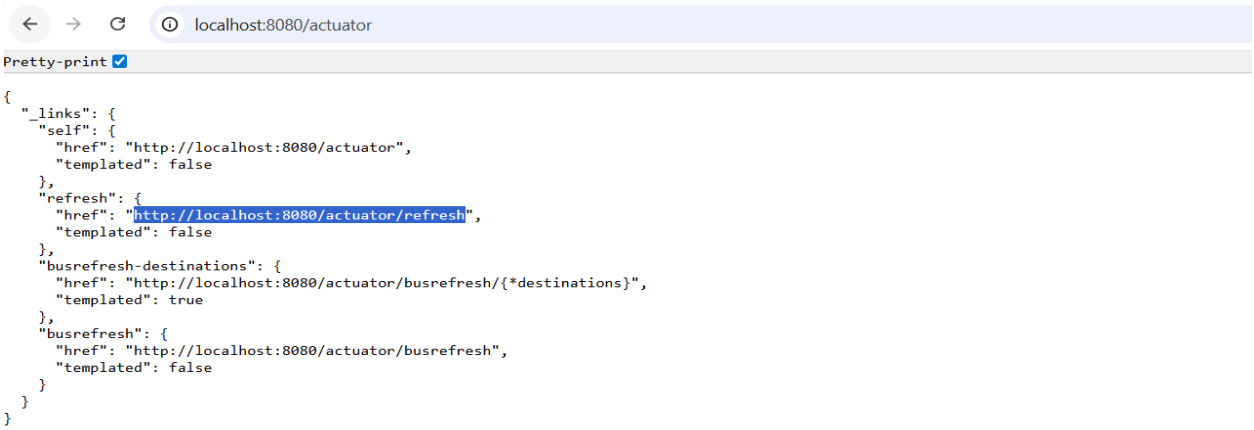
```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
```

- Expose the **/refresh** endpoint

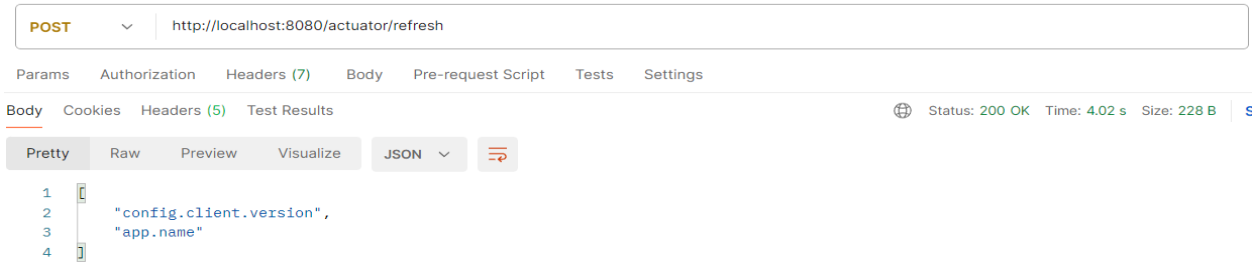
```yaml
management:
  endpoints:
    web:
      exposure:
        include: refresh
```

- Whenever config changes in Git, you must **manually call**:

  First call actuator endpoint :http://localhost:8080/actuator

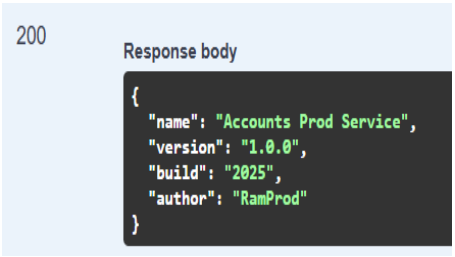


  Then invoke refresh endpoint mentioned in refresh
  POST http://localhost:8080/actuator/refresh



  Before invoking refresh

  After invoking refresh



- The microservice will then pull the latest configuration from the Config Server.

Refresh configurations at runtime using /refresh path

eazy bytes

What occurs when new updates are committed to the Git repository supporting the Config Service? In a typical Spring Boot application, modifying a property would require a restart. However, Spring Cloud Config introduces the capability to dynamically refresh the configuration in client applications during runtime. When a change is pushed to the configuration repository, all integrated applications connected to the config server can be notified, prompting them to reload the relevant portions affected by the configuration modification.

Let's see an approach for refreshing the configuration, which involves sending a specific POST request to a running instance of the microservice. This request will initiate the reloading of the modified configuration data, enabling a hot reload of the application. Below are the steps to follow,

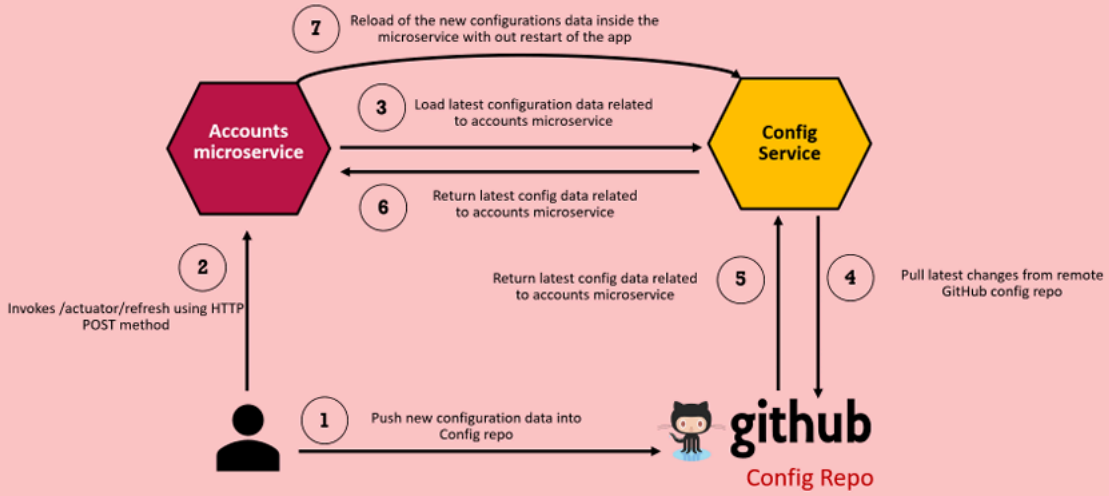
1. Add actuator dependency in the Config Client services: Add Spring Boot Actuator dependency inside pom.xml of the individual microservices like accounts, loans, cards to expose the /refresh endpoint

2. Enable /refresh API : The Spring Boot Actuator library provides a configuration endpoint called "/actuator/refresh" that can trigger a refresh event. By default, this endpoint is not exposed, so you need to explicitly enable it in the application.yml file using the below config,

```
management:
  endpoints:
    web:
      exposure:
        include: refresh
```

Refresh configurations at runtime using /refresh path

eazy bytes



You invoked the refresh mechanism on Accounts Service, and it worked fine, since it was just one application with 1 instance. How about in production where there may be multiple services ?If a project has many microservices, then team may prefer to have an automated and efficient method for refreshing configuration instead of manually triggering each application instance. Let's evaluate other options that we have

❌ Drawbacks

- You must call /refresh **for every microservice**
- If you have multiple instances (ex: 10 pods in Kubernetes), you must refresh **each instance manually**
- Not automated → high maintenance

- Not suitable for production systems with many services

👉 **Because of these limitations, we need a better method.**

Way 2: Automatic Refresh using Spring Cloud Bus

Site : https://spring.io/projects/spring-cloud-bus

Spring Cloud Bus links nodes of a distributed system with a lightweight message broker(ex: RabbitMQ or Kafka). This can then be used to broadcast state changes (e.g. configuration changes) or other management instructions.

A single refresh triggers updates across all services.

Only one manual step is needed: invoking /actuator/bus-refresh on any microservice instance. If all microservices are connected via a message broker like RabbitMQ,any configuration changes will update automatically. Invoking /actuator/bus-refresh on one microservice reloads the configuration for all others, since they are all connected through the same broker.

Ex: Steps to Integrate Spring Cloud Bus With RabbitMQ

1. Install RabbitMQ

We can install it using docker or .exe https://www.rabbitmq.com/docs/download

The rabbitmq:4-management image includes two parts:

1. Management Component (UI)
    - Runs on 15672
    - Lets you monitor and manage queues, exchanges, users, etc.
2. Core Messaging Component
    - Runs on 5672
    - Handles all core messaging operations like queues, exchanges, and message delivery.

```
C:\Users\jakkula.ramesh>docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:4-management
docker: error during connect: Head "http://%2F%2F.%2Fpipe%2FdockerDesktopLinuxEngine/_ping": open //./pipe/dockerDesktopLinuxEngine: The system cannot find
the file specified.

Run 'docker run --help' for more information

C:\Users\jakkula.ramesh>docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:4-management
Unable to find image 'rabbitmq:4-management' locally
4-management: Pulling from library/rabbitmq
41e174afde5a: Downloading [============>                                      ]  3.146MB/12.47MB
5746ab5df185: Downloading [============>                                      ]  2.097MB/8.995MB
342269d3962f: Download complete
2b00085445fe: Download complete
ed0d255a715b: Download complete
f6d381ad6ab2: Download complete
7218fe655bf8: Downloading [=====>                                             ]  3.146MB/27.85MB
3f4461d026ed: Download complete
20043066d3d5: Downloading [============>                                      ]   7.34MB/29.72MB
8e50c11f198e: Downloading [====================>                              ]  19.92MB/46.26MB
```

2. Add Spring Cloud Bus Dependency in Each Microservice

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

3. RabbitMQ Configuration and Expose bus-refresh in application.yml

If we add *, it allows all actuator endpoints. If we specify particular endpoints, then only those endpoints will be allowed.

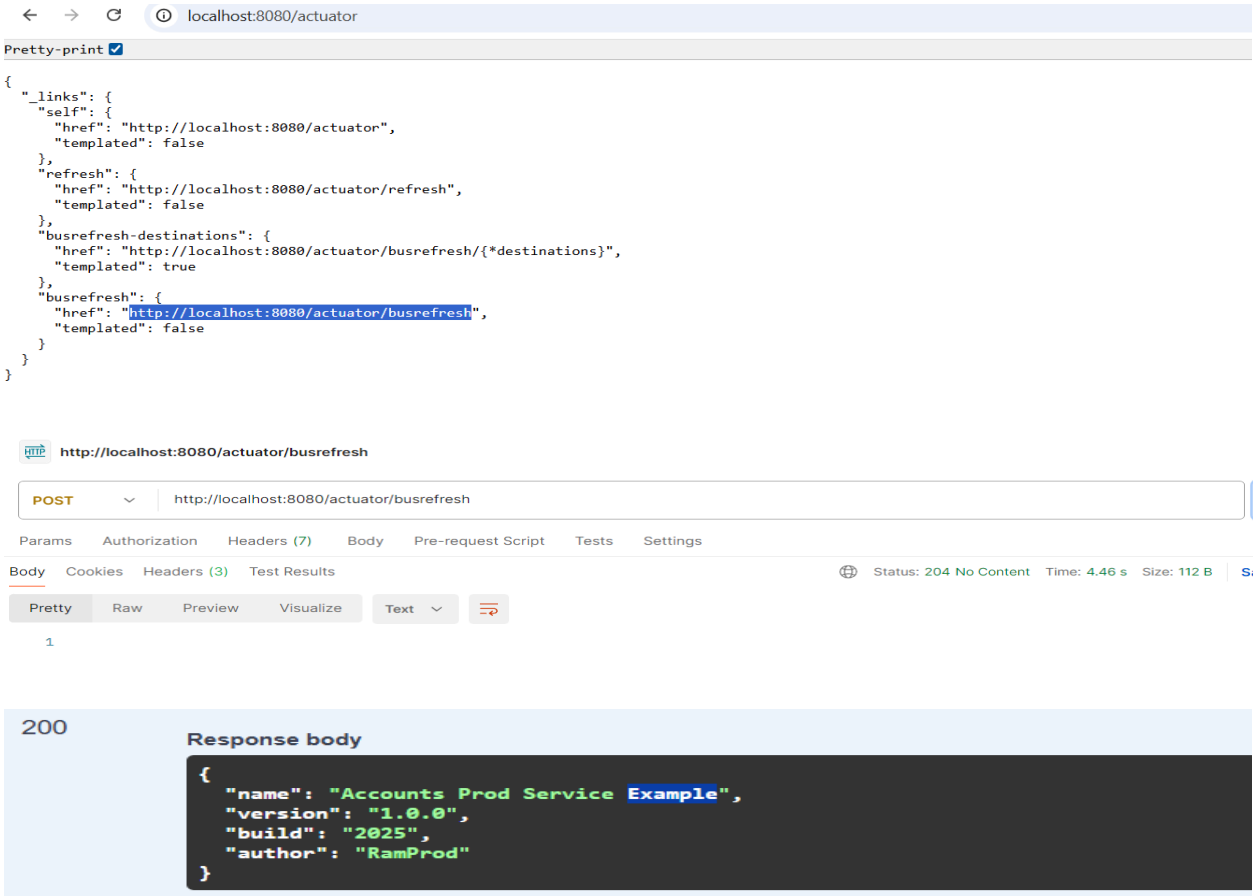
```yaml
rabbitmq:
  host: localhost
  port: 5672
  username: guest
  password: guest
management:
  endpoints:
    web:
      exposure:
        include: refresh, bus-refresh
```

```yaml
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

4. Trigger Config Refresh

After updating configuration in Spring Cloud Config repo, call:

All microservices connected to RabbitMQ will reload their configuration automatically.



Refresh configurations at runtime using Spring Cloud Bus

eazy bytes

Spring Cloud Bus, available at https://spring.io/projects/spring-cloud-bus, facilitates seamless communication between all connected application instances by establishing a convenient event broadcasting channel. It offers an implementation for AMQP brokers, such as RabbitMQ, and Kafka, enabling efficient communication across the application ecosystem.

Below are the steps to follow,

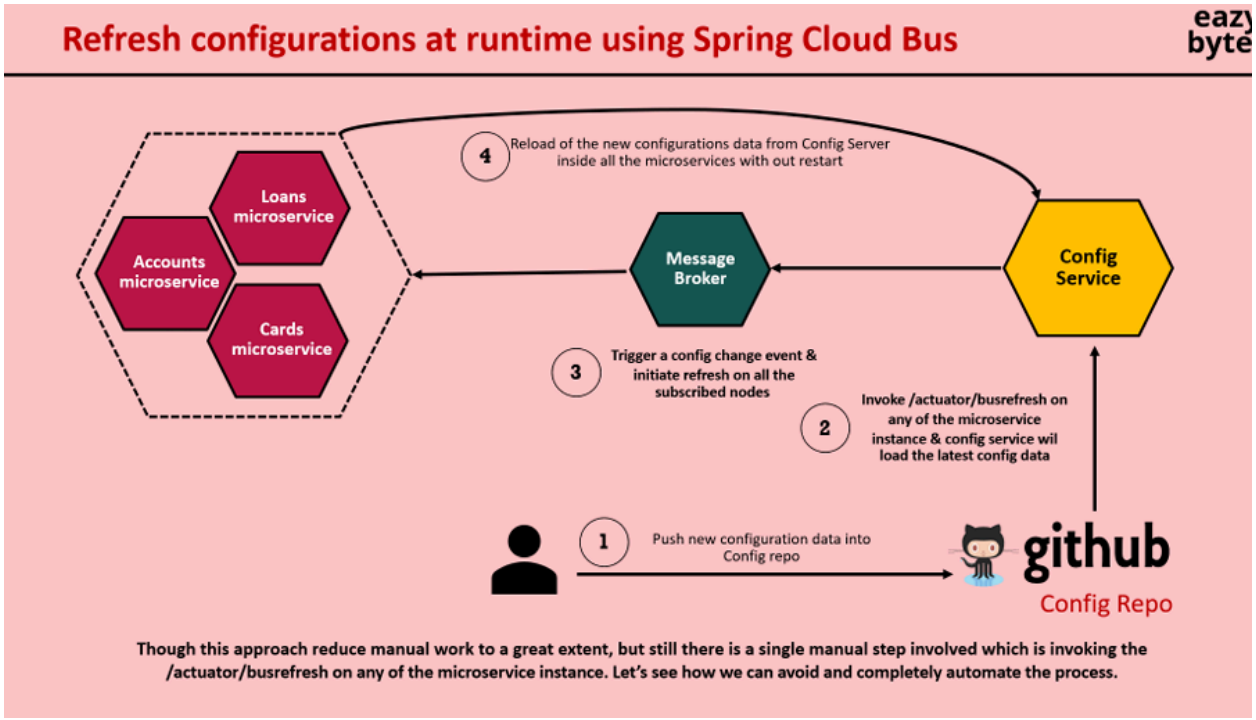
1. **Add actuator dependency in the Config Server & Client services**: Add Spring Boot Actuator dependency inside pom.xml of the individual microservices like accounts, loans and cards to expose the /busrefresh endpoint

2. **Enable / busrefresh API** : The Spring Boot Actuator library provides a configuration endpoint called "/actuator/busrefresh" that can trigger a refresh event. By default, this endpoint is not exposed, so you need to explicitly enable it in the application.yml file using the below config,

```
management:
  endpoints:
    web:
      exposure:
        include: busrefresh
```

3. **Add Spring Cloud Bus dependency in the Config Server & Client services**: Add Spring Cloud Bus dependency (spring-cloud-starter-bus-amqp) inside pom.xml of the individual microservices like accounts, loans, cards and Config server

4. **Set up a RabbitMQ**: Using Docker, setup RabbitMQ service. If the service is not started with default values, then configure the rabbitmq connection details in the application.yml file of all the individual microservices and Config server



So while Spring Cloud Bus removes the need to restart each microservice individually, But

Drawback

The manual step of calling /actuator/busrefresh still exists. there is a single manual step involved which is invoking the /actuator/busrefresh on any of the microservice instances."

Way 3 :Use Spring Cloud Config Monitor (Git Webhooks)

Webhooks (Git) + Spring Cloud Config Monitor + Spring Cloud Bus all work **together** to achieve *full automatic refresh* with no manual /bus-refresh call.

Spring Cloud Config Monitor **automatically triggers** /bus-refresh whenever a change is pushed to Git. So **no manual refresh** is needed.

- we should enable **Spring Cloud Config Monitor** in the Config Server , Add dependency in Config Server.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-monitor</artifactId>
</dependency>
```

- Expose webhook endpoint
-

```yaml
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

Config Server needs RabbitMQ

It must publish Bus Refresh events to all microservices.

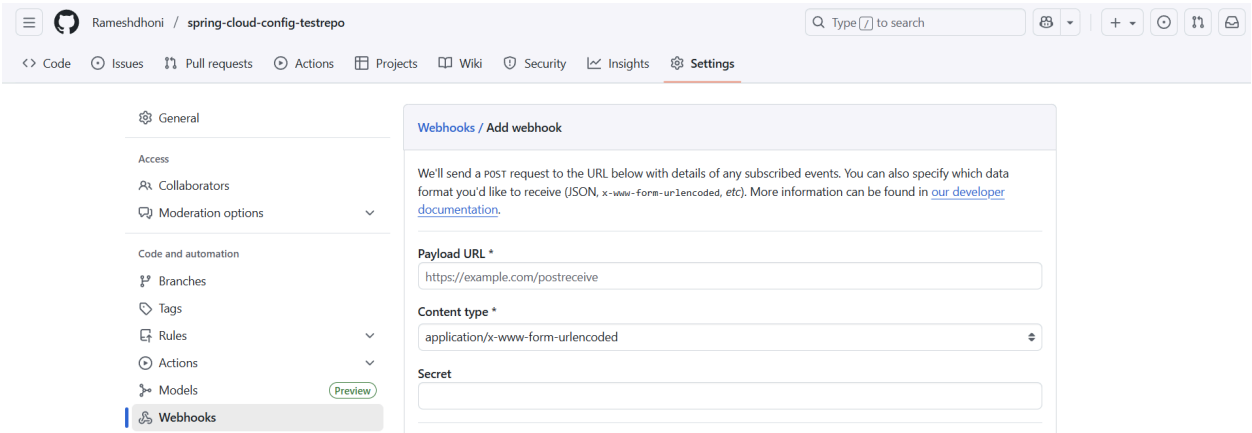
Config Server needs all Actuator endpoints exposed

Especially:

- /actuator/bus-refresh
- /actuator/refresh
- /monitor (for Git webhooks)
- /health
- /info

Including "*" makes all actuator endpoints available.

- Configure a **Git webhook** (GitHub / GitLab / Bitbucket)
  - Go to Repo → Settings → Webhooks → Add Webhook
  - Paste URL in Payload URL (http://<config-server-host>:<port>/monitor)
  - Set Content type = application/json
  - Select Push events and Save



- Whenever you push new configuration:
    - Git sends a webhook POST request to the Config Server
    - Config Server automatically triggers a **Bus Refresh event**
    - All microservices update themselves automatically

No human needs to call /actuator/bus-refresh anymore.

Refresh configurations at runtime using Spring Cloud Bus & Spring Cloud Config Monitor

eazy bytes

Spring Cloud Config offers the Monitor library, which enables the triggering of configuration change events in the Config Service. By exposing the /monitor endpoint, it facilitates the propagation of these events to all listening applications via the Bus. The Monitor library allows push notifications from popular code repository providers such as GitHub, GitLab, and Bitbucket. You can configure webhooks in these services to automatically send a POST request to the Config Service after each new push to the configuration repository. Below are the steps to follow,

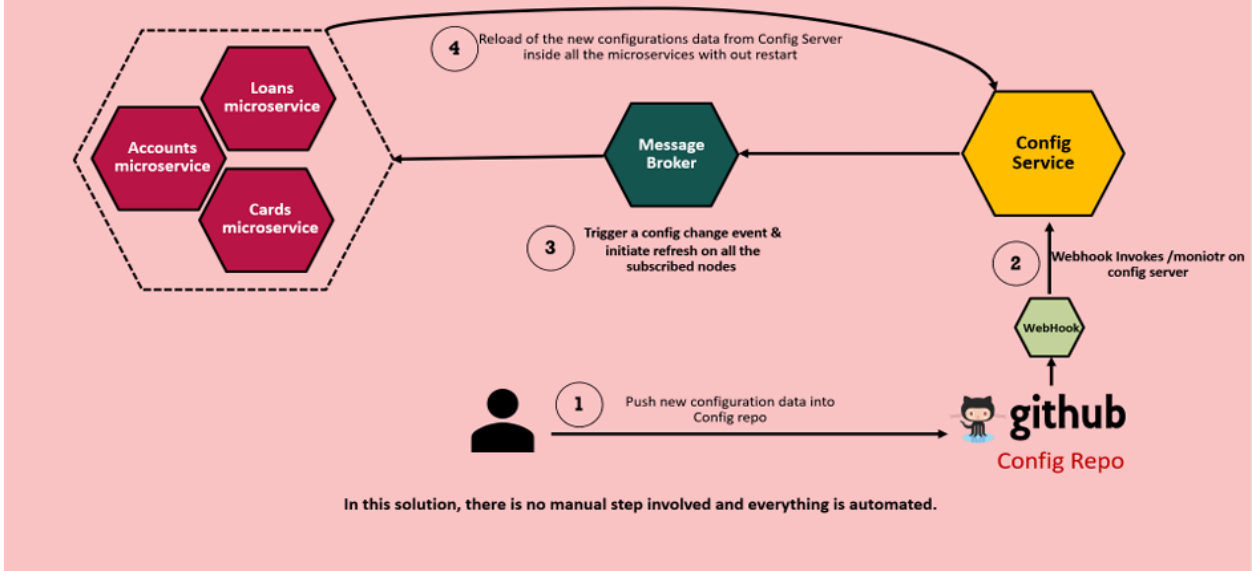
1. Add actuator dependency in the Config Server & Client services: Add Spring Boot Actuator dependency inside pom.xml of the individual microservices like accounts, loans, cards and Config server to expose the /busrefresh endpoint
2. Enable / busrefresh API : The Spring Boot Actuator library provides a configuration endpoint called "/actuator/busrefresh" that can trigger a refresh event. By default, this endpoint is not exposed, so you need to explicitly enable it in the application.yml file using the below config,

```
management:
  endpoints:
    web:
      exposure:
        include: busrefresh
```

3. Add Spring Cloud Bus dependency in the Config Server & Client services: Add Spring Cloud Bus dependency (spring-cloud-starter-bus-amqp) inside pom.xml of the individual microservices like accounts, loans, cards and Config server
4. Add Spring Cloud Config monitor dependency in the Config Server : Add Spring Cloud Config monitor dependency (spring-cloud-config-monitor) inside pom.xml of Config server and this exposes /monitor endpoint
5. Set up a RabbitMQ: Using Docker, setup RabbitMQ service. If the service is not started with default values, then configure the rabbitmq connection details in the application.yml file of all the individual microservices and Config server
6. Set up a WebHook in GitHub: Set up a webhook to automatically send a POST request to Config Service /monitor path after each new push to the config repo.

Refresh configurations at runtime using Spring Cloud Bus & Spring Cloud Config Monitor

eazy bytes

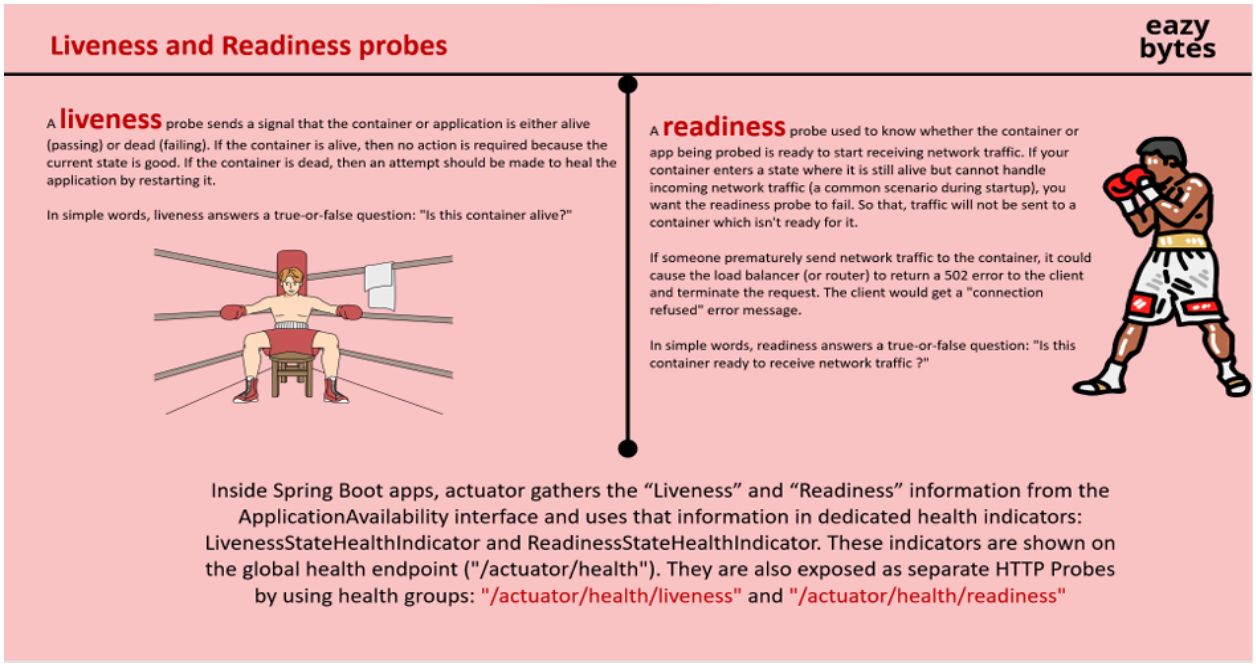


In this solution, there is no manual step involved and everything is automated.

Using Docker Compose With Spring Cloud Config, RabbitMQ, Webhooks & Multiple Environments

When moving microservices to container-based deployments, we must ensure:

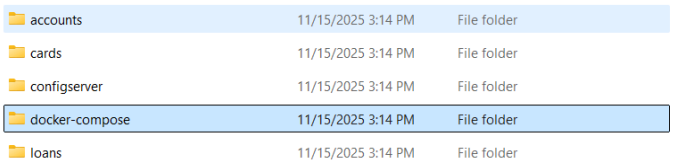
- All services run together
- Config Server reads from Git + Webhook + RabbitMQ
- Microservices auto-refresh on config changes
- Liveness & Readiness probes determine container health
- Multiple environments (dev, qa, prod) run using separate compose files
- Docker images are built once and reused everywhere (immutable infrastructure)



1. Create a docker-compose Folder (Central Folder)

This folder will contain:

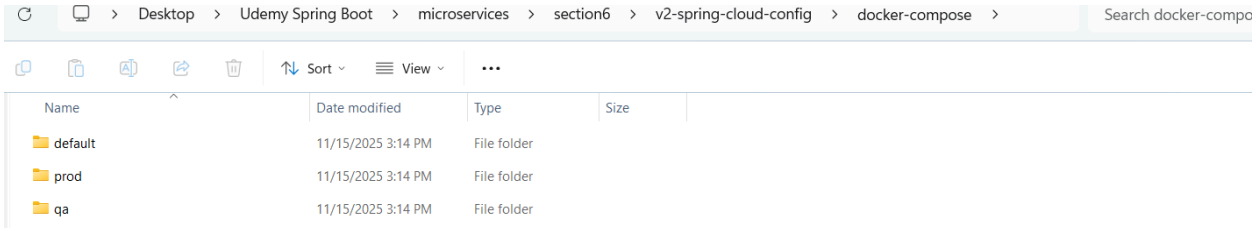
```
docker-compose/
  |-- dev/
  |     |-- docker-compose-dev.yml
  |
  |-- qa/
  |     |-- docker-compose-qa.yml
  |
  |-- prod/
        |-- docker-compose-prod.yml
```

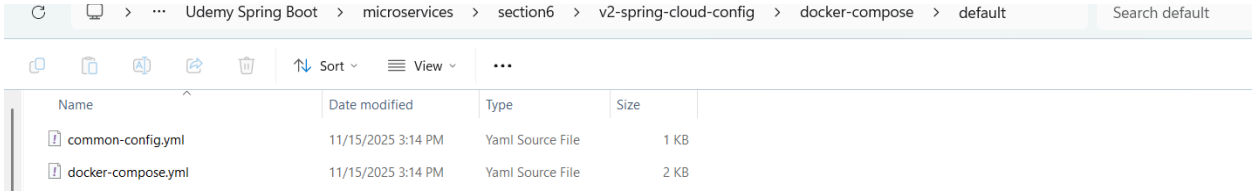


Each environment folder has its own compose file.

✔ You run **dev environment** using docker-compose-dev.yml
✔ You run **qa environment** using docker-compose-qa.yml
✔ You run **prod environment** using docker-compose-prod.yml

This ensures clean separation of ENV-based configs.





2. Add Services to Compose File

(A) docker-compose.yml → main file

(B) common-config.yml → shared settings , the repeated configuration we can write here , and we use this in docker-compose.yml

docker-compose.yml

```
services:
  rabbit:
    image: rabbitmq:3.13-management
    hostname: rabbitmq
    ports:
```

```
    - "5672:5672"
    - "15672:15672"
  healthcheck:
    test: rabbitmq-diagnostics check_port_connectivity
    interval: 10s
    timeout: 5s
    retries: 10
    start_period: 5s
  extends:
    file: common-config.yml
    service: network-deploy-service


configserver:
  image: "eazybytes/configserver:s6"
  container_name: configserver-ms
  ports:
    - "8071:8071"
  depends_on:
    rabbit:
      condition: service_healthy
  healthcheck:
    test: "curl --fail --silent localhost:8071/actuator/health/readiness | grep UP || exit 1"
    interval: 10s
    timeout: 5s
    retries: 10
```

```
    start_period: 10s
  extends:
    file: common-config.yml
    service: microservice-base-config

accounts:
  image: "eazybytes/accounts:s6"
  container_name: accounts-ms
  ports:
    - "8080:8080"
  depends_on:
    configserver:
      condition: service_healthy
  environment:
    SPRING_APPLICATION_NAME: "accounts"
  extends:
    file: common-config.yml
    service: microservice-configserver-config

loans:
  image: "eazybytes/loans:s6"
  container_name: loans-ms
  ports:
    - "8090:8090"
  depends_on:
```

```
    configserver:
      condition: service_healthy
  environment:
    SPRING_APPLICATION_NAME: "loans"
  extends:
    file: common-config.yml
    service: microservice-configserver-config

 cards:
  image: "eazybytes/cards:s6"
  container_name: cards-ms
  ports:
    - "9000:9000"
  depends_on:
    configserver:
      condition: service_healthy
  environment:
    SPRING_APPLICATION_NAME: "cards"
  extends:
    file: common-config.yml
    service: microservice-configserver-config

networks:
 eazybank:
  driver: "bridge"
```

Common-config.yml

```
services:
 network-deploy-service:
  networks:
   - eazybank
 microservice-base-config:
  extends:
   service: network-deploy-service
  deploy:
   resources:
    limits:
     memory: 700m
  environment:
   SPRING_RABBITMQ_HOST: "rabbit"
 microservice-configserver-config:
  extends:
   service: microservice-base-config
  environment:
   SPRING_PROFILES_ACTIVE: default
   SPRING_CONFIG_IMPORT: configserver:http://configserver:8071/
```

Detailed Breakdown of Each Service

1️⃣RabbitMQ Service

Docker Compose:

```
rabbit:
  image: rabbitmq:3.13-management
  hostname: rabbitmq
  ports:
    - "5672:5672"
    - "15672:15672"
```

Explanation:

Setting	Meaning
`image: rabbitmq:3.13-management`	RabbitMQ with admin UI
`hostname: rabbitmq`	Internal DNS name inside docker network
`ports`	
5672 = AMQP messaging	
15672 = management UI	

Health Check:

```
healthcheck:
  test: rabbitmq-diagnostics check_port_connectivity
  interval: 10s
  timeout: 5s
  retries: 10
  start_period: 5s
```

Parameter	Meaning
`test`	Command run inside container to check health
`interval: 10s`	Run test every 10 seconds
`timeout: 5s`	Fail test if no response in 5 seconds
`retries: 10`	Mark container unhealthy after 10 failures
`start_period: 5s`	Wait 5 seconds before starting checks

Extends:

```
extends:
  file: common-config.yml
  service: network-deploy-service
```

Meaning:

- shared network config

2️⃣ Config Server

```
configserver:
  image: "eazybytes/configserver:s6"
  container_name: configserver-ms
  ports:
    - "8071:8071"
  depends_on:
    rabbit:
      condition: service_healthy
```

Explanation:

Setting	Meaning
`depends_on.condition: service_healthy`	Start only after RabbitMQ is healthy
`ports: 8071`	Exposes config server outside

Health Check:

```
healthcheck:
  test: "curl --fail --silent localhost:8071/actuator/health/readiness | grep UP || exit 1"
  interval: 10s
  timeout: 5s
  retries: 10
  start_period: 10s
```

Parameter	Meaning
`curl … /readiness`	Ensures Spring Boot says "ready"
`grep UP`	Only ready if UP
`start_period: 10s`	Wait for Spring Boot startup

Extends:

```
extends:
  file: common-config.yml
  service: microservice-base-config
```

Meaning:

- memory limit
- RabbitMQ host
- network config

3️⃣ Microservices (accounts, loans, cards)

Example: **accounts**

```
accounts:
  image: "eazybytes/accounts:s6"
  container_name: accounts-ms
  ports:
    - "8080:8080"
  depends_on:
    configserver:
      condition: service_healthy
  environment:
    SPRING_APPLICATION_NAME: "accounts"
```

Explanation:

Setting	Meaning
`depends_on.configserver.service_healthy`	Start only when config server is UP
`SPRING_APPLICATION_NAME`	Identifies microservice to config server
`ports`	Exposes service to local machine

Extends → configserver config

```
extends:
  file: common-config.yml
  service: microservice-configserver-config
```

This injects:

```
SPRING_PROFILES_ACTIVE=default
```

SPRING_CONFIG_IMPORT=configserver:http://configserver:8071/

SPRING_RABBITMQ_HOST=rabbit

So:

- It loads config from config server
  It sends bus refresh events
- It uses RabbitMQ for Cloud Bus

Same applies for:

- loans (8090)
- cards (9000)

4. Build Docker Images for All Microservices

```
Last login: Sun Jul 23 15:22:00 on ttys006
eazybytes@Eazys-MBP accounts % mvn compile jib:dockerBuild
```

5. Push Docker All Images to Docker Hub

```
eazybytes@Eazys-MBP configserver % docker image push docker.io/eazybytes/cards:s6
The push refers to repository [docker.io/eazybytes/cards]
4969cc273c51: Layer already exists
```

6. Run Environment Using Docker Compose

Example: run dev

cd docker-compose/dev

docker compose up -d

```
eazybytes@Eazys-MBP docker-compose % cd default
eazybytes@Eazys-MBP default % docker compose up -d
```

```
eazybytes@Eazys-MBP ~ % docker ps
CONTAINER ID   IMAGE                        COMMAND                  CREATED          STATUS                     PORTS
                                                                          NAMES
3446dcb1094a   eazybytes/accounts:s6        "java -cp @/app/jib-…"   54 seconds ago   Up Less than a second      0.0.0.0:8080->8080/tcp
                                                                          accounts-ms
49df7df49ca0   eazybytes/loans:s6           "java -cp @/app/jib-…"   54 seconds ago   Up Less than a second      0.0.0.0:8090->8090/tcp
                                                                          loans-ms
78ae22774607   eazybytes/cards:s6           "java -cp @/app/jib-…"   54 seconds ago   Up Less than a second      0.0.0.0:9000->9000/tcp
                                                                          cards-ms
18e243b04b24   eazybytes/configserver:s6    "java -cp @/app/jib-…"   54 seconds ago   Up 42 seconds (healthy)    0.0.0.0:8071->8071/tcp
                                                                          configserver-ms
74e6d30d3735   rabbitmq:3.12-management     "docker-entrypoint.s…"   54 seconds ago   Up 53 seconds (healthy)    4369/tcp, 5671/tcp, 0.0.0.0:5672->
5672/tcp, 15671/tcp, 15691-15692/tcp, 25672/tcp, 0.0.0.0:15672->15672/tcp   default-rabbit-1
eazybytes@Eazys-MBP ~ %
```

7. Switching Environments

You do **NOT** rebuild Docker images.
You only switch compose files.

Step 1 — Bring down existing containers

docker compose down

- Stops all running containers

Step 2 — Start containers with a new environment

To work in QA:

docker compose -f docker-compose-qa.yml up -d

To work in PROD:

docker compose -f docker-compose-prod.yml up -d

This is the whole point of immutable deployment.

8. Webhook + Config Monitor + RabbitMQ = Full Auto-Refresh

Once containers are running:

Whenever you push config changes to Git:

1. Git sends webhook → Config Server /monitor
2. Config Server publishes bus-refresh through RabbitMQ
3. All microservices receive refresh signal
4. Config updates applied **automatically** without restart

This works even inside Docker.

Connecting MYSQL to microservices

During development, there are two ways to use MySQL with your microservice:

Instead of installing MySQL permanently on your machine, you can run it in a Docker container. Use it when needed and remove it when not. This keeps your system clean and avoids unnecessary resource usage.”

1. Install MySQL directly on your machine

- Install MySQL Server locally (Windows/Linux).
- The microservice connects using localhost:3306.
- This approach works, but MySQL always runs in the background and consumes system resources.

2. Run MySQL using Docker (Recommended)

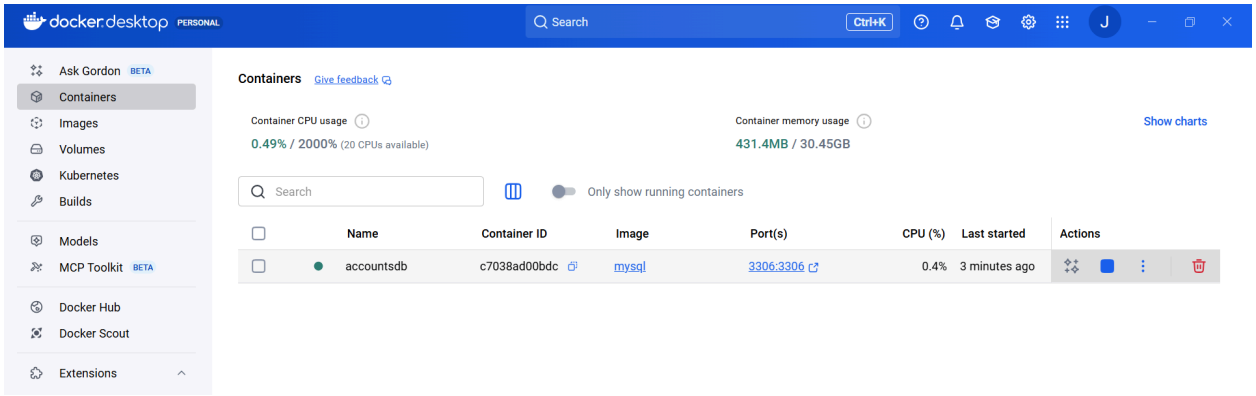
- Pull the official MySQL Docker image from Docker Hub. https://hub.docker.com/_/mysql
- Run MySQL as a container with required environment variables and port mappings.
- Start the container when needed and stop/delete it when not required.
- This avoids permanent installation, saves memory, keeps the system clean, and is ideal for microservice development.
- During development, developers often use **Sqlectron**, a lightweight SQL client, to visually explore the database. https://sqlectron.github.io/

```
C:\Users\jakkula.ramesh>docker run -p 3306:3306 --name accountsdb -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=accountsdb -d mysql
Unable to find image 'mysql:latest' locally
latest: Pulling from library/mysql
480d01bd7a6a: Pull complete
5a3f7744d0e7: Pull complete
494c372d15c3: Pull complete
21aa606d8d58: Pull complete
dcee80f7340c: Pull complete
834e15e3ed24: Pull complete
c276de9b5571: Pull complete
0cd145fbb449: Pull complete
f5f78fcd9ccb: Pull complete
023a182c62a0: Pull complete
Digest: sha256:569c4128dfa625ac2ac62cdd8af588a3a6a60a049d1a8d8f0fac95880ecdbbe5
Status: Downloaded newer image for mysql:latest
c7038ad00bdc121c7cc590acd8e53deb15a6ad50b6c979ffc7b862e7b6b646e7

C:\Users\jakkula.ramesh>
```

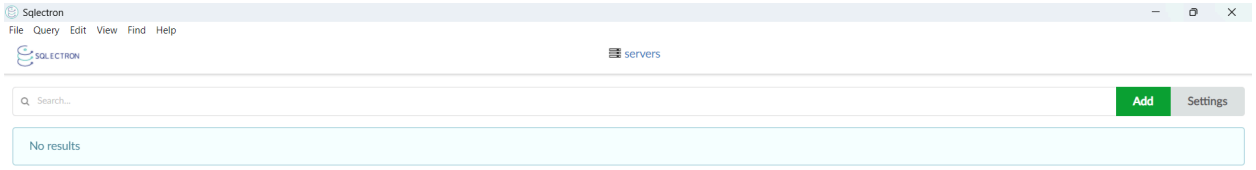
- **--name accountsdb**
  Assign a custom name to the MySQL container so you can easily start, stop, or reference it.
- **-e MYSQL_ROOT_PASSWORD=root**
  Sets the root user password inside the MySQL server.
  (You can replace root with a more secure password.)
- **-e MYSQL_DATABASE=accountsdb**
  Automatically creates a database named accountsdb when the container starts.
- **-p 3306:3306**
  Maps MySQL’s internal port **3306** to your host machine’s port **3306** so external applications (microservices, Sqlectron, etc.) can connect.

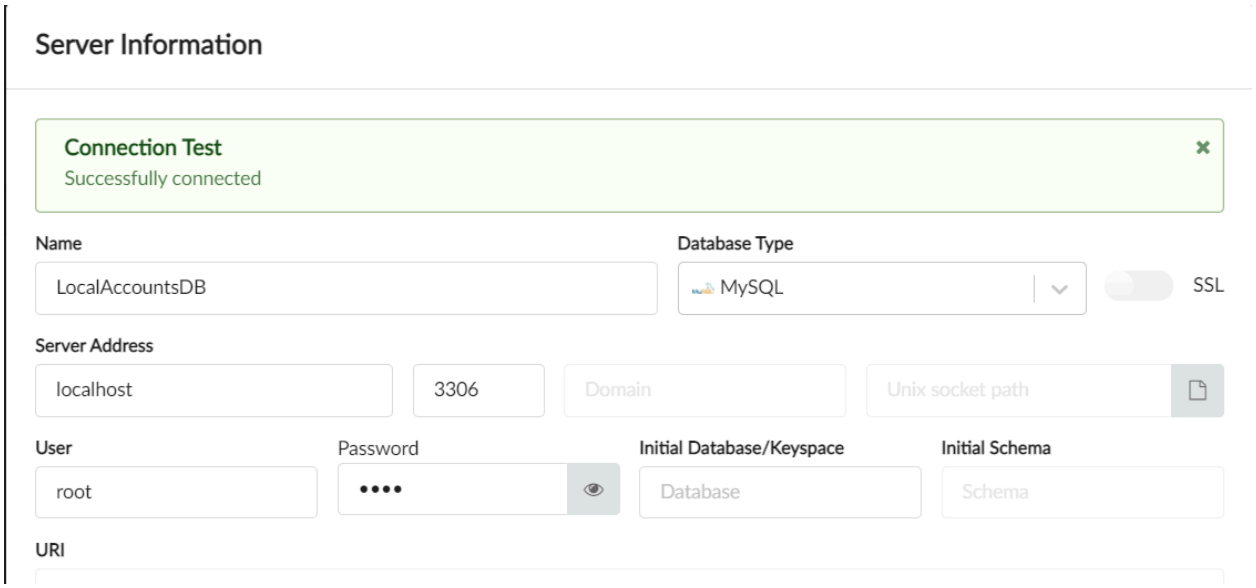
- **-d**
  Runs the container in **detached mode**, meaning it runs in the background.
- **mysql**
  Specifies the image to use — the official MySQL image from Docker Hub.
  (Defaults to the latest version if no tag like :latest is provided.)

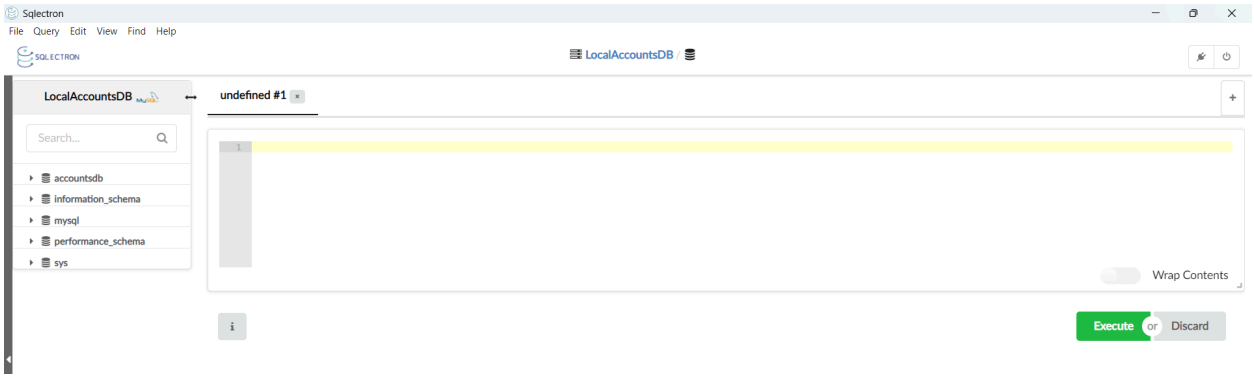
  We can see the created docker container for mysql using the above run command in docker-desktop.



We need to connect with **Sqlectron** as connections to see the UI of the database.







We can create separate MySQL databases for each microservice by running multiple MySQL containers, each with its own database name, container name, root password, and a different exposed port. For example, one microservice can use a container mapped to port **3307**, another on **3308**, and so on. This allows each microservice to have its own isolated database environment without conflicts.

Example: Running Separate MySQL Containers for Multiple Microservices

Microservice A → 3306:3306

Microservice B → 3308:3306

Microservice C → 3309:3306

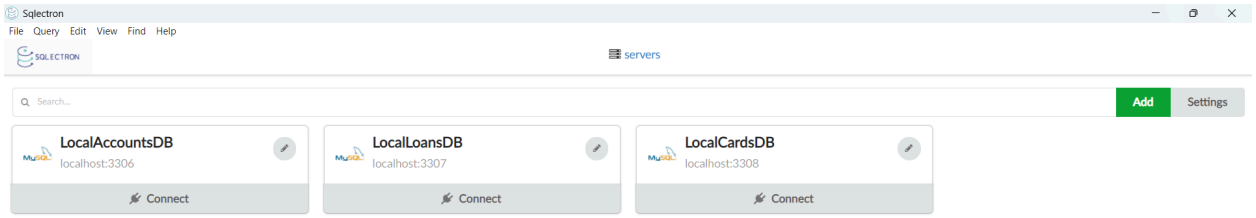
Every Docker container runs in its own isolated environment, so each MySQL container can use the same **internal port (3306)** without any conflict.

Only the **host ports** are different (3306, 3308, 3309).

These host ports must be different because all containers are running on the **same local machine**, and you cannot use the same host port more than once.

```
C:\Users\jakkula.ramesh>docker run -p 3307:3306 --name loansdb -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=loansdb -d mysql
6c2a58266d01b9686dd8a610ddcf7216a519db0db94059be9fc47277f233a9fa

C:\Users\jakkula.ramesh>docker run -p 3308:3306 --name cardsdb -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=cardsdb -d mysql
d675f72bfe21f27e420736c6220ecec7b4145d7eb1e93e279e585b4766ec7b54
```

	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
☐	accountsdb	c7038ad00bdc	mysql	3306:3306	0.43%	21 minutes ago	
☐	loansdb	6c2a58266d01	mysql	3307:3306	0.29%	2 minutes ago	
☐	cardsdb	d675f72bfe21	mysql	3308:3306	0.3%	2 minutes ago	



1. Add MySQL Dependency

2. Add Database Configuration in application.yml

3. Generate Docker Image for the Microservice (Using Jib)

    You can build the Docker image without writing a Dockerfile using the Jib Maven plugin.

    mvn compile jib:dockerBuild

4. Push the Docker Image to Docker Hub

5. Run Microservice Containers Using Docker Compose

    Create a docker-compose.yml file to run both MySQL and your microservice:

    docker compose up -d

Service Discovery & Registration in Microservices?

In microservices, **each service runs on its own IP + port**.
But these IPs keep **changing** because:

- ✔ autoscaling creates new instances
- ✔ crashed instances are replaced
- ✔ services restart with new ports
- ✔ containers get recreated

So the big questions are:

i) How do services find each other inside the network?

Each service must call another service by URL.
But URLs keep changing → so *where is my service?*

ii) How do new service instances join the network?

When new instances are created (due to autoscaling),
how do they become reachable to others?

iii) How do we load balance between multiple instances?

If Loans service has 5 instances:

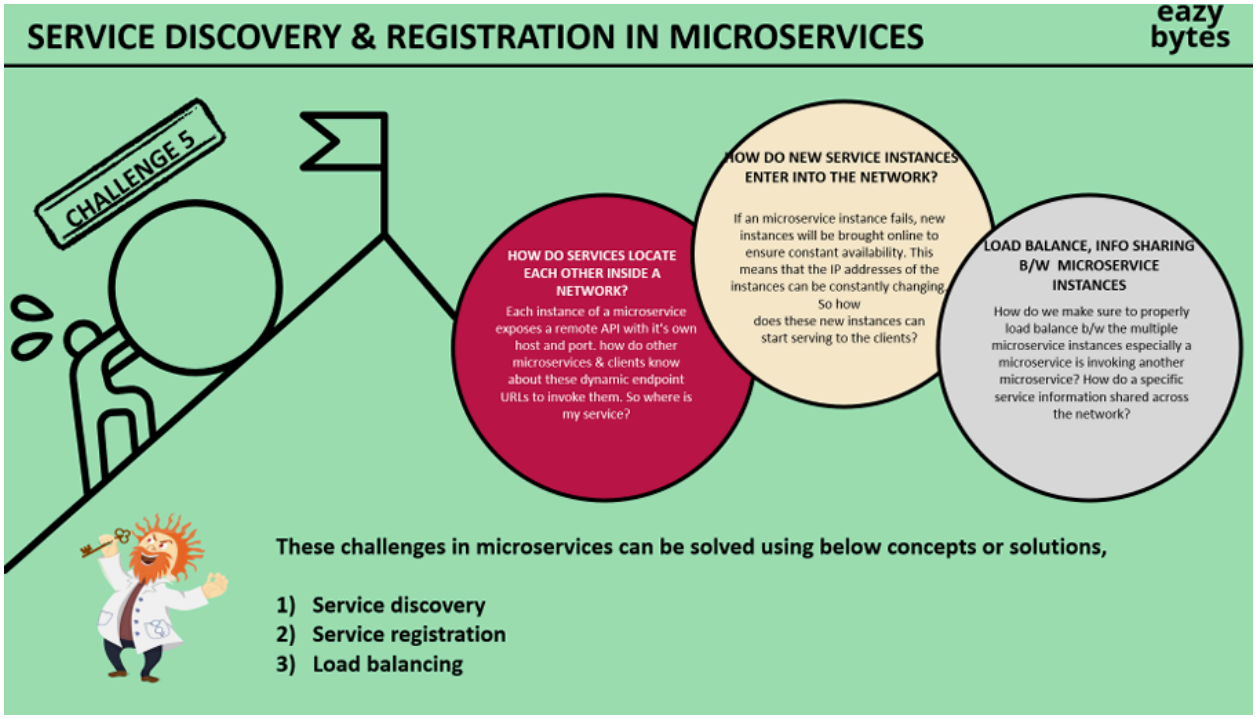
- How does the Accounts service pick one?
- How does it share load properly?

All these problems are solved by:

✔ Service Discovery

✔ Service Registration

✔ Load Balancing



How Cloud-Native Applications Solve This ?

What is a Service Registry

Service Registry is a place where all microservices register their current address (IP + Port).

A Service Registry stores the locations of all running microservice instances so other services can find and communicate with them.

Example:

- Netflix Eureka
- Consul
- Zookeeper

What does a Service Registry do

1. Microservices register themselves

   "Hello, I am the Loans service. My address is 10.0.1.5"

2. Registry stores the live addresses

   Loans =

   - 10.0.1.5
   - 10.0.1.8

   Accounts =

   - 10.0.2.4

   Cards =

   - 10.0.3.7

3. Other microservices ask the registry

   "Give me all Loans service addresses."

4. Registry returns the list

5. If a service dies, registry removes it

   Because no heartbeat was received.

Service Registry solves problems.

It keeps an updated list of:

- which services are alive
- their IPs
- their ports
- their health status

Client-Side Service Discovery

Client-Side Service Discovery (Eureka style)

In **client-side discovery**, the **client microservice itself** is responsible for:

✔ discovering service instances
✔ choosing one
✔ doing load balancing

✅ Who maintains the registry?

A **Service Registry** like **Eureka**, **Consul**, or **Zookeeper**.

Ex; **Accounts** want to call **Loans**.

📌 How it works

1. **Services register** themselves with the registry (IP + port).
2. Another service (Accounts asks registry) asks registry:
   → *"Give me all running instances of Loans service."*
3. Registry returns list:
   - 10.0.1.5
   - 10.0.1.9
4. **Client itself(Accounts)** picks one using load balancing (Round Robin, Random, etc.).
5. Client(Accounts) directly calls that Loans instance.

📘 Example Tools

- Netflix **Eureka**
- HashiCorp **Consul**
- Apache **Zookeeper**

✔ **Client (Accounts) decides**

✔ Client finds address
✔ Client does load balancing
✔ Registry only gives the list

Client-side service discovery and load balancing

eazy bytes

Client-side service discovery is an architectural pattern where client applications are responsible for locating and connecting to services they depend on. In this approach, the client application communicates directly with a service registry to discover available service instances and obtain the necessary information to establish connections.

Here are the key aspects of client-side service discovery:

- Service Registration: Client applications register themselves with the service registry upon startup. They provide essential information about their location, such as IP address, port, and metadata, which helps identify and categorize the service.
- Service Discovery: When a client application needs to communicate with a specific service, it queries the service registry for available instances of that service. The registry responds with the necessary information, such as IP addresses and connection details.
- Load Balancing: Client-side service discovery often involves load balancing to distribute the workload across multiple service instances. The client application can implement a load-balancing strategy to select a specific instance based on factors like round-robin, weighted distribution, or latency.

The major advantage of client-side service discovery is load balancing can be implemented using various algorithms, such as round-robin, weighted round-robin, least connections, or even custom algorithms. A drawback is that client service discovery assigns more responsibility to developers. Also, it results in one more service to deploy and maintain (the service registry). Server-side discovery solutions solve these issues. We are going to discuss the same when we are talking about Kubernetes

The Spring Cloud project provides several alternatives for incorporating client-side service discovery in our Spring Boot based microservices. More details to follow...

Spring Cloud support for Client-side service discovery

eazy bytes

Spring Cloud project makes Service Discovery & Registration setup trivial to undertake with the help of the below components,

- Spring Cloud Netflix's Eureka service which will act as a service discovery agent
- Spring Cloud Load Balancer library for client-side load balancing
- Netflix Feign client to look up for a service b/w microservices

Though in this course we use Eureka since it is mostly used but they are other service registries such as etcd,Consul, and Apache Zookeeper which are also good.

Though Netflix Ribbon client-side is also good and stable product, we are going to use Spring Cloud Load Balancer for client-side load balancing. This is because Ribbon has entered a maintenance mode and unfortunately, it will not be developed anymore

Advantages of Service Discovery approach includes,

- No limitations on availability
- Peer to peer communication b/w Services Discovery agents
- Dynamically managed IPs, configurations & Load balanced
- Fault-tolerant & Resilient in nature

Server-Side Service Discovery (Kubernetes style)

In **server-side discovery**, the **client does nothing**.

"Client does nothing. Server/load-balancer does everything."

The **server/load balancer** resolves which instance to call.

✅ Who maintains registry?

The platform itself:

- **Kubernetes API Server + Endpoints**
- **AWS ELB / ALB**
- **NGINX / HAProxy**

📌 How it works

1. Service instances register automatically with the **platform** (Kubernetes creates Endpoints).

2. Client calls a **fixed DNS name**: http://loans-service

3. Kubernetes load balancer / service proxy selects an instance.

4. Traffic is forwarded internally to:

    - 10.0.2.12
      10.0.2.15
    - 10.0.2.20

📘 Example Tools

- **Kubernetes Services + CoreDNS**

- **AWS Elastic Load Balancer**

- **NGINX / HAProxy**

✔ Client (Accounts) does NOT choose

✔ Client does NOT ask registry
✔ Kubernetes decides the instance

✔ Kubernetes does load balancing
✔ Client only uses a fixed name (DNS)

Client-Side Discovery (Eureka)

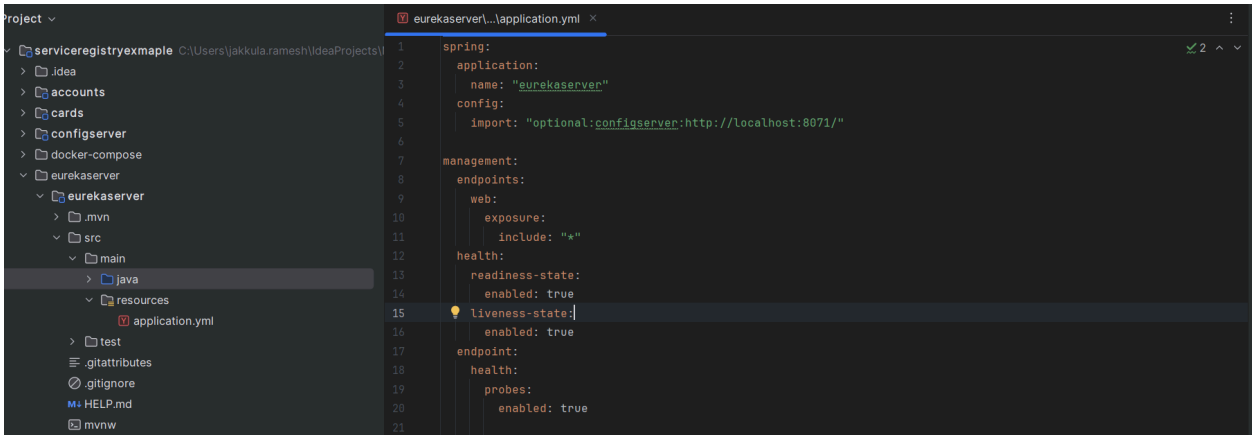
Step 1: Create the Eureka Server Project

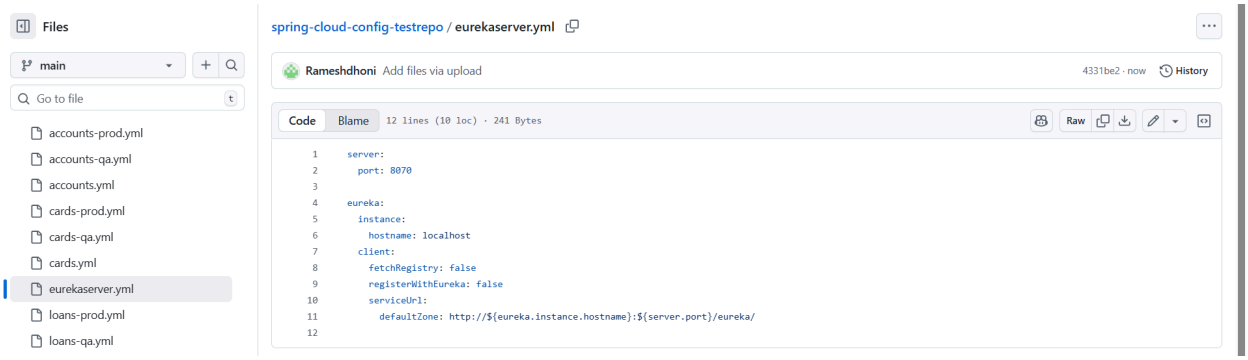
Dependencies required , When creating the Spring Boot project (via Spring Initializr), include:

1. Eureka Server
2. Spring Boot Actuator
3. Config Client

Add @EnableEurekaServer in the main class ,This turns your Spring Boot project into a **Service Registry**. Also add basic configuration in application.yml and also if we externalize any configuration also.

```java
@SpringBootApplication
@EnableEurekaServer
public class EurekaserverApplication {

    public static void main(String[] args) { SpringApplication.run(EurekaserverApplication.class, args); }

}
```

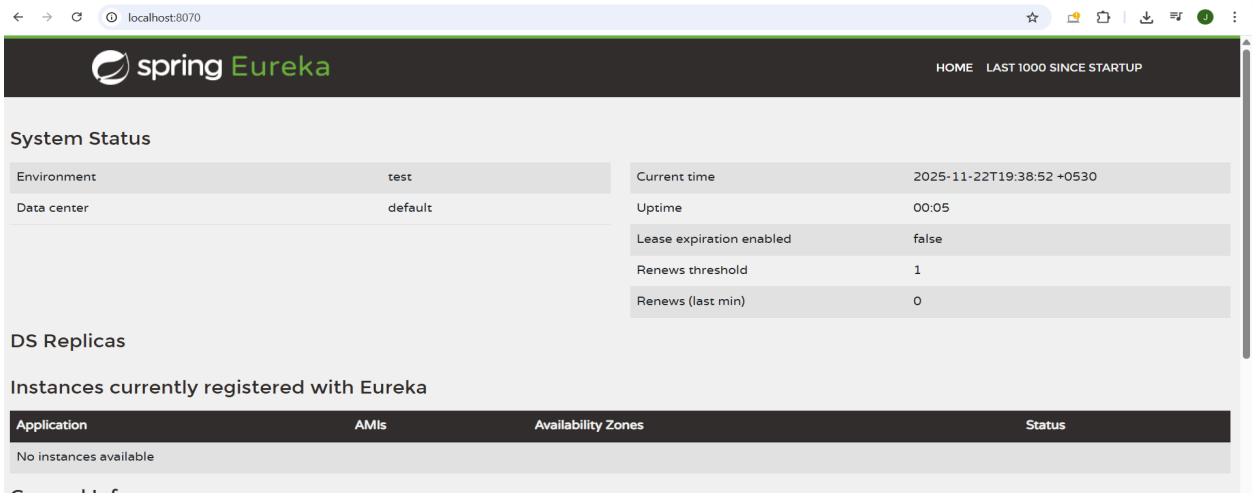


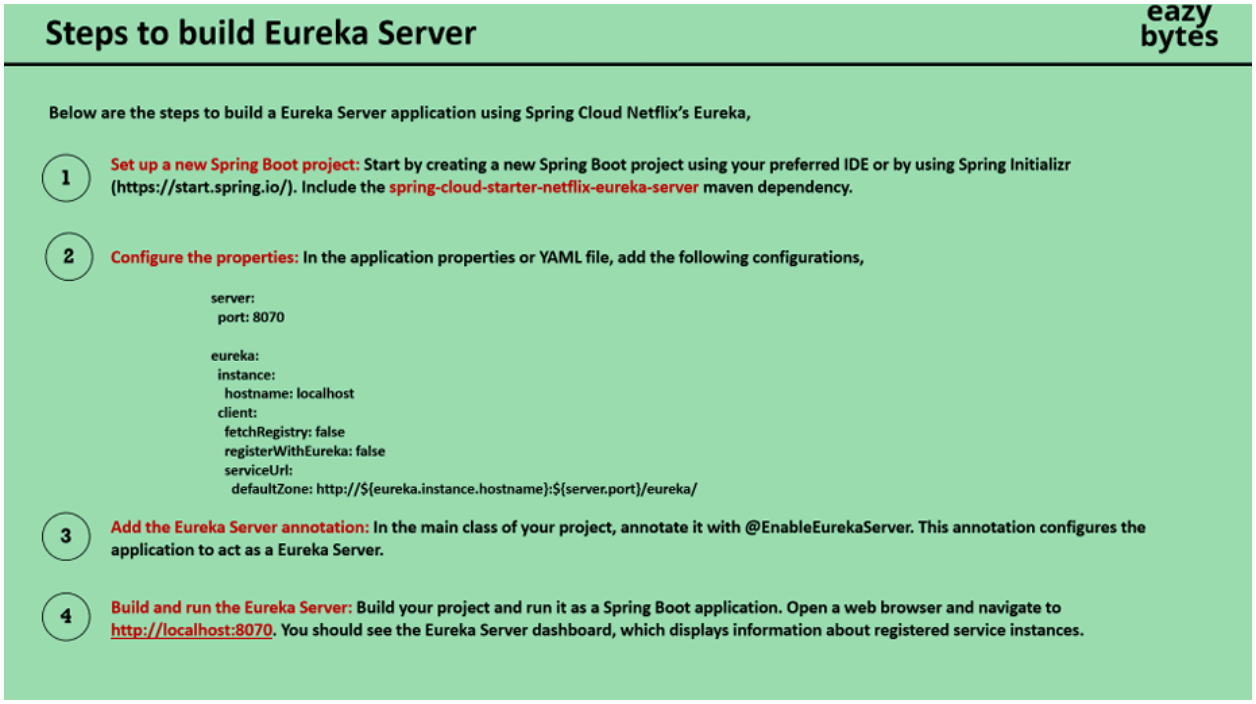


register-with-eureka: false → Eureka server should NOT register itself

fetch-registry: false → Eureka server should NOT fetch other registries

Start the config server and then eureka will get the UI of eureka.





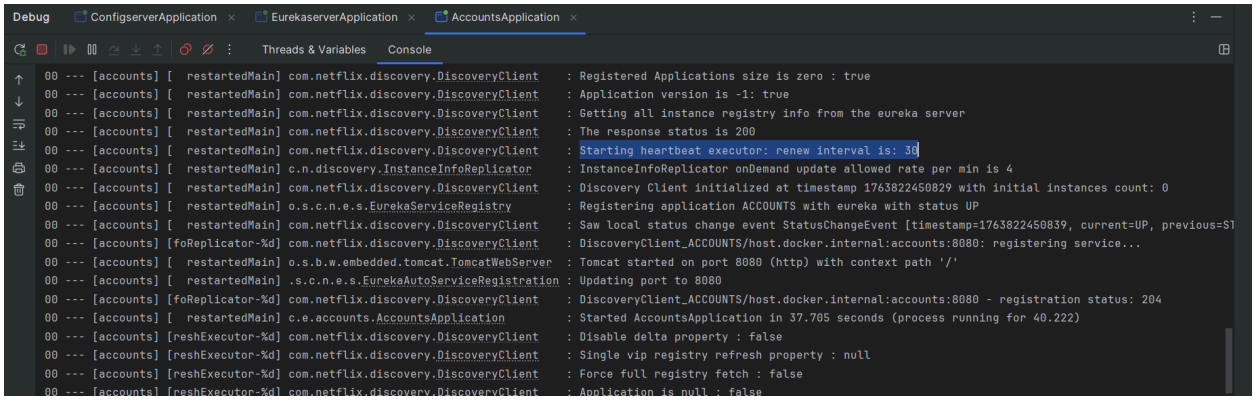
Step 2: Establish the connection between each microservice and the Eureka Server

Add **Eureka Client dependency** in Each microservice

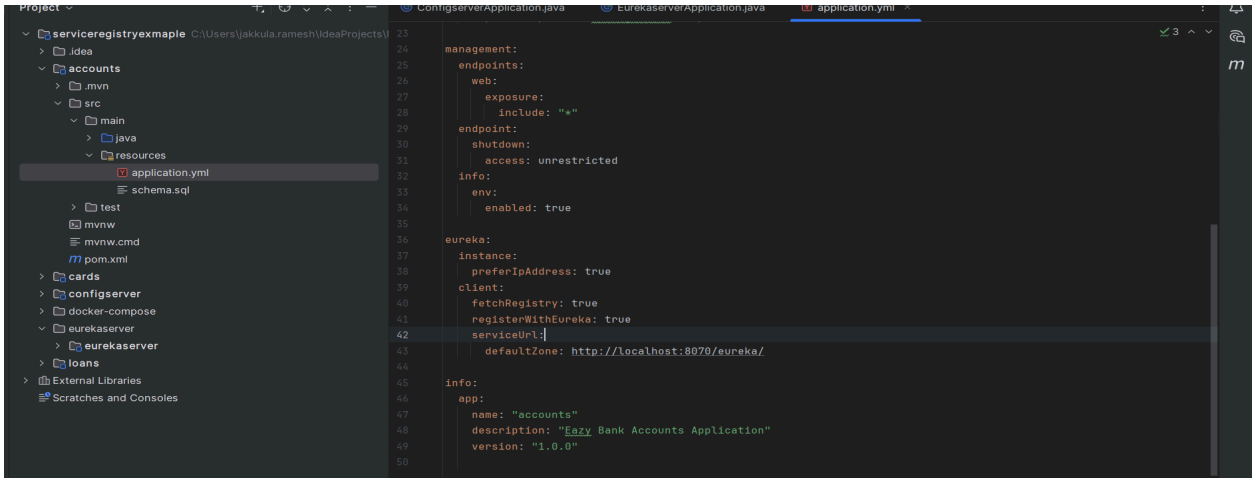
```
71  <dependency>
72      <groupId>org.springframework.cloud</groupId>
73      <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
74  </dependency>
```

@EnableDiscoveryClient is optional in Spring Boot 2.x and above when using Eureka. Because **Spring Cloud Netflix Eureka Client auto-configures itself** when it detects these dependencies:

Heartbeats: The client sends periodic heartbeat messages (renewals) to the Eureka server every 30 seconds by default to confirm it is still alive. If the server does not receive heartbeats for a configurable period (default is 90 seconds, or three renewal periods), the instance is typically removed from the registry. We can see the log message account microservices when we start.



Configure application.yml to register with Eureka Server



management: Controls Actuator endpoints.

- exposure.include: "*" → expose all endpoints
- shutdown.access: unrestricted → allow /actuator/shutdown
- info.env.enabled: true → show environment info in /actuator/info

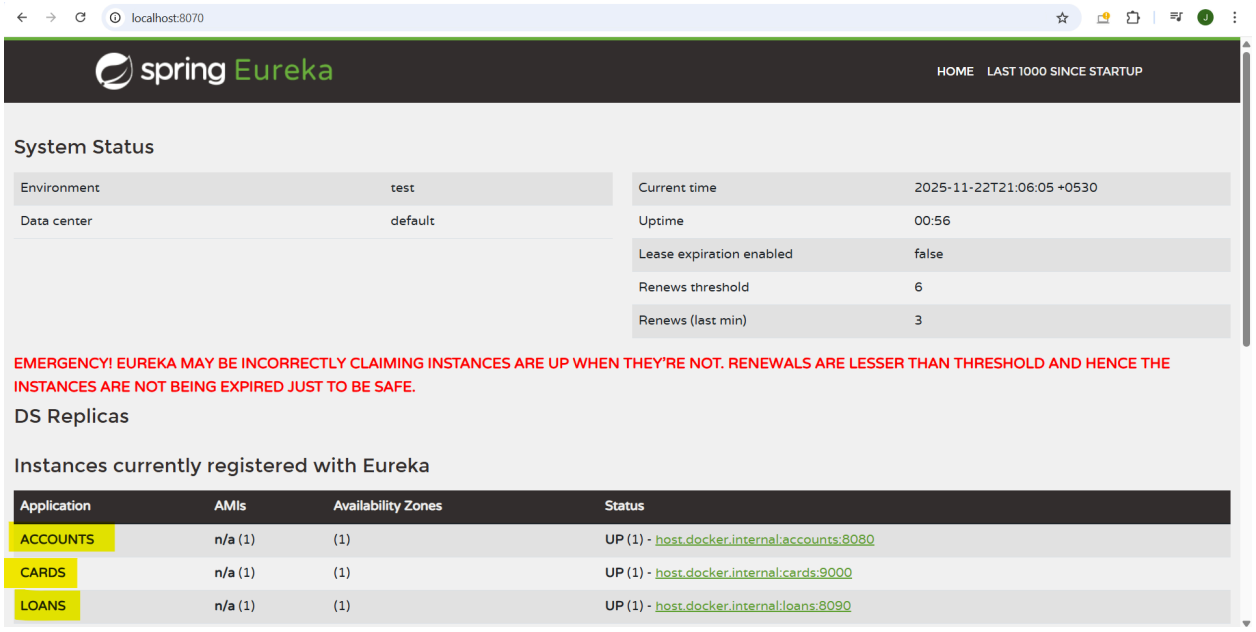
eureka: Connect microservice to Eureka server.

- preferIpAddress: true → register using IP, not hostname
- fetchRegistry: true → download list of other services
- registerWithEureka: true → register itself in Eureka
- defaultZone: http://localhost:8070/eureka/ → Eureka server URL

Start the microservices in below order:

- Spring Cloud Config Server
- Eureka Service Registry

- Individual microservices (Accounts, Loans, Cards, etc



Microservices Communicate With Each Other

There are two main ways:

i) Synchronous Communication : Microservices communicate using **synchronous methods** like REST APIs over HTTP/HTTPS and gRPC, where one service makes a request and waits for a response,
One service sends a request to another service and waits for a response before continuing.

REST APIs (HTTP) : RestTemplate, Feign Client, and WebClient

Imperative Approach :

You **write everything manually** —
You tell *HOW* the call should be executed.

You control:

- How the request is sent
- How headers are set
- How to handle errors
- How to parse the response

Ex: RestTemplate ,WebClient.

Declarative Approach :

You only **declare WHAT** you want to call.
Framework handles **HOW** the call happens.

You do NOT write:

- HTTP call code
- Request creation
- Error handling boilerplate
- Load balancing logic

Framework does it automatically.

Ex: Feign Client

i) RestTemplate (Imperative Approach)

Old, synchronous, blocking HTTP client.

blocking HTTP client means When you call another service, the **thread waits** (stops) until it gets the response. **Your thread is blocked** during the HTTP call.

- Uses **Thread-per-request** → not good for high load.
- Very easy to use.
- Spring marked it as **deprecated / not recommended** for new apps (but still widely used in old real-time projects). N***ote:*** *RestTemplate is officially deprecated in Spring 5+. Use WebClient for new applications*

Example

```
RestTemplate restTemplate = new RestTemplate();
String result = restTemplate.getForObject(
"http://books-service/books/1", String.class);
```

ii) Feign Client (Declarative Approach)

Declarative HTTP client means we **declare** (define) an interface, and Spring/Feign automatically generates the HTTP code for you.

You only **declare WHAT** you want to call.

Framework handles **HOW** the call happens.

- You just write an **interface**, and Feign generates HTTP calls.
- Works best with **Spring Cloud + Eureka**.
- Used a lot in **microservices** for synchronous communication.
- Auto-load balancing using **Ribbon / Spring Cloud LoadBalancer**.

Example

```
@EnableFeignClients
@SpringBootApplication
public class StudentServiceApplication { }
```

```
@FeignClient(name = "books-service")
public interface BookClient {
@GetMapping("/books/{id}")
Book getBook(@PathVariable("id") Long id);
}
```

iii) WebClient

Modern, non-blocking (reactive) HTTP client means When you make an HTTP call, your thread **does NOT wait** for the response.It can continue doing other work.The call runs **asynchronously** (non-blocking). When the response arrives, it is processed via a reactive pipeline (Mono/Flux).

- Recommended by Spring for new projects.
- Part of **Spring WebFlux**.
- Supports **async + reactive** programming.
- Best choice for **high-performance, scalable** microservices.

Example

```
WebClient webClient = WebClient.create();
String result = webClient.get()
        .uri("http://service/api")
        .retrieve()
        .bodyToMono(String.class)
        .block();
```

Type	Example	Meaning
Blocking HTTP Client	RestTemplate	Thread waits for the response
Declarative HTTP Client	FeignClient	You write only interface → client auto-generated
Non-blocking Reactive HTTP Client	WebClient	Thread does not wait → async & scalable

2) Asynchronous Communication: Asynchronous methods like message brokers (e.g., RabbitMQ, Kafka) or event streaming, where a service publishes a message without waiting for an immediate reply.
No waiting for a response.

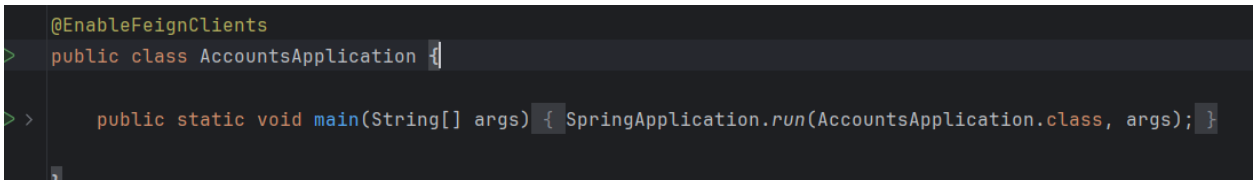
Both approaches rely on APIs for communication, not direct calls, and often involve using lightweight protocols and data formats like JSON.

Feign Client Example

Steps to Integrate Feign Client in Microservices (Accounts → Loans, Cards).

Step 1: Add Feign Client Dependency : Add to accounts microservice:

Step 2: Enable Feign Clients in Accounts Application : This activates Feign functionality.



Step 3: Make Sure Services Are Registered in Eureka

Each microservice must have:

```yaml
spring:
  application:
    name: accounts / loans / cards
eureka:
  client:
    register-with-eureka: true
    fetch-registry: true
```

🔸 **Important:**

`@FeignClient(name = "loans")` must match the **application name** registered in Eureka.

Step 4: Create Feign Client Interfaces in Accounts

Example: Loans Client

```java
@FeignClient(name = "loans")
public interface LoansClient {

    @GetMapping("/loans/{customerId}")
    LoansDto getLoanDetails(@PathVariable("customerId") Long customerId);
}
```

REST endpoint path **must match** the Loans/Card controller path

Method signature must match

DTOs used in response MUST be recreated in Accounts

Step 5: DTO MUST be created in Accounts microservice based on the return type of the Feign abstract method.

```java
@Data
public class LoansDto {
    private Long loanNumber;
    private String loanType;
    private Long totalLoan;
}
```

Loans and Cards microservices return JSON response.
Feign needs a Java class to convert that JSON into an object.
Even if Loans MS already has LoansDto, you MUST create this same class inside Accounts.

Step 6: Call Feign Client Inside Service Layer

```
@Service
public class AccountsService {

    @Autowired
    private LoansClient loansClient;

    public LoansDto getLoansOfCustomer(Long customerId) {
        return loansClient.getLoanDetails(customerId);
    }
}
```

Feign + Eureka will:

- **Feign asks Eureka :** "Where are the instances of Loans service?"
- **Eureka responds :** Returns all running instances, e.g.:
  http://localhost:9090

  http://localhost:9091
- **Load Balancing (client-side)**

  Spring Cloud LoadBalancer (inside Accounts) picks **one instance** from the list.

  Example: picks http://localhost:9091.
- **Feign builds REST call** : GET http://localhost:9091/loans/101
- **Loans service responds :** Returns JSON data.
- **Feign converts JSON → LoansDto**

  Accounts receive fully mapped objects.

➡ This is synchronous HTTP communication via service discovery.

Eureka Self-Preservation

Before Self-Preservation Mode

Eureka maintains a registry of all microservice instances.

- Each instance sends a **heartbeat** every 30 seconds to show it's alive.
- If Eureka stops receiving heartbeats, it assumes the instance is **down** and removes it from the registry.

Issue:

- If there's a **temporary network glitch**, Eureka might wrongly remove perfectly healthy services.
- This can cause **service discovery failures** and cascading outages.

Self-Preservation Mode

- Monitors the rate of heartbeats from all registered instances.
- If the heartbeat rate falls below a safe threshold, it **stops removing (eviction)** of instances.
- Ensures that **temporary network issues or brief outages** don't cause healthy services to be wrongly removed.
- Keeps the **service registry stable and reliable** even during partial failures.

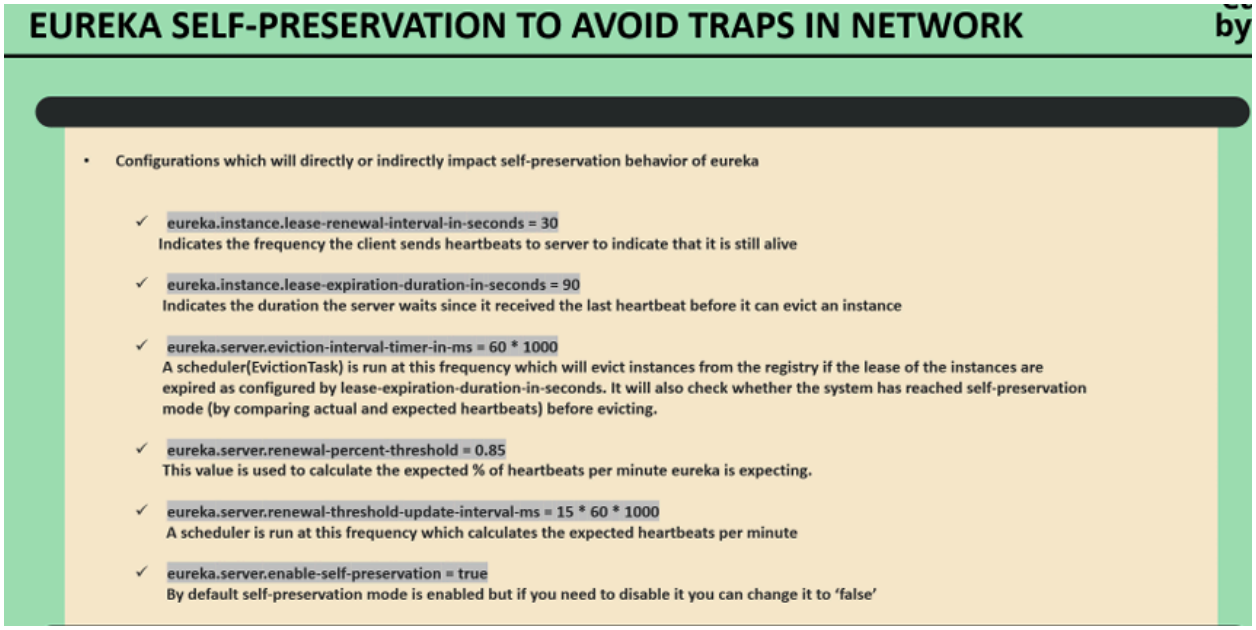
How It Works

1. Each client (microservice) sends a **heartbeat** to Eureka every 30 seconds (default).
2. Eureka expects a certain number of heartbeats based on the number of registered instances.
3. If the number of received heartbeats **drops below a threshold**, Eureka enters **self-preservation mode**.

4. In this mode:

    - Eureka **stops expiring instances** from its registry.
    - Prevents mass eviction that could cause a system-wide outage.

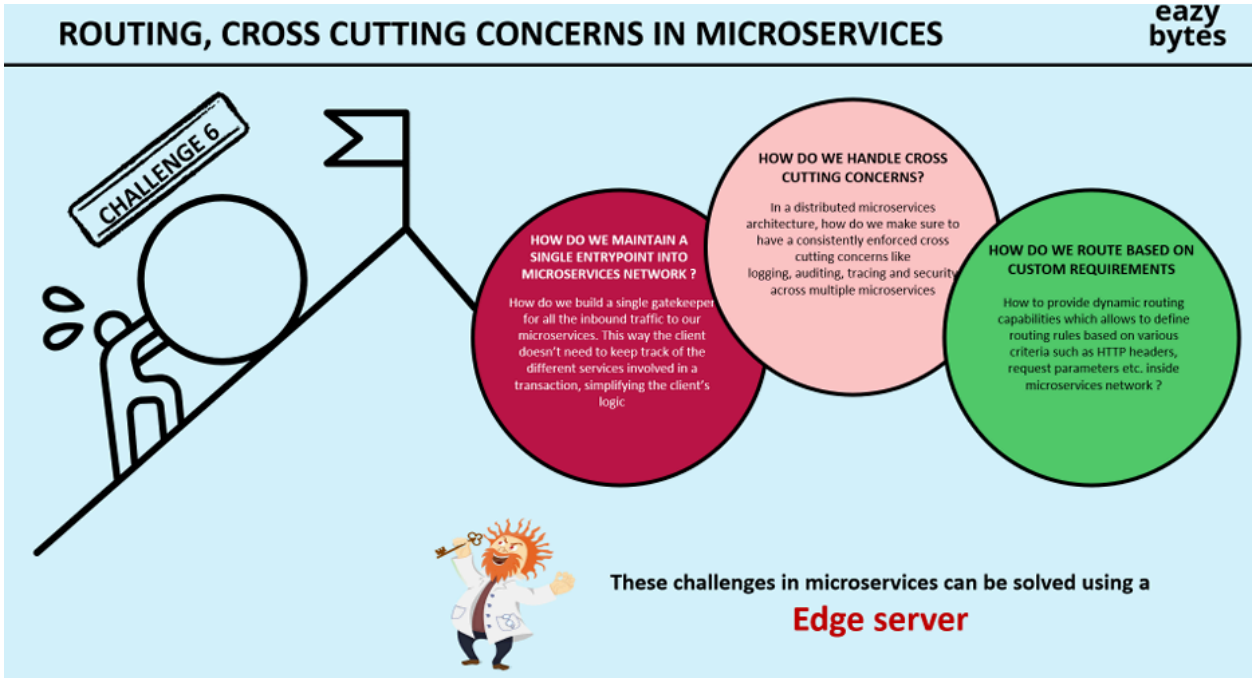
Key Properties



When to Disable

- Usually, you **do not disable self-preservation** in production.
  Disabling might be useful **in development or isolated environments** where heartbeats are unreliable.

```yaml
eureka:
  server:
    enable-self-preservation: false
```



API Gateway

What is an API Gateway?

An **API Gateway** is like the **front door** for all your backend microservices.

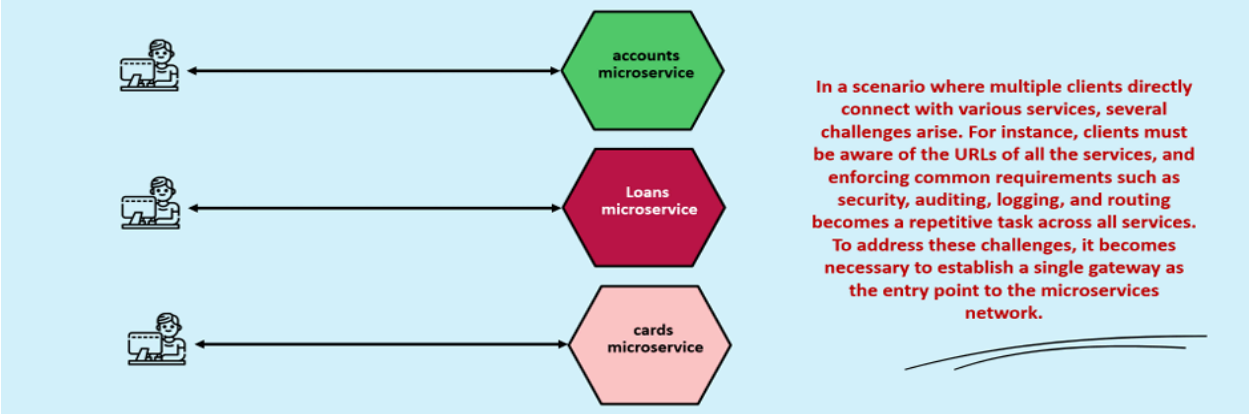
- It sits **between clients (web, mobile, etc.) and microservices**.
- Clients don't call microservices directly → Instead, they call the Gateway, and it forwards requests to the correct service.
- It provides **centralized features:**
  - ○ Routing (send request to right microservice)
  - ○ Load balancing
  - ○ Security (authentication/authorization)
  - ○ Rate limiting (avoid overload)
  - ○ Logging & monitoring
  - ○ Request/response transformations

Before and After API Gateway

Before API Gateway (Direct Communication)
Client → User Service (http://localhost:8081/users)
Client → Order Service (http://localhost:8082/orders)
Client → Payment Service (http://localhost:8083/payments)

Problems:

- Clients must know all microservice URLs.
- Hard to manage changes (if a service port changes, the client must update).
- No centralized logging/security.



After API Gateway (Single Entry Point)

Client → API Gateway (http://localhost:8080/api)
↳ /users → User Service ↳ /orders → Order Service
↳ /payments → Payment Service Benefits:

- Client only talks to one URL.
- Gateway routes request to correct microservice.
- Security, logging, rate limiting handled at one place.

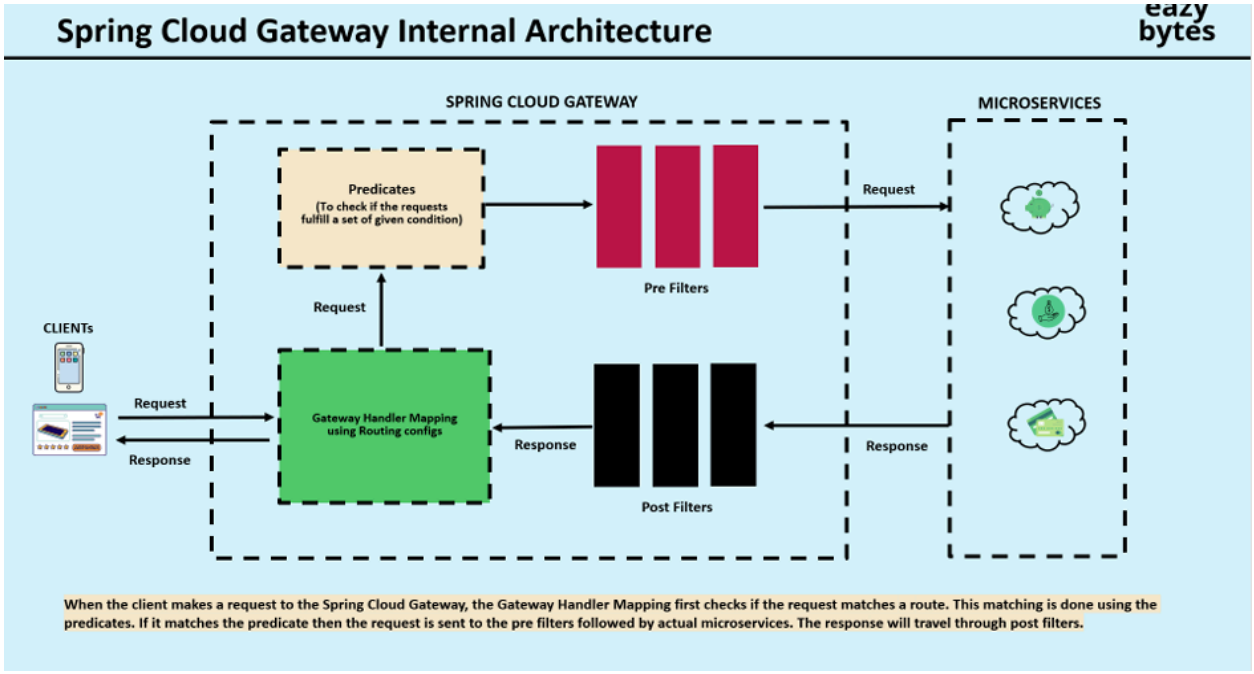
These are real-world, widely used API gateways:
Open-source API Gateways

- Spring Cloud Gateway (Java, Spring Boot) ‘: https://spring.io/projects/spring-cloud-gateway
- Netflix Zuul
- NGINX API Gateway

Cloud Provider API Gateways

- **AWS API Gateway**

- Azure API Management
- Google Cloud API Gateway



Step 1: Create Spring Boot Project

Add the following dependencies

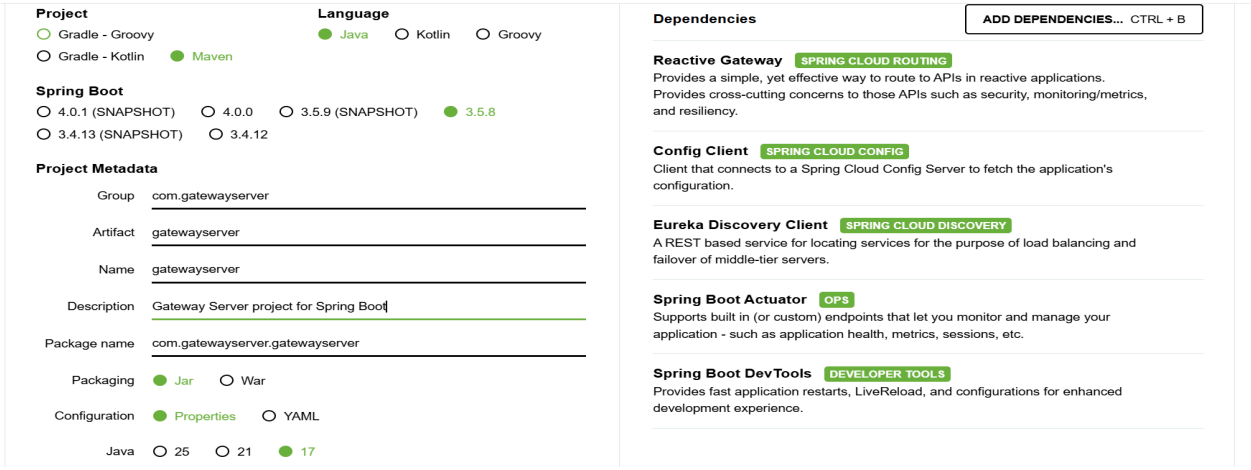
Spring Cloud Gateway (Reactive)

Eureka Discovery Client : So the Gateway can discover microservices dynamically.

Spring Cloud Config Client : Gateway will read config from Config Server (git-based).

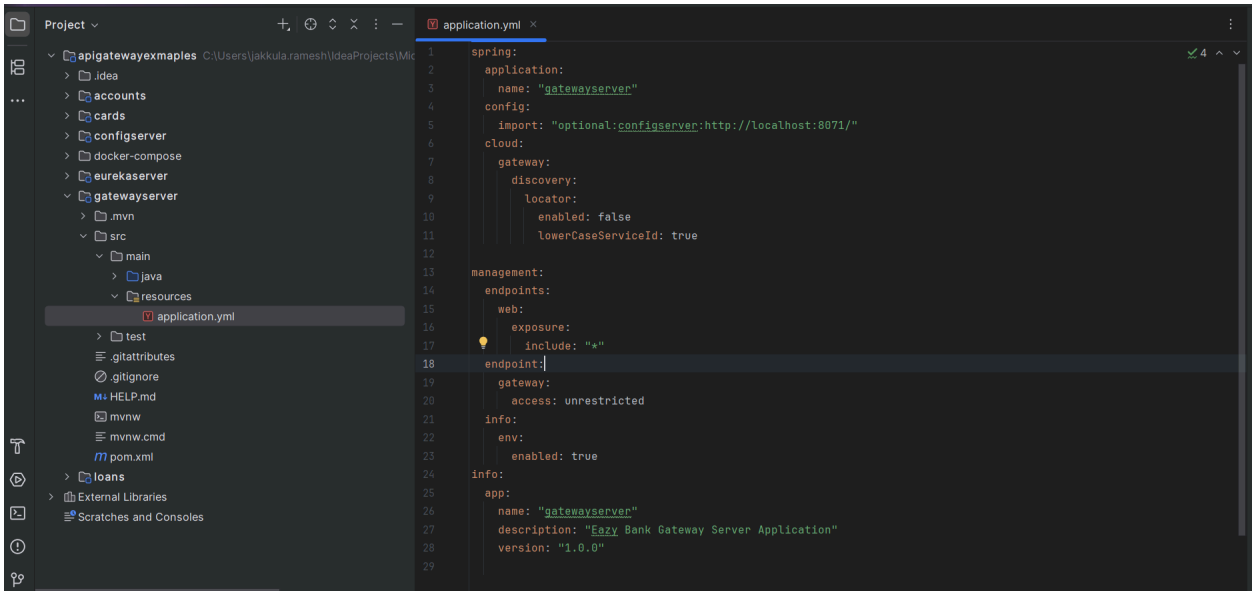
Spring Boot Actuator : For health, metrics, refresh endpoint.

Spring Boot DevTools : For auto restart during development.



Step 2 : Add configuration for API Gateway : This file contains basic startup configuration like:

- service name
- config server URL
- actuator settings
- gateway discovery settings



management:
Expose all actuator endpoints

```yaml
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

management.endpoint.gateway.access: unrestricted
Allows access to sensitive actuator gateway endpoints like /actuator/gateway/routes, /actuator/gateway/globalfilters. Otherwise, they are restricted.

```yaml
management:
  endpoints:
    web:
      exposure:
        include: "*"
  endpoint:
    gateway:
      access: unrestricted
```

spring.cloud.gateway.discovery.locator

```yaml
spring:
  application:
    name: "gatewayserver"
  config:
    import: "optional:configserver:http://localhost:8071/"
  cloud:
    gateway:
      discovery:
        locator:
          enabled: false
          lowerCaseServiceId: true
```

enabled: false

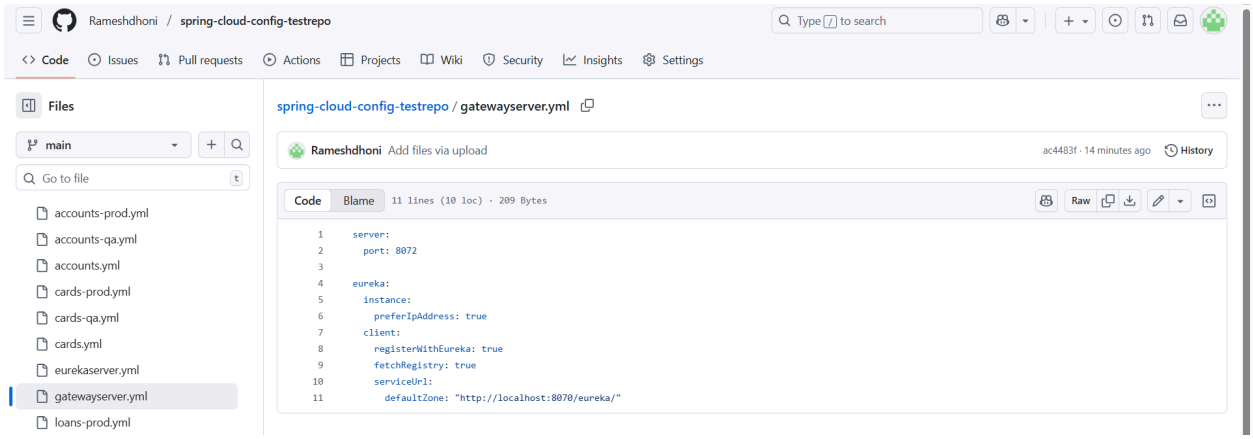
- *If true, Gateway automatically creates routes for each Eureka service.*
- *If false, you will write routes manually in YAML.*

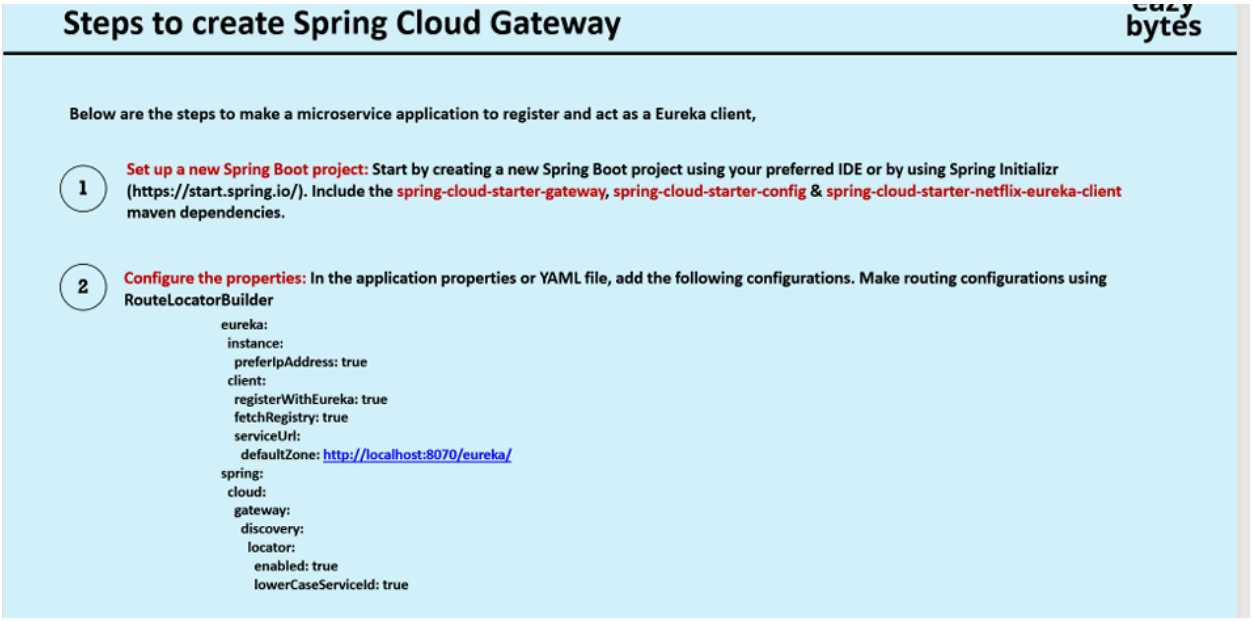
lowerCaseServiceId: true

- *When enabled, service IDs become lowercase.*
  Helpful because URLs become cleaner.

  ACCOUNTS → accounts

  LOANS → loans



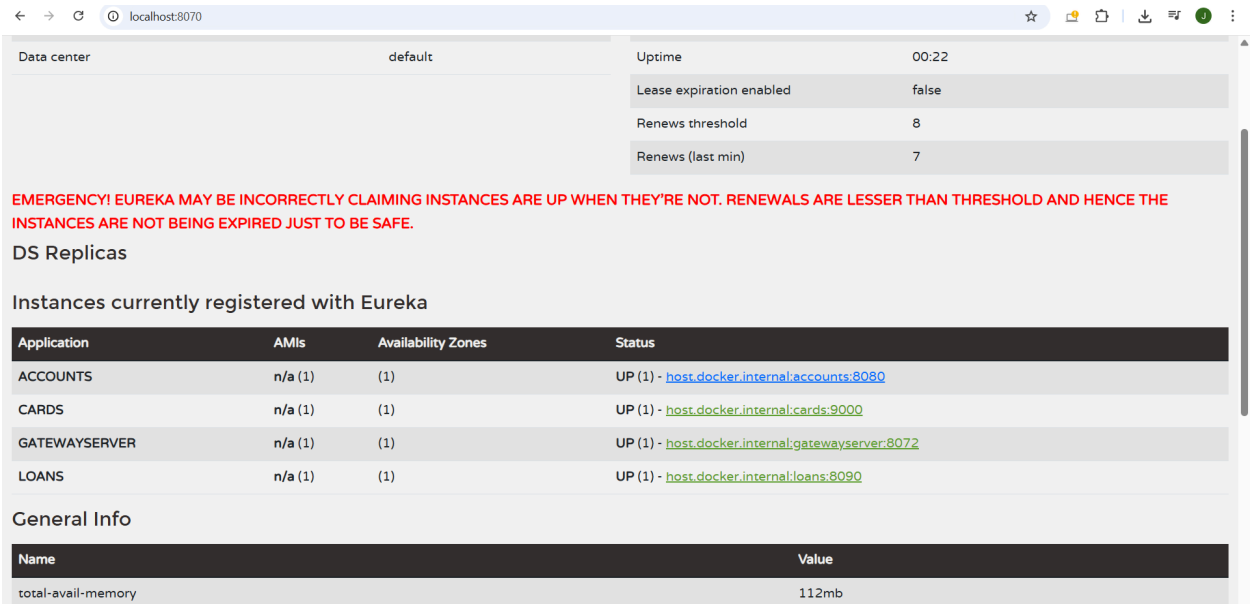


Step 3: Start All Microservices

Start the **Config Server**, then the **Eureka Server**.
Next, start **all microservices**, and lastly start the **API Gateway** so they can:

✔ Load their configuration from Config Server
✔ Register themselves with Eureka
✔ Become available for API Gateway routing

Eureka dashboard



Actuator endpoints of gateway server


```
{
  "_links": {
    "self": {
      "href": "http://localhost:8072/actuator",
      "templated": false
    },
    "beans": {
      "href": "http://localhost:8072/actuator/beans",
      "templated": false
    },
    "caches-cache": {
      "href": "http://localhost:8072/actuator/caches/{cache}",
      "templated": true
    },
    "caches": {
      "href": "http://localhost:8072/actuator/caches",
      "templated": false
    },
    "health-path": {
      "href": "http://localhost:8072/actuator/health/{*path}",
      "templated": true
    },
    "health": {
      "href": "http://localhost:8072/actuator/health",
      "templated": false
    },
    "info": {
      "href": "http://localhost:8072/actuator/info",
      "templated": false
    },
    "conditions": {
      "href": "http://localhost:8072/actuator/conditions",
      "templated": false
    },
    "configprops-prefix": {
      "href": "http://localhost:8072/actuator/configprops/{prefix}",
      "templated": true
    },
    "configprops": {
      "href": "http://localhost:8072/actuator/configprops",
      "templated": false
    },
    "env": {
      "href": "http://localhost:8072/actuator/env",
      "templated": false
```


Copy the url of gateway and add /routes at end see the info related to each microservices .

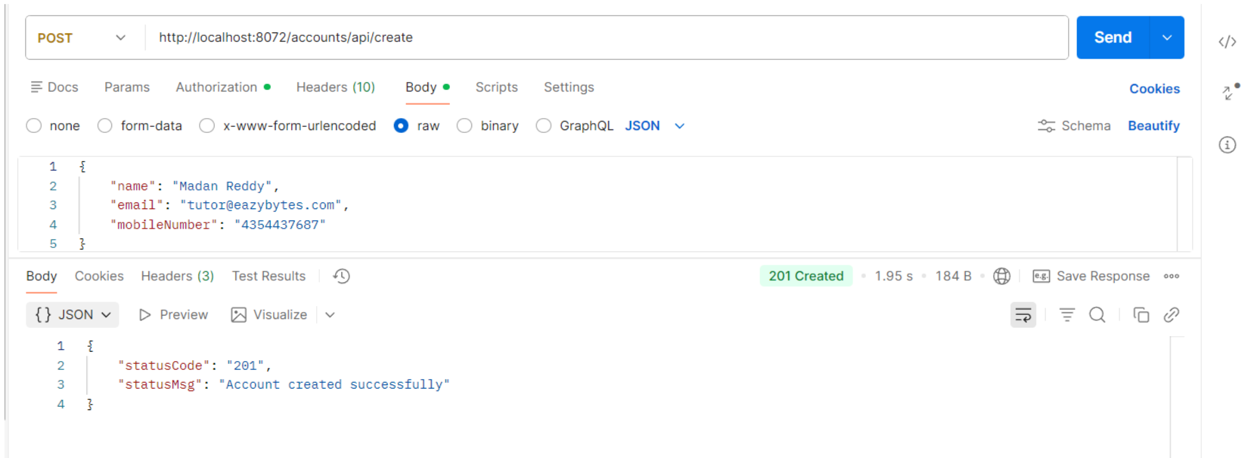

```
    },
    "gateway": {
      "href": "http://localhost:8072/actuator/gateway",
      "templated": false
    }
  }
}
```



```
[
  {
    "predicate": "Paths: [/loans/**], match trailing slash: true",
    "metadata": {
      "management.port": "8090"
    },
    "route_id": "ReactiveCompositeDiscoveryClient_LOANS",
    "filters": [
      "[[RewritePath /loans/?(?\u003Cremaining\u003E.*) = '/${remaining}'], order = 1]"
    ],
    "uri": "lb://LOANS",
    "order": 0
  },
  {
    "predicate": "Paths: [/gatewayserver/**], match trailing slash: true",
    "metadata": {
      "management.port": "8072"
    },
    "route_id": "ReactiveCompositeDiscoveryClient_GATEWAYSERVER",
    "filters": [
      "[[RewritePath /gatewayserver/?(?\u003Cremaining\u003E.*) = '/${remaining}'], order = 1]"
    ],
    "uri": "lb://GATEWAYSERVER",
    "order": 0
  },
  {
    "predicate": "Paths: [/accounts/**], match trailing slash: true",
    "metadata": {
      "management.port": "8080"
    },
    "route_id": "ReactiveCompositeDiscoveryClient_ACCOUNTS",
    "filters": [
      "[[RewritePath /accounts/?(?\u003Cremaining\u003E.*) = '/${remaining}'], order = 1]"
    ],
    "uri": "lb://ACCOUNTS",
    "order": 0
  },
  {
    "predicate": "Paths: [/cards/**], match trailing slash: true",
    "metadata": {
      "management.port": "9000"
    },
    "route_id": "ReactiveCompositeDiscoveryClient_CARDS",
    "filters": [
      "[[RewritePath /cards/?(?\u003Cremaining\u003E.*) = '/${remaining}'], order = 1]"
    ],
```


If a client sends a request whose path starts with /loans/** to the gateway server, the predicate becomes TRUE and the Gateway forwards the request to the microservice whose name in Eureka is **LOANS** via load-balancing (lb://LOANS).

Before forwarding, the RewritePath filter removes the /loans prefix so that the backend service receives only the internal path (/${remaining}).
Ex:



Your API Gateway is working correctly.

- You hit: http://localhost:8072/accounts/api/create
- Gateway matches /accounts/** route
- It rewrites path to /api/create
- Sends request to Accounts service through Eureka
- Accounts service returns **201 – created successfully**

Dynamic /Custom routing

Instead, Gateway discovers services **automatically** from Eureka and creates routes. We can create custom routes either in application.yml or java.
https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway-server-webflux/request-predicates-factories.html or
https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway-server-webflux/fluent-java-routes-api.html

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: accounts_route
          uri: lb://ACCOUNTS
          predicates:
            - Path=/bank/accounts/**
          filters:
            - RewritePath=/bank/accounts/(?&lt;segment&gt;.*), /${segment}
```

Java-based routing using **RouteLocator + RouteLocatorBuilder**.

Every route() defines:

1. Which incoming path to match
2. How to rewrite the path
3. Which microservice (Eureka ID) to forward to

```java
@Bean
public RouteLocator eazyBankRouteConfig(RouteLocatorBuilder routeLocatorBuilder) {
  return routeLocatorBuilder.routes()
              .route(p -> p
                  .path("/eazybank/accounts/**")
                  .filters( f -> f.rewritePath("/eazybank/accounts/(?&lt;segment&gt;.*)","/${segment}"))
                  .uri("lb://ACCOUNTS"))
            .route(p -> p
                .path("/eazybank/loans/**")
                .filters( f -> f.rewritePath("/eazybank/loans/(?&lt;segment&gt;.*)","/${segment}"))
                .uri("lb://LOANS"))
            .route(p -> p
                .path("/eazybank/cards/**")
                .filters( f -> f.rewritePath("/eazybank/cards/(?&lt;segment&gt;.*)","/${segment}"))
                .uri("lb://CARDS")).build();
```

What happens?

Step	Meaning
Path match	When request URL starts with `/eazybank/accounts/**`
Rewrite	Remove `/eazybank/accounts` prefix

Example:

Incoming request: http://localhost:8072/eazybank/accounts/api/create

Rewrite path becomes: /api/create

Forwarded to: ACCOUNTS microservice

Steps to create Spring Cloud Gateway

eazy bytes

3 **Configure the routing config:** Make routing configurations using RouteLocatorBuilder like shown below,

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
        return builder.routes()
                .route(p -> p
                        .path("/eazybank/accounts/**")
                        .filters(f -> f.rewritePath("/eazybank/accounts/(?&lt;segment&gt;.*)","/${segment}")
                                .addResponseHeader("X-Response-Time",new Date().toString()))
                        .uri("lb://ACCOUNTS"))
                .route(p -> p
                        .path("/eazybank/loans/**")
                        .filters(f -> f.rewritePath("/eazybank/loans/(?&lt;segment&gt;.*)","/${segment}")
                                .addResponseHeader("X-Response-Time",new Date().toString()))
                        .uri("lb://LOANS"))
                .route(p -> p
                        .path("/eazybank/cards/**")
                        .filters(f -> f.rewritePath("/eazybank/cards/(?&lt;segment&gt;.*)","/${segment}")
                                .addResponseHeader("X-Response-Time",new Date().toString()))
                        .uri("lb://CARDS")).build();
}
```

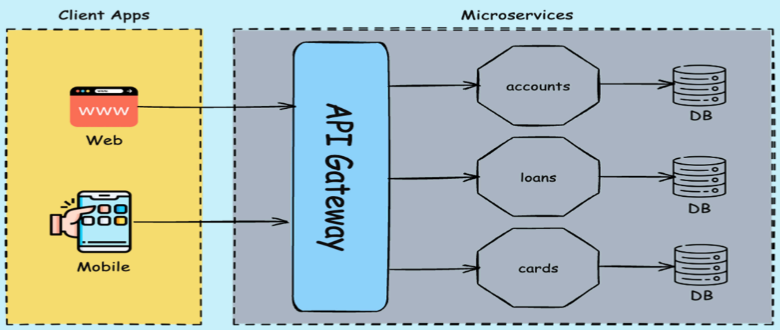
4 **Build and run the application:** Build your project and run it as a Spring Boot application. Invokes the APIs using http://localhost:8072 which is the gateway path.

Design Patterns around api gateway

API Gateway Pattern

eazy bytes

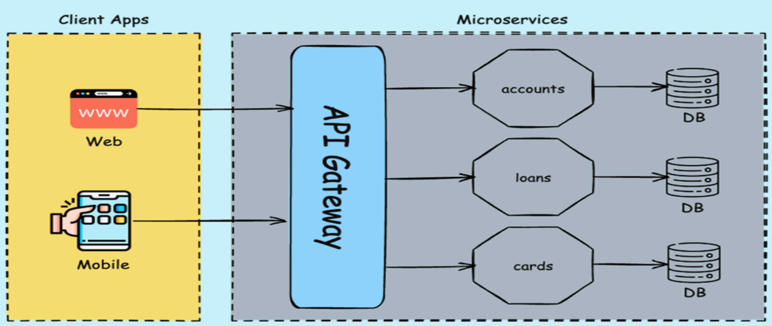
The API Gateway Pattern is a critical architectural component in microservices design, offering a unified entry point for multiple microservices. It acts as a gateway between the external clients (e.g., web apps, mobile apps) and the internal microservices, helping streamline communication, security, and routing. This pattern is essential when managing the complexities of microservice-based applications.



Gateway Routing pattern

eazy bytes

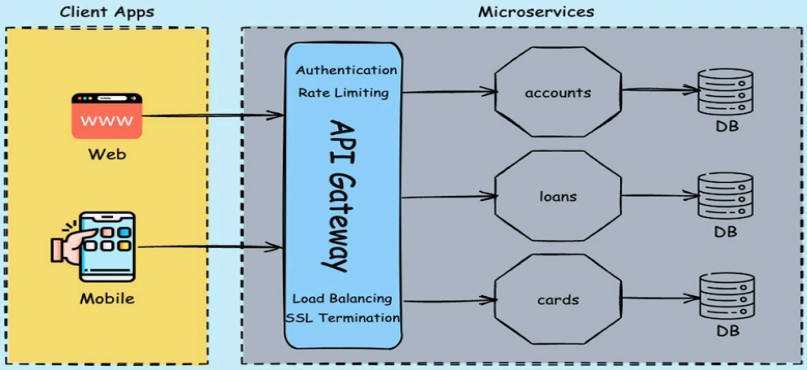
The Gateway Routing pattern is a design pattern used in microservices architectures where an API Gateway routes incoming client requests to the appropriate backend microservices based on various factors like the URL, headers, or request parameters.



Gateway offloading Pattern

eazy bytes

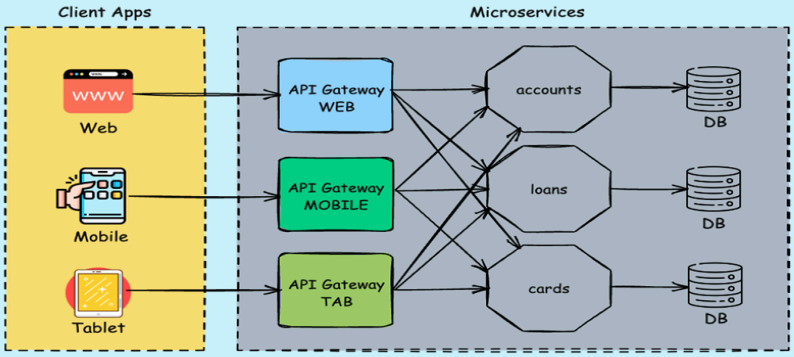
The Gateway Offloading Pattern is an architectural pattern used in microservices to offload certain cross-cutting concerns—such as security, caching, rate limiting, and monitoring—from individual microservices to the API Gateway. This pattern helps centralize and simplify the implementation of these concerns, allowing the microservices to focus solely on business logic.



Backend for Frontend (BFF) Pattern

eazy bytes

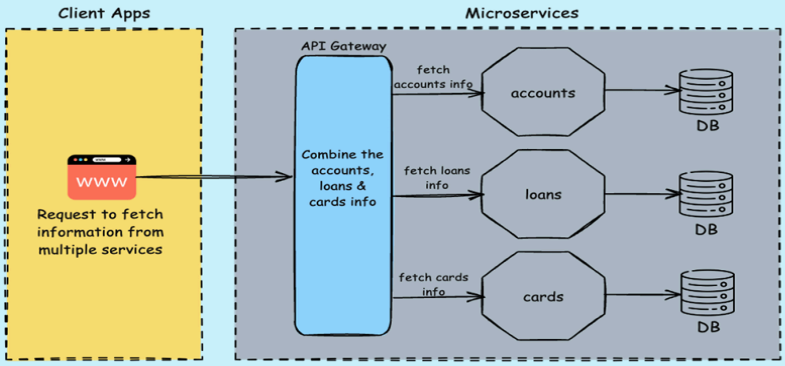
The Backend for Frontend (BFF) Pattern is a design pattern used in microservices architectures where a separate backend service is created for each client type (e.g., web, mobile, tablet). Each frontend (client) has its own specialized backend to optimize communication between the frontend and the microservices, providing a tailored experience for different clients.



Gateway Aggregator/Composition pattern

eazy bytes

In microservices architecture, a Gateway Aggregator or Gateway Composition pattern is used when a request from a client needs to retrieve or process data from multiple backend microservices. Instead of having the client make multiple calls to various microservices, the API Gateway consolidates the requests into a single response.



Resiliency in Microservices

Resiliency in microservices means **the ability of a microservice system to keep working even when some of its services fail, slow down, or face unexpected problems.**
It ensures that one service failure does **not** crash the entire application.

Failures will happen (network down, service slow, service crash). Resiliency patterns help your system **remain stable** instead of crashing.

Why Resilience Is Needed?

Because microservices run on:

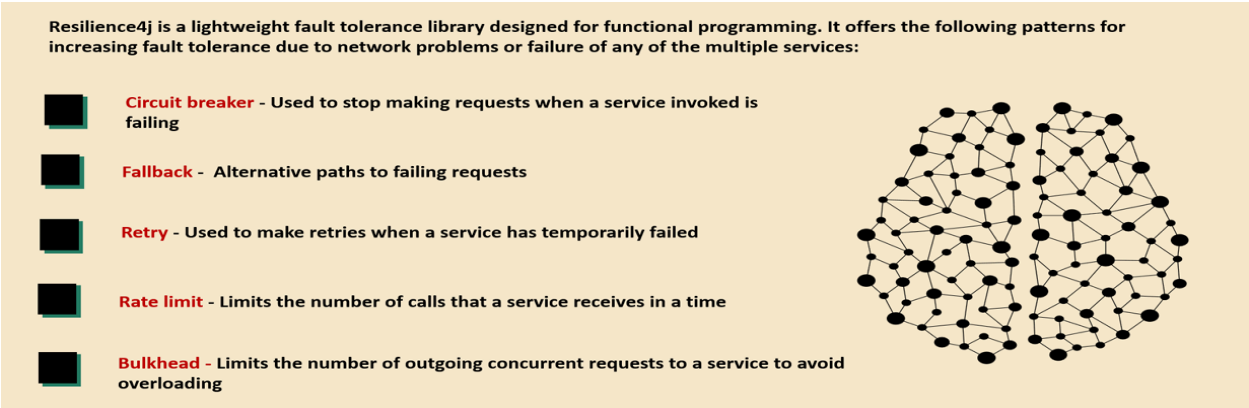
- multiple servers
- multiple networks
- talk to each other frequently

So failure chances are high. Resiliency ensures your app stays UP even when some services are DOWN. There are several patterns used to build resilient applications.

In the Java ecosystem, **Hystrix**, a fault-tolerance library developed by Netflix, was once widely used for implementing these resilience patterns. However, Hystrix entered **maintenance mode in 2018** and is no longer actively developed.

To fill this gap, **Resilience4j** https://resilience4j.readme.io/docs/getting-started emerged as the modern and lightweight alternative. It is now the **standard library** for resiliency in microservices. It provides a set of resilience features :

- Circuit Breaker
- Retry
- Rate Limiter
- Fallback
- Bulkhead
- Timeouts



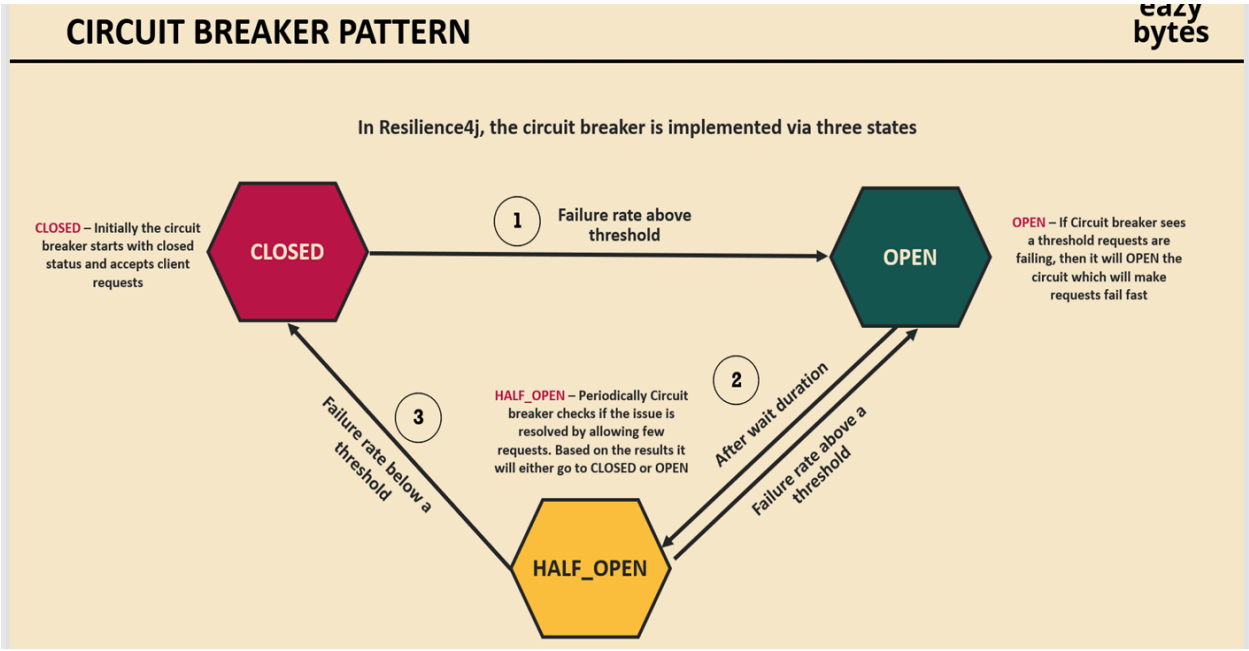
i) Circuit Breaker Pattern

The **Circuit Breaker pattern** is a software design pattern used in distributed systems to **detect failures and prevent an application from repeatedly calling a failing service.**

If a service is slow, unresponsive, or down, the **circuit breaker opens** and stops sending requests to that service. This **prevents cascading failures** in the system.

Benefits of Using Circuit Breakers

- **Prevents Cascading Failures:** Isolates failures so that one service failure does not affect others.
- **Graceful Degradation:** Provides fallback responses to maintain a good user experience.



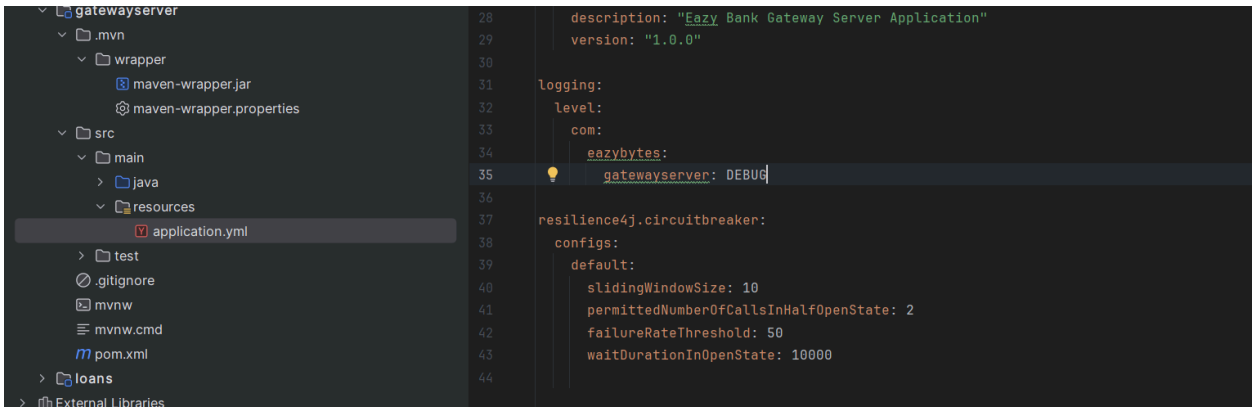
How to Integrate Circuit Breaker in Spring Boot (Using Resilience4j)

Circuit Breaker pattern in gateway server

1. **Add Dependency in API Gateway pom.xml**

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-reactor-resilience4j</artifactId>
</dependency>
```

2. **Add Configuration in application.yml**



✔ **slidingWindowSize: 10**

- Circuit Breaker observes the **last 10 requests**.
- After these 10 requests, it **calculates failure percentage**.
- Based on this, it decides **open or closed** state.

👉 *It must receive 10 calls before it decides.*

✔ **permittedNumberOfCallsInHalfOpenState: 2**

- When the circuit breaker moves from **OPEN → HALF-OPEN**, it allows **only 2 test requests** to check if the service is healthy.

👉 *If these 2 calls succeed → move to CLOSED*
👉 *If any fails → go back to OPEN*

✔ **failureRateThreshold: 50**

- If **50% or more** of the monitored calls fail (out of 10 sliding window calls), the circuit breaker **opens**.

Example:

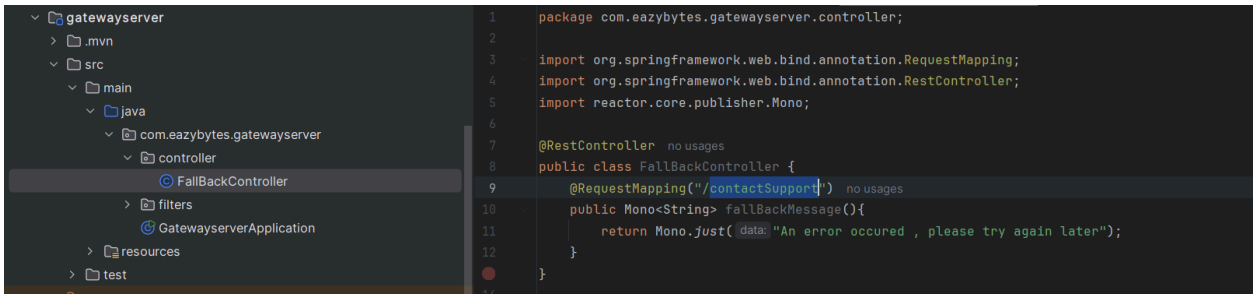
- 10 calls → 5 fail → **OPEN**
- 10 calls → 4 fail → **stay CLOSED**

✔ **waitDurationInOpenState: 10000 (10 seconds)**

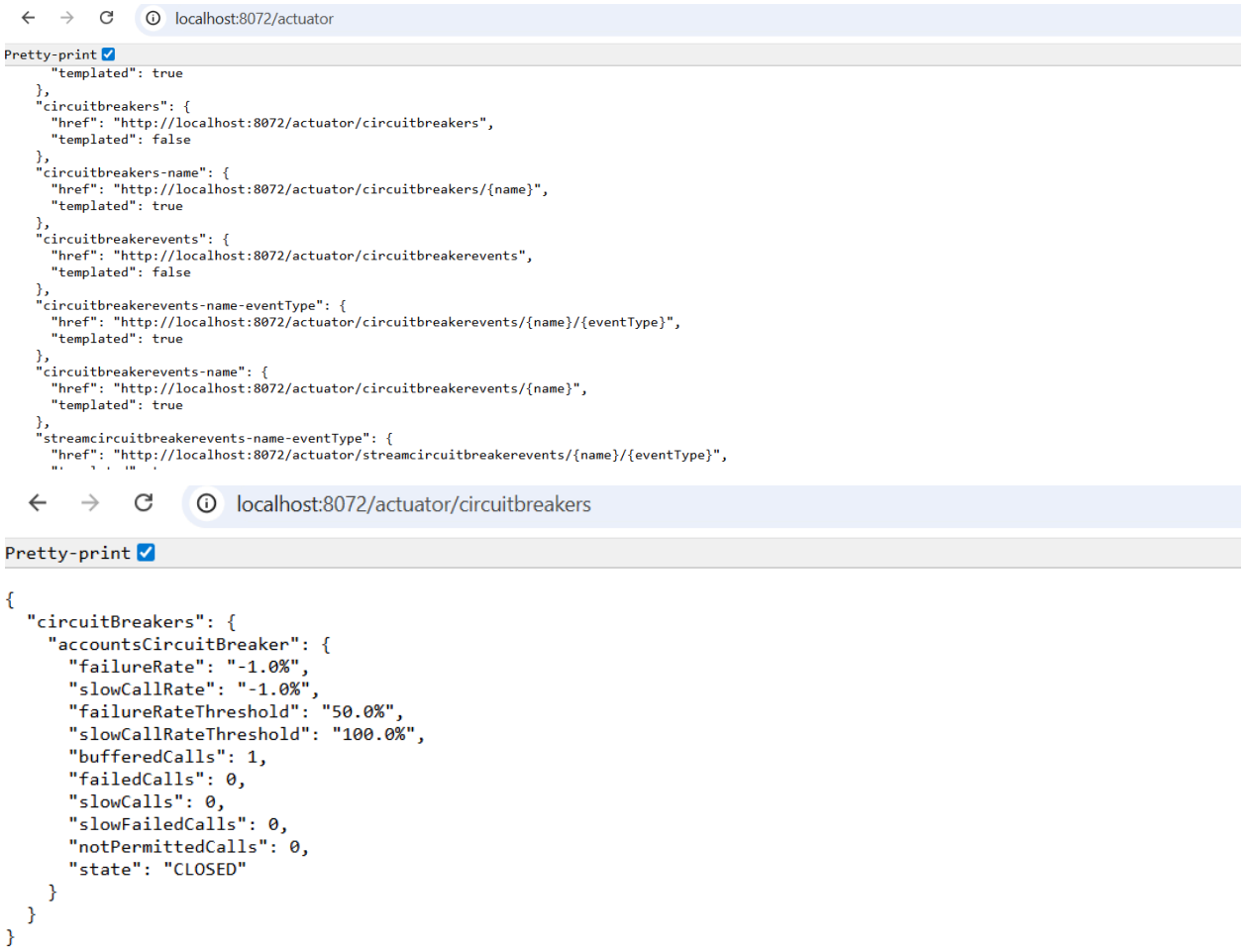
- Once the circuit breaker becomes **OPEN**,
- It stays OPEN for **10 seconds**.
- During this time, **all requests are blocked** and fallback is executed.
- After 10 seconds → transitions to **HALF-OPEN** → allows 2 test requests.

3. **Create Method Using Circuit Breaker + Fallback**

```
@Bean  no usages
public RouteLocator eazyBankRouteConfig(RouteLocatorBuilder routeLocatorBuilder) {
    return routeLocatorBuilder.routes()
                    .route( PredicateSpec p -> p
                            .path( ...patterns: "/eazybank/accounts/**") BooleanSpec
                            .filters( GatewayFilterSpec f -> f.rewritePath( regex: "/eazybank/accounts/(?<segment>.*)", replacement: "/${segment}")
                                    .addResponseHeader( headerName: "X-Response-Time", LocalDateTime.now().toString()).
                                    circuitBreaker( Config config->config.setName("accountsCircuitBreaker").
                                            setFallbackUri("forwad:/contactSupport"))) UriSpec
                            .uri("lb://ACCOUNTS"))
```



Step 4: Start application and Actuator for Circuit Breaker Monitoring
Circuit Breakers list: http://localhost:8072/actuator/circuitbreakers
Events for specific Circuit Breaker:
http://localhost:8072/actuator/circuitbreakerevents?name=loanCB



will see events like:

- SUCCESS
- ERROR
- FAILURE_RATE_EXCEEDED
- STATE_TRANSITION (CLOSED → OPEN → HALF_OPEN → CLOSED)
- NOT_PERMITTED

localhost:8072/actuator/circuitbreakerevents?name=accountsCircuitBreaker

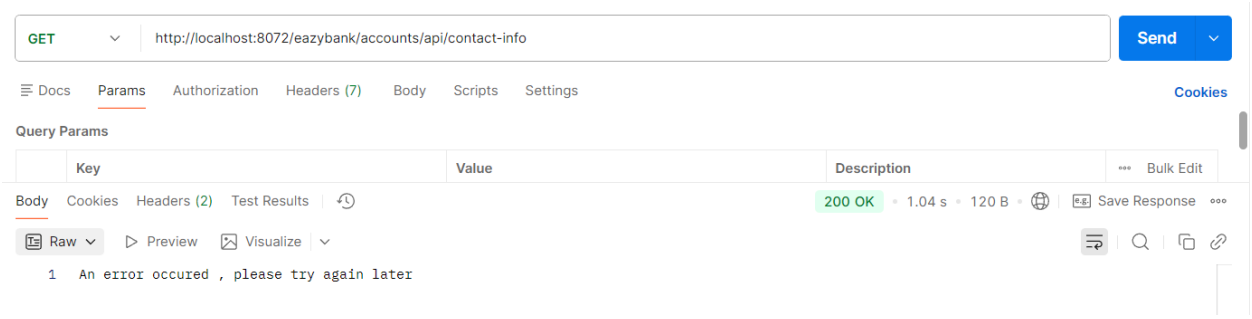
Pretty-print


```
{
  "circuitBreakerEvents": [
    {
      "circuitBreakerName": "accountsCircuitBreaker",
      "type": "SUCCESS",
      "creationTime": "2025-11-25T15:58:26.512861600+05:30[Asia/Calcutta]",
      "errorMessage": null,
      "durationInMs": 293,
      "stateTransition": null
    },
    {
      "circuitBreakerName": "accountsCircuitBreaker",
      "type": "SUCCESS",
      "creationTime": "2025-11-25T16:01:47.026475700+05:30[Asia/Calcutta]",
      "errorMessage": null,
      "durationInMs": 19,


    {
      "circuitBreakerName": "accountsCircuitBreaker",
      "type": "NOT_PERMITTED",
      "creationTime": "2025-11-25T16:09:30.514647+05:30[Asia/Calcutta]",
      "errorMessage": null,
      "durationInMs": null,
      "stateTransition": null
    },
    {
      "circuitBreakerName": "accountsCircuitBreaker",
      "type": "NOT_PERMITTED",
      "creationTime": "2025-11-25T16:09:31.449518900+05:30[Asia/Calcutta]",
      "errorMessage": null,
      "durationInMs": null,
      "stateTransition": null
    },
    {
      "circuitBreakerName": "accountsCircuitBreaker",
      "type": "STATE_TRANSITION",
      "creationTime": "2025-11-25T16:09:32.354920100+05:30[Asia/Calcutta]",
      "errorMessage": null,
      "durationInMs": null,
      "stateTransition": "OPEN_TO_HALF_OPEN"
    },
```


If you keep a **breakpoint** in your service method, the request will be held (delayed). This may cause **timeouts** or **NOT_PERMITTED** events in the circuit breaker.

```
247     )
248     @GetMapping("/contact-info")
        public ResponseEntity<AccountsContactInfoDto> getContactInfo() {
250         return ResponseEntity
251                 .status(HttpStatus.OK)
252                 .body(accountsContactInfoDto);
253     }
```

This is the response returned when Circuit Breaker is **OPEN** or the service is slow/down.



CIRCUIT BREAKER PATTERN

eazy bytes

Below are the steps to build a circuit breaker pattern using Spring Cloud Gateway filter,

1. **Add maven dependency:** Add spring-cloud-starter-circuitbreaker-reactor-resilience4j maven dependency inside pom.xml

2. **Add circuit breaker filter:** Inside the method where we are creating a bean of RouteLocator, add a filter of circuit breaker like highlighted below and create a REST API handling the fallback uri `/contactSupport`

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
    return builder.routes()
                .route(p -> p.path("/eazybank/accounts/**")
                        .filters(f -> f.rewritePath("/eazybank/accounts/(?&lt;segment&gt;.*)","/${segment}")
                        .addResponseHeader("X-Response-Time",new Date().toString())
                        .circuitBreaker(config -> config.setName("accountsCircuitBreaker")
                        .setFallbackUri("forward:/contactSupport")))
                        .uri("lb://ACCOUNTS")).build();
}
```

3. **Add properties:** Add the below properties inside the application.yml file,

```
resilience4j.circuitbreaker:
  configs:
    default:
      slidingWindowSize: 10
      permittedNumberOfCallsInHalfOpenState: 2
      failureRateThreshold: 50
      waitDurationInOpenState: 10000
```

Circuit Breaker pattern individual in microservices

Circuit Breaker + Feign Client in Microservices (Resilience4j)
https://docs.spring.io/spring-cloud-openfeign/reference/spring-cloud-openfeign.html

This is for scenarios like:

Accounts → Loans
Accounts → Cards
via **Feign Client**, with **Circuit Breaker protection**.

1. Add Dependencies (in Accounts microservice)

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-reactor-resilience4j</artifactId>
</dependency>
```

If Spring Cloud CircuitBreaker is on the classpath and spring.cloud.openfeign.circuitbreaker.enabled=true, Feign will wrap all methods with a circuit breaker.

3 **Add properties:** Add the below properties inside the application.yml file,

```
spring:
 cloud:
  openfeign:
   circuitbreaker:
    enabled: true
resilience4j.circuitbreaker:
    configs:
      default:
        slidingWindowSize: 5
        failureRateThreshold: 50
        waitDurationInOpenState: 10000
        permittedNumberOfCallsInHalfOpenState: 2
```

2. Enable Feign Client and Create Feign Client Interface

In your main class:

```
@SpringBootApplication
@EnableFeignClients
public class AccountsApplication {
}
```

Create Feign Client Interface

```
@FeignClient(name = "LOANS", fallback = LoansFallback.class)
public interface LoansClient {

   @GetMapping("/loans/{id}")
   String getLoanDetails(@PathVariable("id") long id);
}
```

3. Create Fallback Class for Feign Client

```
@Component
public class LoansFallback implements LoansClient {

   @Override
   public String getLoanDetails(long id) {
      return "Loan service is unavailable. Please try again later.";
   }
}
```

✔ This fallback will be triggered when Feign client fails
✔ Only Feign fallback does NOT manage OPEN / HALF-OPEN states
👉 For real circuit breaker behavior, we add **Resilience4j** as well.

Below are the steps to build a circuit breaker pattern using normal Spring Boot service,

1. Add maven dependency: Add spring-cloud-starter-circuitbreaker-resilience4j maven dependency inside pom.xml

2. Add circuit breaker related changes in Feign Client interfaces like shown below:

```
@FeignClient(name= "cards", fallback = CardsFallback.class)
public interface CardsFeignClient {

   @GetMapping(value = "/api/fetch",consumes = "application/json")
   public ResponseEntity<CardsDto> fetchCardDetails(@RequestHeader("eazybank-correlation-id")
                                    String correlationId, @RequestParam String mobileNumber);

}
```

```
@Component
public class CardsFallback implements CardsFeignClient{
   @Override
   public ResponseEntity<CardsDto> fetchCardDetails(String correlationId, String mobileNumber){
      return null;
   }
}
```

ii) Timeouts pattern

https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway-server-webflux/http-timeouts-configuration.html

If a service does not respond within a fixed time, stop waiting and fail the request.

Timeout Pattern is a resiliency pattern in microservices where a service stops waiting for a response from another service after a fixed time limit.

If the response does not arrive within that time, the request is automatically aborted to prevent blocked threads, slowdowns, and cascading failures.

Ex : Set timeout to **2 seconds**.

If Loan service takes 5 seconds to respond:

❌ Timeout happens
❌ Request stops
⚠️ Circuit Breaker counts it as a failure
✔ Fallback message is returned

global http timeouts example

```yaml
spring:
  cloud:
    gateway:
      httpclient:
        connect-timeout: 1000
        response-timeout: 5s
```

per-route http timeouts configuration via configuration

```yaml
      - id: per_route_timeouts
        uri: https://example.org
        predicates:
          - name: Path
            args:
              pattern: /delay/{timeout}
        metadata:
          response-timeout: 200
          connect-timeout: 200
```

per-route timeouts configuration using Java DSL

```java
      @Bean
      public RouteLocator customRouteLocator(RouteLocatorBuilder routeBuilder){
         return routeBuilder.routes()
               .route("test1", r -> {
                  return r.host("*.somehost.org").and().path("/somepath")
                        .filters(f -> f.addRequestHeader("header1", "header-value-1"))
                        .uri("http://someuri")
                        .metadata(RESPONSE_TIMEOUT_ATTR, 200)
                        .metadata(CONNECT_TIMEOUT_ATTR, 200);
               })
               .build();
      }
```

A per-route `response-timeout` with a negative value will disable the global `response-timeout` value.

```
      - id: per_route_timeouts
        uri: https://example.org
        predicates:
          - name: Path
            args:
              pattern: /delay/{timeout}
        metadata:
          response-timeout: -1
```

iii) Retry Pattern

The **Retry Pattern** means **trying the same request again** when a service fails temporarily. If a request fails due to issues like **network glitches or temporary service downtime**, the system will **automatically retry it a few times** before giving up.

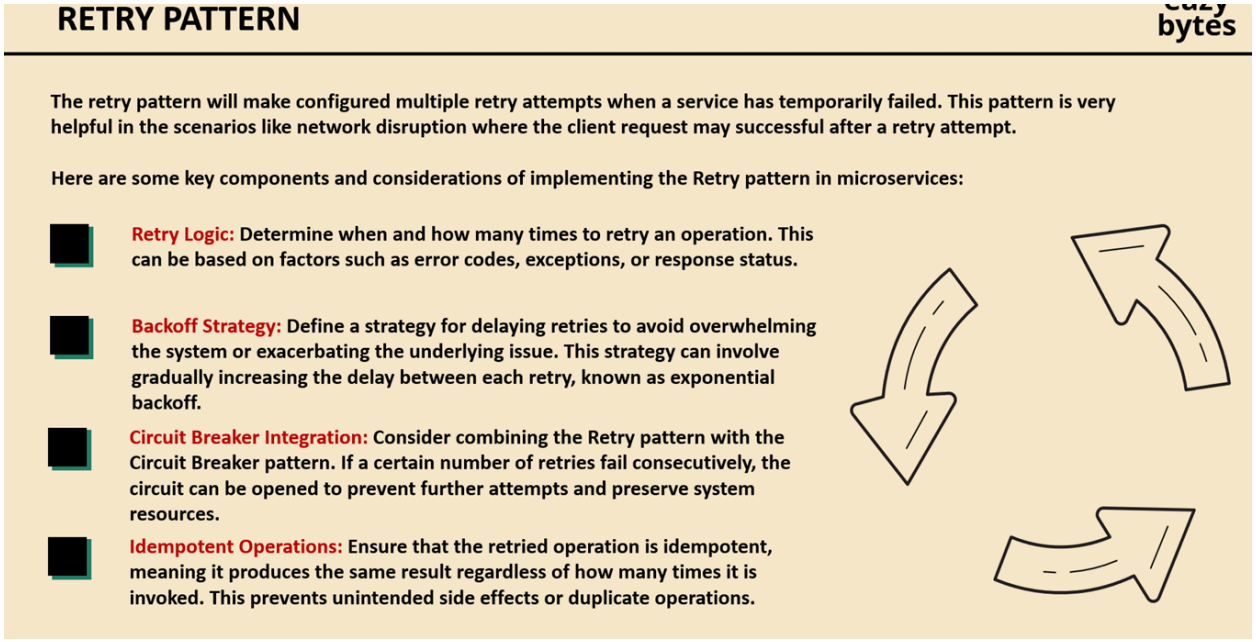
It helps improve the reliability and stability of the system by reducing the impact of transient errors. https://resilience4j.readme.io/docs/getting-started-3

🧠 **Why we use Retry?**

Because sometimes services fail for **small, temporary reasons**, like:

- Network delay
- Service takes time to start
- Service gives timeout
- Server responds slowly

Instead of failing immediately, Retry says: 👉 **"Wait… try again."**



Some operations **can be retried safely** (idempotent), others **cannot**.

✔ **Safe (idempotent)**

- GET (fetch data)

❌ **Not Safe (non-idempotent)**

- POST (creating a new order → retrying may create duplicate orders)

- Payment processing (retrying can charge twice)

If the operation is **idempotent**, retrying is safe

If the operation is **not idempotent**, retry may cause **duplicate data, double charges, repeated inserts**

Retry pattern in gateway server

Below are the steps to build a retry pattern using Spring Cloud Gateway filter,

1. **Add Retry filter: Inside the method where we are creating a bean of RouteLocator, add a filter of retry like highlighted below,**

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
  return builder.routes()
      .route(p -> p.path("/eazybank/loans/**")
          .filters(f -> f.rewritePath("/eazybank/loans/(?&lt;segment&gt;.*)","/${segment}")
              .addResponseHeader("X-Response-Time",new Date().toString())
              .retry(retryConfig -> retryConfig.setRetries(3).setMethods(HttpMethod.GET)
                  .setBackoff(Duration.ofMillis(100),Duration.ofMillis(1000),2,true)))
          .uri("lb://LOANS")).build();
}
```

setRetries(3) → try 3 more times if request fails

setMethods(GET) → retry only GET calls

setBackoff(100ms, 1000ms, 2, true)

- 100ms → first delay
- 1000ms → max delay
- 2 → exponential factor
- true → enable jitter (random small variation to avoid spikes)

100ms → First wait time

After first failure, wait **100 milliseconds** before retrying.

1000ms → Maximum wait time

The wait time will increase, but **never more than 1000 ms** (1 second).

2 → Exponential factor

Each retry waits **double** the previous delay.

Ex:

Fails → retry #1
Delay: 100 ms

Fails → retry #2
Delay: 200 ms

Fails → retry #3
Delay: 400 ms

Let's see how we use retry pattern in individual microservices

Below are the steps to build a retry pattern using normal Spring Boot service,

1 **Add Retry pattern annotations:** Choose a method and mention retry pattern related annotation along with the below configs. Post that create a fallback method matching the same method signature like we discussed inside the course,

```
@Retry(name = "getBuildInfo", fallbackMethod = "getBuildInfoFallBack")
@GetMapping("/build-info")
public ResponseEntity<String> getBuildInfo() {

}

private ResponseEntity<String> getBuildInfoFallBack(Throwable t) {

}
```

2 **Add properties:** Add the below properties inside the application.yml file,

```
resilience4j.retry:
  configs:
    default:
      maxRetryAttempts: 3
      waitDuration: 500
      enableExponentialBackoff: true
      exponentialBackoffMultiplier: 2
      retryExceptions:
       - java.util.concurrent.TimeoutException
      ignoreExceptions:
       - java.lang.NullPointerException
```

Ex: 1. Adding @Retry on one if a method in the controller class.

Fallback method **must be inside the same class.**

It must accept the same arguments as your retry method .

Plus one extra argument → Throwable

Main method with Retry

```
@Retry(name = "retryLoansAPI", fallbackMethod = "fallbackLoans")
public String getLoanDetails(String customerId) {
    return loansClient.getLoans(customerId);
}
```

Matching Fallback Method

```
public String fallbackLoans(String customerId, Throwable t) {
    return "Loans service is temporarily unavailable. Please try again later.";
}
```

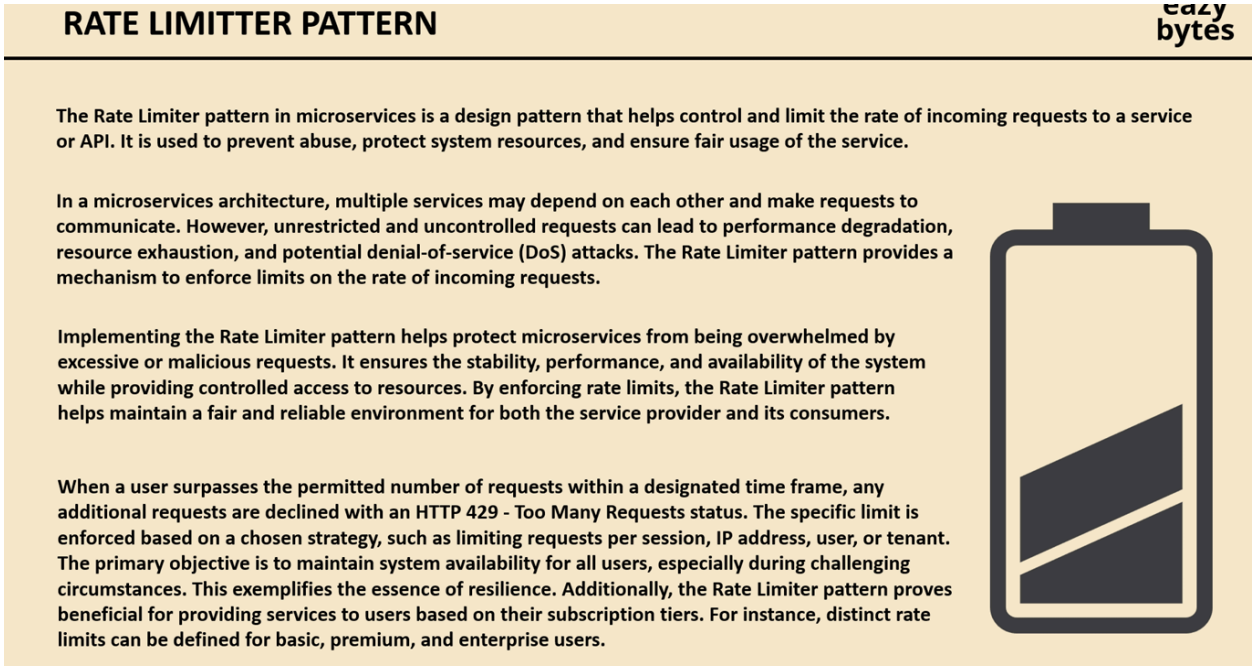
iv) Rate Limiter pattern

Rate Limit Pattern is a resiliency pattern in microservices used to **control how many requests a client or service can make within a specific time period**.
https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway-server-webflux/gatewayfilter-factories/requestratelimiter-factory.html#redis-ratelimiter

It sets a **maximum number of requests** allowed per second/minute/hour.

Example : If the user exceeds the allowed number of requests, the system rejects extra calls with HTTP 429 – Too Many Requests.

- "Only 100 requests per minute allowed per user."
- If the 101st request comes → API returns 429 Too Many Requests.



Rate limiter pattern in gateway server

Redis Rate Limiter, Apache Bench (ab) load testing, Docker Redis

1. Run Redis Using Docker (Prepare environment — run Redis)

Spring Cloud Gateway's rate limiter needs Redis.

Run Redis with one simple command: docker run -d --name redis -p 6379:6379 redis

Now Redis is ready to store:

- tokens for rate limiting
- counters
- user request keys

2. Add Maven dependencies (pom.xml) and Configure Redis connection (application.yml)

```yaml
spring:
  redis:
    host: localhost
    port: 6379
```

Start Your Gateway Application

Your Gateway has:

✔ **Rate Limiter**
✔ **KeyResolver**
✔ **Retry Filter** (if added)

Rate Limiter (Token Bucket)

```
@Bean
public RedisRateLimiter redisRateLimiter() {
   return new RedisRateLimiter(1, 1, 1);  // replenishRate=1, burst=1
}
```

- Only **1 request per second** is allowed per user.
- If the user sends >1 request → **HTTP 429** returned.

3. **How User Request Works with Rate Limiter**

Gateway extracts key using KeyResolver
Example: `exchange.getRequest().getHeaders().getFirst("user")`

If no header → user = "anonymous"

Based on "user", Gateway checks Redis bucket:

- If token exists → allow request
- If token not available → **429 Too Many Requests**

4) Test with Apache Bench (ab)

Example:

```
# 20 requests, concurrency 5, include header used by KeyResolver
ab -n 20 -c 5 -H "user: ramesh" http://localhost:8080/eazybank/cards/123
```

Expectations:

- If replenishRate=1 and burstCapacity=2:
  - First 1–2 quick requests may pass (burst).
  - Subsequent requests within the same second -> 429 Too Many Requests.
- Over time (1 token/second) requests are allowed again.

Below are the steps to build a rate limitter pattern using Spring Cloud Gateway filter,

1 **Add maven dependency:** **Add spring-boot-starter-data-redis-reactive maven dependency inside pom.xml and make sure a redis container started. Mention redis connection details inside the application.yml file**

2 **Add rate limitter filter:** **Inside the method where we are creating a bean of RouteLocator, add a filter of rate limitter like highlighted below and creating supporting beans of RedisRateLimiter and KeyResolver**

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
    return builder.routes()
            .route(p -> p.path("/eazybank/cards/**")
                    .filters(f -> f.rewritePath("/eazybank/cards/(?&lt;segment&gt;.*)","/${segment}")
                        .addResponseHeader("X-Response-Time",new Date().toString())
                        .requestRateLimiter(config ->
                          config.setRateLimiter(redisRateLimiter()).setKeyResolver(userKeyResolver())))
                    .uri("lb://CARDS")).build();
}

@Bean
public RedisRateLimiter redisRateLimiter() {
    return new RedisRateLimiter(1, 1, 1);
}

@Bean
KeyResolver userKeyResolver() {
    return exchange -> Mono.justOrEmpty(exchange.getRequest().getHeaders().getFirst("user"))
            .defaultIfEmpty("anonymous");
}
```

Let's see how we use **Rate limiter** in individual microservices

Below are the steps to build a rate limiter pattern using normal Spring Boot service,

1 **Add Rate limitter pattern annotations:** **Choose a method and mention rate limitter pattern related annotation along with the below configs. Post that create a fallback method matching the same method signature like we discussed inside the course,**

```
@RateLimiter(name = "getJavaVersion", fallbackMethod = "getJavaVersionFallback")
@GetMapping("/java-version")
public ResponseEntity<String> getJavaVersion() {

}

private ResponseEntity<String> getJavaVersionFallback(Throwable t) {

}
```

2 **Add properties:** **Add the below properties inside the application.yml file,**

```
resilience4j.ratelimiter:
  configs:
    default:
      timeoutDuration: 5000
      limitRefreshPeriod: 5000
      limitForPeriod: 1
```

Two Common Ways to Add Rate Limiting in a Microservice

1️⃣ Using Resilience4j RateLimiter

2️⃣ Using Bucket4j (popular & powerful)

Using Resilience4j RateLimiter (Most common)

Step-1: Add Dependency (in microservice's pom.xml)

Step-2: Add configuration in application.yml

```
resilience4j:
  ratelimiter:
    instances:
      userServiceLimiter:
        limit-for-period: 5      # allow only 5 calls
        limit-refresh-period: 10s # every 10 seconds bucket resets
        timeout-duration: 0      # do not wait
```

📌 **Step-3: Apply RateLimiter on a REST method**

```
@RestController
public class UserController {

    @RateLimiter(name = "userServiceLimiter", fallbackMethod = "rateLimiterFallback")
    @GetMapping("/users")
    public String getUsers() {
        return "Users fetched!";
    }

    public String rateLimiterFallback(Throwable t) {
        return "Too many requests — please try after some time!";
    }
}
```

✔ This protects **only this microservice**
✔ Blocks requests when limit is crossed
✔ Sends fallback message

v) Bulkhead pattern

Bulkhead Pattern is a resiliency pattern where you **divide your system into isolated compartments** so that if one part fails or becomes slow, it **does NOT affect the others**.

Bulkhead Pattern isolates resources (threads, connections, pools) for each service or function so that a failure in one area does not impact the entire system.
https://resilience4j.readme.io/docs/bulkhead

Imagine a ship.

A ship has *separate compartments*.
If one compartment gets filled with water,
only that part sinks — the whole ship does NOT sink.

Imagine you have **3 features** in your application:

1. **Login Service**
2. **Order Service**
3. **Payment Service**

❌ Without Bulkhead

All three services share the **same thread pool / same resources**.

If **Payment Service** becomes slow or gets heavy traffic:

- it consumes all threads
- Login and Order also become slow
- system hangs

This is a system-wide failure.

✅ With Bulkhead

We separate resources:

- Login → **Thread Pool A**
- Orders → **Thread Pool B**
- Payments → **Thread Pool C**

So if **Payment** becomes slow:

- Only **Thread Pool C** is busy

- Login and Order **still work normally**
- No system crash
- No cascading failure

This is the Bulkhead **Pattern**.

Aspect Order in Resilience4
https://resilience4j.readme.io/docs/getting-started-3#aspect-order

- **Bulkhead** runs first
- then **TimeLimiter**
- then **RateLimiter**
- then **CircuitBreaker**
- **Retry runs last**,

  Bulkhead → isolate threads
  TimeLimiter → apply timeouts
  RateLimiter → limit requests
  CircuitBreaker → detect failures
  Retry → if all above fail, then retry

If you need a different order, you must use the functional chaining style instead of the Spring annotations style or explicitly set aspect order using the following properties:

Text

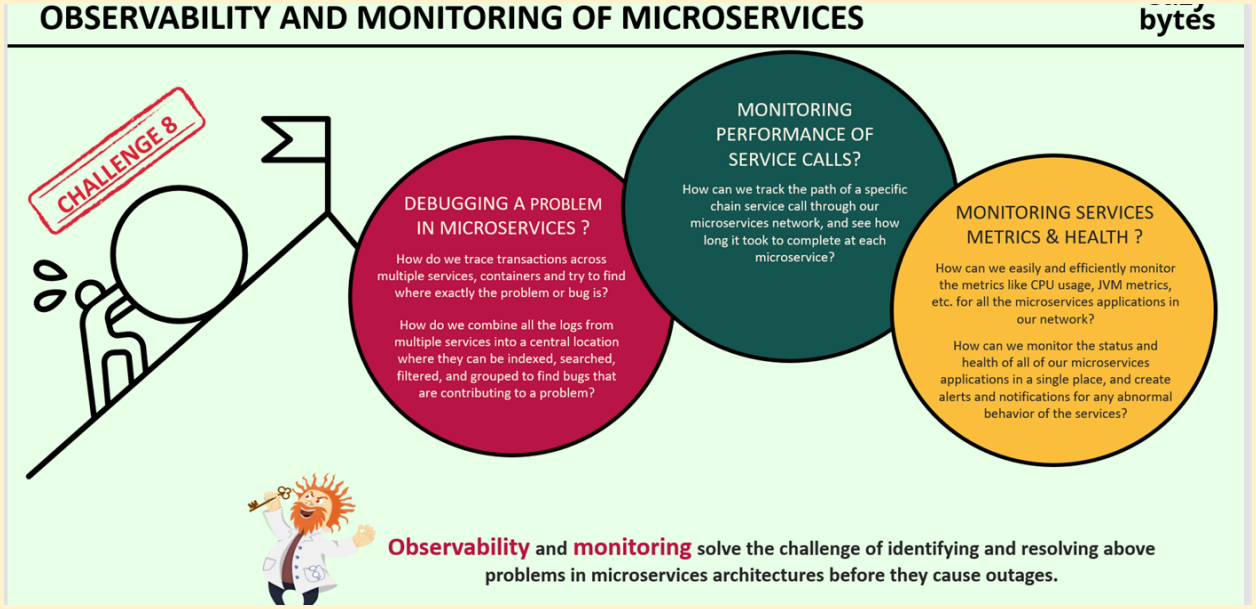
```
- resilience4j.retry.retryAspectOrder
- resilience4j.circuitbreaker.circuitBreakerAspectOrder
- resilience4j.ratelimiter.rateLimiterAspectOrder
- resilience4j.timelimiter.timeLimiterAspectOrder
- resilience4j.bulkhead.bulkheadAspectOrder
```

For example - to make Circuit Breaker starts after Retry finish its work you must set `retryAspectOrder` property to greater value than `circuitBreakerAspectOrder` value (the higher value = the higher priority).

YAML

```yaml
resilience4j:
  circuitbreaker:
    circuitBreakerAspectOrder: 1
  retry:
    retryAspectOrder: 2
```

Observability and Monitoring of Microservices



Microservices architecture consists of many small, independent services communicating over networks.
Because of this distributed nature, debugging, performance tracking, and error detection become complex.

Monitoring

Monitoring is the process of **collecting predefined metrics and logs** to check the health, performance, and availability of applications . **Monitoring** means **continuously checking the health of your microservices**.

Alerts when something goes wrong.
Tracks system health using dashboards.

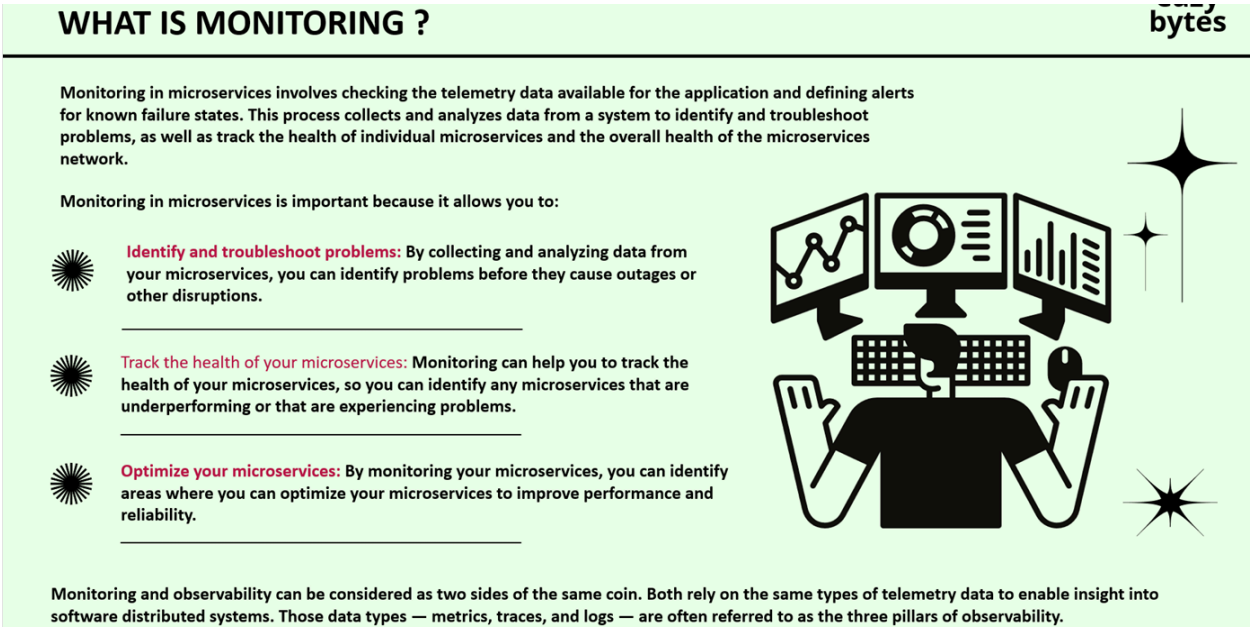
In simple words:
Monitoring = “Are my services healthy right now?”
“Is my system working fine right now?”
“Is anything going wrong?”
“Is any service down?”

Why is monitoring important?

1. **Find problems early**
   Collecting data like errors, response time, and failures helps you detect issues before the system crashes.

2. **Track service health**
   You can see which microservice is slow, overloaded, or not responding properly.

3. **Improve performance**
   By watching your system, you can identify what needs optimization.



Observability

Observability means **understanding what is happening inside the system by analyzing its output**. **Observability means understanding WHY something is wrong.**

In microservices, observability helps you understand:

- The exact root cause
- Where the issue started
- Why a service is slow
- Which service is causing the failure

- Where the request got stuck

It uses **three pillars**:

🔺 **1. Metrics**

Numbers showing system performance.
Examples: CPU, memory, response time.

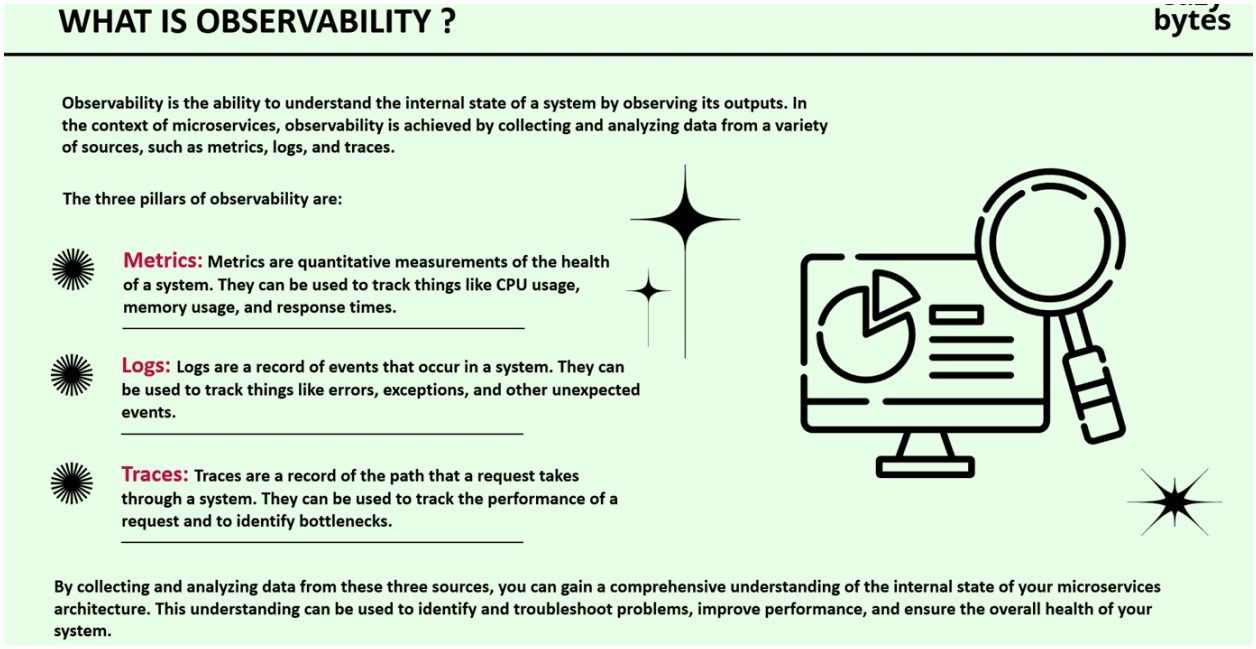
🔺 **2. Logs**

Text messages describing events.
Examples: errors, warnings, exceptions.

🔺 **3. Traces**

A map of how a request moves across microservices.
This shows where delays or failures happen.

Observability gives a **complete understanding** of the internal behavior of your services.



Feature	Monitoring	Observability
Purpose	Shows **what** is wrong	Shows **why** it is wrong
Approach	Reactive (after issue happens)	Proactive (finds issues early)
Data	Metrics, logs	Metrics, logs, traces
Goal	Identify problems	Understand the full system behavior

What is Grafana

Grafana is an **open-source web application** used to monitor, analyze, and visualize data from many different sources . https://grafana.com/

A powerful **dashboard tool that visualizes data from many systems, helping you monitor, analyze, and alert on everything in one place.**

It allows **users to query, visualize, and set alerts on their data**, making it a powerful tool for IT operations, application performance monitoring, and business intelligence.

Grafana does **not collect data by itself**. Instead, it connects to various data sources—like databases, cloud platforms, and monitoring systems—and presents the information using graphs, charts, tables, and heatmaps.

⭐ Grafana – Key Features

1. **Data Visualization**
   Offers many chart types like time-series graphs, heatmaps, histograms, gauges, and tables to visualize data effectively.

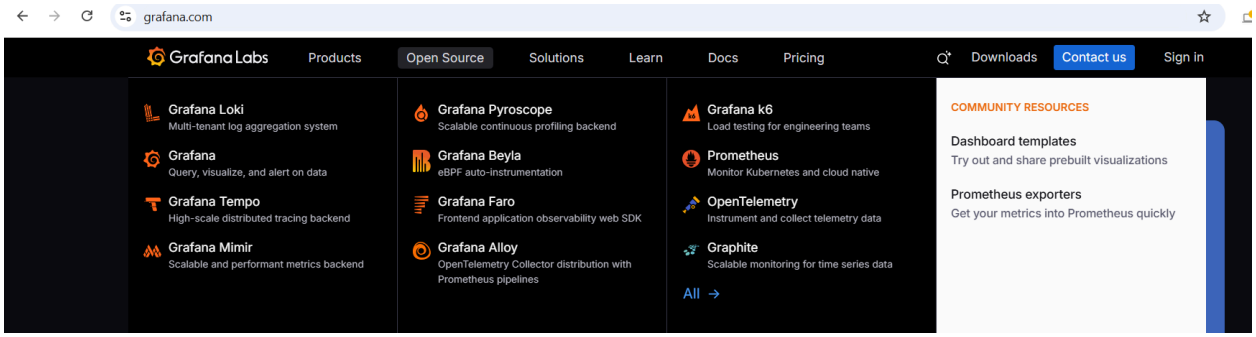
2. **Multi-Source Support**
   Connects to Prometheus, Loki, InfluxDB, cloud services, and many more—showing all data in one dashboard.

3. **Customizable Dashboards**
   Allows complete customization to create unified, real-time dashboards for different systems and applications.

4. **Alerting**
   Provides a powerful alerting system that notifies users when certain conditions or

thresholds are reached.

5. **Open-Source Platform**
   Grafana's core is free and open-source, with paid versions (Grafana Cloud, Grafana Enterprise) for advanced features.



i) Logging / Logs

Logs are records of what happens inside an application.

They help us **debug**, **troubleshoot**, and **understand** system behavior.

Logging in Monolithic Apps

- All code is inside one codebase.
- All logs are in a **single place**.
- Easy to search, find, and troubleshoot issues.

Logging in Microservices

- Each microservice has its **own logs**.
- Logs are stored in different locations across multiple services → difficult to trace issues.
- So it becomes **hard to find logs**, **trace a request**, and **fix issues**.

So we need **centralized logging** → all logs in **one place (**Collects logs from all services into one location).

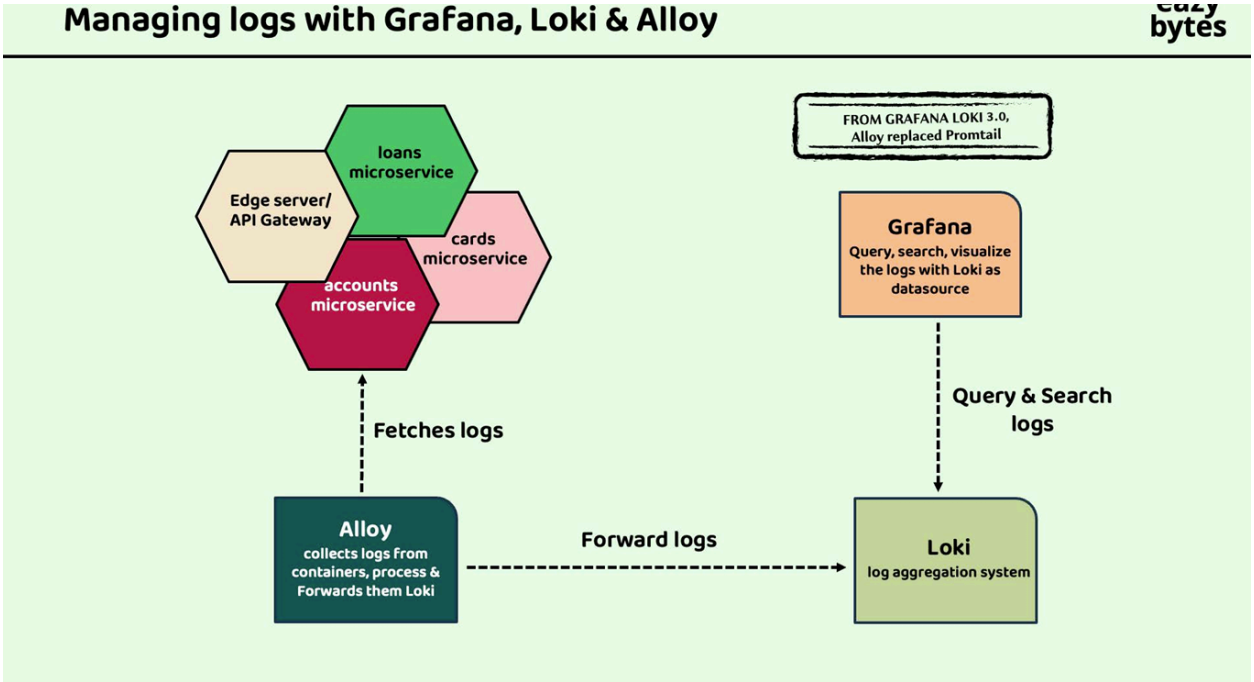
Solution: Grafana + Loki + Alloy

Alloy : Collects logs from each microservice and sends them to Loki.

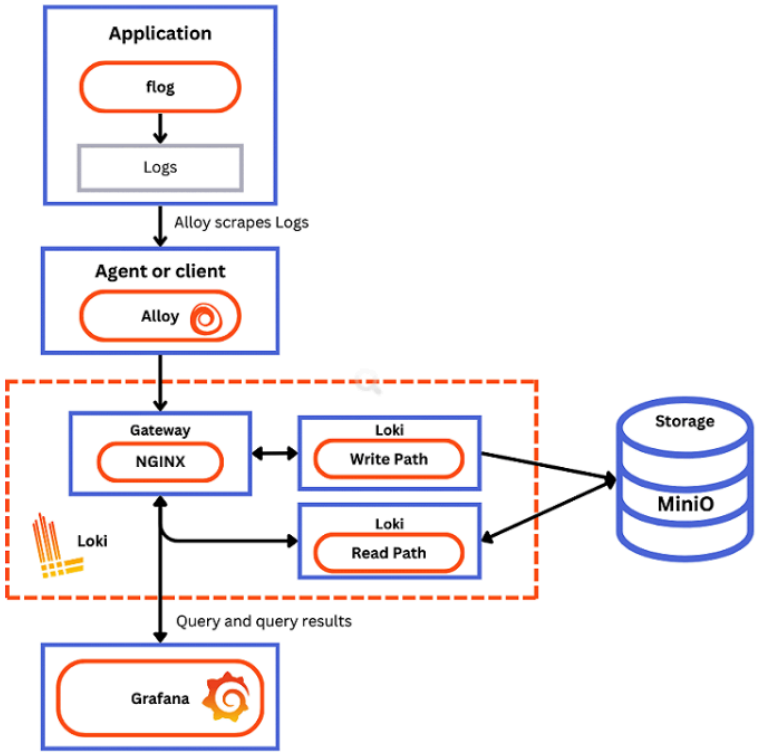
Loki : Stores and aggregates logs in a scalable and cost-effective way.

Grafana : Visualizes the logs with dashboards, filters, search, and alerts.

Workflow: Microservices → Alloy → Loki → Grafana Dashboard



https://grafana.com/docs/loki/latest/get-started/quick-start/quick-start/



- Your **application**(ex:microservices,accounts,flog) **prints logs** to stdout. These logs are NOT stored by the application itself.

- **Alloy** collects those logs (replaced Promtail in Loki 3.0).

- Logs go through an **NGINX gateway** to Loki.

- **Loki stores logs** (MinIO is used as storage).

  Loki has two main internal paths:

  - **Write Path** → accepts and stores logs
  - **Read Path** → returns logs when Grafana queries them
- **Grafana reads logs** from Loki and shows them in dashboards.

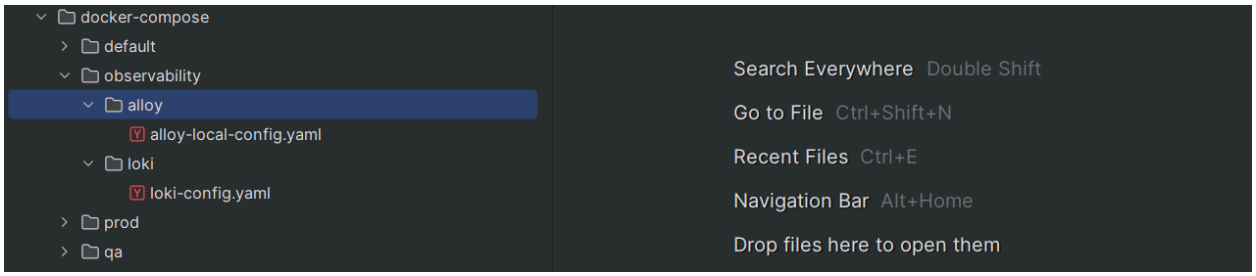
Steps to integrate with microservices :

1. Create Folder Structure

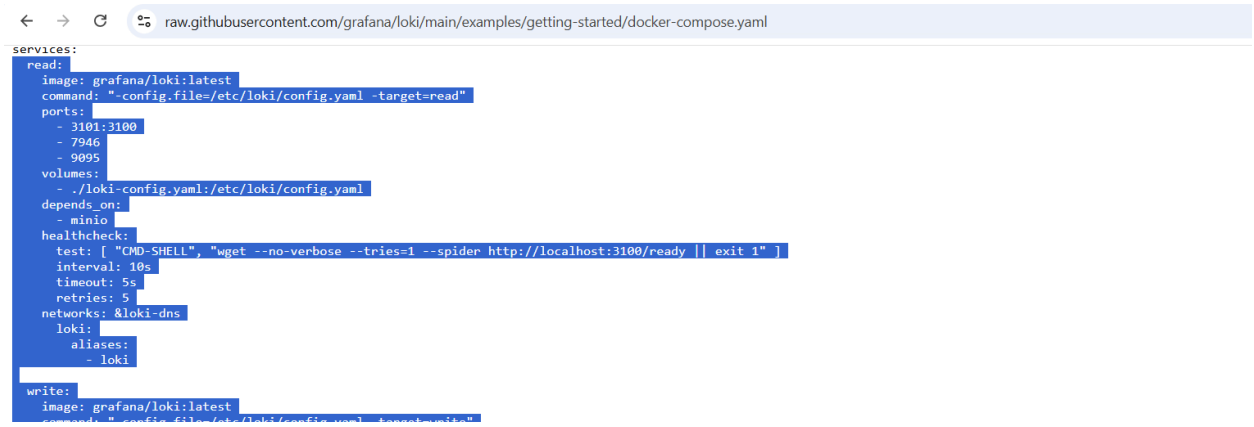
Inside your main **docker-compose** folder , create an **observability** folder. Then create **loki** and **alloy** folders inside the **observability** folder.

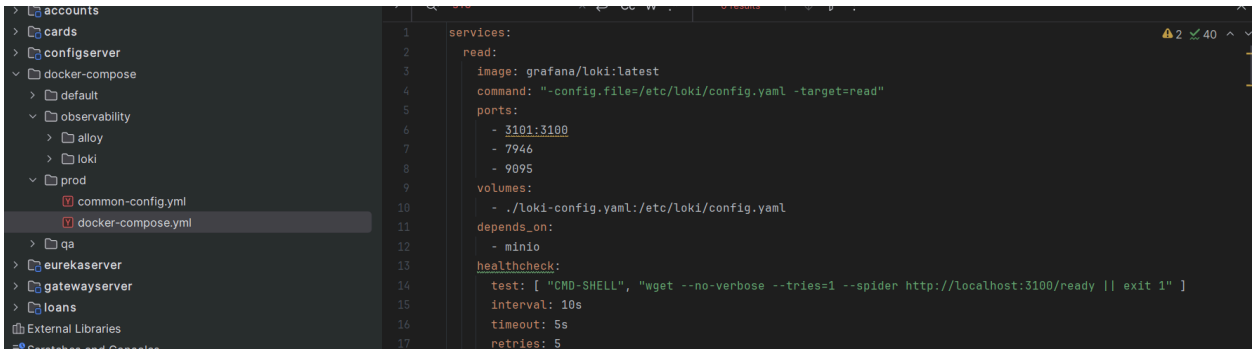
Add the **loki-config.yaml** file inside the *loki* folder, and the **alloy-local-config.yaml** file inside the *alloy* folder.

You will get these .yml files from the URL provided.
https://grafana.com/docs/loki/latest/get-started/quick-start/quick-start/



Copy the **docker-compose.yaml** from the mentioned URL and paste the required sections into your environment-specific docker-compose.yml file (such as **dev**, **qa**, or **prod**).

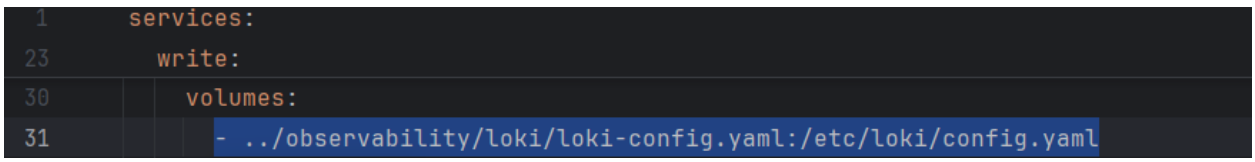




After copying the required sections from the URL into your environment-specific docker-compose.yml (such as dev, qa, or prod), update a few details—especially the **volumes** for Loki and Alloy.

1. **Update Loki volume path**
   You must point to the loki-config.yaml file created under the observability/loki folder.
   Example:

   - ../observability/loki/loki-config.yaml:/etc/loki/config.yaml



Update Alloy volume path
Similarly, update the volume mapping for alloy-local-config.yaml:

- ./observability/alloy/alloy-local-config.yaml:/etc/alloy/config.alloy:ro

If needed, update any references that connect to common-config.yml

Check whether any services require configuration values from common-config.yml and update the paths accordingly.

```
116    extends:
117      file: common-config.yml
118      service: network-deploy-service
```

build **Docker images for all microservices using Maven + Jib**

(the command you meant is: mvn compile jib:dockerBuild).

```
C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\grafanaexample\accounts>mvn compile jib:dockerBuild
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< com.eazybytes:accounts >----------------------
[INFO] Building accounts 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
```

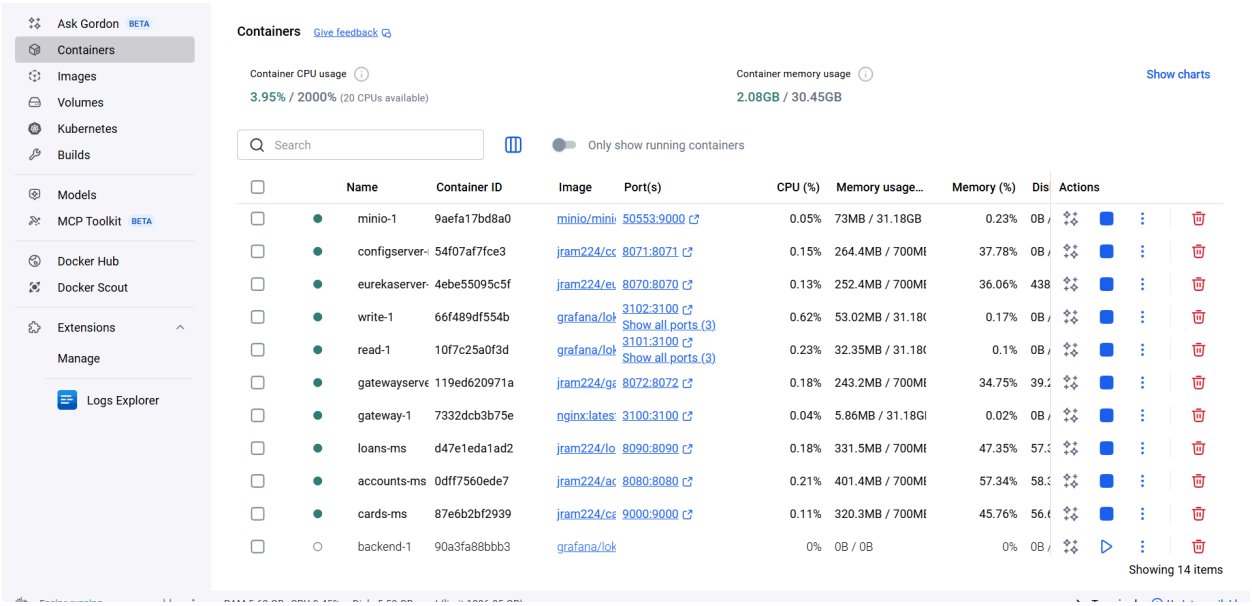
```
C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\grafanaexample\cards>mvn compile jib:dockerBuild
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------< com.eazybytes:cards >-------------------------
[INFO] Building cards 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
```

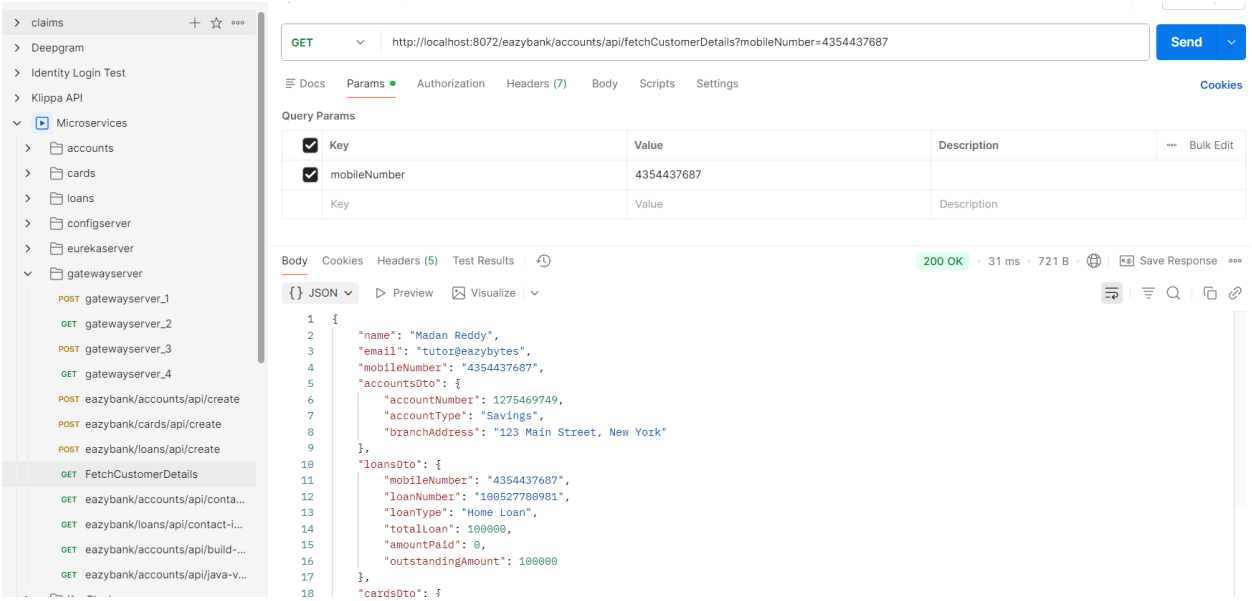
And all other services also like loans, eureka,gateway,config…

Name	Tag	Image ID	Created	Size	Actions
jram224/accounts	s11	7e91fb53bab0	N/A	542.79 MB	
jram224/cards	s11	cd6c0fd85e26	N/A	538.91 MB	
jram224/loans	s11	94b6e4dce3a3	N/A	539.29 MB	
jram224/configserver	s11	822ea2d2498c	N/A	454.8 MB	
jram224/eurekaserver	s11	9229adae8170	N/A	481.08 MB	
jram224/gatewayserver	s11	156bda84a13d	N/A	482.78 MB	

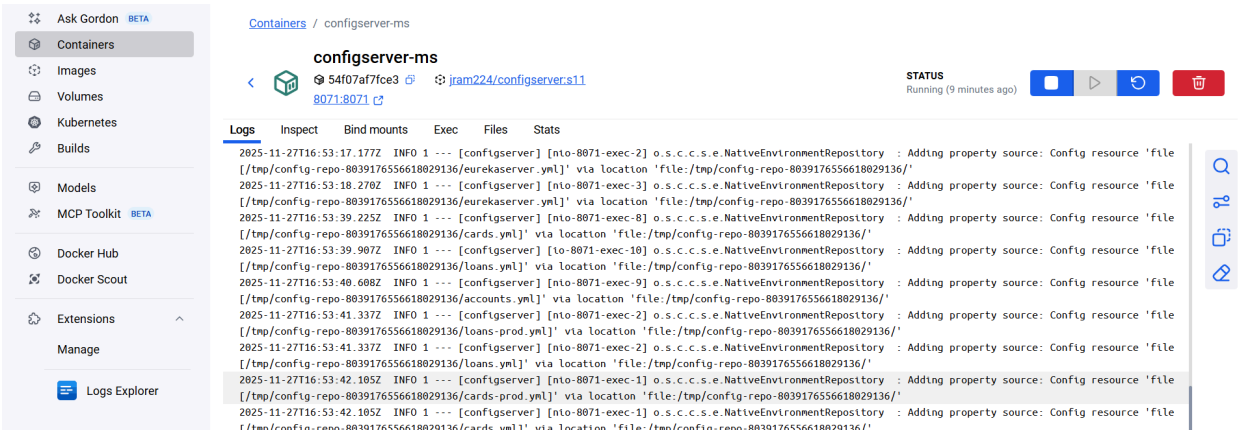
Running docker run command from prod folder . **docker compose up -d**

```
C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\grafanaexample\docker-compose\prod>docker compose up -d
[+] Running 14/14
 ✔Network prod_eazybank          Created                                    0.0s
 ✔Container prod-minio-1         Started                                    0.9s
 ✔Container configserver-ms      Healthy                                   22.2s
 ✔Container eurekaserver-ms      Healthy                                   42.1s
 ✔Container prod-write-1         Started                                    1.7s
 ✔Container prod-read-1          Started                                    1.4s
 ✔Container cards-ms             Healthy                                   83.5s
 ✔Container loans-ms             Healthy                                   83.5s
 ✔Container accounts-ms          Healthy                                   83.5s
 ✔Container prod-gateway-1       Started                                    2.1s
 ✔Container gatewayserver-ms     Started                                   83.8s
 ✔Container prod-grafana-1       Started                                    3.2s
 ✔Container prod-backend-1       Started                                    3.0s
 ✔Container prod-alloy-1         Started                                    3.4s

C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\grafanaexample\docker-compose\prod>
```





If any issue occurs in the system, and we don't have log segregation, we must manually check logs in each individual microservice. This makes troubleshooting slow and complicated because logs are scattered across multiple services.

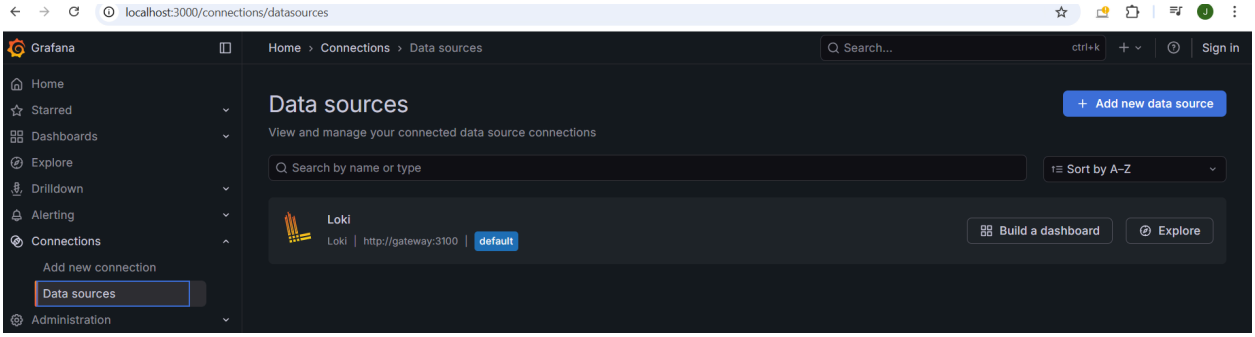


. With Loki + Alloy + Grafana, all logs are collected centrally, so we can easily analyze issues from a single place without jumping into each service container.

After integration, once you run docker compose up, open Grafana at
👉 **http://localhost:3000**

The Grafana home page will appear. Then:

1. Click the **hamburger menu** (☰) on the left.
2. Go to **Connections → Data sources**.
3. You will see **Loki** listed as a connected data source.



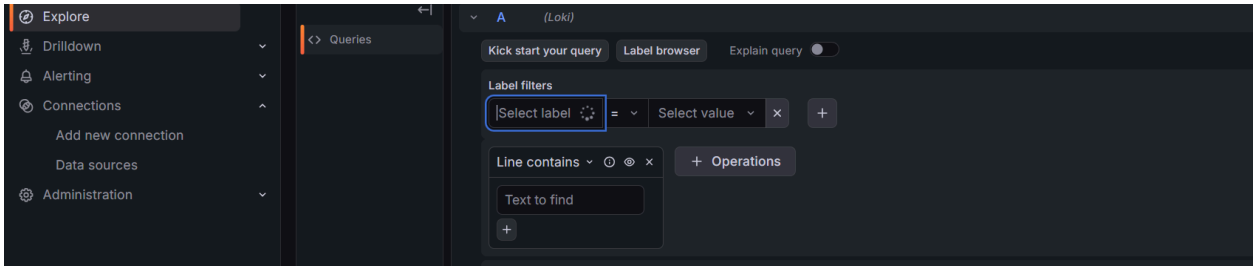
In the **Explore** page, choose **Loki** as the data source.

Under **“Select label”**, choose **container**.

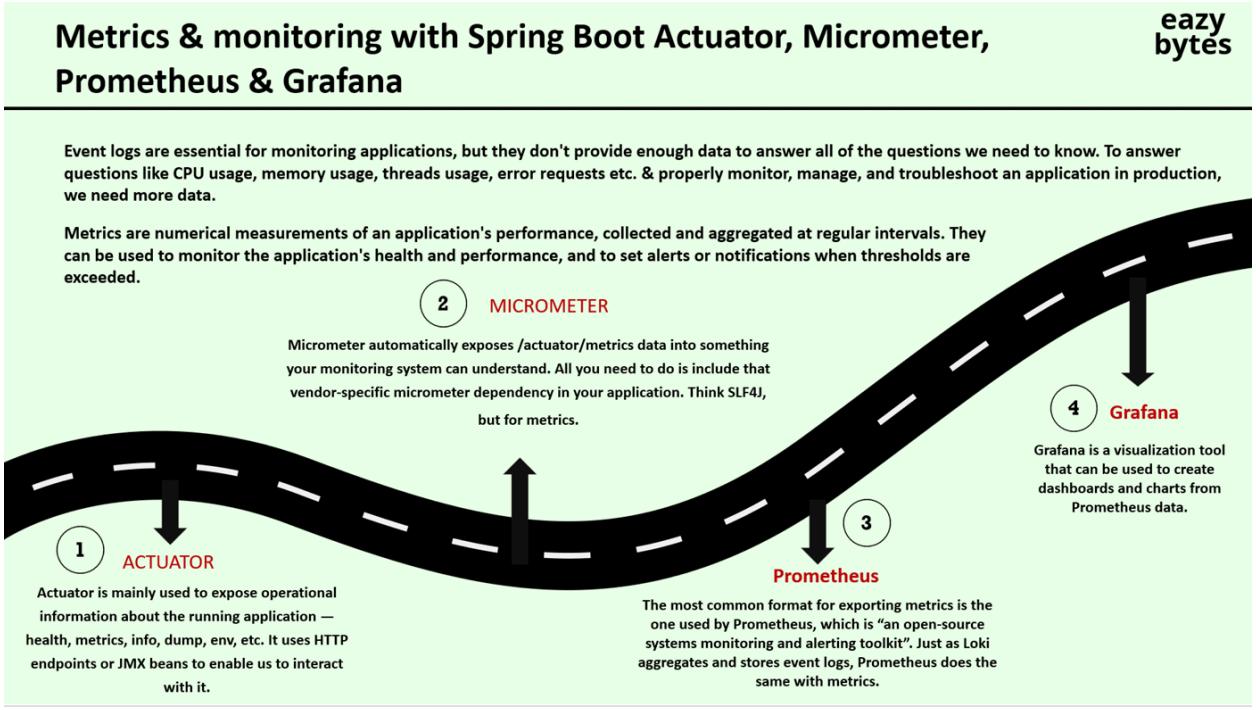
Under **“Value”**, you will see a list of all container names collected by Alloy.

Select any container to view only that service’s logs.

Click **Run query** — Grafana will show filtered logs for the selected container



ii) Metrics



Actuator → Micrometer → Prometheus → Grafana

- Your application (ex: microservices, accounts, flog) exposes metrics via **Actuator** endpoints.
- These metrics are NOT stored by the application itself.
- **Micrometer** collects metrics from Actuator endpoints and formats them for monitoring systems.
- **Prometheus** pulls metrics from Micrometer, aggregates, and stores them in its time-series database.
- **Grafana** reads metrics from Prometheus and shows them in dashboards.

1) Add Micrometer Dependency in Every Microservice

In each microservice's pom.xml, add:

```xml
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

2. Add Metrics Properties in application.yml Inside every microservice: ex

```yaml
metrics:
  tags:
    application: ${spring.application.name}
```

Start the microservices → and check:

All metrics http://<service-host>:<port>/actuator/metrics

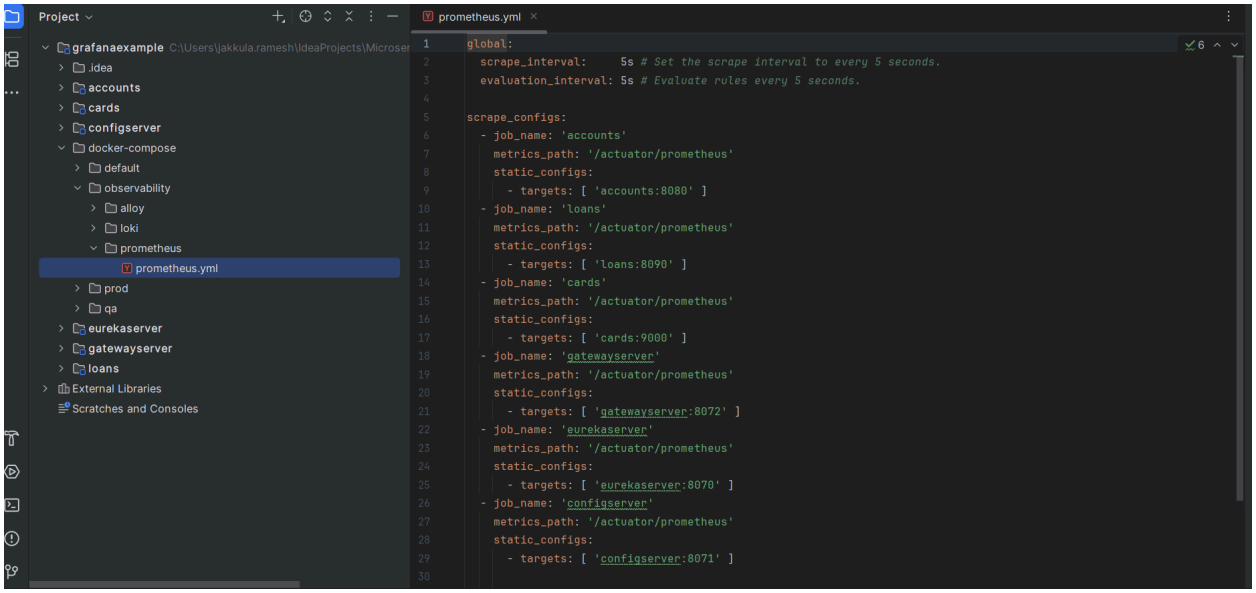
Specific metric /actuator/metrics/http.server.requests

Prometheus-formatted metrics /actuator/prometheus

3. **Create a folder prometheus inside the observability folder in docker-compose directory**

Create a file prometheus.yml inside it (from Prometheus docs).
https://prometheus.io/docs/prometheus/latest/getting_started/

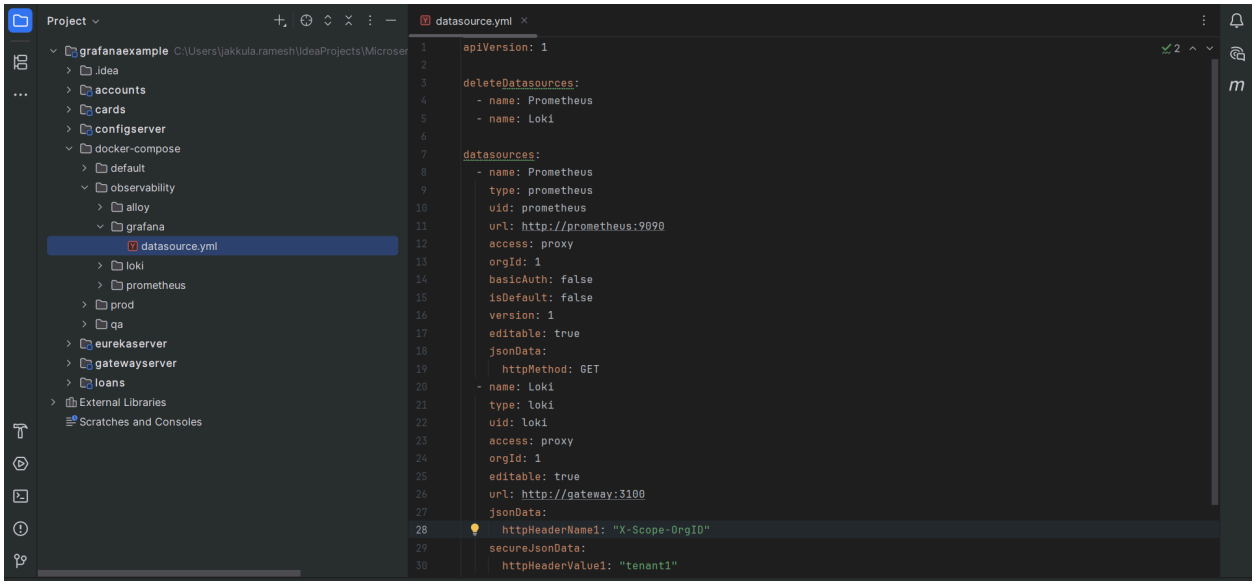
Add each microservice and its actuator Prometheus endpoint.



4. **Add Prometheus related configuration in production docker-compose.yml**
Just like earlier configurations, mount the Prometheus file.

5. **Next step → Connect Prometheus and Grafana**

Grafana will read data from Prometheus.**Create a folder grafana inside observability**, then create datasource.yml This file is reused for Grafana config (for both Loki and Prometheus).



Add this file path in the production docker-compose for Grafana.

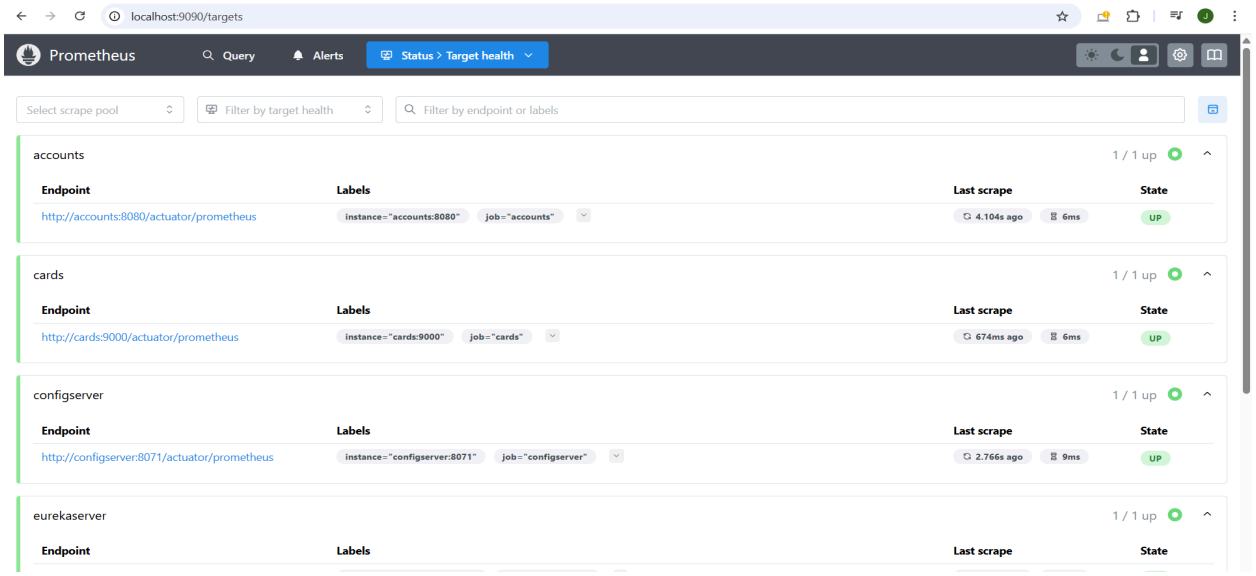
6. Now generate Docker images for all microservices and Run docker-compose from the environment folder (ex: prod).

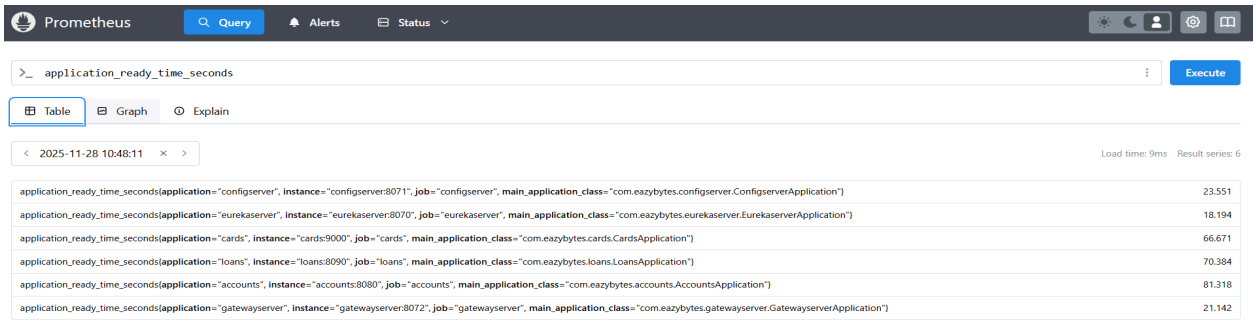
```
mvn compile jib:dockerBuild

docker compose up -d
```

To check Prometheus dashboard : Use the Prometheus port number.

Example: http://localhost:9090/targets

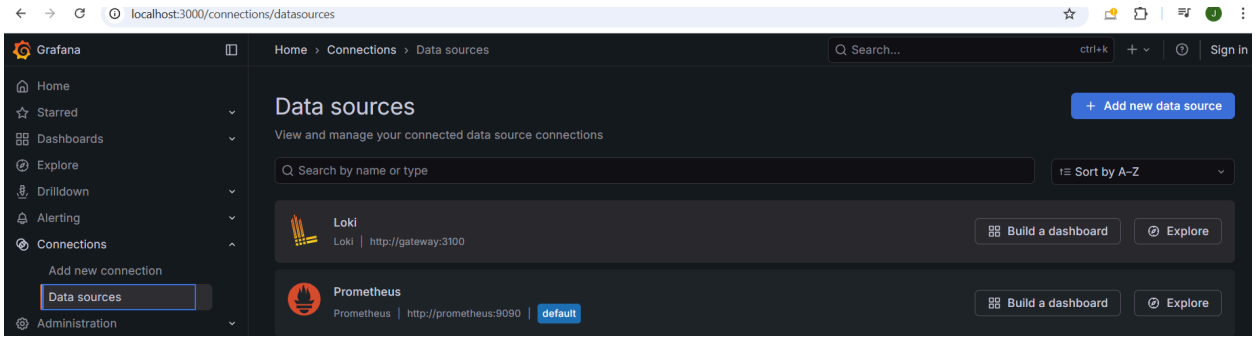
we can see all running microservices health or unhealthy container info and other details.

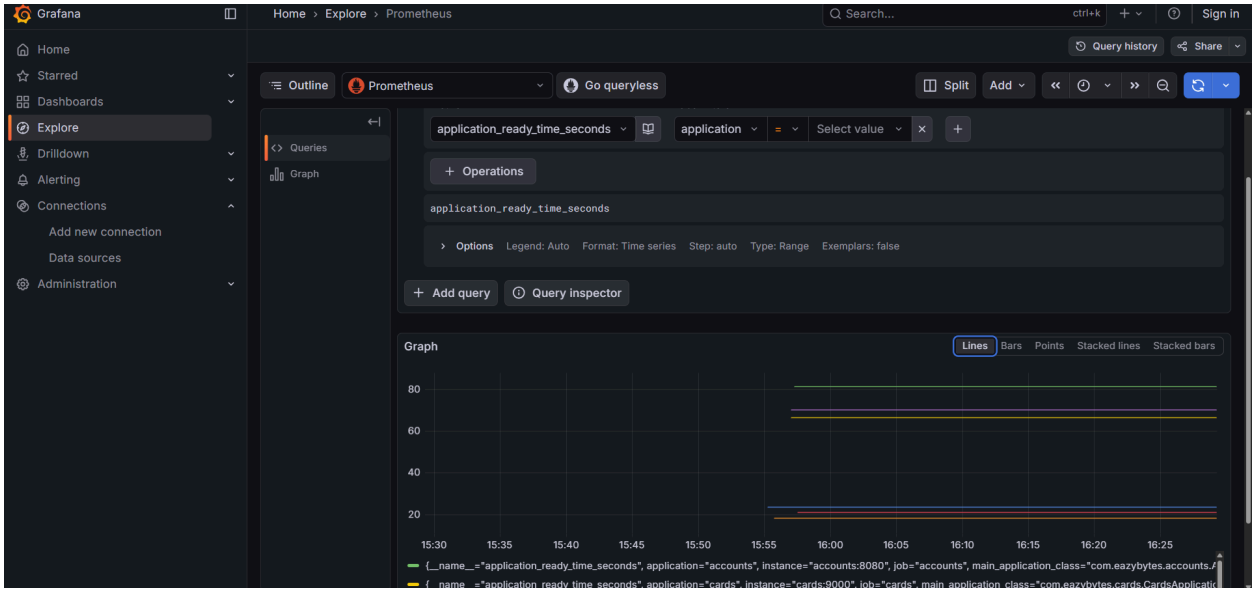




for bigger application, better to use Grafana
so we can monitor the application in production better.

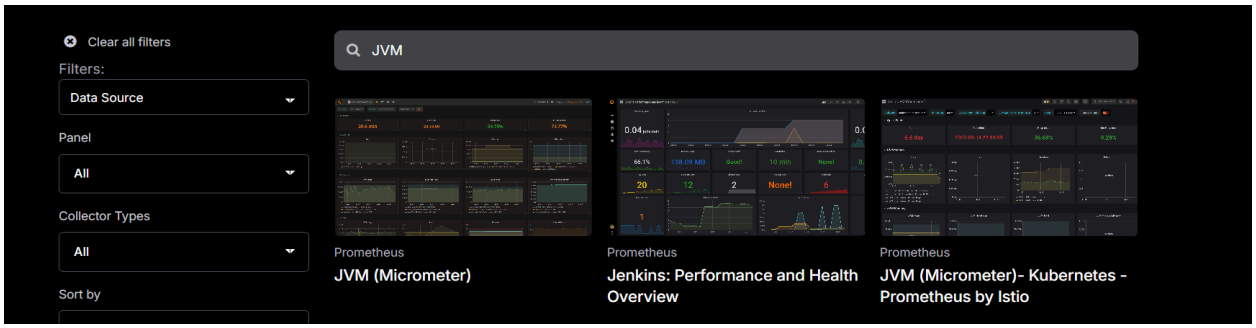
To check the Grafana dashboard : Use the grafana port number. Example: http://localhost:3000

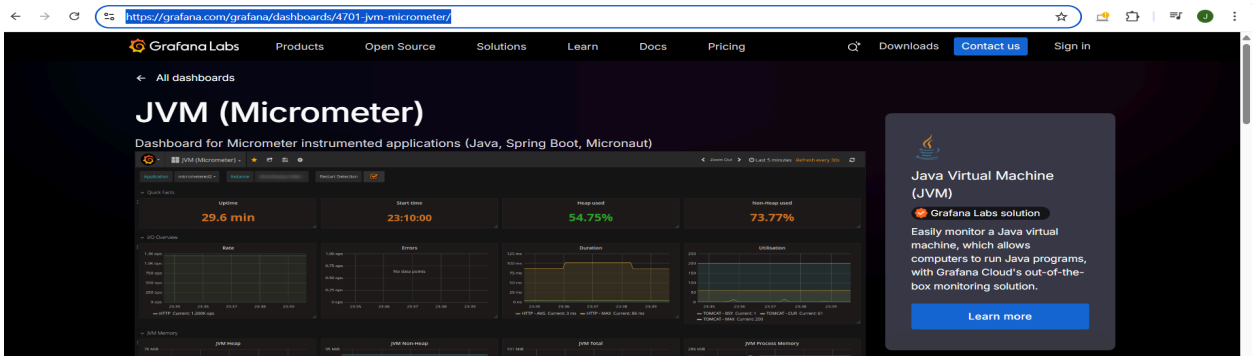




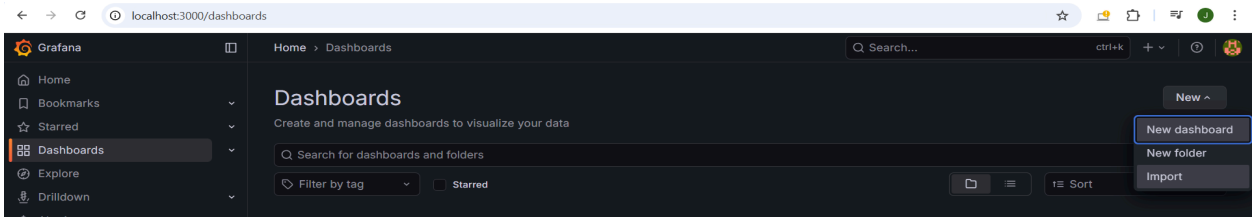
Prometheus and Grafana together, with inbuilt custom dashboards

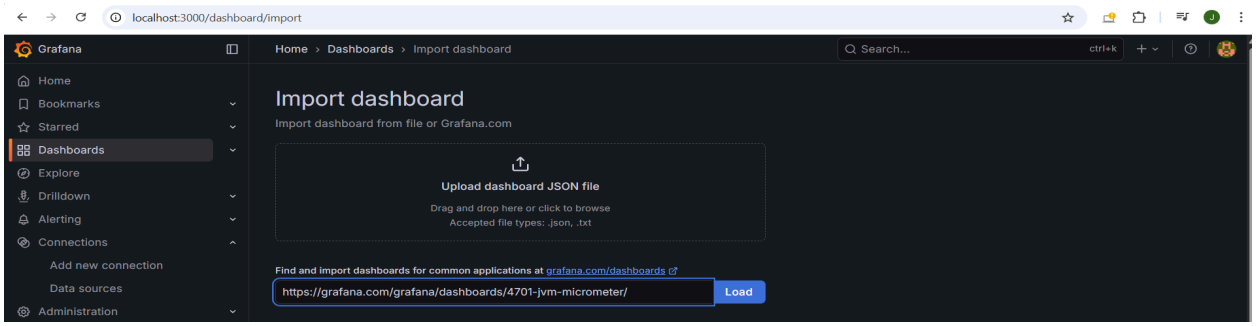
We can copy pre-built dashboards and use them in Grafana, so no need to create dashboards from scratch. https://prometheus.io/docs/visualization/grafana/ , Copy pre built dashboard from https://grafana.com/grafana/dashboards/



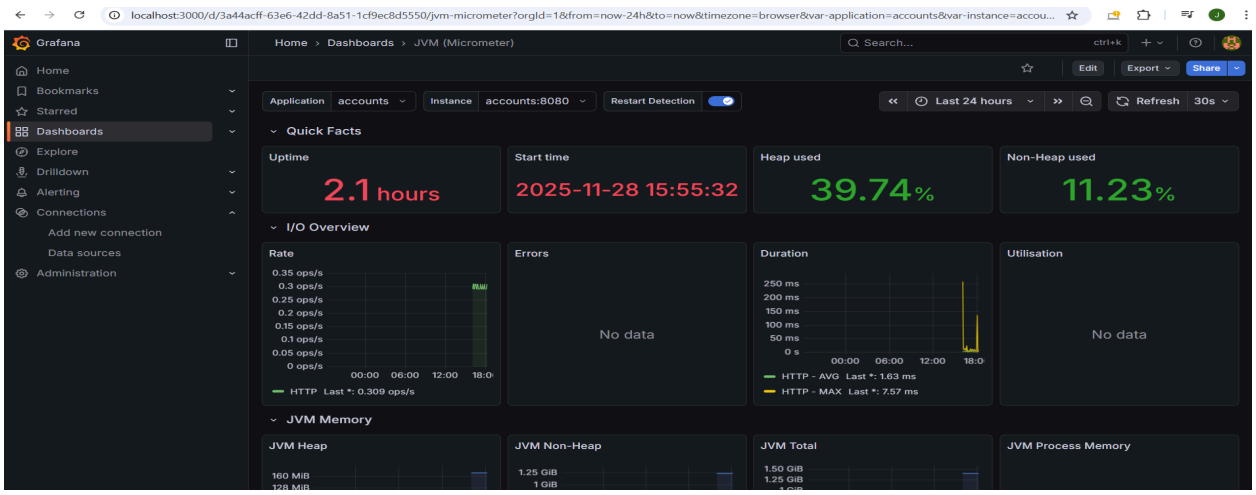


Open grafana and sign with default credentials admin, admin.
Go to the dashboard and click new ->import. We can also create a dashboard by clicking on the new dashboard.



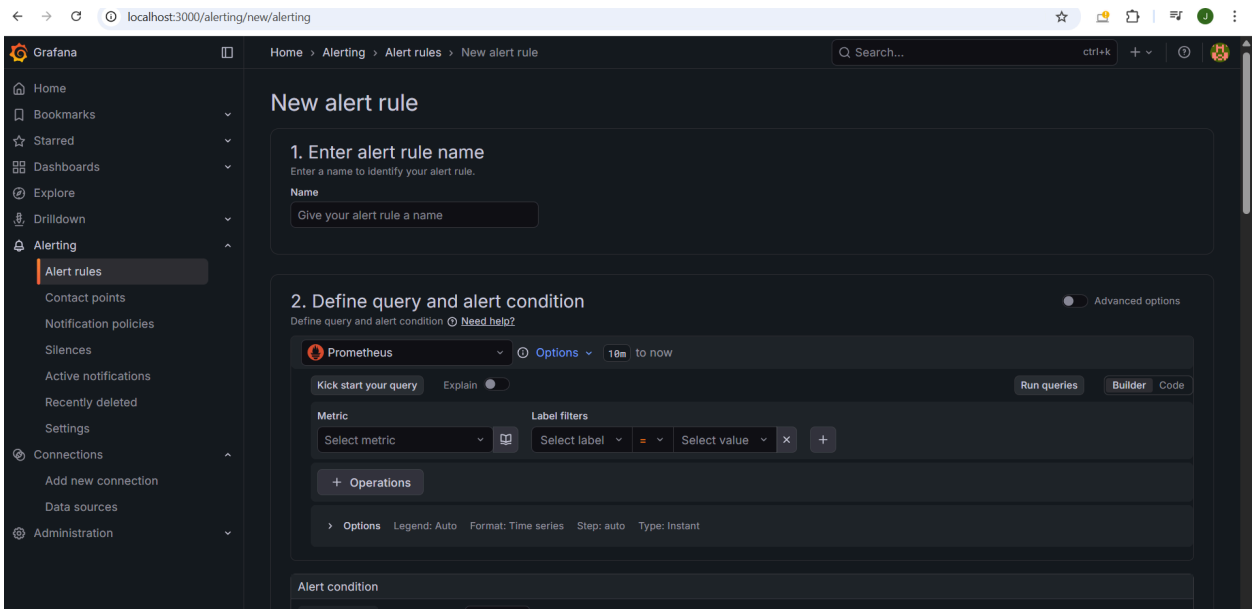


We can see the dashboard layout ..



We can select alerts by clicking on alerting in grafana. To use alerting in Grafana:

1. open Grafana → Alerting
2. create alert rule
3. write Prometheus query
4. set condition (ex: error > 5)
5. choose notification channel (email/slack)
6. save → Grafana sends alerts automatically.

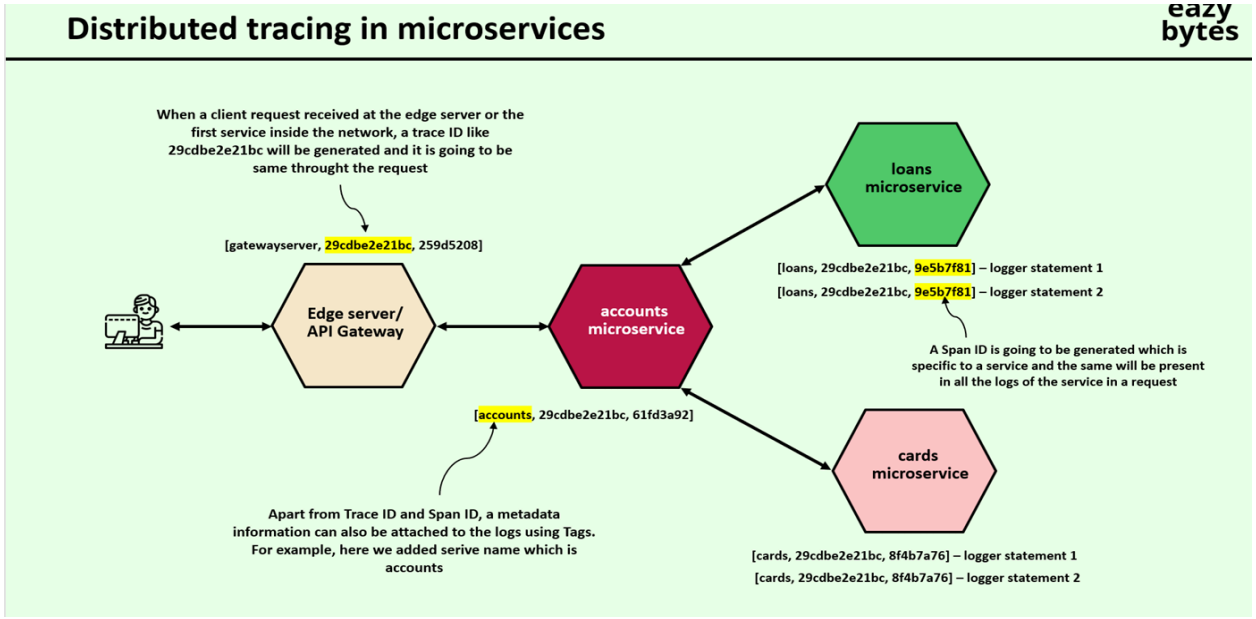


iii) Tracing

When a client request travels through **many microservices**, it's hard to know:

- Where the request went
- Which service caused the delay
- Where the error happened
- How long each step took

Distributed tracing solves this. It helps track a request **end-to-end across all microservices**.



1. Trace ID → Same for Entire Request

When a request enters the system (API Gateway / first service), a **Trace ID** is generated.

Example: 29cdbe2e21bc

This **Trace ID stays the same across all microservices** handling that request.

2. Span ID → Different for Each Microservice

Each microservice generates its own **Span ID**. Example:

- Loans MS span → 9e5b7f81
- Cards MS span → 8f4b7a76
- Accounts MS span → 61fd3a92

Span ID = which service did which part of the work

3. Tags → Extra Information

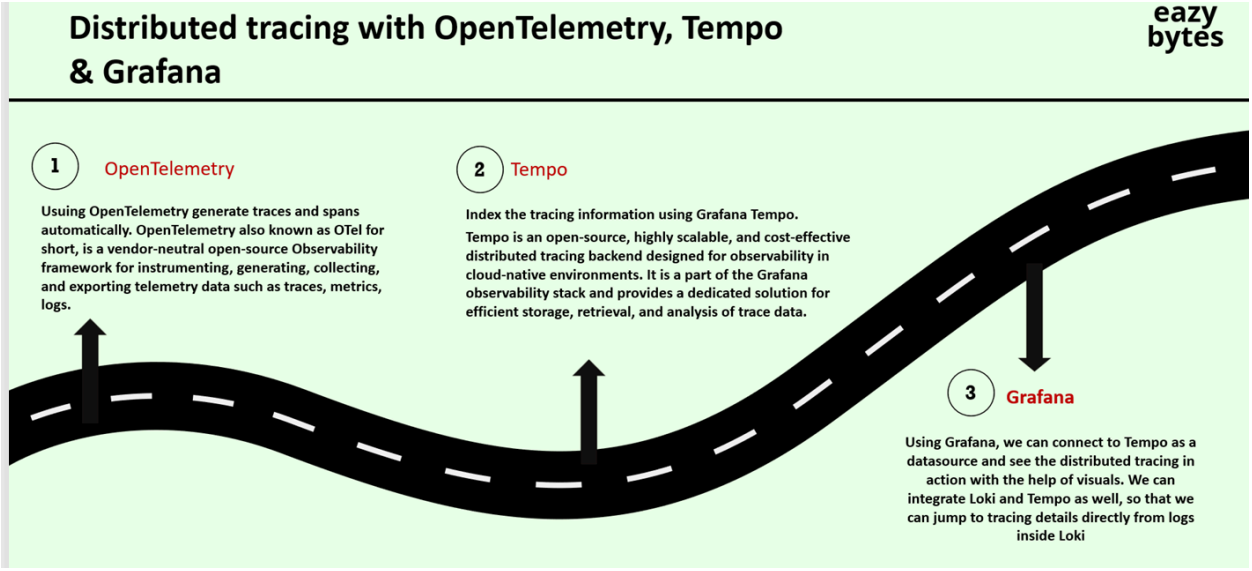
Tags add additional metadata, e.g.:

- service name
- URL
- user ID
- tenant ID

Example tag: service=accounts

We can implement distributed tracing using **Spring Cloud Sleuth**, but Sleuth is deprecated and not recommended.https://spring.io/projects/spring-cloud-sleuth
Spring Boot 3 introduced **Micrometer Tracing**, but it requires more configuration and mainly supports Java only. https://micrometer.io/docs/tracing

To support **multiple languages** and get an **industry-standard** solution, we will use **OpenTelemetry**, which supports automatic instrumentation, works across all microservices, and integrates with Jaeger/Tempo/Grafana easily. https://opentelemetry.io/



OpenTelemetry -> Tempo -> Grafana

- Your application (ex: microservices, accounts, flog) generates tracing data using OpenTelemetry libraries or auto-instrumentation. These traces are NOT stored by the application.
- **OpenTelemetry (OTel)** automatically creates **traces and spans** for incoming requests, outgoing calls, DB calls, messaging, etc., and exports them.
- **Tempo** receives these traces from OpenTelemetry, indexes them, and stores them as distributed trace data in an efficient, low-cost backend.
- **Grafana** reads trace data from Tempo and displays them visually as timelines, flame graphs, and span details.
  Grafana can also link **Loki ↔ Tempo**, allowing you to jump from **logs → trace**.

Steps to Integrate

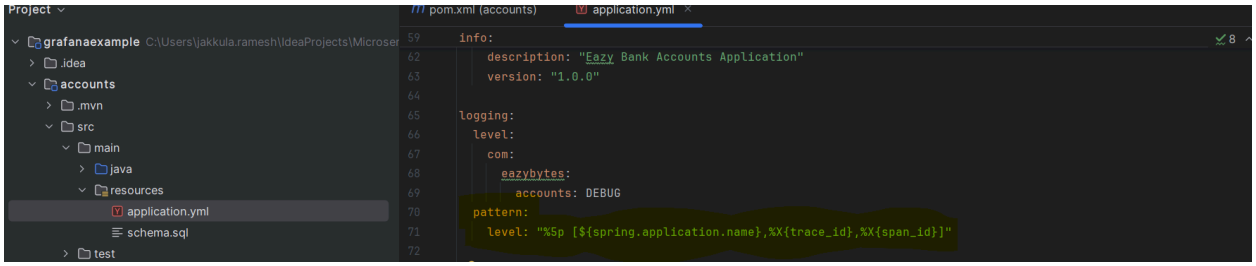
1. **Add OpenTelemetry dependencies and Add OTel Configuration in application.yml**

   Add required OpenTelemetry (OTel) dependencies in each microservice (pom.xml).

```
    <properties>
        <java.version>17</java.version>
        <spring-cloud.version>2024.0.0</spring-cloud.version>
        <otelVersion>2.11.0</otelVersion>
```

```xml
<dependency>
    <groupId>io.opentelemetry.javaagent</groupId>
    <artifactId>opentelemetry-javaagent</artifactId>
    <version>${otelVersion}</version>
    <scope>runtime</scope>
</dependency>
```

logging patterns like logging.pattern.level in each service.



2. **Add Optional Custom Logs**

   Use logger.debug() or logger.info() in code wherever additional details are needed.

```java
@GetMapping("/fetchCustomerDetails")
public ResponseEntity<CustomerDetailsDto> fetchCustomerDetails(@RequestHeader("eazybank-correlation-id")
                                                               String correlationId,
                                                               @RequestParam @Pattern(regexp="(^$|[0-9]{10})",
                                                                       message = "Mobile number must be 10 digits")
                                                               String mobileNumber) {
    logger.debug("fetchCustomerDetails method start");
    CustomerDetailsDto customerDetailsDto = iCustomersService.fetchCustomerDetails(mobileNumber, correlationId);
    logger.debug("fetchCustomerDetails method end");
    return ResponseEntity.status(HttpStatus.SC_OK).body(customerDetailsDto);

}
```

```java
@GetMapping("/fetch")
public ResponseEntity<LoansDto> fetchLoanDetails(@RequestHeader("eazybank-correlation-id") String correlationId,
                                                 @RequestParam
                                                 @Pattern(regexp="(^$|[0-9]{10})",message = "Mobile number m
                                                 String mobileNumber) {
    logger.debug("fetchLoanDetails method start");
    LoansDto loansDto = iLoansService.fetchLoan(mobileNumber);
    logger.debug("fetchLoanDetails method end");
    return ResponseEntity.status(HttpStatus.OK).body(loansDto);
}
```

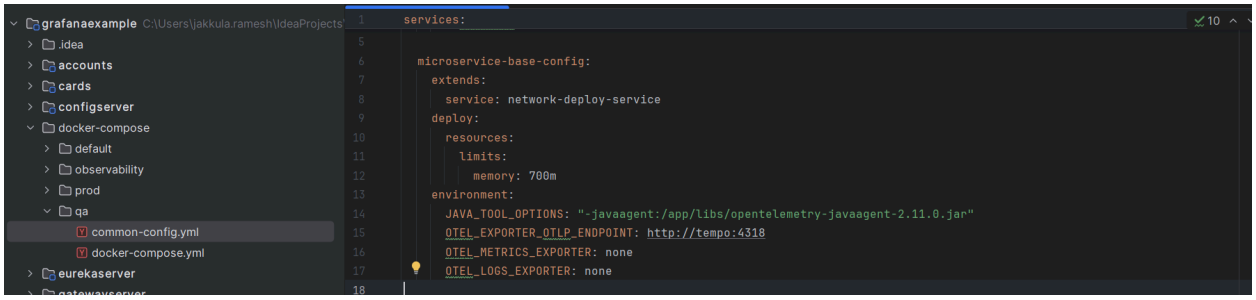
3. Update Docker Configuration : Open the **Docker Compose file of any env folder(ex: prod)**.

   Update configuration details in:

   a. common-config.yml (shared configuration for services)

b. docker-compose.yml (service-specific environment overrides)

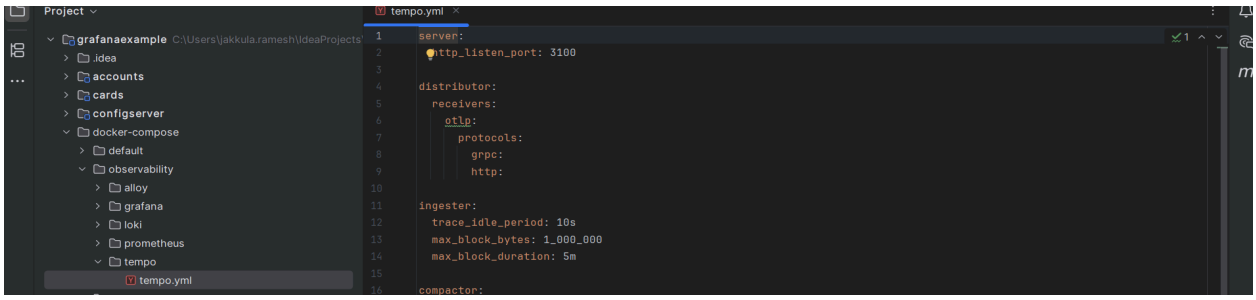
Ensure OTel, Tempo, and Grafana endpoints are correctly mapped.



4. Create Tempo Configuration

Open the docker-compose folder, Inside the **observability** folder, create a new folder named **tempo**. Inside tempo, create tempo.yml.

Add Tempo-specific configuration to tempo.yml. Map tempo.yml in docker-compose.yml under the Tempo service.

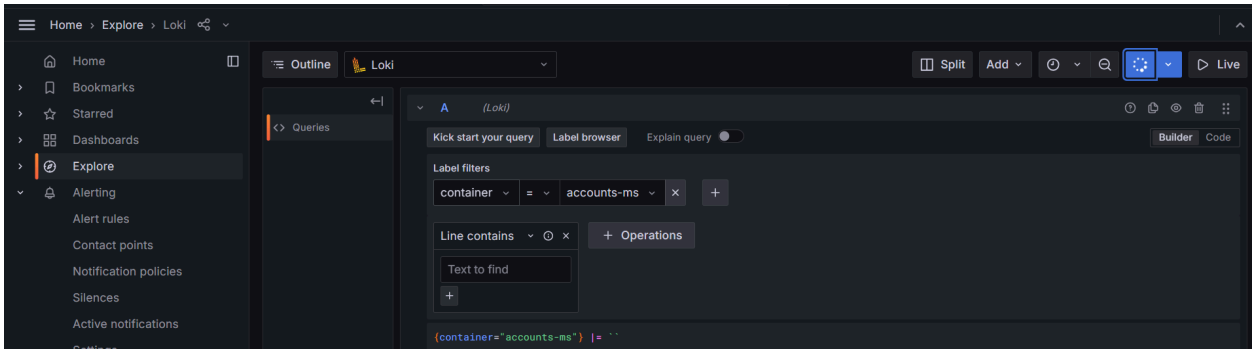


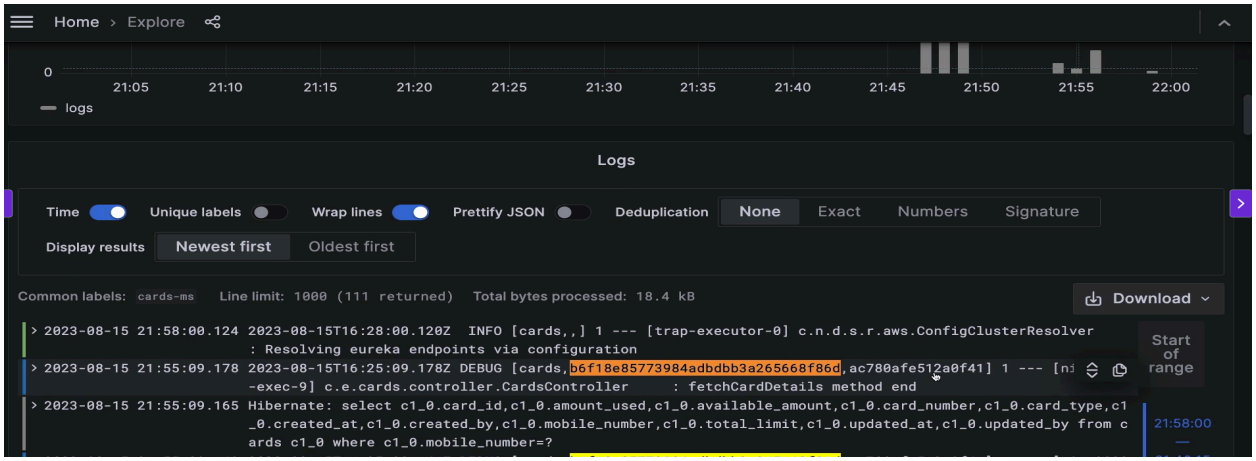
5. Now generate Docker images for all microservices and Run docker-compose from the environment folder (ex: prod).

mvn compile jib:dockerBuild

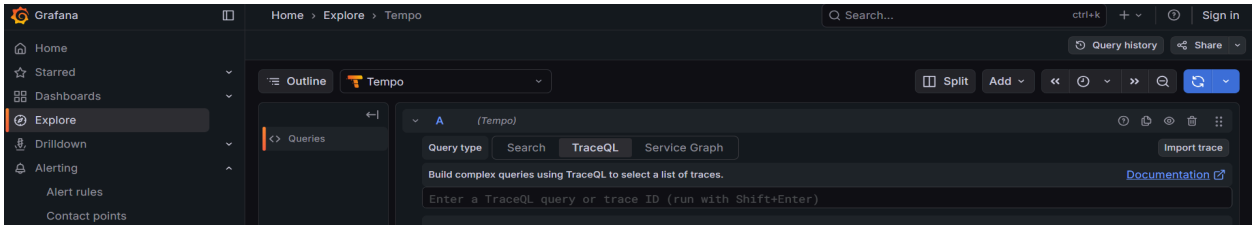
docker compose up -d

Grafana Home ->Explore , select loki and run query and you will get logs with tags,trace id , span id.

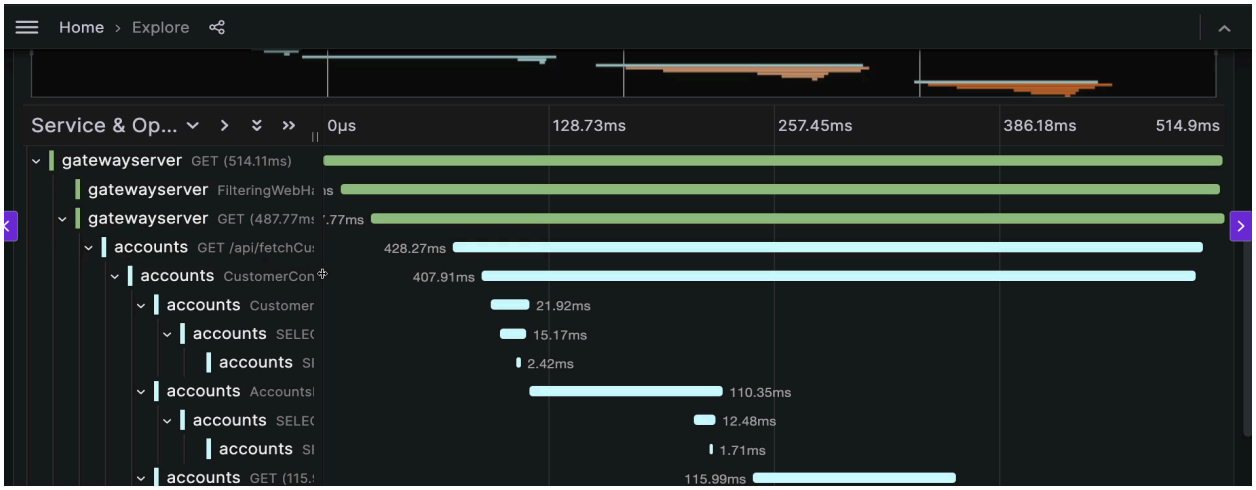


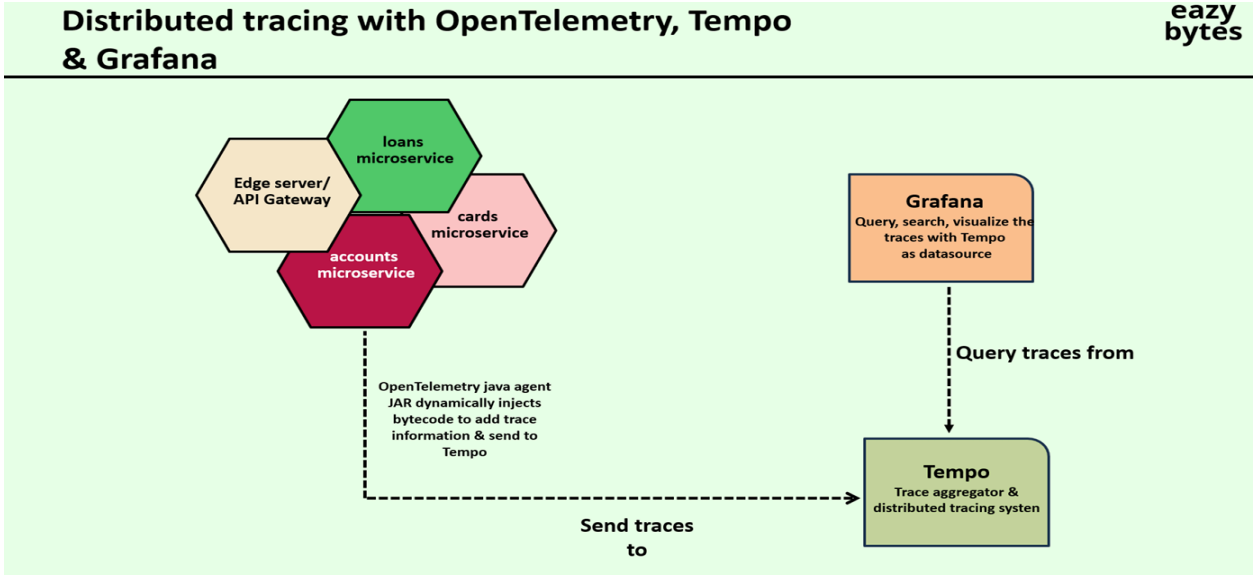


Grafana Home ->Explore , select temp and enter the id then we can see how the request travelled across all microservices.



We can copy the trace id and search in the different containers so we can access the logs easily.





Microservices Security

🔐 Securing Microservices from Unauthorized Access

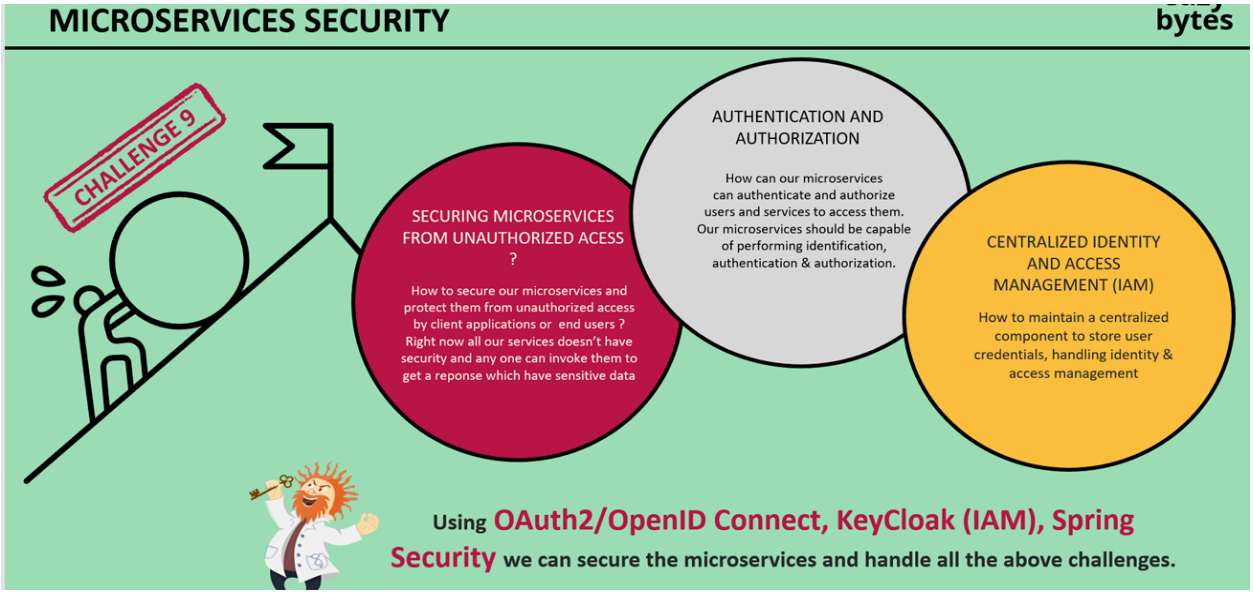
- Microservices currently expose sensitive endpoints.
- Need protection against unauthorized clients or users.

👤 Authentication & Authorization

- Microservices require a way to authenticate users and authorize access to APIs.
- Each service shouldn't handle auth logic individually.

🏢 Centralized Identity and Access Management (IAM)

- A centralized place to store identities and handle login, tokens, and permissions.

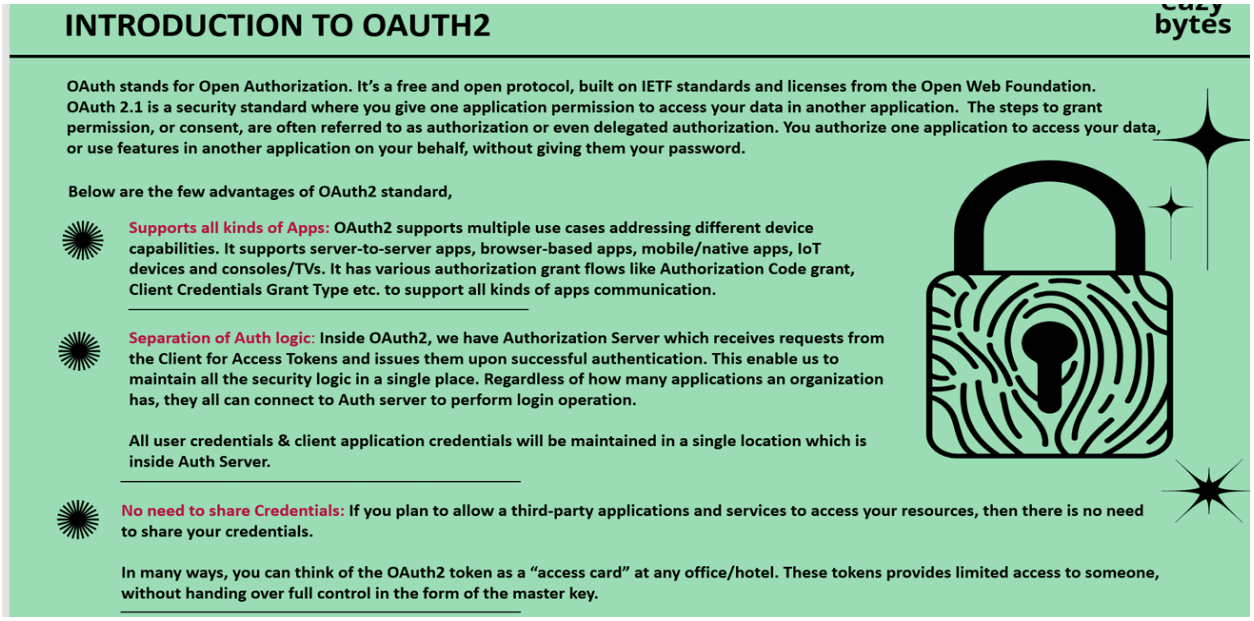


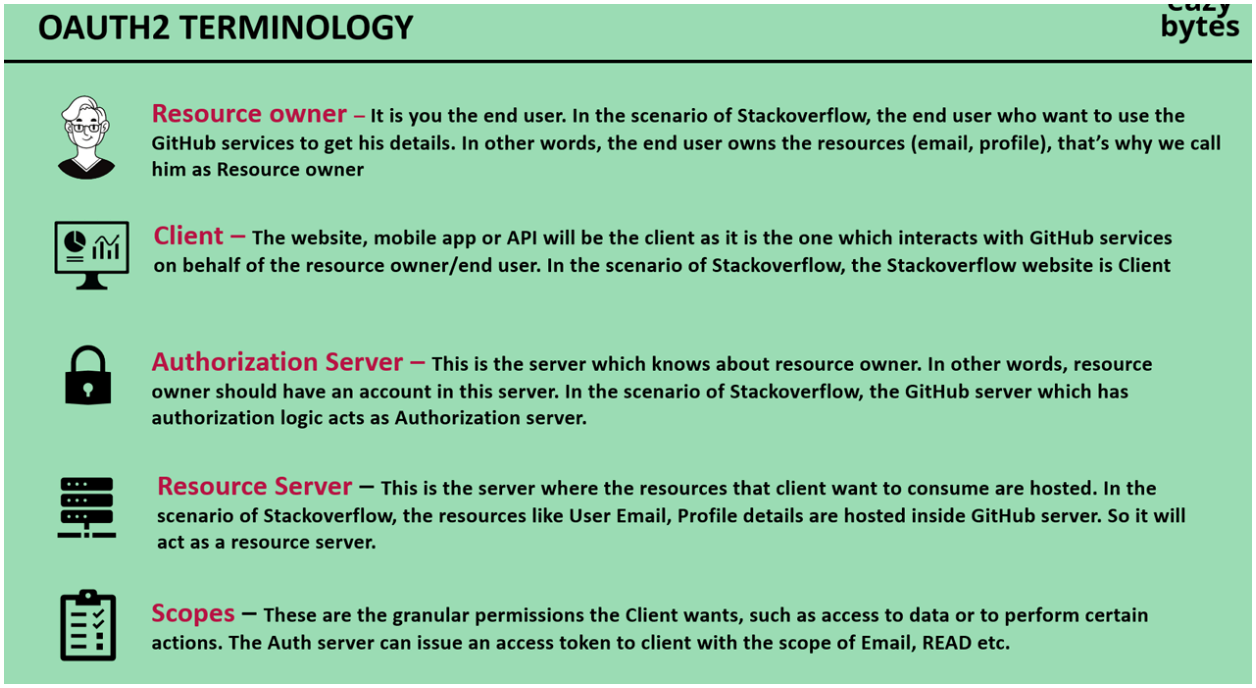
Authentication : It verifies **WHO** you are.

Authorization : It decides what **permissions and access** you have **after login.**

OAuth 2.0 (Open Authorization 2.0(version)):

OAuth 2.0 (Open Authorization 2.0) is a standard protocol that allows applications (clients) to **access resources on behalf of a user** without needing to know the user's credentials (username/password). It provides **secure, delegated access** to APIs or services by issuing **Access Tokens** ((often JWT) that define what the client can do.
https://auth0.com/intro-to-iam/what-is-oauth-2





Resource Owner

- **Who:** You, the user.
- **Role:** Owns the resources (like email, profile info).
- **Example:** You want to log in to Stack Overflow using your GitHub account.

Client

- **Who:** The app or website that wants access. **Client application** (website, mobile app, desktop app, smart TV, etc.)
- **Role:** Requests access to the user's resources on their behalf.
- **Example:** Stack Overflow website is the client.

Authorization Server

- **Who:** Server that manages authentication and permissions.
- **Role:** Confirms your identity and decides what the client can access.
- **Example:** GitHub's login server.

Resource Server

- **Who:** Server where the resources actually reside.
- **Role:** Delivers the requested resources if the access token is valid.
- **Example:** GitHub server hosting your email and profile data.

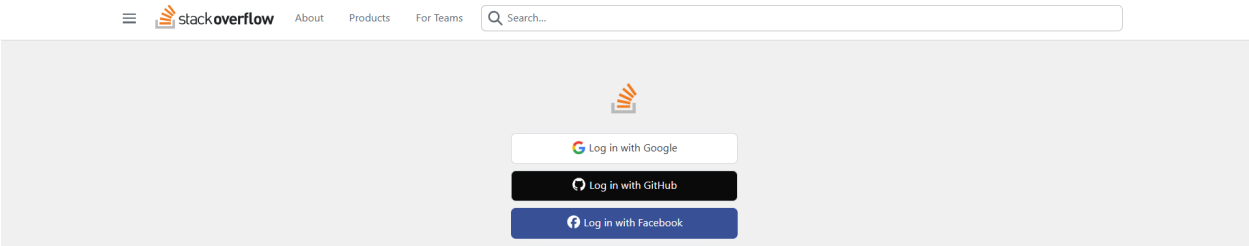
Scopes

- **What:** Specific permissions the client is requesting.

- **Role:** Limits what the client can access.
- **Example:** Stack Overflow asks for **email and profile info** only, not your private repositories.

Ex: You visit **Stack Overflow** and click **"Login with Google/GitHub ."** You want to log in **without giving Stack Overflow your Google/GitHub password**.

- **You click "Login with Google/GitHub."**
- Stack Overflow **redirects you to Google/GitHub** to verify your identity.
- **You log in on Google's/GitHub site** (not on Stack Overflow).
- Google/GitHub tells Stack Overflow: "Yes, this user is authenticated."
- Stack Overflow **lets you in** without ever seeing your Google/**GitHub** password.



Grant Types

In OAuth 2.0, grants are the set of steps a Client has to perform to get resource access authorization.

Or

A Grant Type is the method/way a client application uses to get an access token from the Authorization Server.

Different scenarios require different grant types because applications behave differently (web apps, mobile apps, backend services, devices, etc.) .

The authorization framework provides several grant types to address different scenarios.

1) Authorization Code Grant

The Authorization server returns a single-use Authorization Code to the Client, which is then exchanged for an Access Token.

This is the best option for traditional web apps where the exchange can securely happen on the server side.

The Authorization Code flow might be **used by Single Page Apps (SPA) and mobile/native apps**.

However, here, the client secret cannot be stored securely, and so authentication, during the exchange, is limited to the use of *client id* alone.

A better alternative is the *Authorization Code with PKCE grant*, below.

2) Authorization Code with PKCE (Proof Key for Code Exchange)

This authorization flow is similar to the *Authorization Code* grant, but with additional steps that make it more secure for mobile/native apps and SPAs.

PKCE adds an extra verification step (code challenge + code verifier).

3) **Implicit Grant (Deprecated)**

A simplified flow where the Access Token is returned directly to the Client.

In the Implicit flow, the authorization server may return the Access Token as a parameter in the callback URI or as a response to a form post.

The first option is now deprecated due to potential token leakage.

4) **Resource Owner Password Credentials (ROPC) Grant – Not Recommended**

- User gives the username + password directly to the client.
- Client sends them to Authorization Server to get Access Token.

  Why discouraged:

- User passwords are exposed to the application.
- Not suitable unless the app is 100% trusted (rare today).
- No redirect → used only when redirection is impossible**.**

  This flow is considered legacy/insecure.

5) Client Credentials Grant

Best for:

- Machine-to-machine communication
- Microservice → Microservice
- Backend → Backend
- API - API

- Server-to-server communication

How it works:

- No **user** and **UI** involved.
- Client uses **client ID + client secret** to get an Access Token

Example: Microservice A calling Microservice B using a token issued by IAM (Keycloak/Okta/Cognito).

Used when **no user is involved**.

- Microservice sends:
  → client_id + client_secret
- Gets back Access Token

6) Refresh Token Grant

To obtain a **new Access Token** without requiring the user to log in again. The flow that involves the **exchange of a Refresh Token** for **a new Access Token**.

When used:

- App sessions need to remain active for a long time
- Access tokens expire frequently for security

How it works:

- Refresh Token → exchanged for a new Access Token
- No user interaction required

7) Device Authorization Grant

A grant that enables use by apps on input-constrained devices, such as smart TVs.

OpenID Connect (OIDC)

OpenID Connect (OIDC) is an identity layer built on top of the OAuth 2.0 framework. It allows third-party applications to verify the identity of the end-user and to obtain basic user profile information.

- Identity layer **on top of OAuth 2.0**.
- Standardizes **authentication** — proves who the user is and provides profile info.

- Apps **trust a third-party** (Google, GitHub, Okta) to verify identity.

Why OIDC?

- OAuth alone is about **accessing data**, not authentication.
- Without OIDC: Apps must guess who the user is.
- With OIDC:
    - Google/GitHub send a **special ID token (JWT)** proving exactly who you are.
    - Apps can trust this token.
    - Secure, standardized, works across multiple sites.

Ex: 1️⃣ **What happens when you log in using Google/GitHub (the Stack Overflow example):**

- You click **"Login with Google."**
- Stack Overflow sends you to Google to log in.
- Google asks: "Who are you?" You log in there.
- Google tells Stack Overflow: **"Yes, this person is authenticated. Here's some info about them."**
- Stack Overflow logs you in — **without seeing your Google password.**

This works even **without knowing OpenID explicitly**, because OAuth can send basic info like your email.

2️⃣ **So why OpenID Connect (OIDC)?**

OAuth alone is for **accessing data**, not for **authentication**. That means:

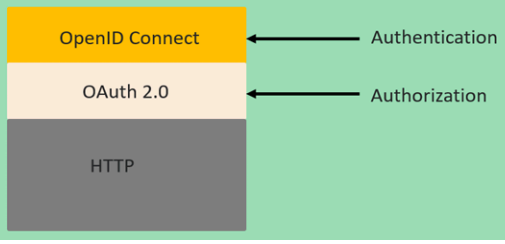
- Without OIDC:
    - OAuth gives you a token to access resources (like email), and you can "guess" the user's identity.
    - Each app might handle the identity differently — no standard way.
- With OIDC:
    - It standardizes **identity verification**.
    - Google/GitHub send a **special ID token** (JWT) that proves exactly **who you are**.
    - Apps can trust this token instead of guessing.
    - It's safe, secure, and consistent across sites.

WHAT IS OPENID CONNECT & WHY IT IS IMPORTANT ?

eazy bytes

What is OpenID Connect?

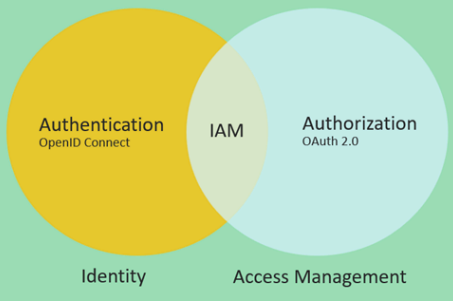
- OpenID Connect is a protocol that sits on top of the OAuth 2.0 framework. While OAuth 2.0 provides authorization via an access token containing scopes, OpenID Connect provides authentication by introducing a new ID token which contains a new set of information and claims specifically for identity.
- With the ID token, OpenID Connect brings standards around sharing identity details among the applications.



The OpenID Connect flow looks the same as OAuth. The only differences are, in the initial request, a specific scope of openid is used, and in the final exchange the client receives both an Access Token and an ID Token.

Why is OpenID Connect important?

- Identity is the key to any application. At the core of modern authorization is OAuth 2.0, but OAuth 2.0 lacks an authentication component. Implementing OpenID Connect on top of OAuth 2.0 completes an IAM (Identity & Access Management) strategy.
- As more and more applications need to connect with each other and more identities are being populated on the internet, the demand to be able to share these identities is also increased. With OpenID connect, applications can share the identities easily and standard way.



OpenID Connect add below details to OAuth 2.0

1. OIDC standardizes the scopes to openid, profile, email, and address.
2. ID Token using JWT standard
3. OIDC exposes the standardized “/userinfo” endpoint.

Feature	OAuth 2.0	OpenID Connect (OIDC)
Purpose	Authorization (who can access resources)	Authentication (who the user is)
What it provides	Access Token → allows client to access resources	ID Token → provides verified user identity; Access Token can also be included
User info	Not standardized	Standardized (name, email, profile info, etc.)
Login	Not required; focuses on granting permissions	Required; lets apps verify user identity
Example	A site asks GitHub: "Can I read the user's repositories?"	A site asks GitHub: "Who is this user?"
Use case	Giving third-party apps access to your data without sharing password	Logging in to apps using your existing account (SSO) safely

Centralized Identity and Access Management (IAM)

Definition: A **central system** that manages:

1. **User identities** – who the users are.
2. **Authentication** – verifying user logins (passwords, social login, etc.).
3. **Authorization** – deciding what users or apps are allowed to do.
4. **Tokens & sessions** – issuing and managing access tokens, refresh tokens, or ID tokens for apps.

Why use a centralized IAM?

- Microservices **should not implement their own login/auth logic** individually — it's hard to maintain and insecure.
- Central IAM provides a **single source of truth** for identity.
- Supports **single sign-on (SSO)** across multiple applications.
- Makes it easier to **manage permissions, policies, and tokens** consistently.

Examples of Centralized IAM systems : Keycloak https://www.keycloak.org/ ,Okta https://www.keycloak.org/ ,Amazon Cognito ,ForgeRock and others.

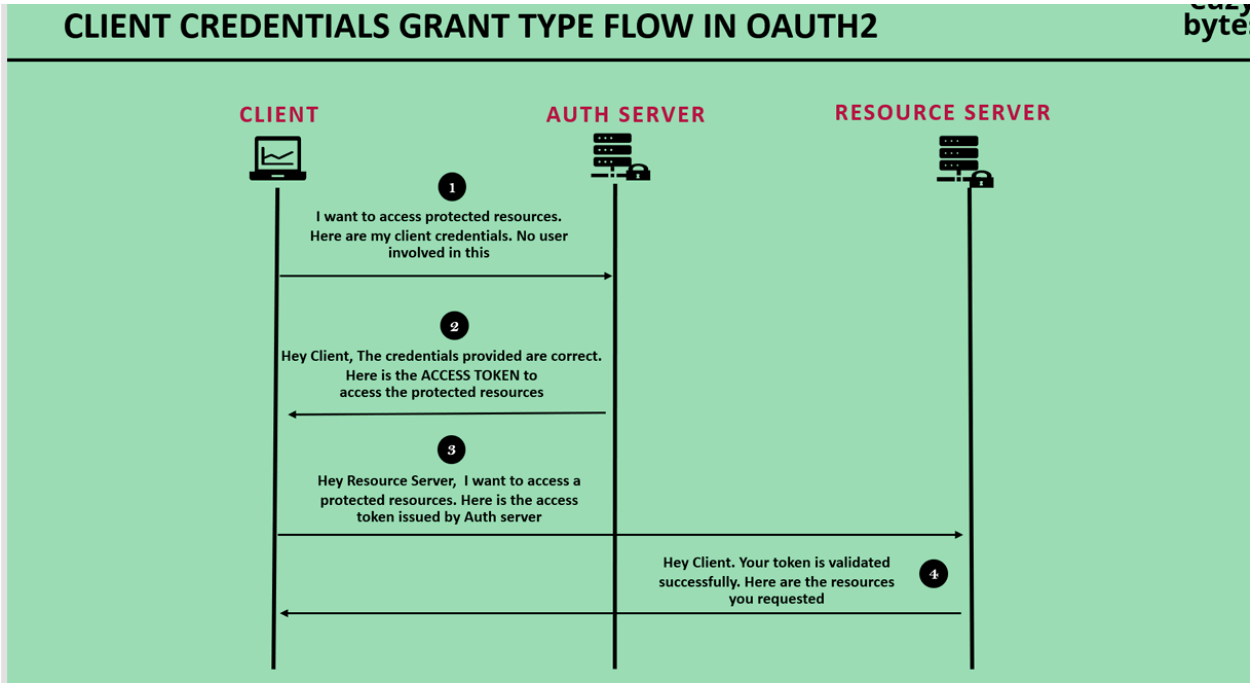
IAM System	Key Feature
Okta	Cloud-based identity management, supports SSO, MFA, OAuth/OIDC
Keycloak	Open-source IAM, can self-host, supports OAuth/OIDC, SAML
Amazon Cognito	AWS service, integrates with AWS apps, handles user pools and identity pools

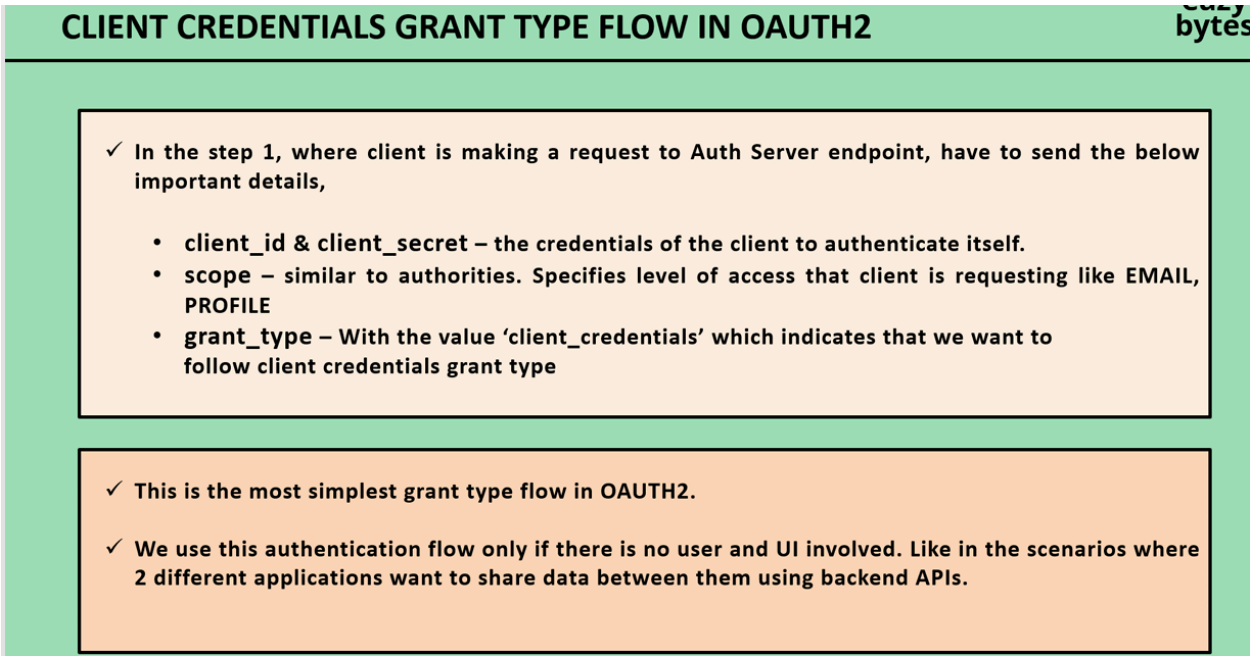
ForgeRock Enterprise IAM, supports authentication, authorization, directory services

Client Credentials Grant Type Example

Used when **no user and UI is involved**. For only Mostly used for

microservices-microservices ,server-to-server communication.





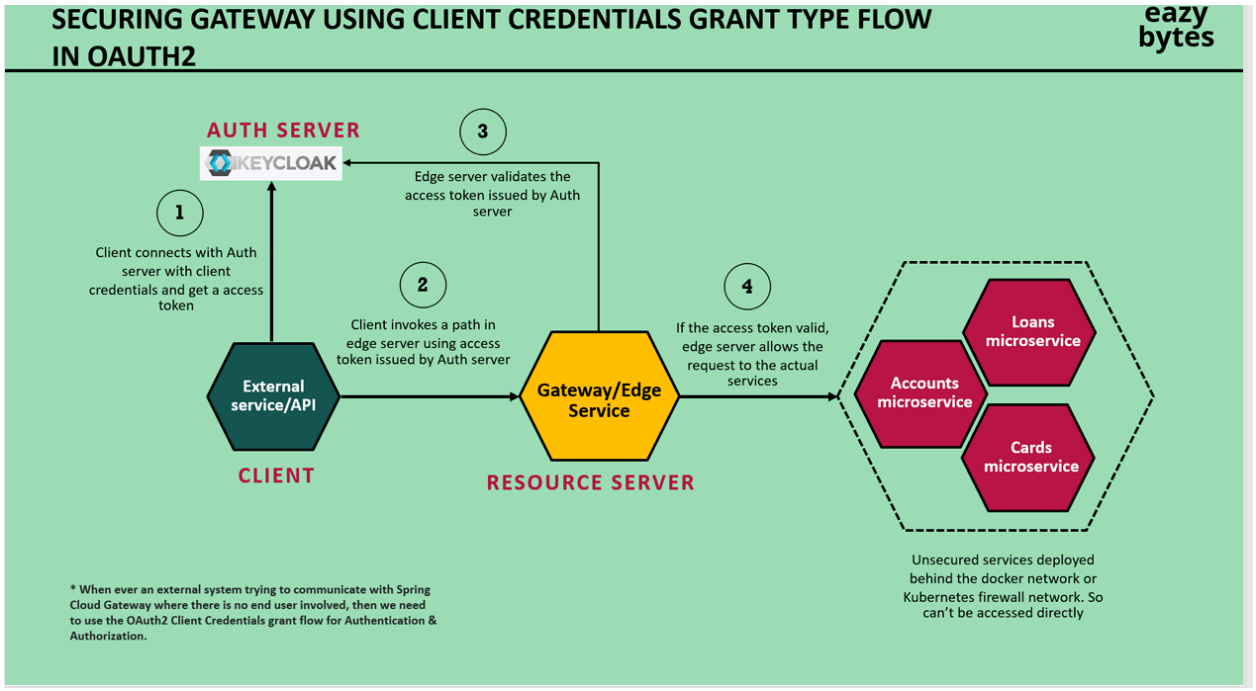
Ex:

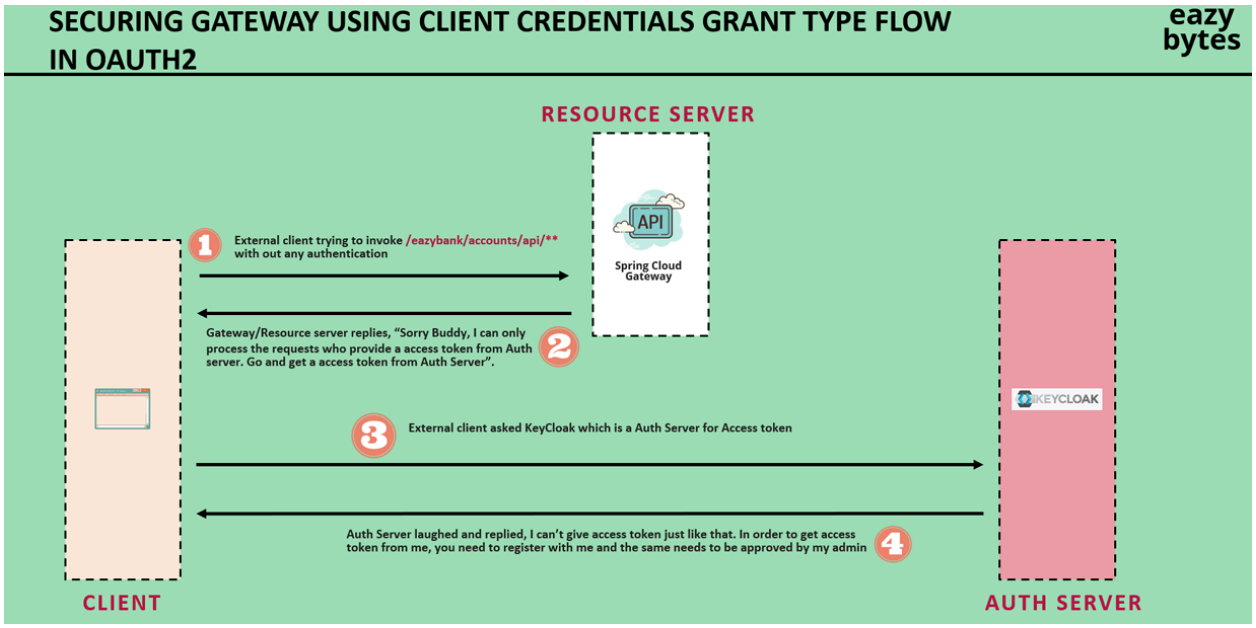
External Client → Any third-party system calling your APIs

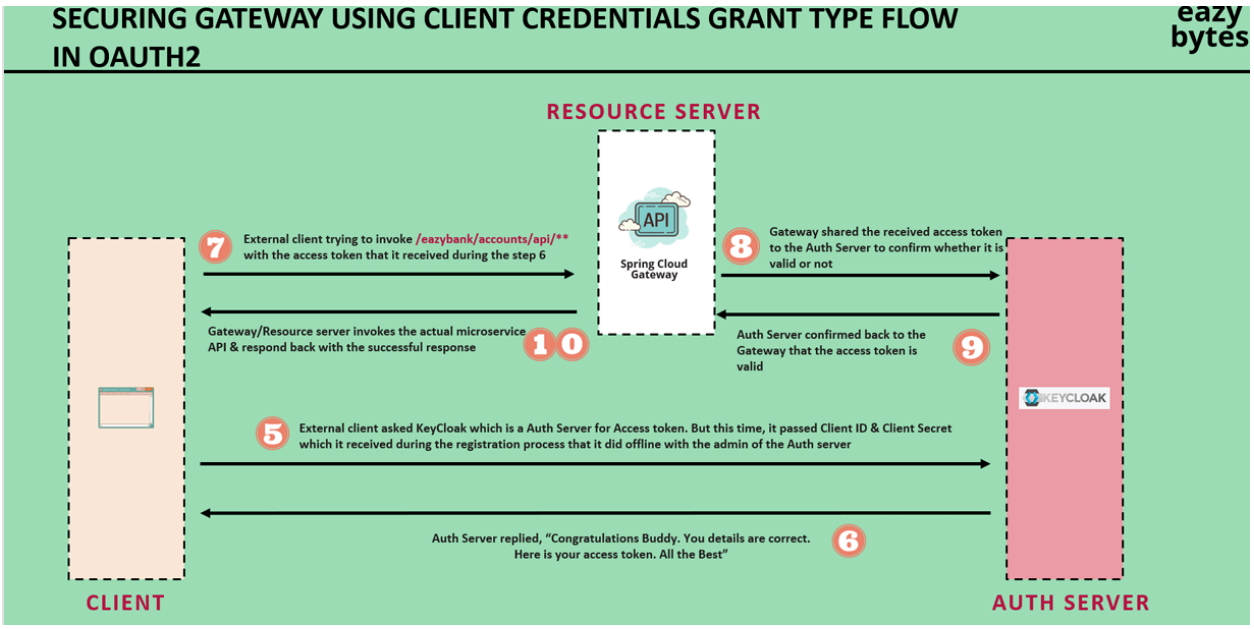
Keycloak → Authorization Server (Auth Server)

API Gateway (Spring Cloud Gateway / Zuul / APIGW) → Resource Server

Microservices → Protected resources behind the gateway.







Integration

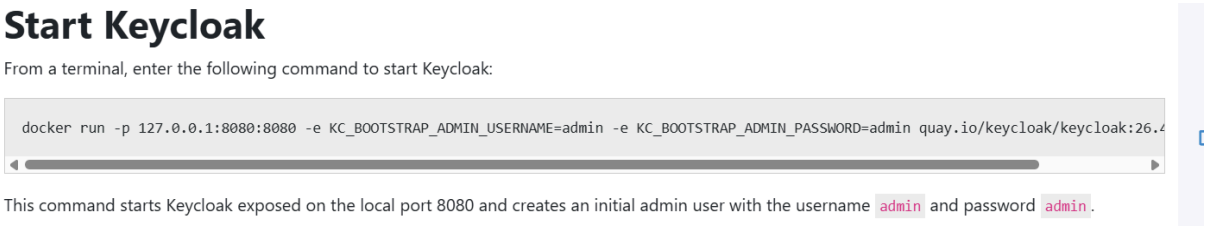
1) Set up an **Authorization Server** using **Keycloak .**

i) **Go to the official Keycloak website** https://www.keycloak.org/guides . Use the "Get Started" guides:https://www.keycloak.org/guides . Keycloak can be started in many ways (OpenJDK, Docker, Podman, etc.).

ii) Start Keycloak using Docker

Click the Docker option : https://www.keycloak.org/getting-started/getting-started-docker

Copy the command from the website and run it in your local system.



Change the exposed port to avoid conflicts (if port 8080 already exists).

```
C:\Users\jakkula.ramesh>docker run -d -p 127.0.0.1:7080:8080 -e KC_BOOTSTRAP_ADMIN_USERNAME=admin -e KC_BOOTSTRAP_ADMIN_PASSWORD=admin quay.io/keycloak/keyc
loak:26.4.6 start-dev
```

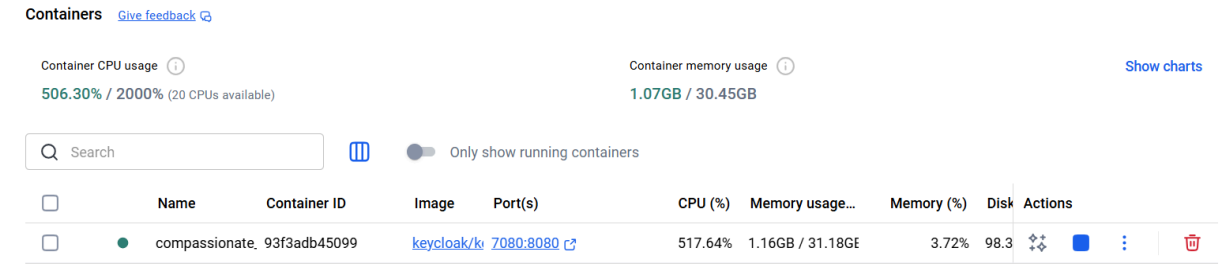
iii) Open Docker Desktop and confirm that the Keycloak container is running. Open browser http://localhost:

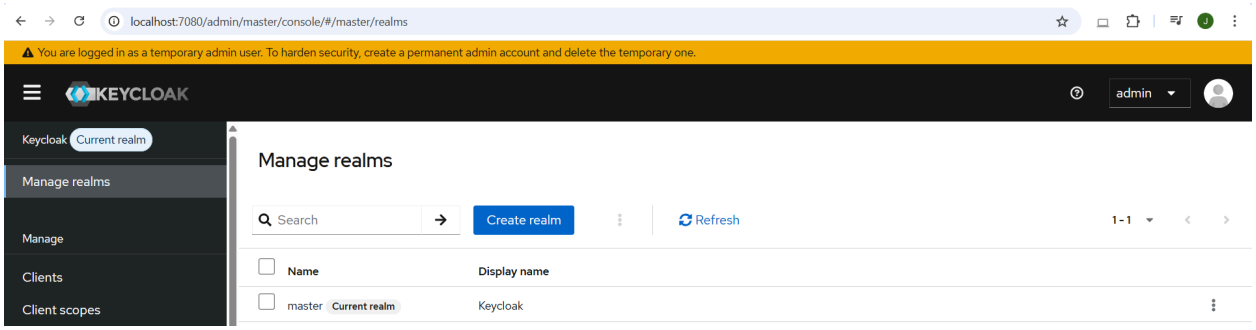
A **Realm** in Keycloak is simply a **separate security domain**.Think of it as a **folder that contains its own users, clients, roles, and settings**, completely isolated from others.

- A realm = **one project / one application environment**

- Inside a realm you create:
  - Users
  - Roles
  - Clients
  - Groups
  - Security settings

Everything inside one realm is isolated from other realms.





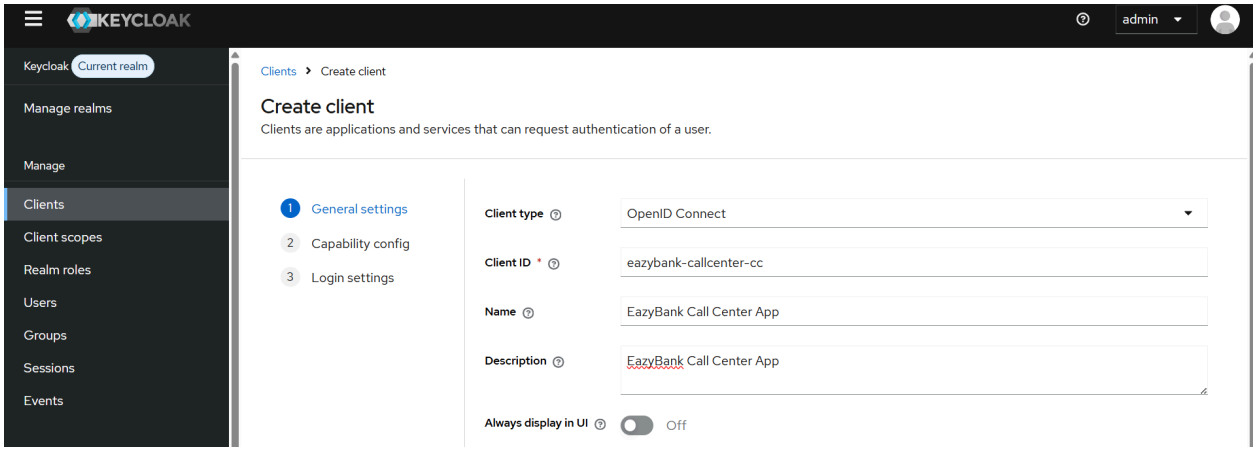
2) Register Client Application with Authorization Server (Keycloak)

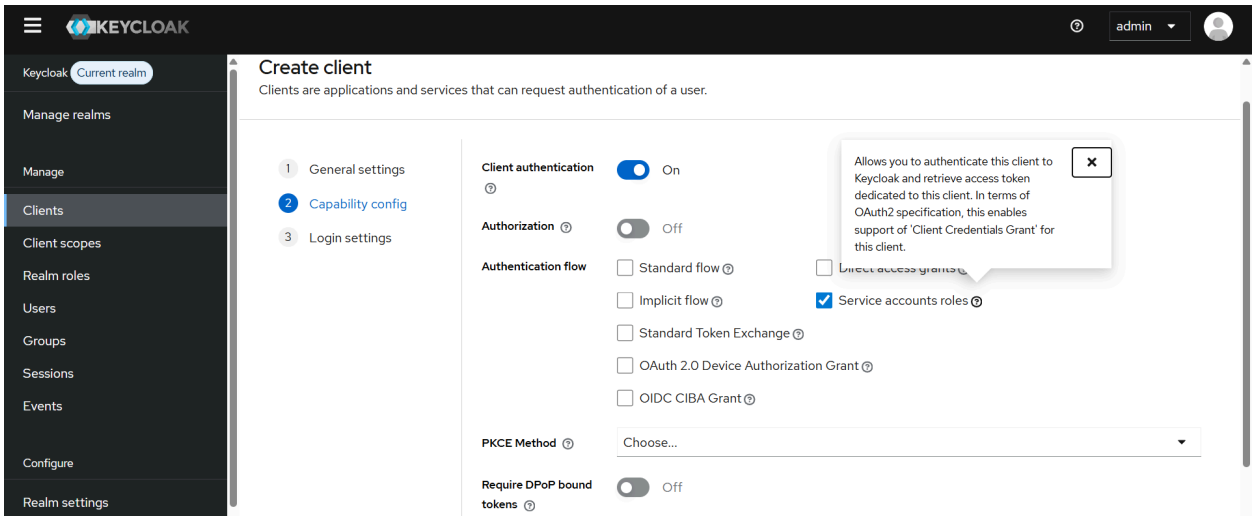
i) Open the **Keycloak dashboard**. Select the **Realm** you created. Click **Clients → Create Client**.

Enter the required details and save :

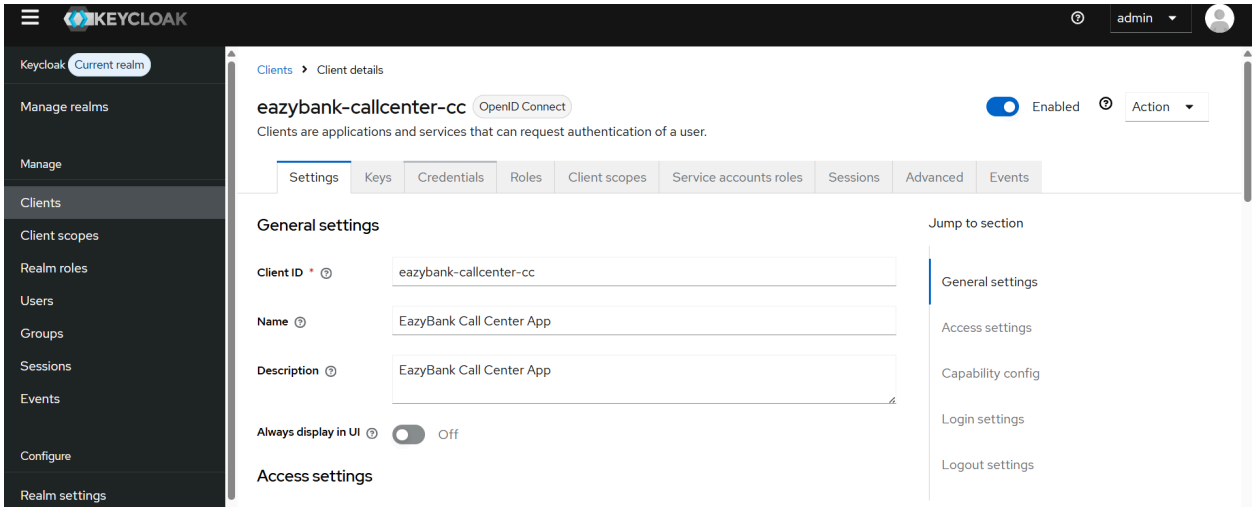
- **Client ID**
- **Client type** (e.g., OpenID Connect / SAML)

- **Client authentication settings**
- **Redirect URIs** (if needed)





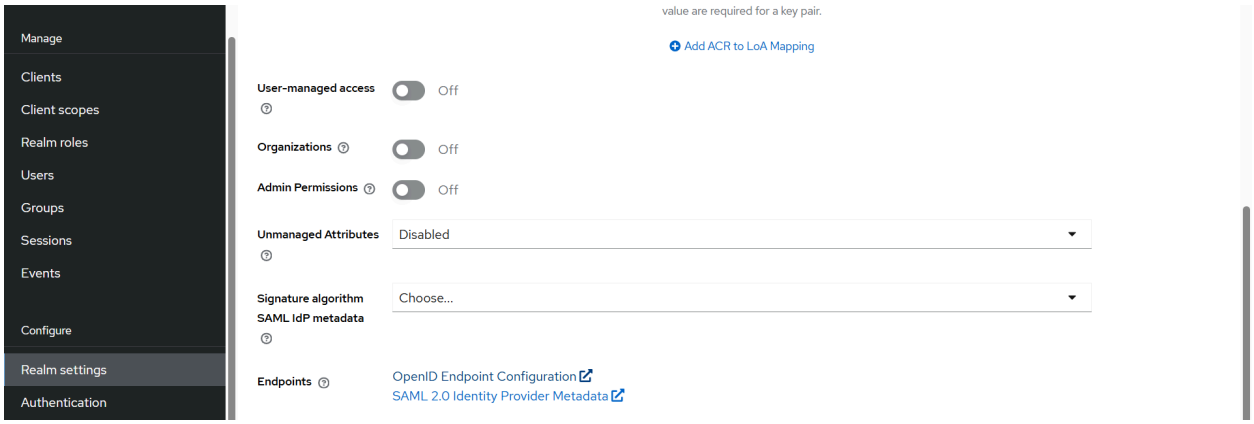
ii) Once created, Keycloak will show



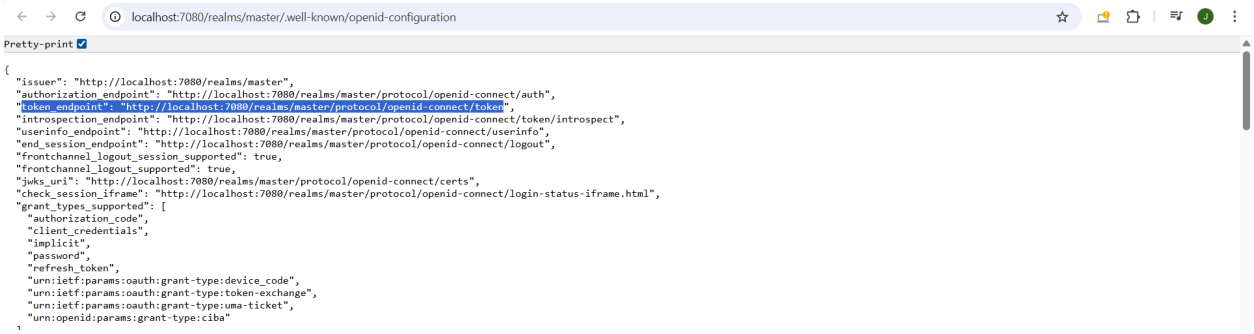
3) Getting Access Token from Keycloak(auth server)

i) the client application needs the **Token Endpoint URL** to get the access token.

Go to Keycloak -> Select your **Realm -> Go to Realm Settings → OpenID Endpoint Configuration**



There you will see all **OpenID Endpoint Configuration** and open it , copy the token_endpoint.



ii) Open Postman and create a POST request to the **token endpoint**.

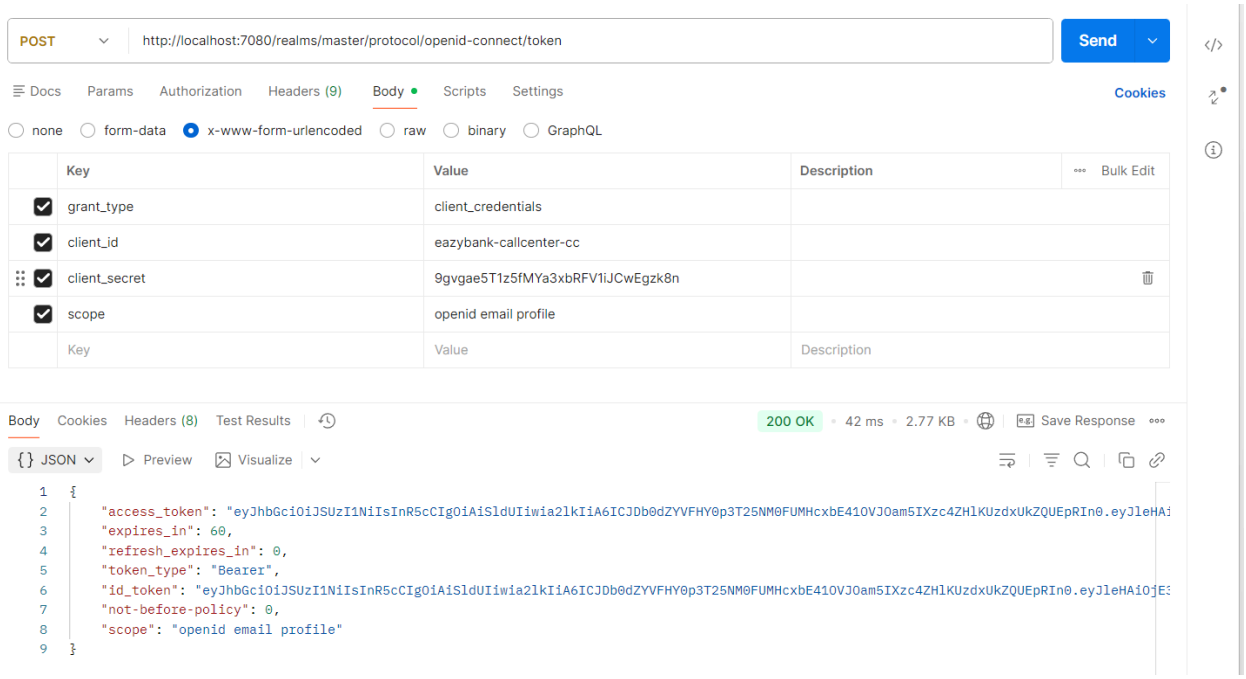
Whatever scopes you mention must exist in your Keycloak client settings. In **Body → x-www-form-urlencoded** add: grant_type, client_id , client_secret , scope.

- Select your **Client -> credentials** for cred details / Go to **Client Scopes**
- You will see
    - **Default Scopes**
    - **Optional Scopes**

Only scopes allowed here can be requested.

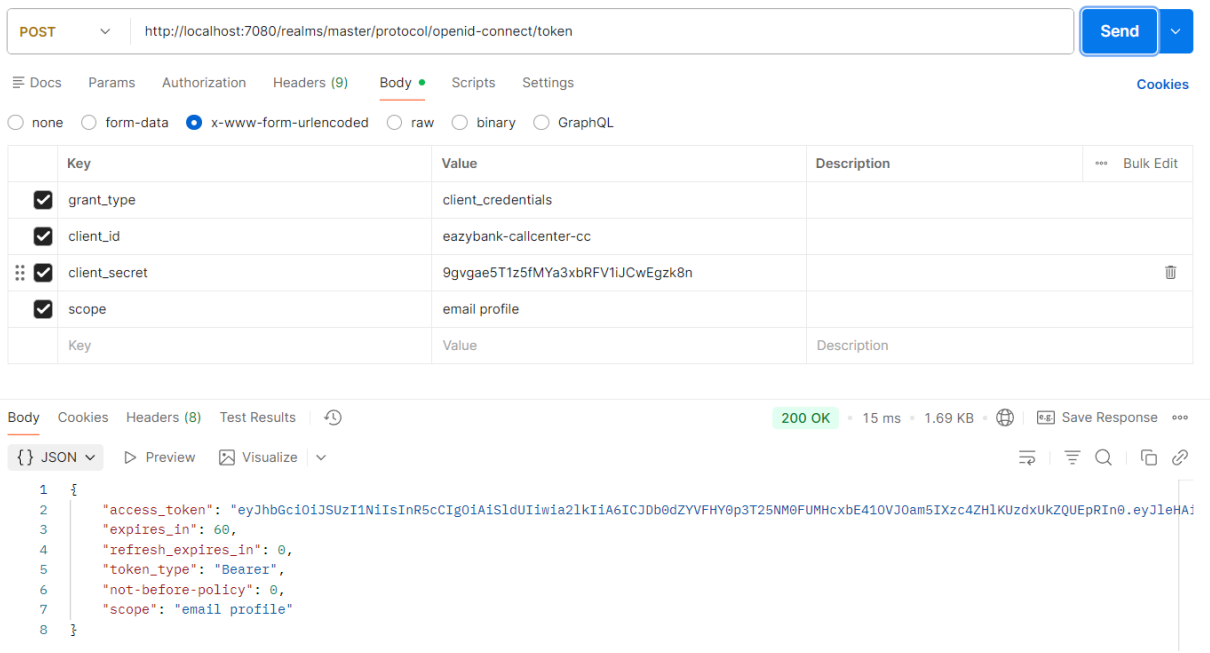
If you include **openid** in the scope → you get **Access Token + ID Token.**

scope : openid email profile



If you **do not** include openid → you get **only Access Token**

scope : email profile



4) Set Up **Resource Server** using **Gateway**

i) Add Required Dependencies in Gateway

Your API Gateway must validate incoming Access Tokens (JWT).

Add the following **three dependencies** in the gateway's pom.xml:

```xml
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-oauth2-resource-server</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-oauth2-jose</artifactId>
        </dependency>
```

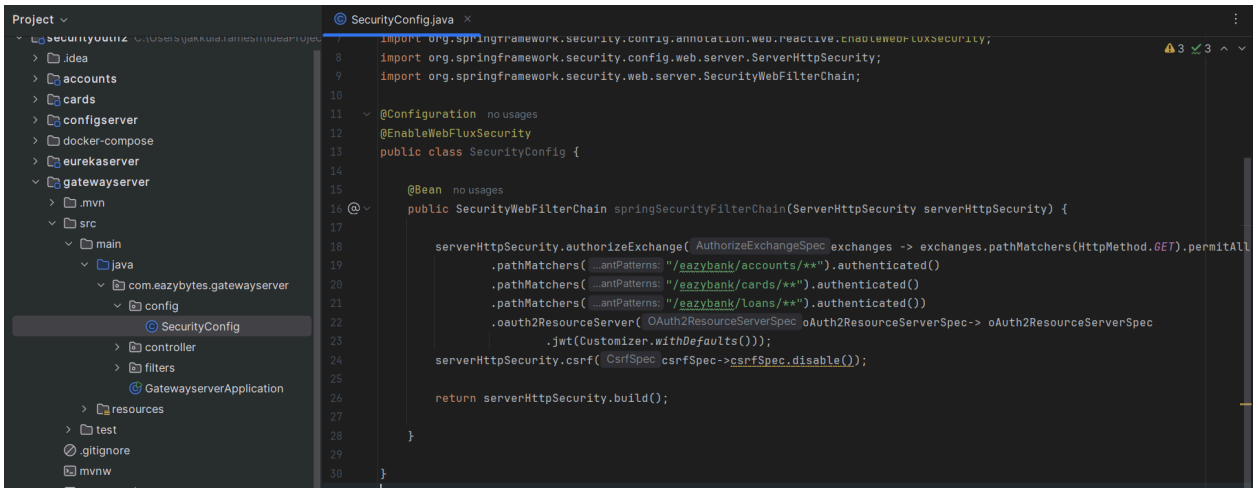
spring-boot-starter-security → enables Spring Security

spring-security-oauth2-resource-server → turns gateway into a resource server

spring-security-oauth2-jose → required to verify JWT signatures (RS256)

ii) Create SecurityConfig Class

Inside the **config** package, create SecurityConfig.java. Annotate with @configuration as we can create beans inside it.



Define which API paths require authentication
All requests to:

- /eazybank/accounts/**
- /eazybank/cards/**
- /eazybank/loans/**

must have a valid JWT access token.

Define which paths are public
All GET requests are allowed without authentication (permitAll()).

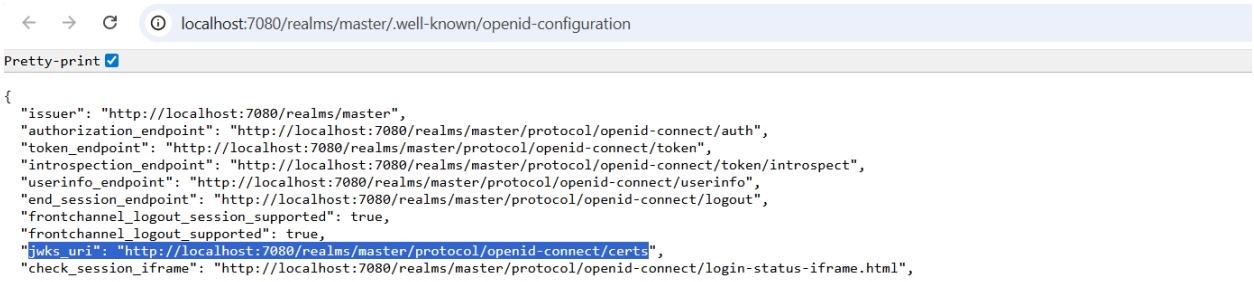
Enable JWT validation
oauth2ResourceServer().jwt() tells Spring to treat this service as a **Resource Server** and validate incoming JWT tokens issued by Keycloak.

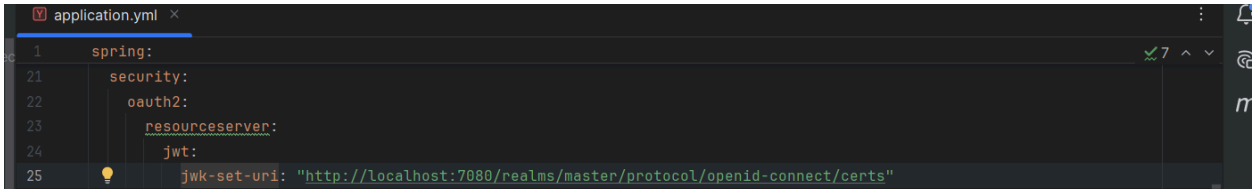
Disable CSRF (no browser)
Because this is an API Gateway (no login page / no sessions), CSRF is disabled.

iii) Add OAuth2 Resource Server Properties in application.yml

Your Gateway needs to know **where to fetch Keycloak's public keys (JWKS)**.

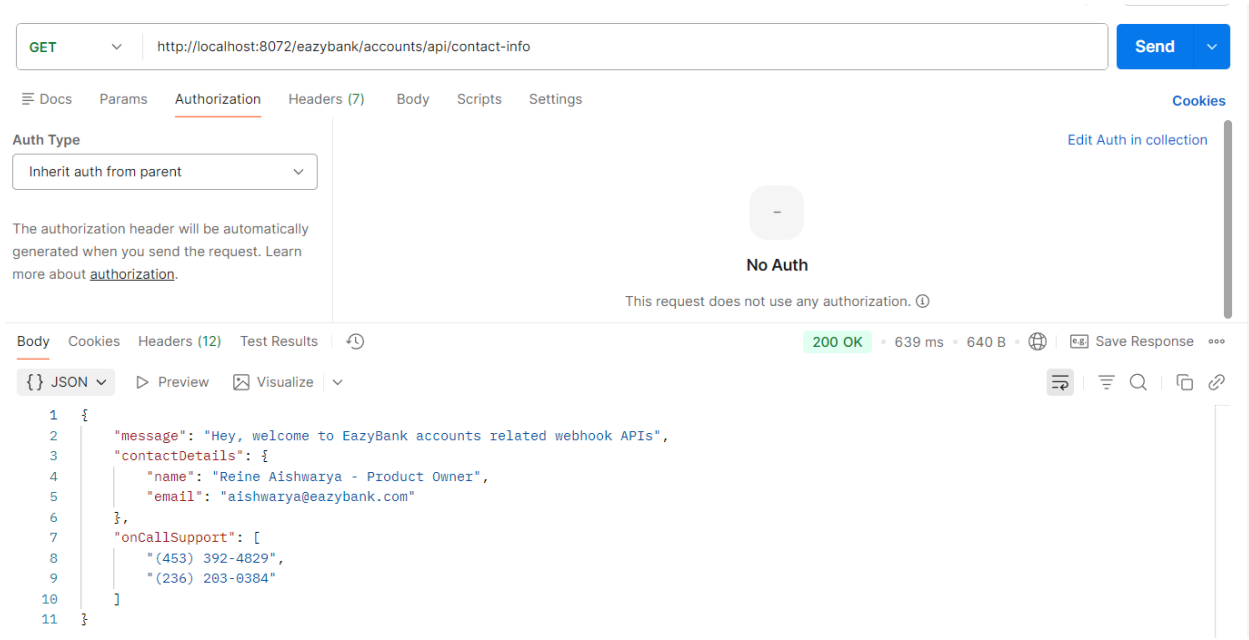
Go to: **Keycloak → Your Realm → Realm Settings → OpenID Endpoint Configuration**
Copy the **issuer** or **jwks_uri**.



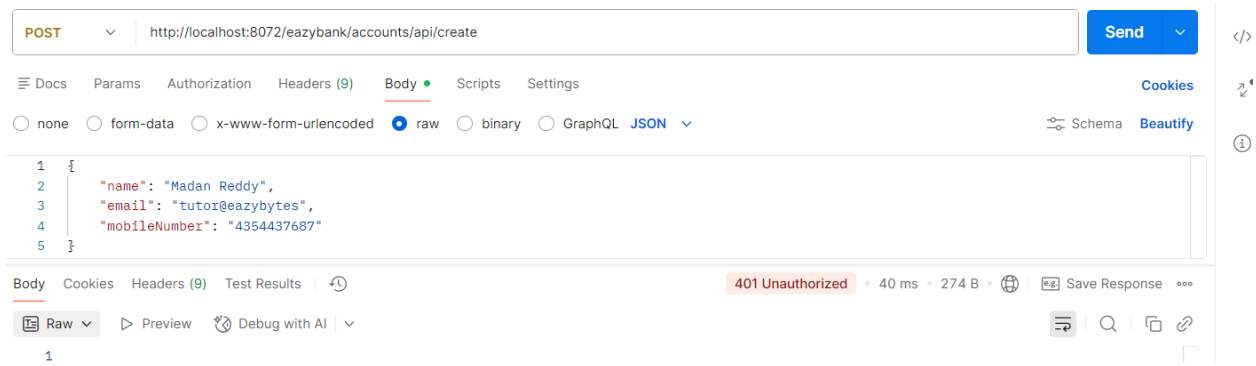


iv) start all services config,eureka and all microservices then gateway server.

Hit any public API(GET as we don't define any authentication)



Hit any protected API → **You will get 401 Unauthorized**

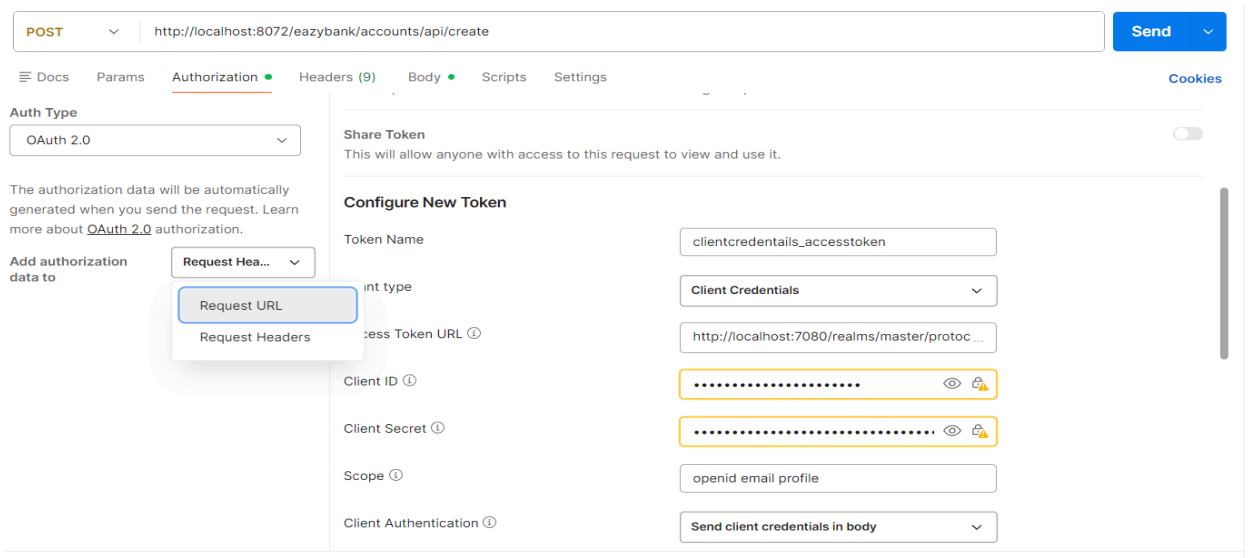


This means the gateway is now a Resource Server

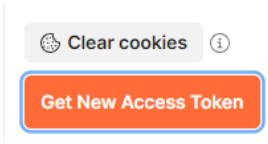
We can either hit token endpoint url , copy the access token and pass in the header of api and test

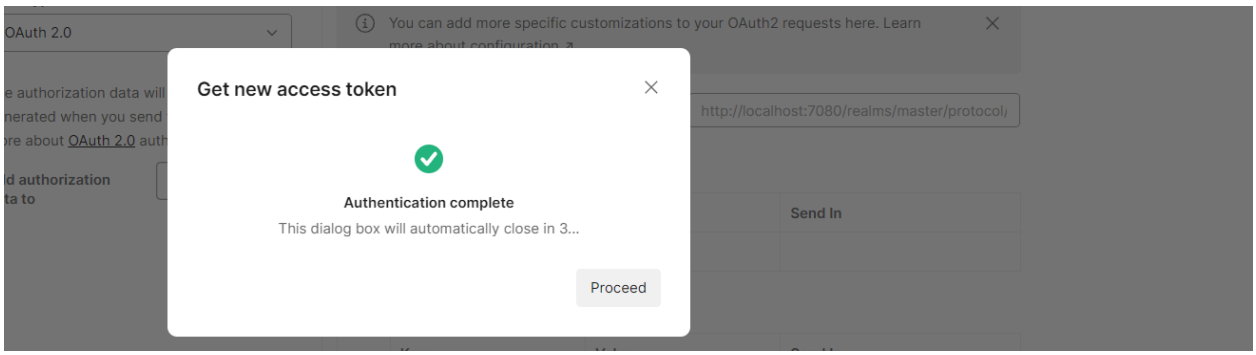
Authorization: Bearer <ACCESS_TOKEN>

Or we can directly provide auth details in the api itself so it will hit the endpoint url and pass access in header in api request .



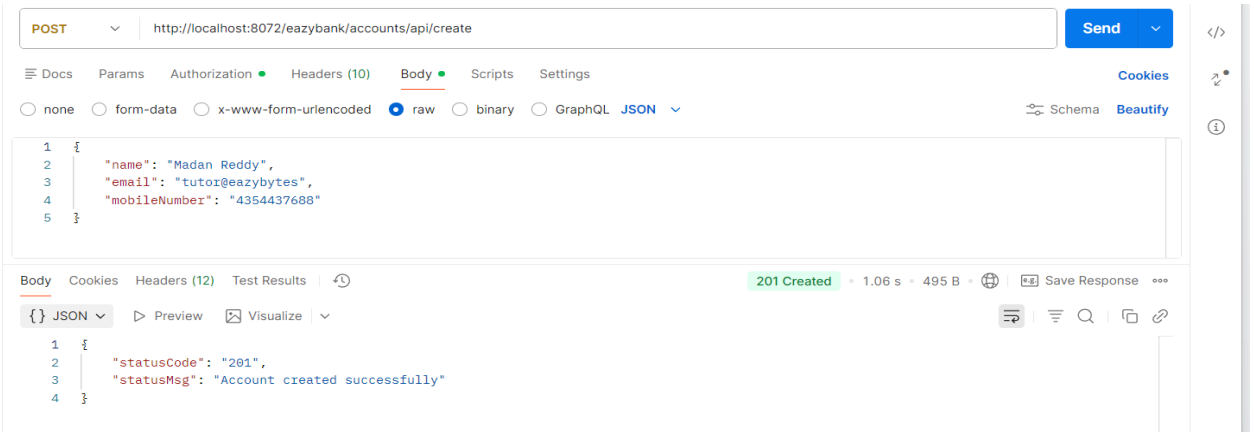
Click on Get New Access Token





Then click on **Use Token** then hit the api.

If the token is valid → API works.



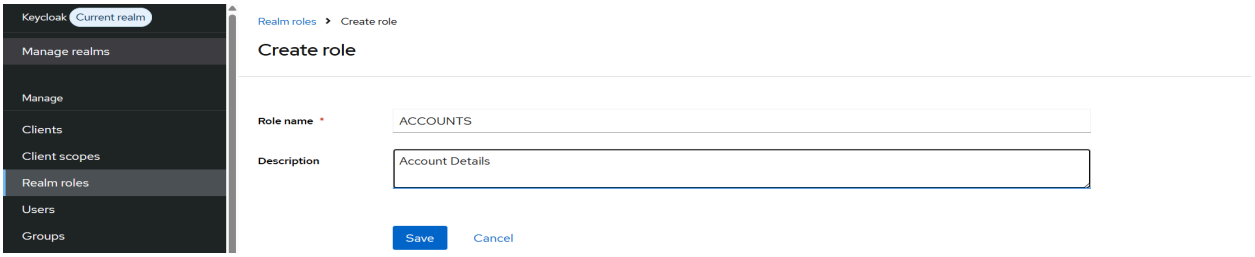
Right now we have completed only *Authentication*

- Gateway checks **WHO** the user/client is
- Validates JWT token
- Allows or blocks based on token validity
- No role-based permissions yet

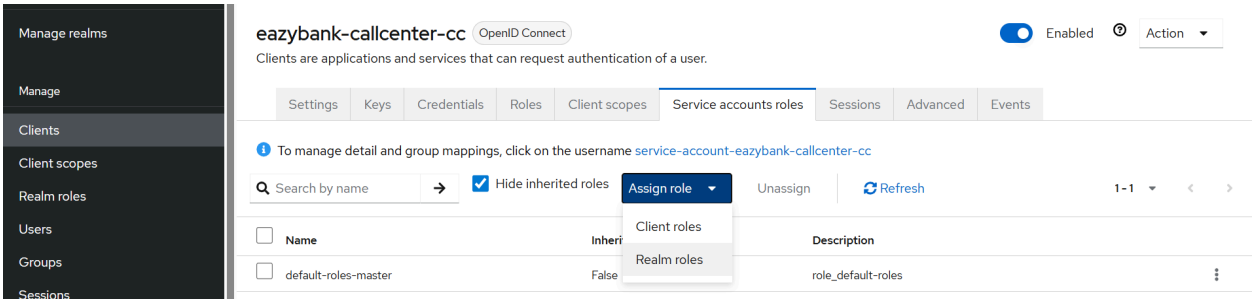
This is **Authentication only**.

Authorization (Role-Based Access Control)

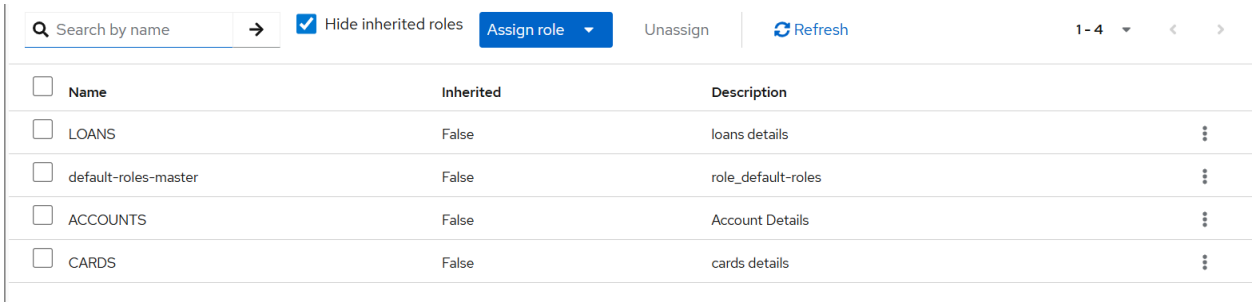
i) To enable Authorization in your Gateway: First we need to create Roles in **Keycloak**.



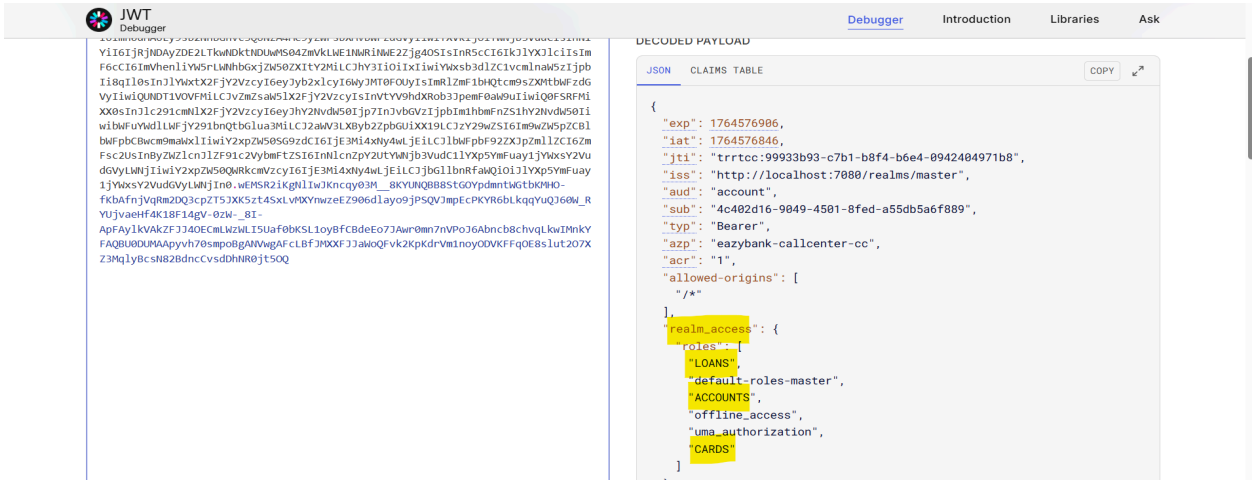
Assign these roles to a client application.



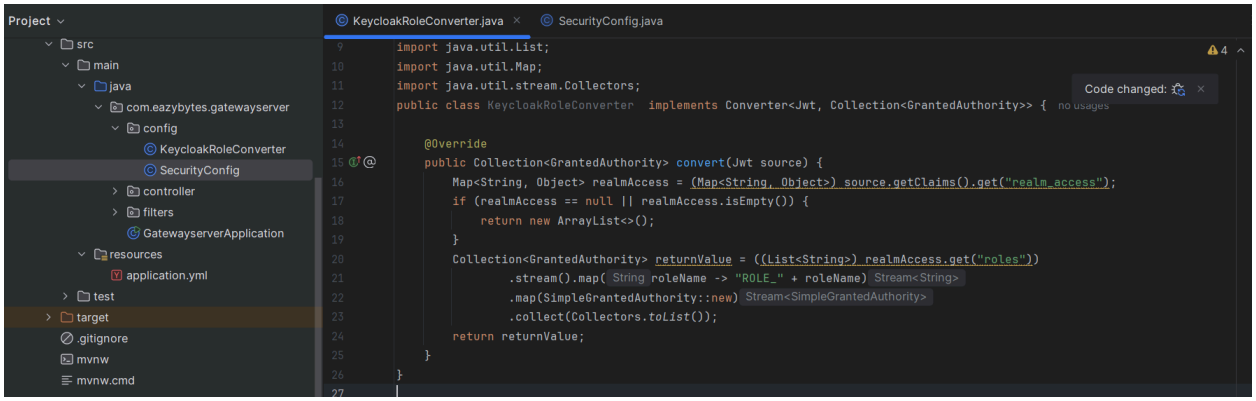
Select the roles and Assign

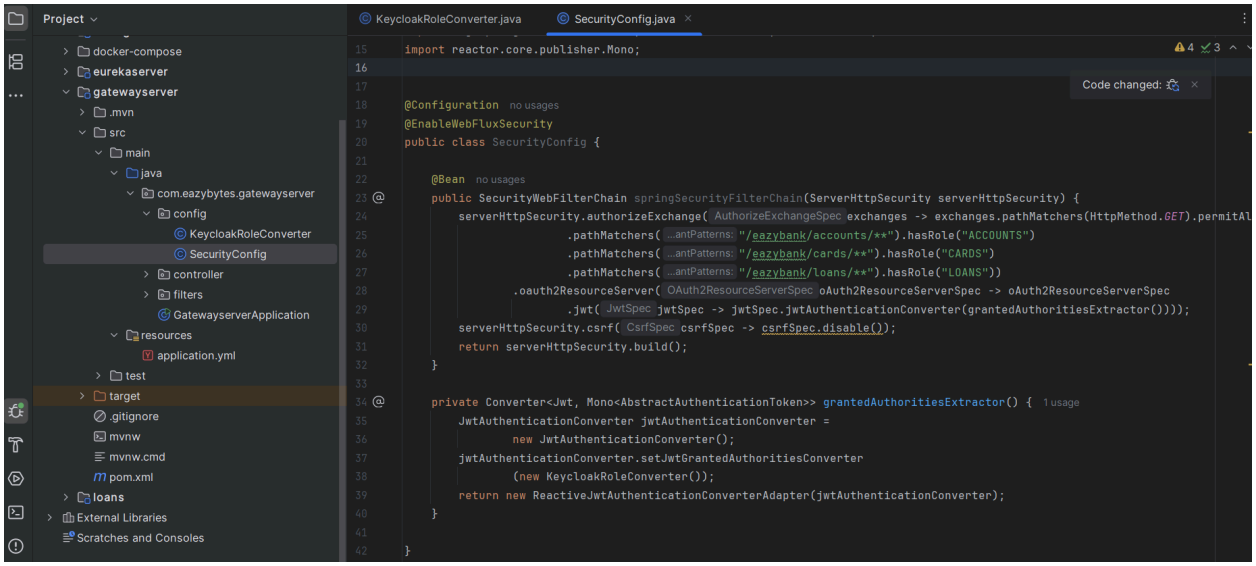


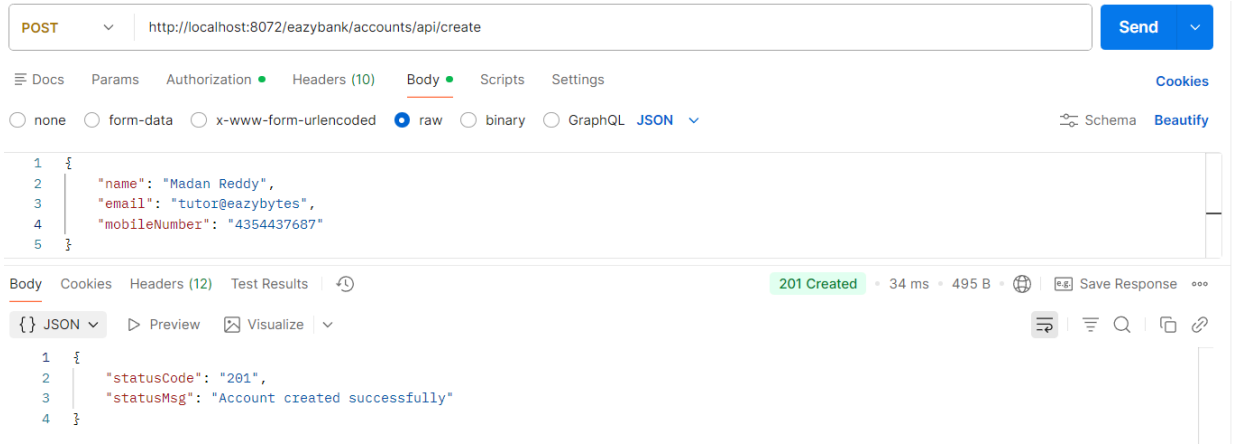
We can see those roles in access token using [jwt.io](jwt.io)



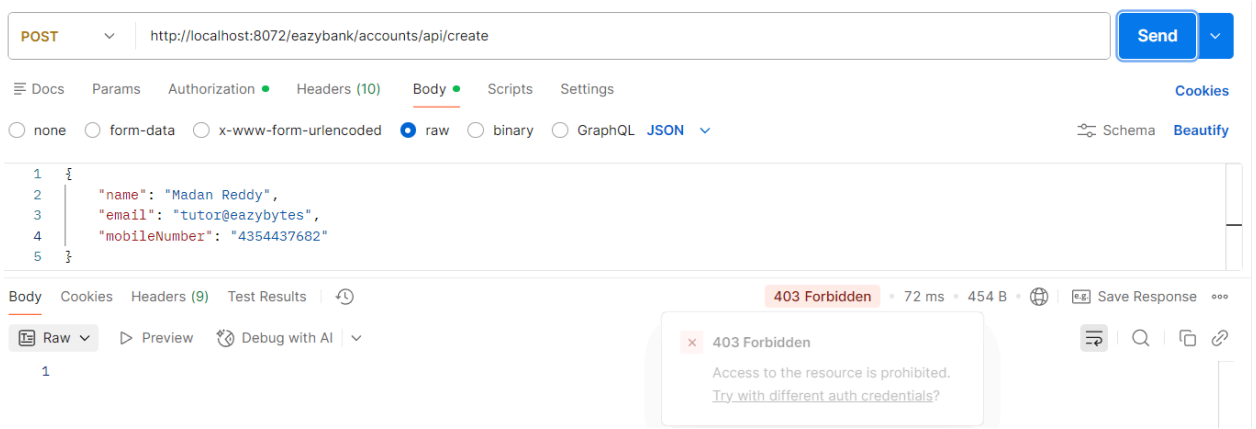
To use these roles from the access token in the gateway server , we needed to create a class for logic and use them in the class where you mention pathMatchers.





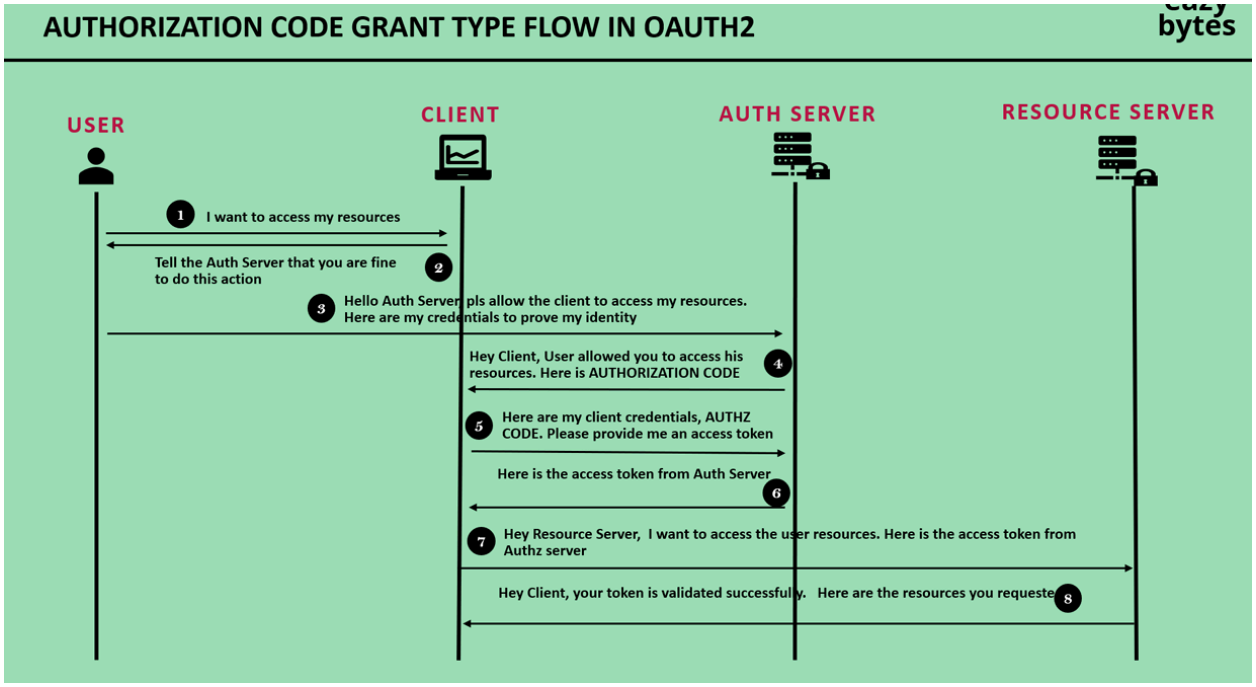


For example if we delete Account role in keycloak and test.



Authorization Code Grant Type Example

Authorization Code Grant Flow is used **when a USER is involved** and a **UI (web/mobile)** is present.



- ✓ In the steps 2 & 3, where client is making a request to Auth Server endpoint have to send the below important details,
  - **client_id** – the id which identifies the client application by the Auth Server. This will be granted when the client register first time with the Auth server.
  - **redirect_uri** – the URI value which the Auth server needs to redirect post successful authentication. If a default value is provided during the registration then this value is optional
  - **scope** – similar to authorities. Specifies level of access that client is requesting like READ
  - **state** – CSRF token value to protect from CSRF attacks
  - **response_type** – With the value 'code' which indicates that we want to follow authorization code grant

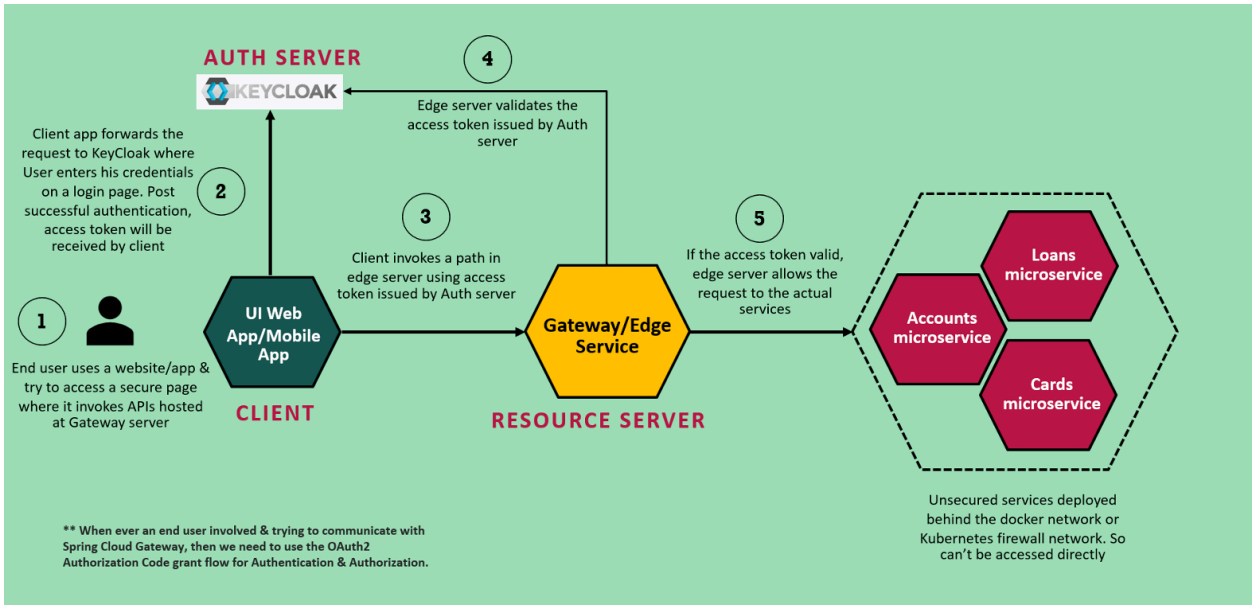
- ✓ In the step 5 where client after received a authorization code from Auth server, it will again make a request to Auth server for a token with the below values,
  - **code** – the authorization code received from the above steps
  - **client_id & client_secret** – the client credentials which are registered with the auth server. Please note that these are not user credentials
  - **grant_type** – With the value 'authorization_code' which identifies the kind of grant type is used
  - **redirect_uri**

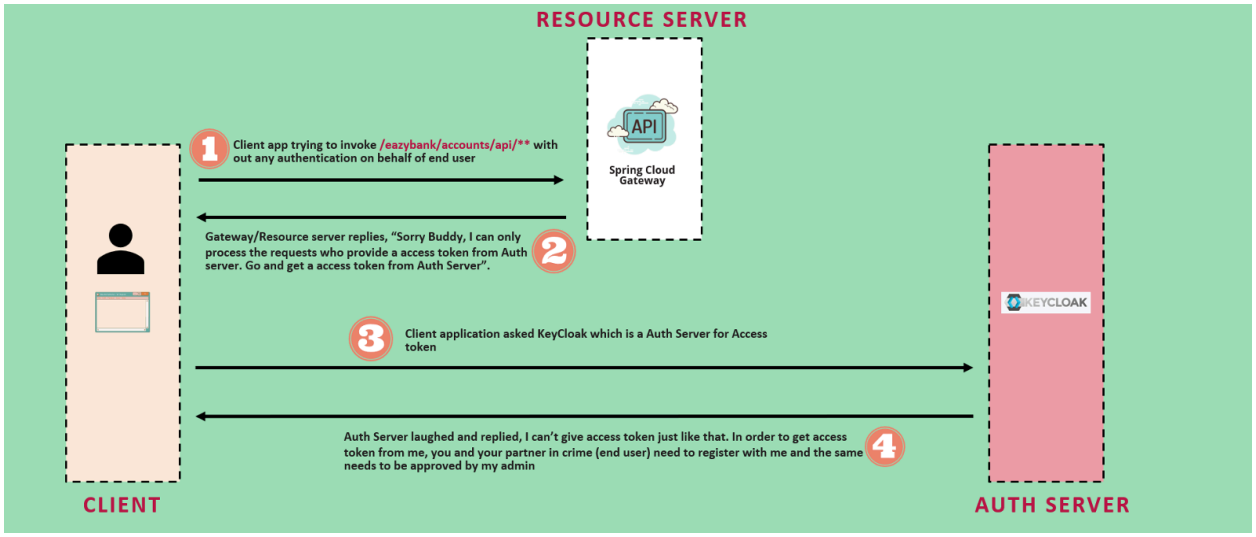
✓ We may wonder that why in the Authorization Code grant type client is making request 2 times to Auth server for authorization code and access token.

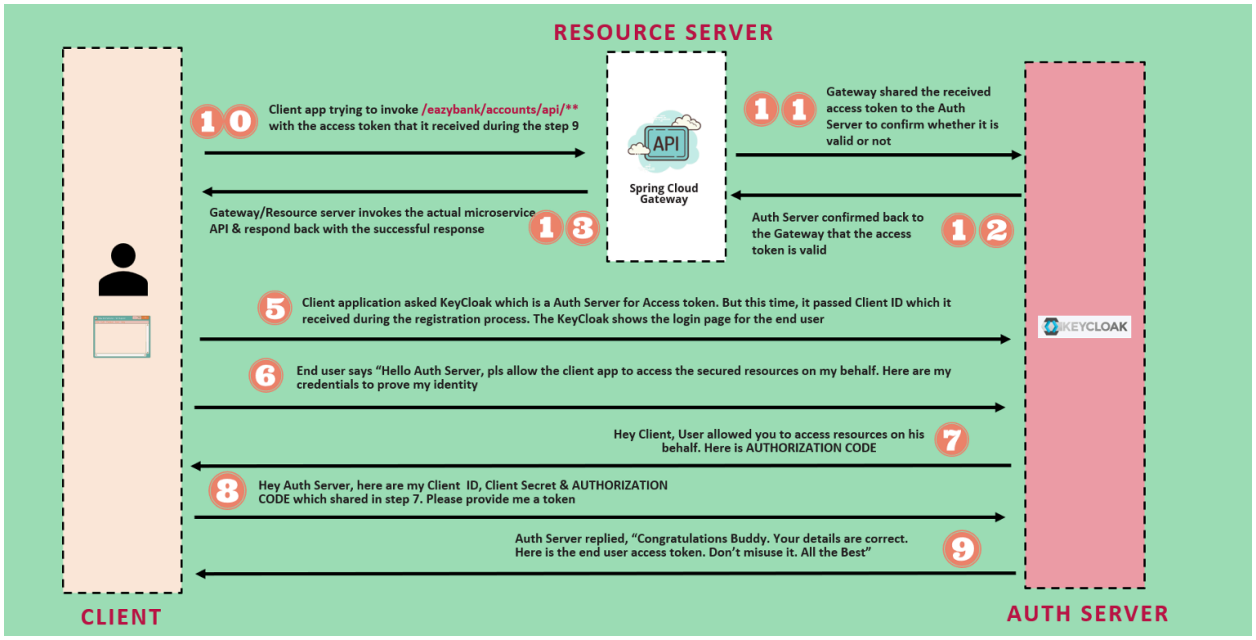
- In the first step, authorization server will make sure that user directly interacted with it along with the credentials. If the details are correct, auth server send the authorization code to client
- Once it receives the authorization code, in this step client has to prove it's identity along with the authorization code & client credentials to get the access token.

✓ Well you may ask why can't Auth server directly club both the steps together and provide the token in a single step. The answer is that we used to have that grant type as well which is called as **'implicit grant type'**. But this grant type is deprecated as it is less secure.

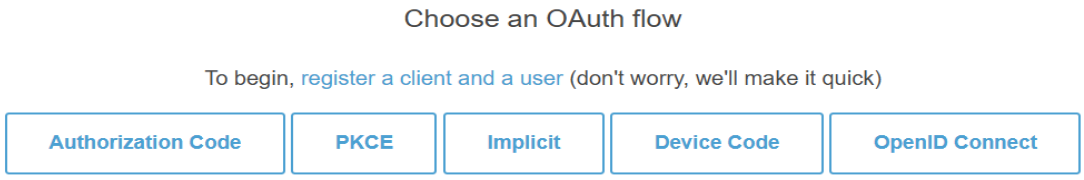
Sample example



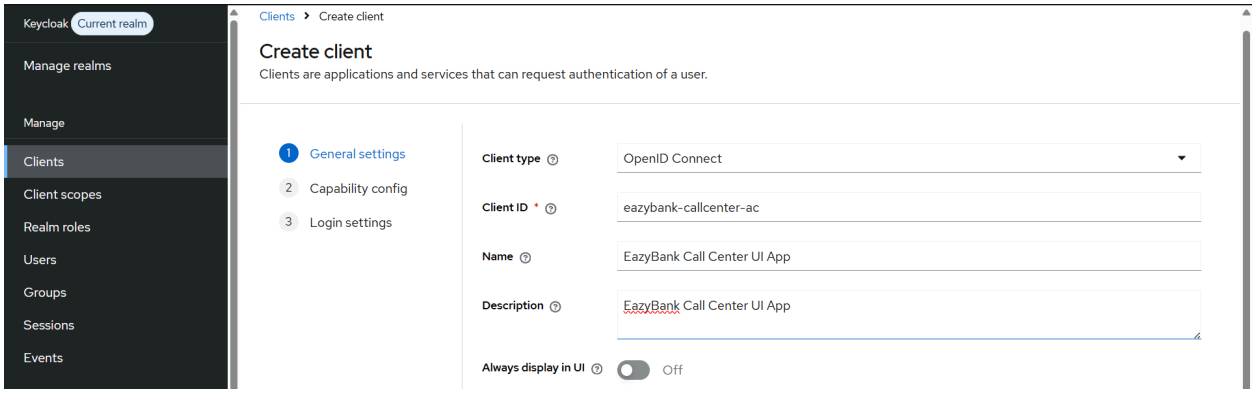


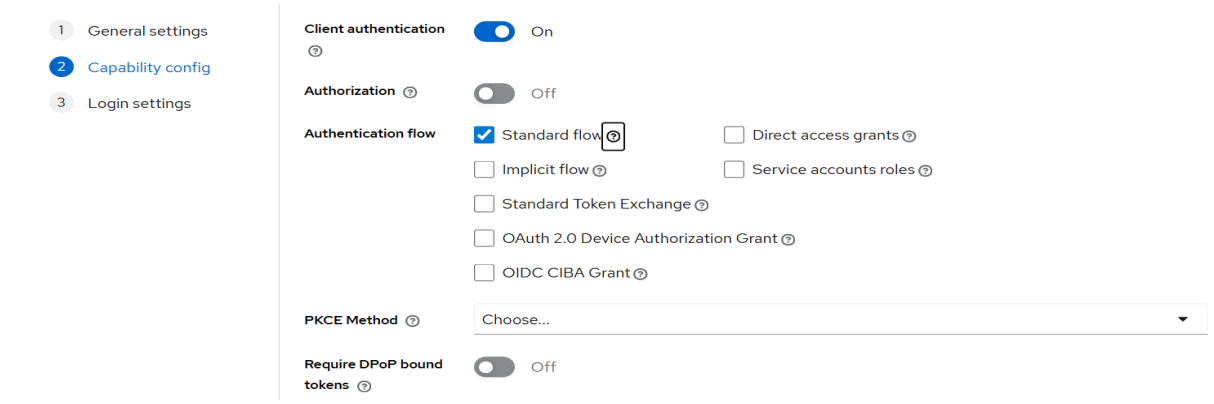


We use the oAuth playground to understand better https://www.oauth.com/playground/

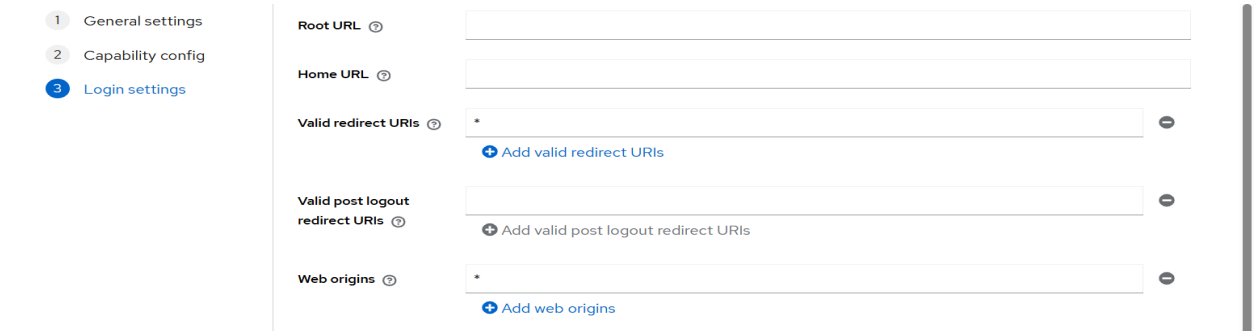


To integrate this first we need to create a client and user in keycloak. And also assign roles to users. Go to **Keycloak Dashboard → Your Realm → Clients → Create Client**

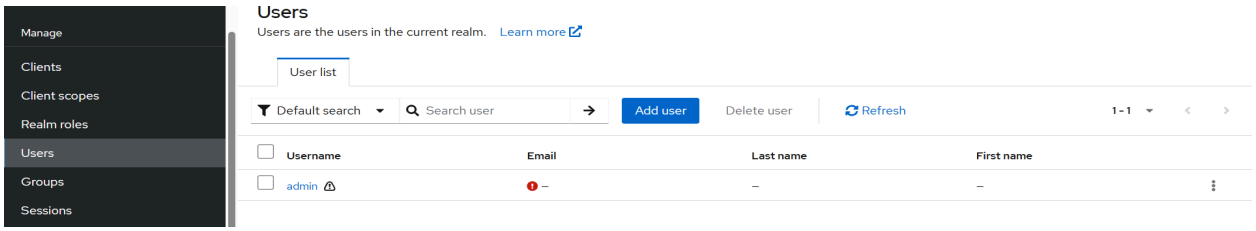


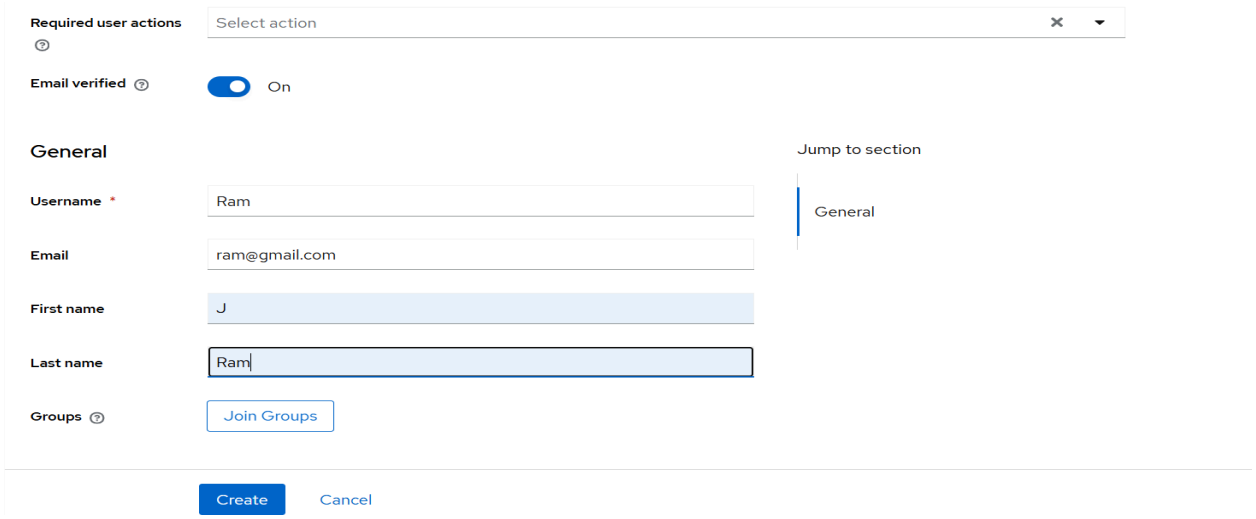


We should mention the popper redirect uri , as we don't have UI , mentioning * allow all.



Go to **Users → Add User**

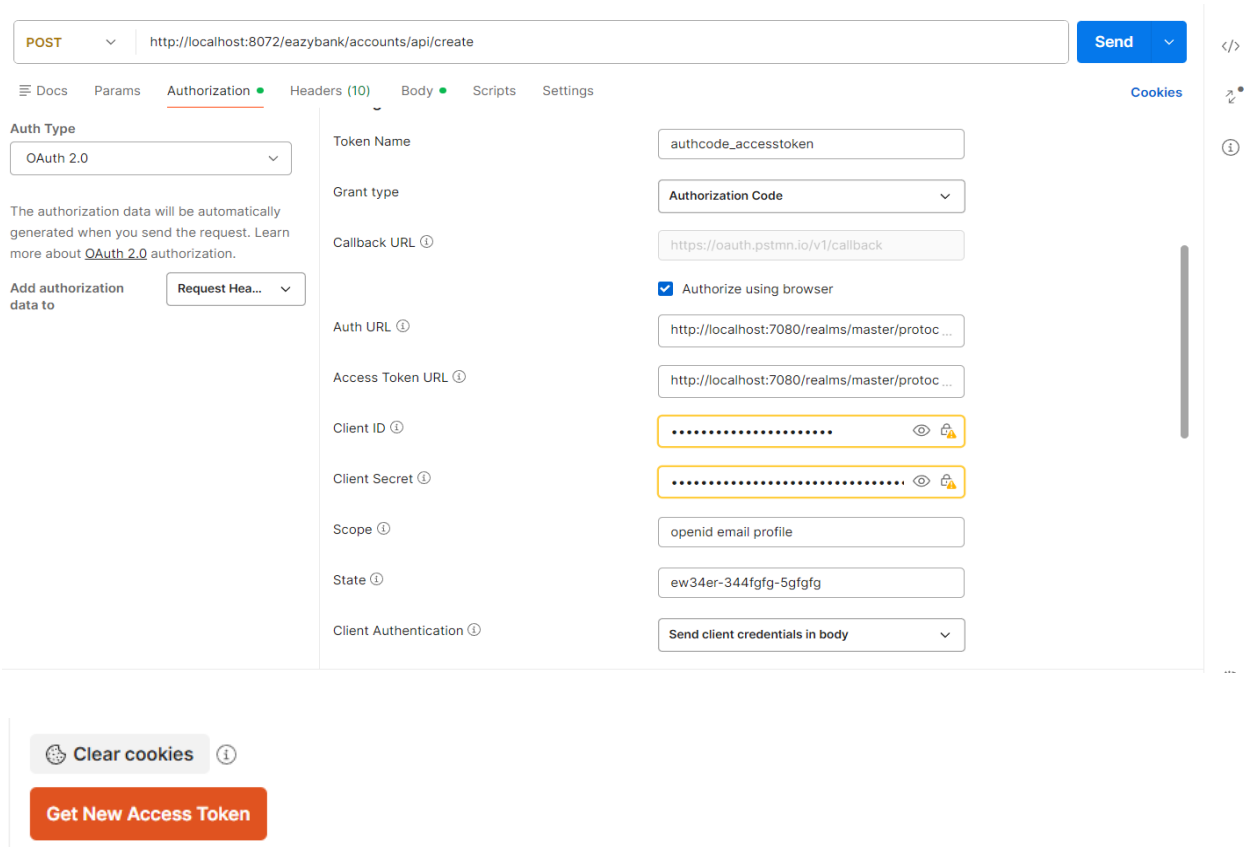




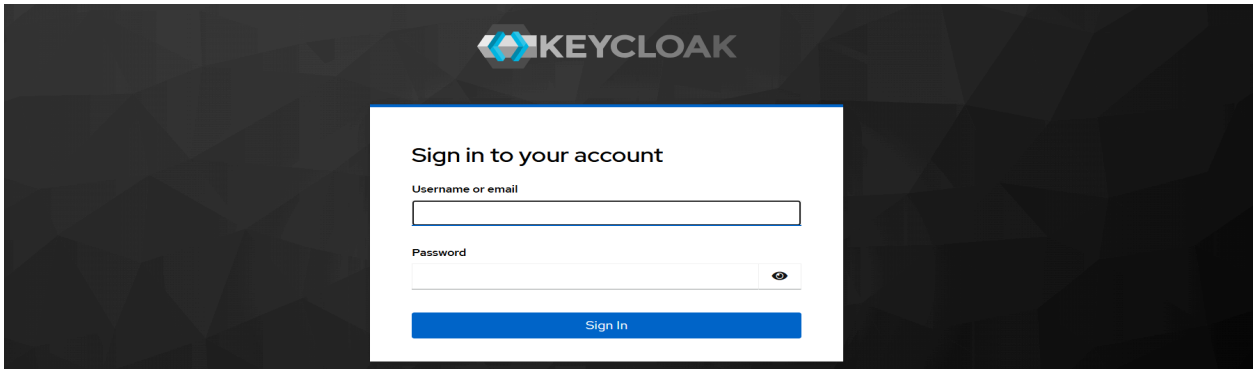
Go to the credentials of the user and set the password. Go to the Role Mapping and assign roles.

As we don't design any UI, we can test it in the postman by providing all details in the api.

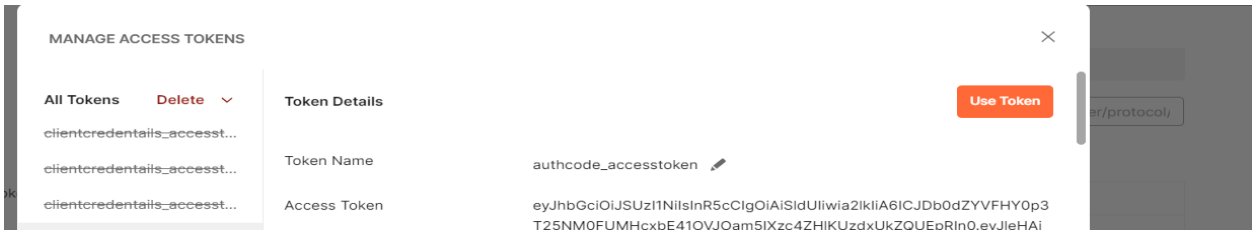
Select grant type and other details you will get from Go to: **Keycloak → Your Realm → Realm Settings → OpenID Endpoint Configuration**

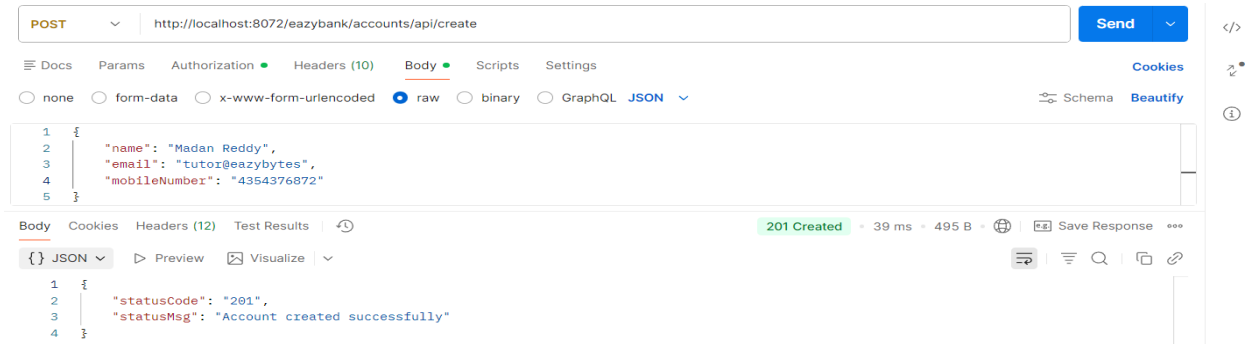


When we click on the "Get New Access Token" it will redirect a web page to login with a user which you create in keycloak.

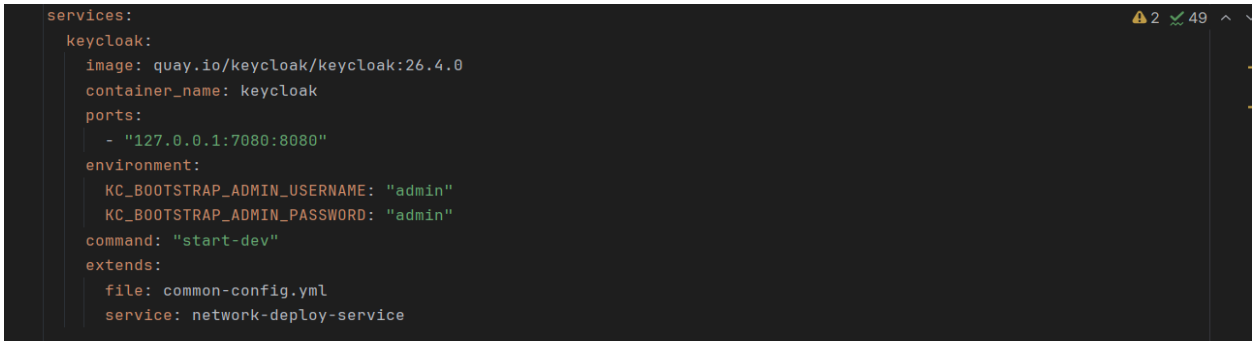


After successful sign in , it redirects to postman (as we don't hav ebay particular redirect uri , we just mention *)



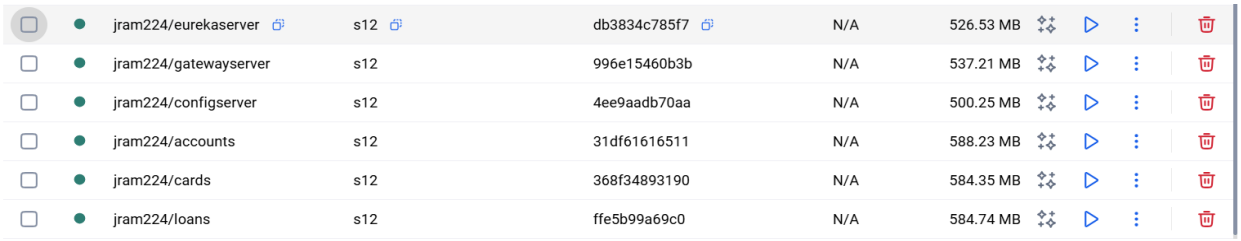


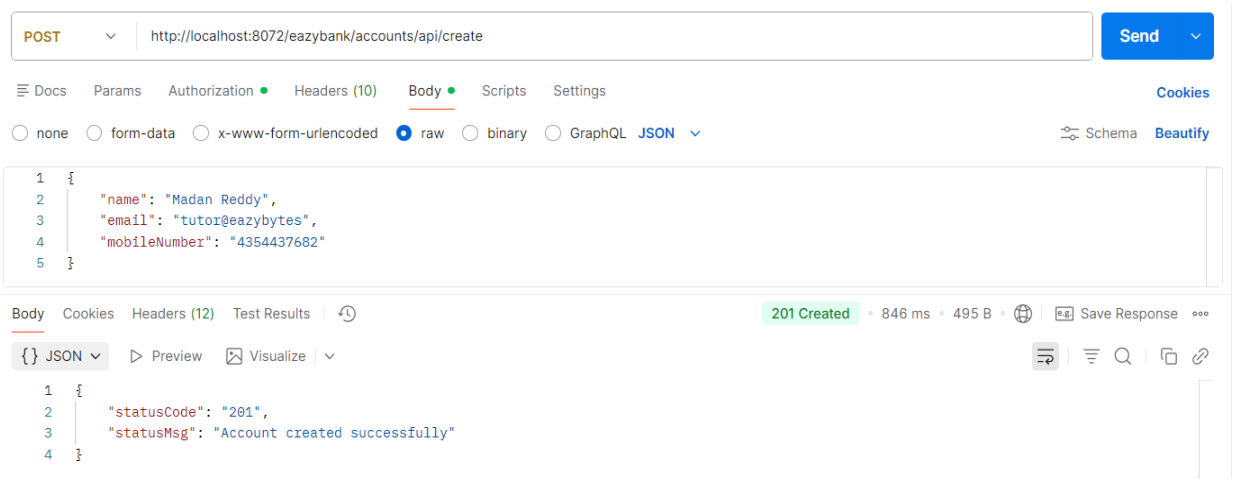
Generate docker images for services , add configuration related to keycloak and keycloak details in gateway server inside the docker - compose.yml inside docker -compose file.



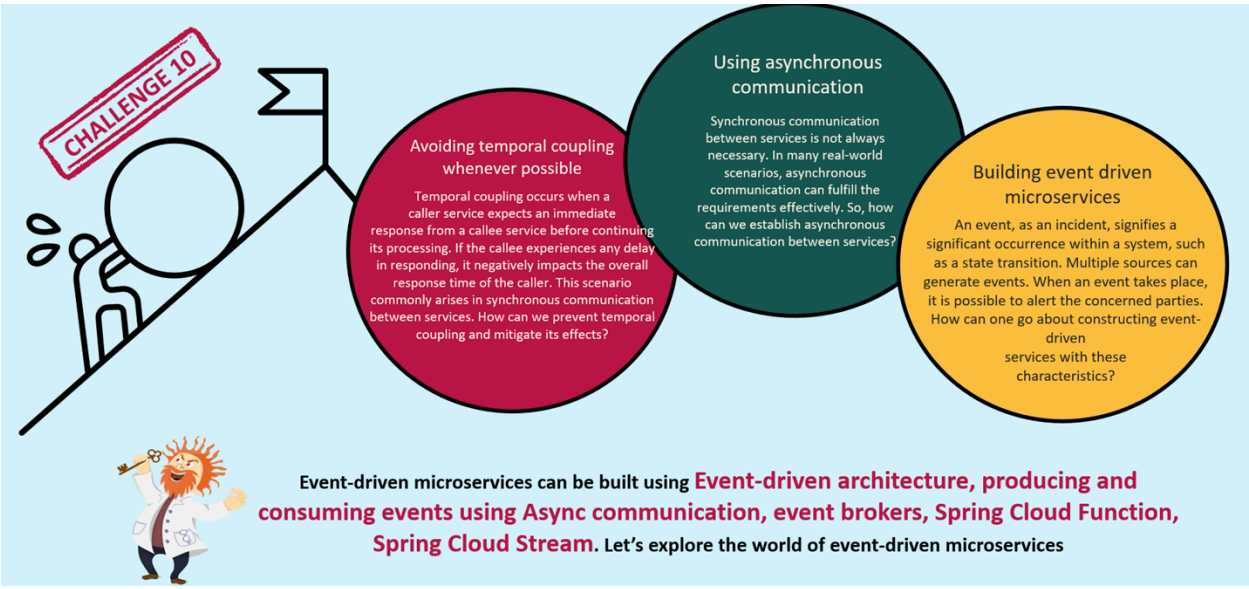
We can remove port mapping of cards,loans and accounts in docker-compose.yml. So it is secure that no one will access the service using their specific ports. As anyway we are calling through the gateway server port only right.

```yaml
loans:
  image: "jram224/loans:s12"
  container_name: loans-ms
  depends_on:
    configserver:
      condition: service_healthy
    eurekaserver:
      condition: service_healthy
```





Event Driven Microservices



[Communication in microservices]

Synchronous communication between services is not always necessary.

Temporal coupling = when Service A cannot continue unless Service B responds immediately.
This ONLY happens in **synchronous communication**.

- Account Service → Payment Service
- Payment Service is slow → Account Service becomes slow
- If Payment Service is down → Account Service fails

This makes the whole system tightly coupled and fragile.

Why avoid temporal coupling?

Because:

- A delay in one service affects all others
- A failure in one service cascades to others
- User experience becomes slow

Using Asynchronous Communication to Remove Temporal Coupling

In many real-world scenarios, asynchronous communication can fulfill the requirements effectively.

This means:

✔️ Instead of waiting for an immediate response,
Service A publishes an **event** and continues.

✔️ Service B processes the event **in the background**.

✔️ No blocking, no waiting, no dependency on timing.

This removes temporal coupling completely.

Event-Driven Microservice

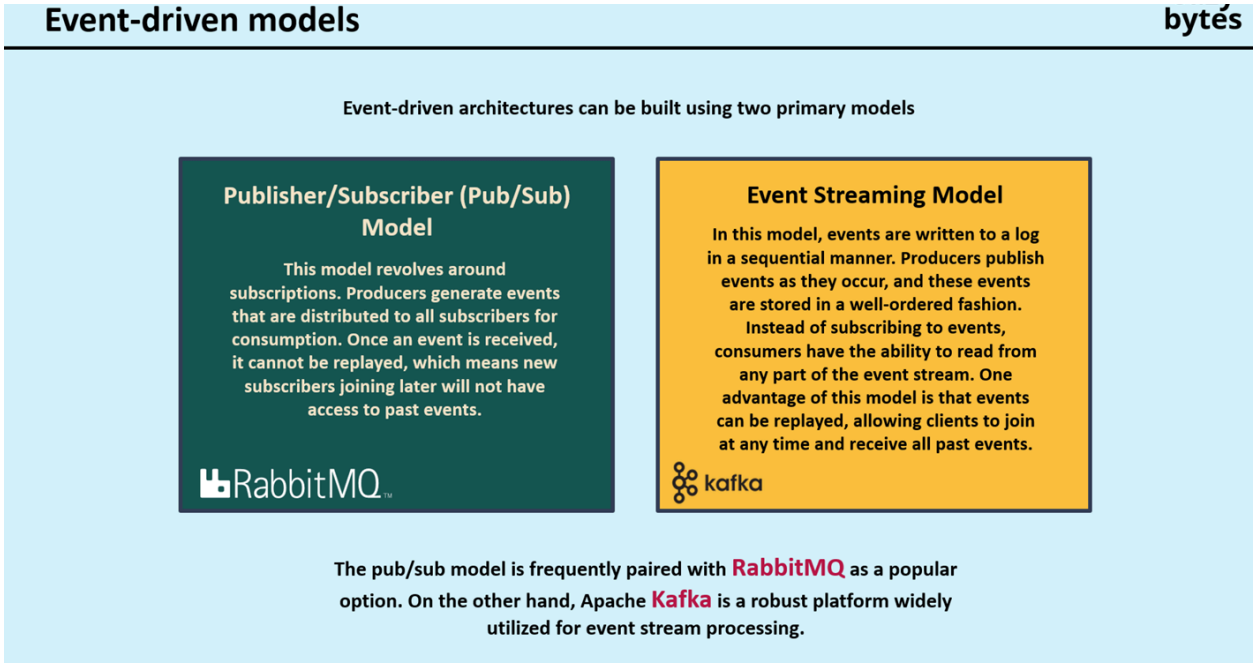
Event-driven microservices are microservices that **react to events** instead of waiting for request/response calls.

An **event** = "Something happened in the system."

Services communicate by **sending events**, not making API calls.

Example events:

- *UserCreated*
- *PaymentCompleted*
- *OrderPlaced*
- *InventoryUpdated*



A. Publisher/Subscriber (Pub/Sub) Model

✔️ Producer → publishes an event
✔️ Consumers → receive the event immediately
✔️ Events **cannot be replayed** (temporary)

Good for:

- Notifications
- Emails
- Live communication

Ex: **Snapchat Message :** You see it once → it disappears → you cannot open it again.

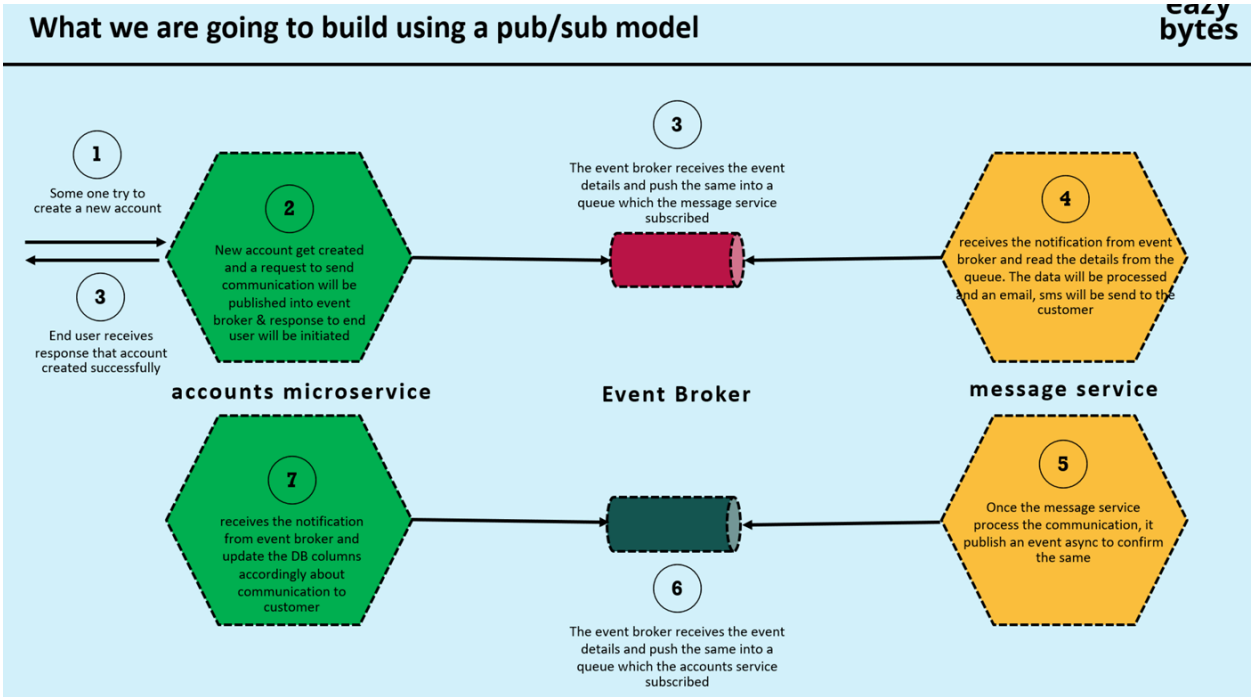
B. Event Streaming Model

✔️ Events stored in a log

✔️ Consumers can read from any point

✔️ Events can be replayed anytime

✔️ Perfect ordering of events

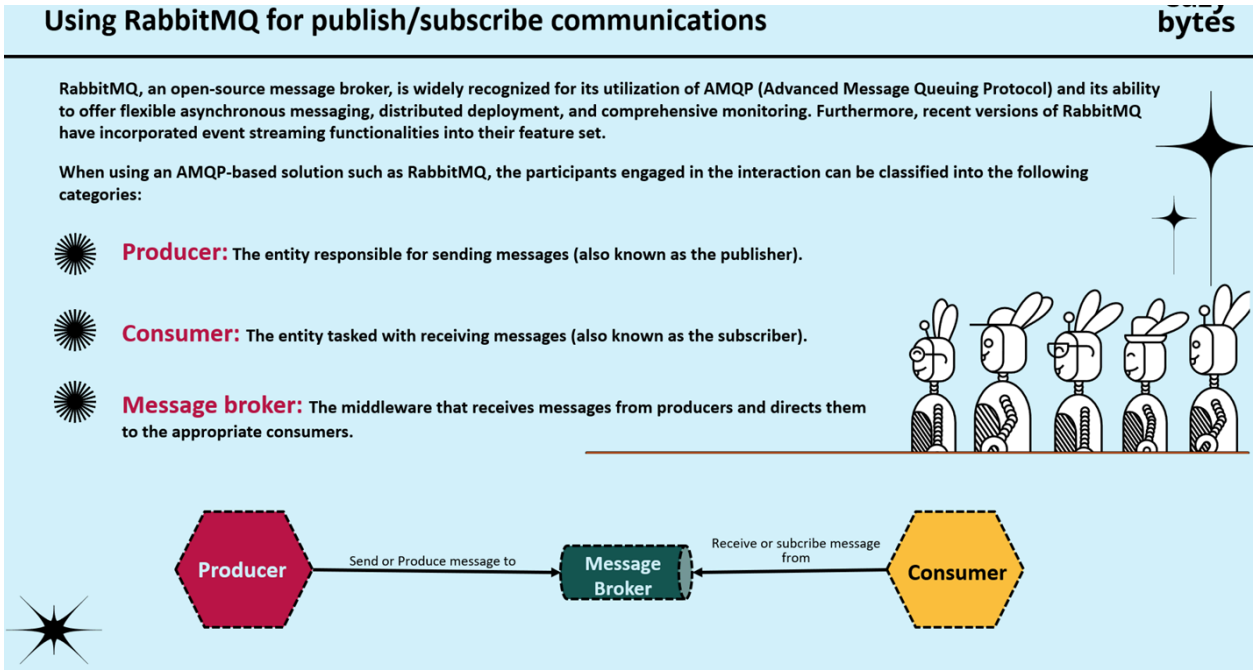


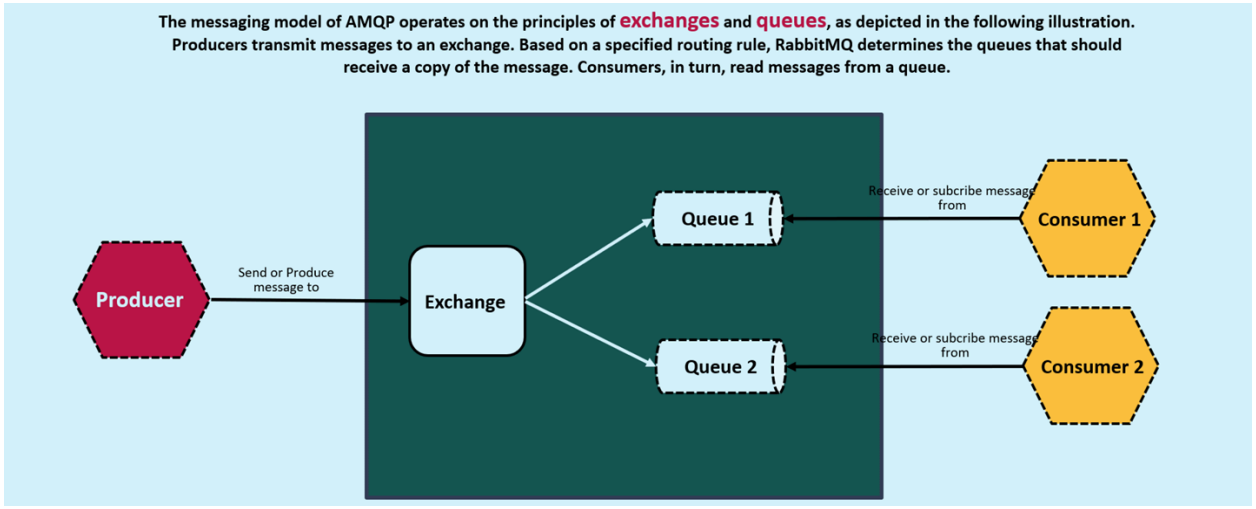
Ex: RabbitMQ

AMQP (Advanced Message Queuing Protocol)

AMQP is a messaging protocol that supports reliable, scalable, and flexible messaging.

- It is most commonly used with **RabbitMQ**, a widely adopted message broker.
- Unlike JMS, AMQP is not Java-specific and supports multiple programming languages.





Spring Cloud Function:

Spring Cloud Function is a framework from Spring that allows **developers to write business logic as standalone functions** and **run them in multiple environments (HTTP, messaging, serverless platforms) without rewriting code for each platform**.

You write a function **once**, and Spring maps it to whatever platform you need.

https://spring.io/projects/spring-cloud-function

🔥 What it gives you

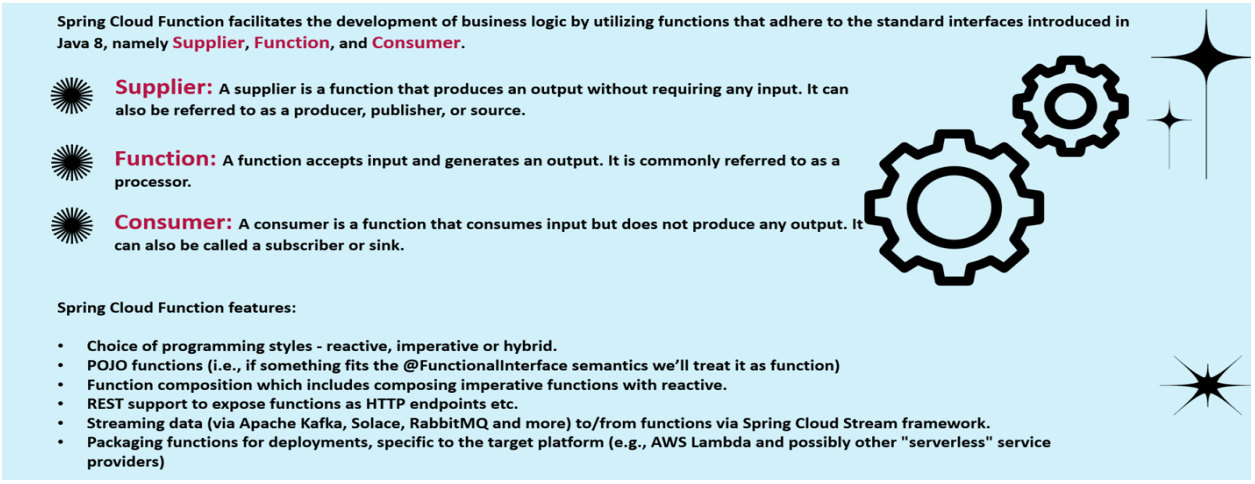
Capability	What it means for you
Write only functions → no controllers, no boilerplate	Focus purely on business logic
Deploy same function to HTTP, messaging systems, Lambda, Azure, GCP	**One function → multiple runtimes**
Works with imperative & reactive programming	Flexible based on application need
Type conversion, routing, composition	Combine functions, chain them, auto-convert I/O
Works standalone on your machine or in cloud	Local testing becomes easy
Uses Spring Boot features	DI, auto-config, metrics — everything still works

Why it is useful

- Reduces infrastructure code
- Ideal for microservices & serverless workloads
- Same function runs everywhere — no rewriting for cloud platforms
- Faster development + easier testing
- Lightweight and portable deployments

You write simple Java functions like:

- Function<Input, Output>
- Consumer<Input>
- Supplier<Output>



Before Spring Cloud Function (Traditional Spring Boot CRUD)

Architecture: Client → Controller → Service → Repository → DB

✔ Works perfectly for monoliths & REST
✖ Business logic tightly coupled with HTTP layer
✖ Hard to reuse same logic for Kafka, SQS, Lambda, CLI
✖ Need controllers/endpoints for each use case

Example: Customer CRUD (Traditional way)

```
@RestController
@RequestMapping("/customers")
public class CustomerController {
    @Autowired
    CustomerService service;
    @PostMapping
    public Customer create(@RequestBody Customer c) {
        return service.create(c);
    }
    @GetMapping("/{id}")
    public Customer get(@PathVariable Long id) {
        return service.get(id);
    }
}


@Service
public class CustomerService {
    @Autowired
    CustomerRepository repo;
    public Customer create(Customer c) { return repo.save(c); }
```

```
    public Customer get(Long id) { return repo.findById(id).orElse(null); }
}
```

🔁 REST dependant
🐌 Cannot directly run as Lambda/Event stream
🎯 Business logic is not reusable outside controller context

After Spring Cloud Function (CRUD as Functions)

We replace controller methods with Functions.

Architecture:

```
Client / Kafka / AWS Lambda / CLI
        ↓
    Function Bean(s)
        ↓
   Repository → DB
```

✔ No @RestController required
✔ Same logic callable by REST/Kafka/AWS Lambda without rewrite
✔ Highly reusable, serverless friendly
✔ Business logic is independent of transport

CRUD using Functions

```
@SpringBootApplication
public class CustomerFunctionApp {
    @Autowired CustomerRepository repo;
    // Create customer
    @Bean
    public Function<Customer, Customer> create() {
        return c -> repo.save(c);
    }
    // Fetch customer
```

```
    @Bean
    public Function<Long, Customer> get() {
        return id -> repo.findById(id).orElse(null);
    }
}
```

Expose as REST (No Controller Needed)

i) Multiple Independent Functions (using ;)

```
spring:
  cloud:
    function:
      definition: create;get
```

Semicolon (**;**) is used to list multiple functions separately.

ii) Function Chaining (using |)

```
definition: create|get
```

Output of create → input to get

Used only when functions must process data in sequence.

Now you can call like REST:

Function	Endpoint
create	POST /create
get	POST /get

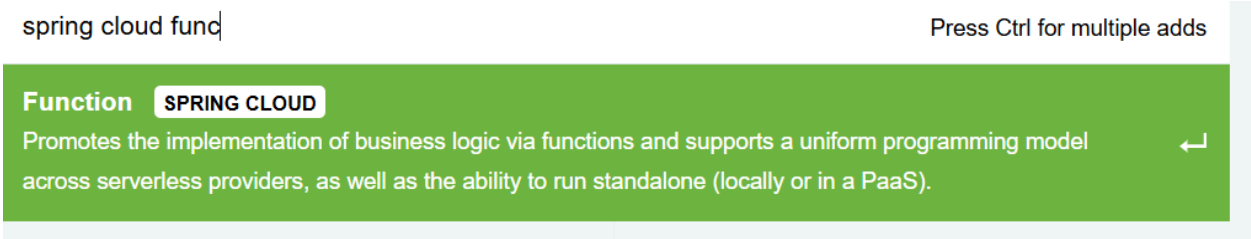
📌 In one deployment, the same code also works for

Without rewriting code	Works as
REST API	Yes
Kafka Listener	Yes
AWS Lambda	Yes
GCP / Azure Function	Yes

Simple Example:

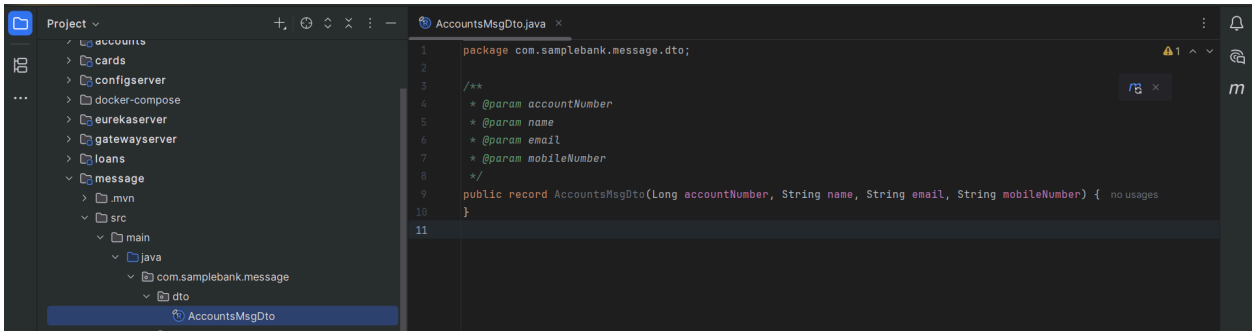
Steps to Create Functions Using Spring Cloud Function

i) create a microservice with spring cloud function dependency.

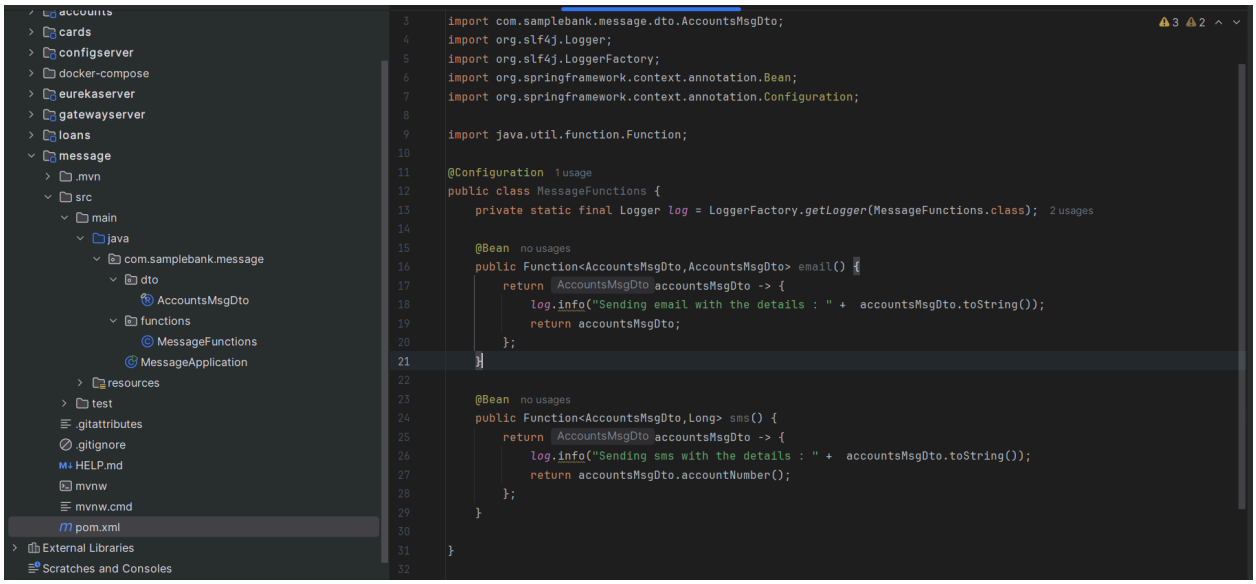


spring-boot-starter-web *(only if you want REST support)*

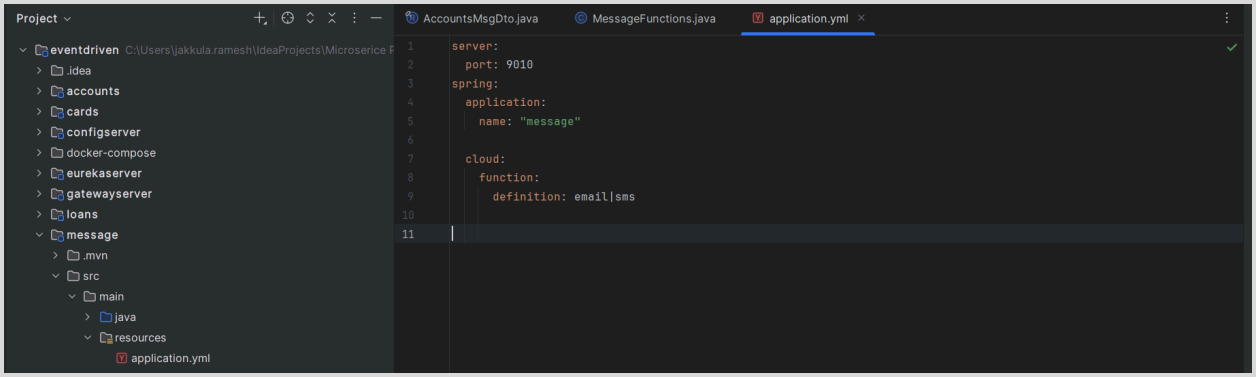
ii) Create dto package, which accepts the message going to receive from the message broker.



Create function package ,Where will create function for our business logic.



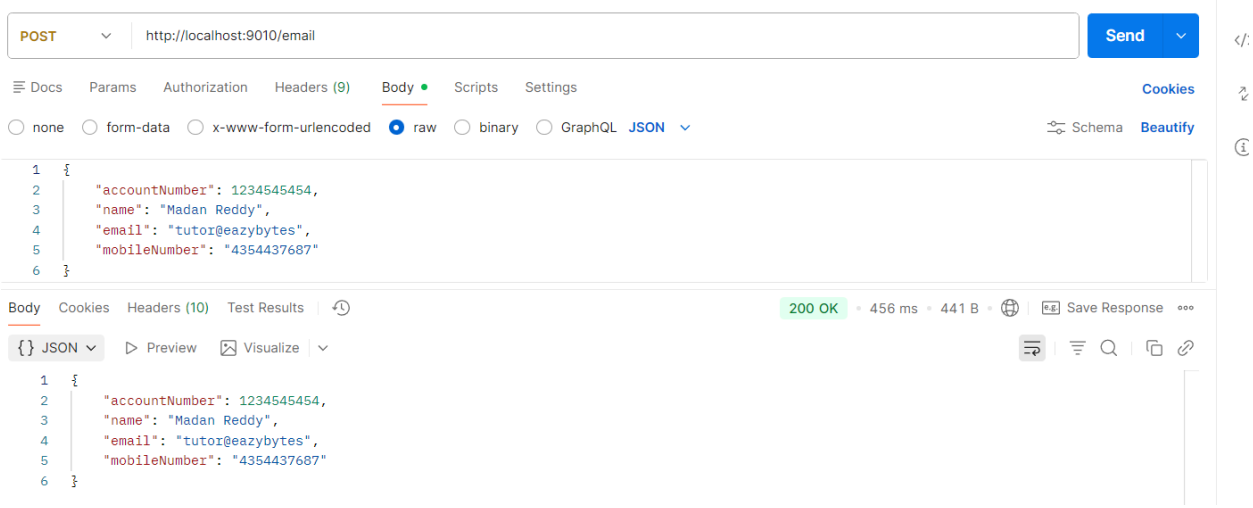
iii) Configure function exposure (application.yml)

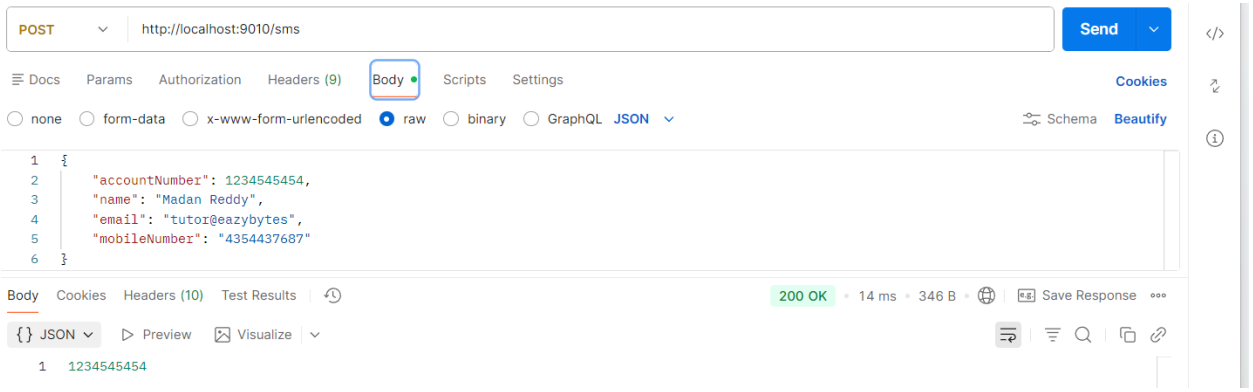


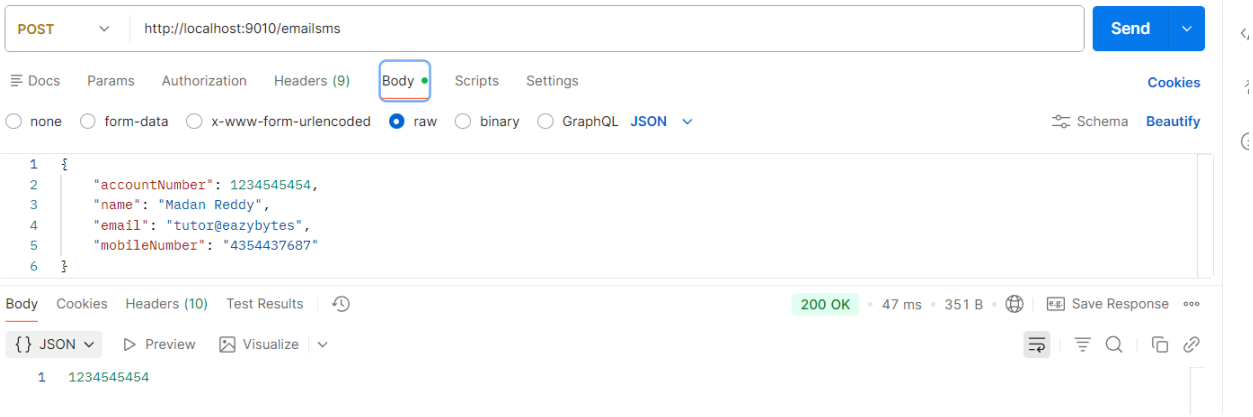
email and sms will be your **function names**, and when using Spring Cloud Function + Web (HTTP adapter) you can call/invoke them by URL using the function name as the endpoint.

How to call them via HTTP , just testing in postman

http://localhost:9010/email and http://localhost:9010/sms







Steps to create functions using Spring Cloud Functions

bytes

Below are the steps to create functions using Spring Cloud Functions,

1. **Initialize a spring cloud function project:** Start by creating a new Spring Boot project using your preferred IDE or by using Spring Initializr (https://start.spring.io/). Include the spring-cloud-function-context maven dependency

2. **Implement the business logic using functions**

   Develop two functions with the name email() and sms() like in the image. To make it simple, for now they just have logic of logging the details. But in real projects you can write logic to send emails and messages.

   To enable Spring Cloud Function to recognize our functions, we need to register them as beans. Proceed with annotating the MessageFunctions class as @Configuration and the methods email() & sms() as @Bean to accomplish this.

```java
@Configuration
public class MessageFunctions {

  private static final Logger log = LoggerFactory.getLogger(MessageFunctions.class);

  @Bean
  public Function<AccountsMsgDto, AccountsMsgDto> email() {
    return accountsMsgDto -> {
      log.info("Sending email with the details : " +  accountsMsgDto.toString());
      return accountsMsgDto;
    };
  }

  @Bean
  public Function<AccountsMsgDto, Long> sms() {
    return accountsMsgDto -> {
      log.info("Sending sms with the details : " + accountsMsgDto.toString());
      return accountsMsgDto.accountNumber();
    };
  }
```

3. **Composing functions:** If our scenario needs multiple functions to be executed, then we need to compose them otherwise we can use them as individual functions as well. Composing functions can be achieved by defining a property in application.yml like shown below,

```yaml
spring:
  cloud:
    function:
      definition: email|sms
```

The property spring.cloud.function.definition enables you to specify which functions should be managed and integrated by Spring Cloud Function, thereby establishing a specific data flow. In the previous step, we implemented the email() and sms() functions. We can now instruct Spring Cloud Function to utilize these functions as building blocks and generate a new function derived from their composition.

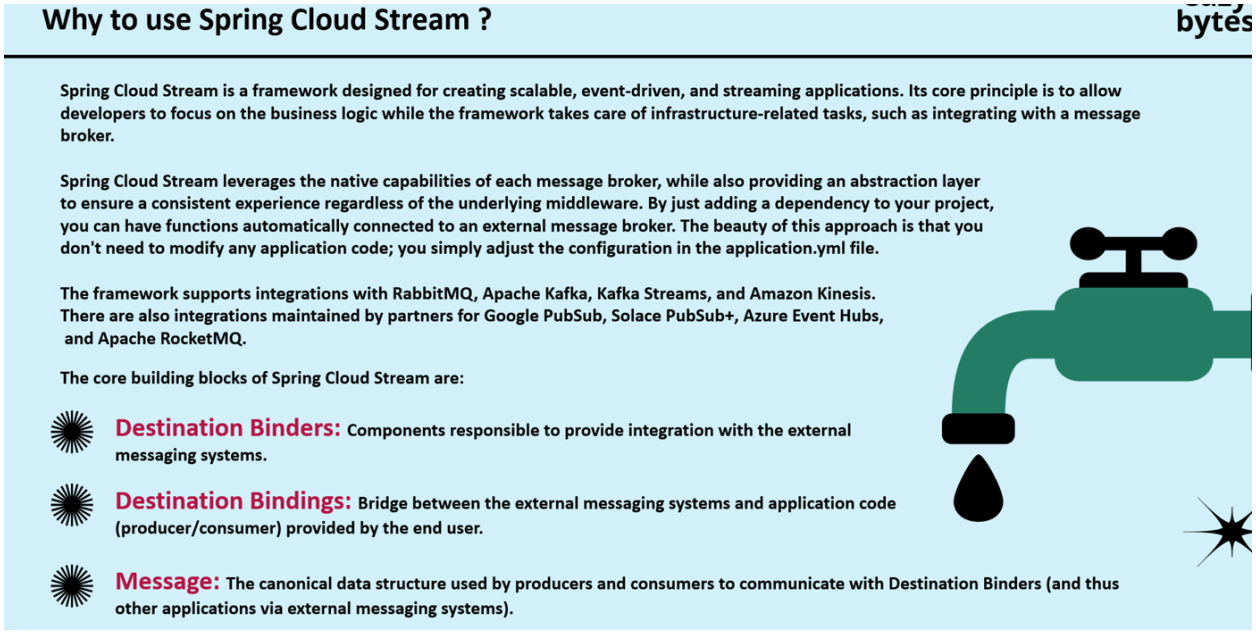
In serverless applications designed for deployment on FaaS platforms like AWS Lambda, Azure Functions, Google Cloud Functions, or Knative, it is common to have one function defined per application. The definition of cloud functions can align directly with functions declared in your application on a one-to-one basis. Alternatively, you can employ the pipe (|) operator to compose functions together in a data flow. In cases where you need to define multiple functions, the semicolon (;) character can be used as a separator instead of the pipe (|).

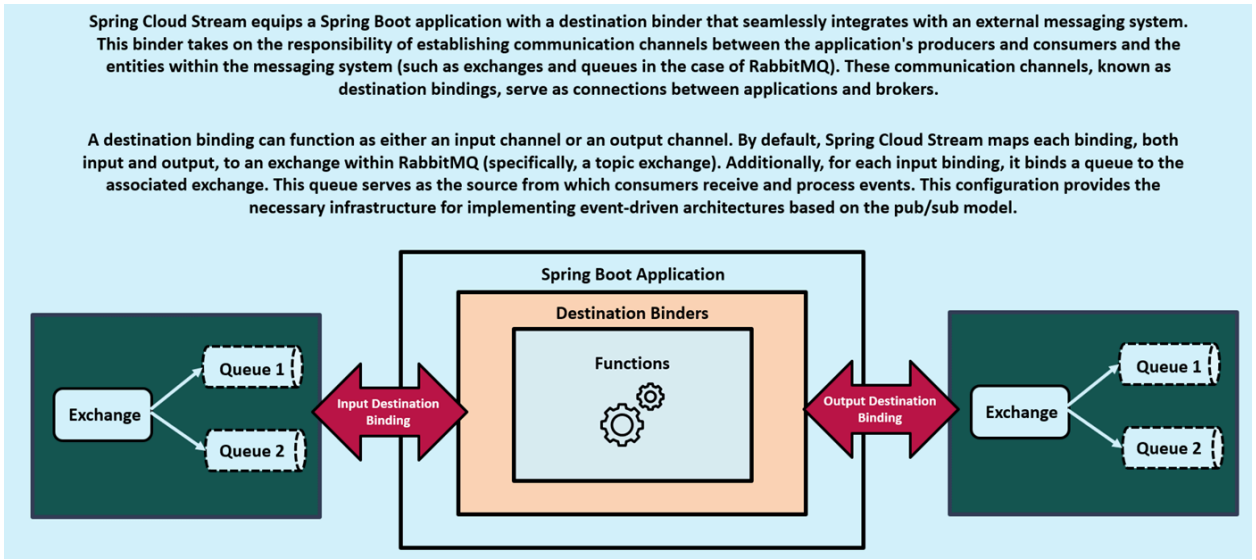
Based on the provided functions, the framework offers various ways to expose them according to our needs. For instance, Spring Cloud Function can automatically expose the functions specified in spring.cloud.function.definition as REST endpoints. This allows you to package the application, deploy it on a FaaS platform such as Knative, and instantly have a serverless Spring Boot application. But that is not what we want. Moving forward, the next step involves integrating it with **Spring Cloud Stream** and binding the function to message channels within an event broker like RabbitMQ.

Spring Cloud Stream:

Spring Cloud Stream is a framework built on top of Spring Boot that helps you build **event-driven microservices**. It helps to connect your application to message brokers in a simple & scalable way.

They communicate by **sending and receiving messages through messaging systems** like **Kafka, RabbitMQ, Azure Event Hubs**, etc. https://spring.io/projects/spring-cloud-stream





Your app uses **bindings** to send or receive messages through the binder.

Input Binding ← Receive message ← Binder ← Broker

Output Binding → Send message → Binder → Broker

You have a **house**. You want to **send** and **receive** water. To do that, you need **two different things**:

1. A connection to water supply

This is what brings water from the city to your house.

➡️ This is **Destination Binder** (Connector between your house and the water system)

2. Taps & Pipes inside your house

You open one tap to get water . You use another pipe to send water out

➡️ These are **Destination Bindings** (Input & output pipes used by you to receive/send)

Now relate to Spring Cloud Stream

Real Life	Spring Cloud Stream
Water supply connection	**Destination Binder**
Taps/Pipes you use	**Destination Bindings** (input/output)
Water	Messages/Events

🟦 Spring Cloud Function

Spring Cloud Function allows you to write business logic as reusable functions that can run on HTTP, messaging, or serverless platforms without code changes.

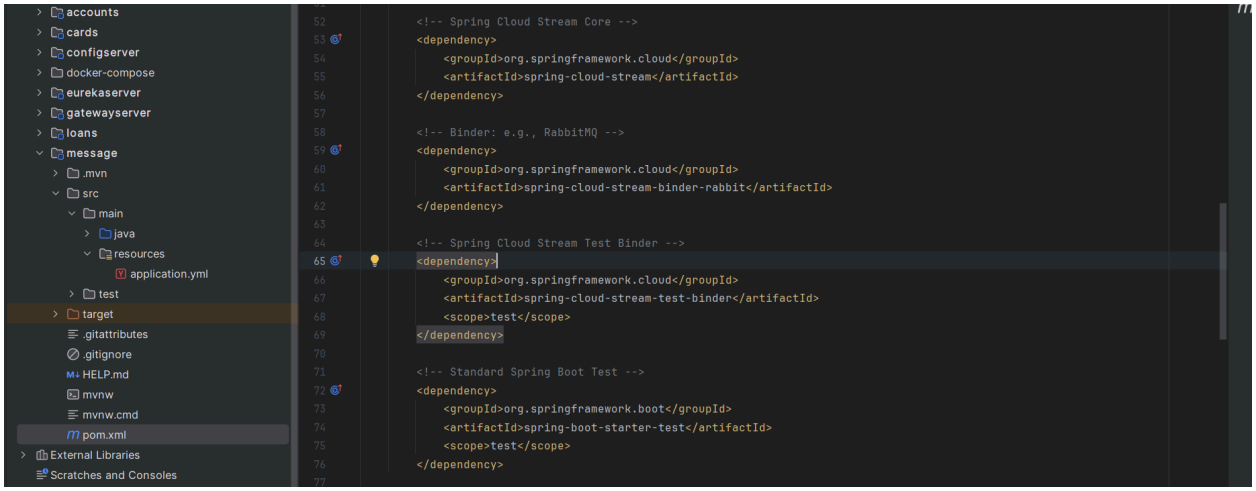
🟩 Spring Cloud Stream

Spring Cloud Stream enables event-driven microservices by connecting applications to message brokers like Kafka/RabbitMQ with simple input/output bindings.

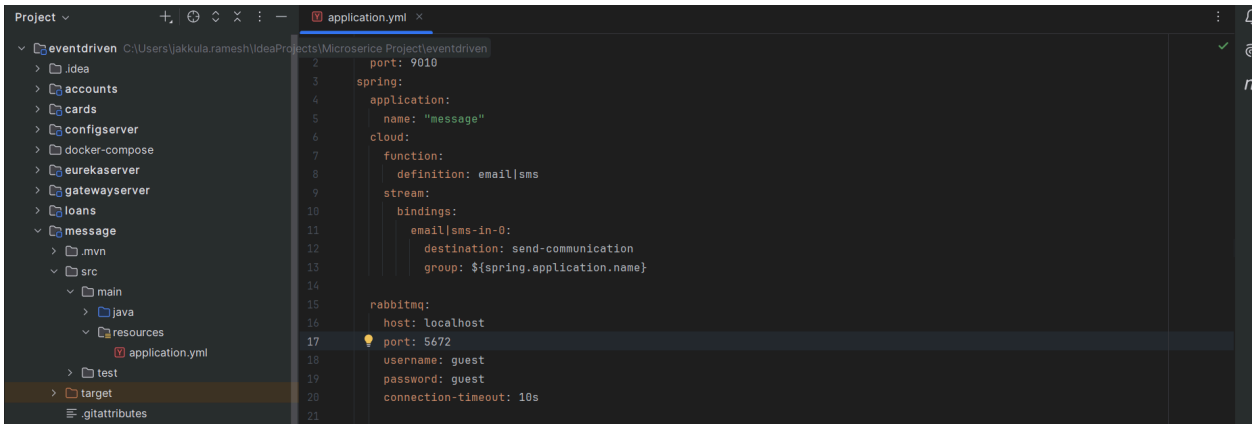
Using **Spring Cloud Function , Spring Cloud Stream with RabbitMQ(**Publisher/Subscriber Model **):**

i) creating bindings using spring cloud stream in message service;

Remove all dependencies related to Spring Cloud Function and add dependencies for Spring Cloud Stream in message service if any exists. When you include Spring Cloud Stream dependencies, it will automatically support Spring Cloud Function.



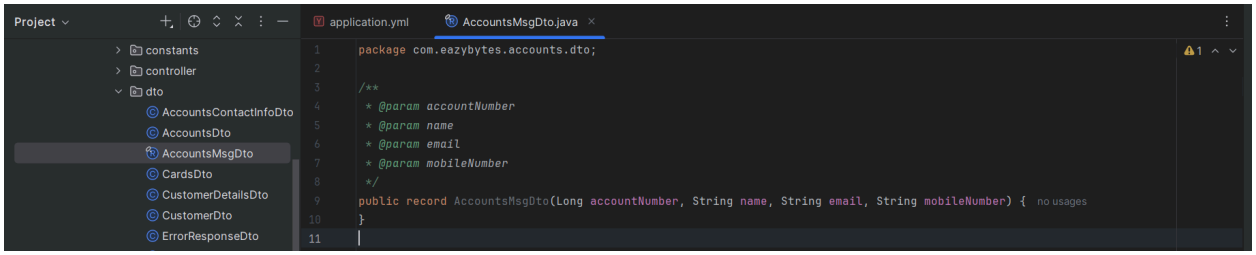
ii) add configuration related details about bindings and rabbitMq in application.yml

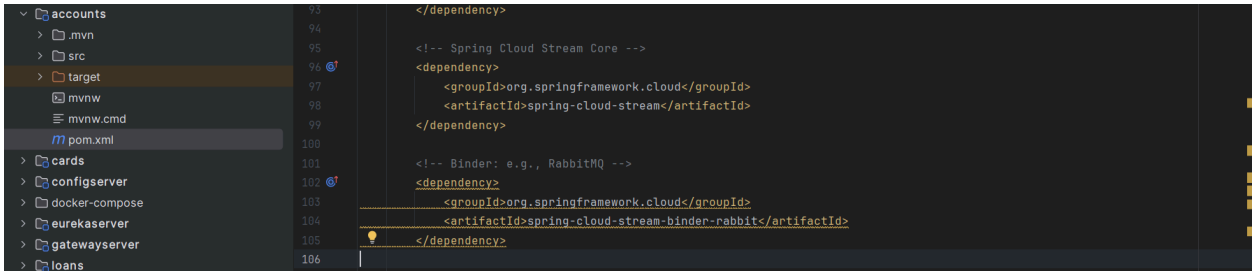


- spring.cloud.function.definition = email|sms → defines your functions to be used.
- email|sms-in-0 → the **input binding for the first function**.
    - -in-0 is automatically derived: if you don't define a binding explicitly, Spring Cloud Stream **assumes the input binding name is <function-name>-in-0**.
- destination: send-communication → the RabbitMQ queue/topic to listen to.
- group: ${spring.application.name} → consumer group for message partitioning/load-balancing.

So with this config, your service **listens to messages from the queue send-communication** inside the rabbitmq **.**

iii) step by step and set up your **Account microservice** to send events to RabbitMQ using **Spring Cloud Stream** with **StreamBridge** whenever a new account is created.





Add application.yml properties

Add the following properties in application.yml for RabbitMQ and bindings. Since we don't have functions in this microservice, we can give any name for the binding:

sendCommunication-out-0 → output binding name. You can give it any name since this microservice only sends messages.

destination: send-communication → the **exchange** in RabbitMQ.

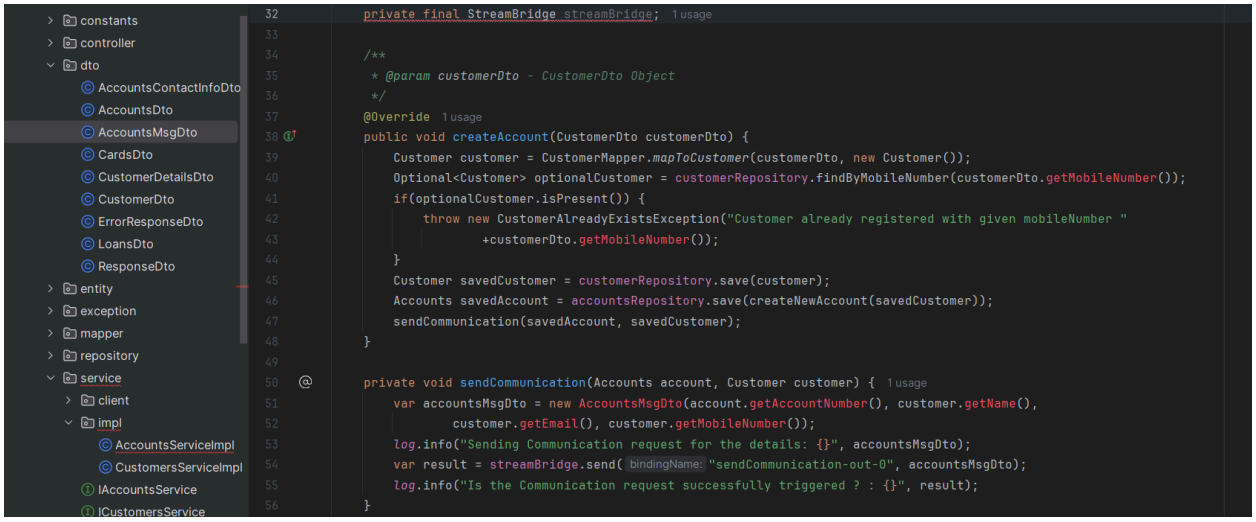
RabbitMQ will bind this destination to the exchange available in RabbitMQ, and the exchange name is send-communication.

Note: When using **out bindings**, the destination binds to an **exchange**.

When using **in bindings**, the destination binds to a **queue**.

Now, create a method inside AccountServiceImpl using **StreamBridge**. The logic should be:

Whenever a new account is created, an event is triggered and sent to RabbitMQ.



iv) run the rabbitmq using docker command and test

We can run it using docker or .exe https://www.rabbitmq.com/docs/download

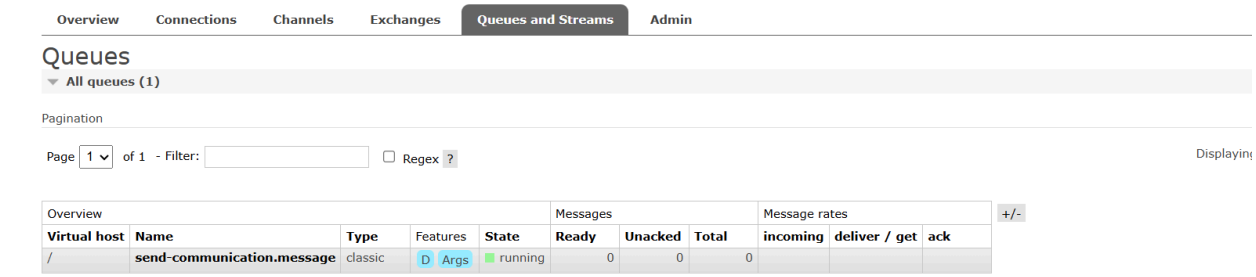
```
C:\Users\jakkula.ramesh>docker run -d -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:4-management
Unable to find image 'rabbitmq:4-management' locally
4-management: Pulling from library/rabbitmq
ea5be5a66a88: Pulling fs layer
945e0aae58bc: Download complete
```

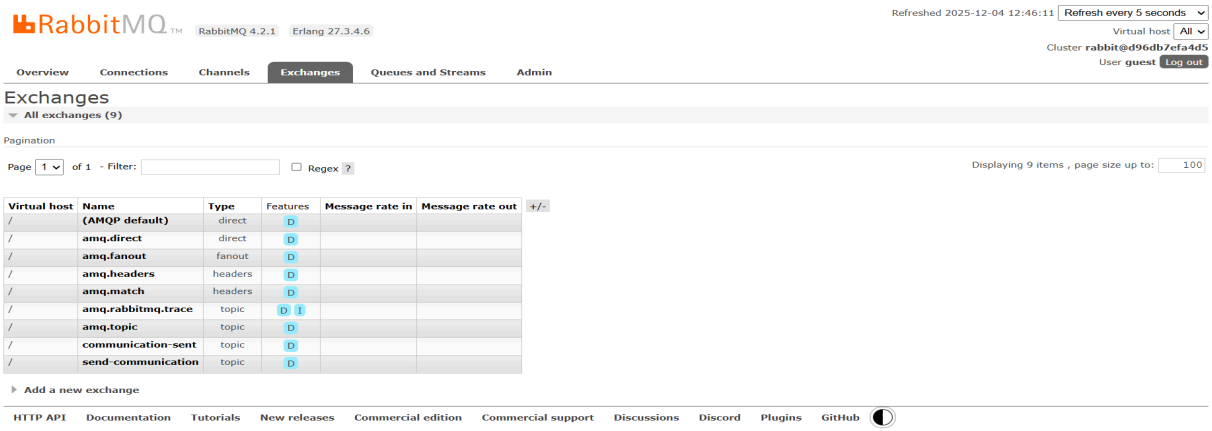
Run the microservice in sequence order to see the rabbit dashboard .

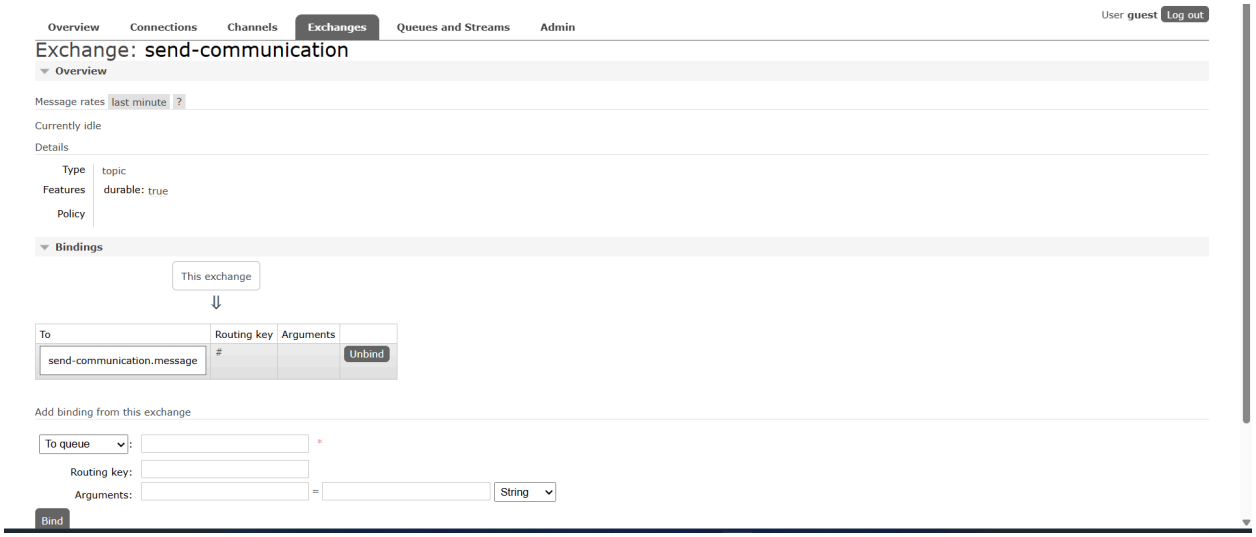
So far we have developed an account microservice -> rabbit mq -> message.
we can do message -> rabbit mq -> account microservice . For that we need to in and out bindings and

In **Message Service**, when we want to publish using *out binding*,

there is **no need to write publishing logic** because Spring Cloud Function automatically routes input → function → output.. As we are using functions to develop logic here. So the spring cloud function will take care of everything.


Overview
Connections
Channels
Exchanges
Queues and Streams
Admin
Queues
All queues (1)
Pagination
Page 1 of 1 - Filter:
Regex ?
Displaying
Overview
Messages
Message rates
+/-
Virtual host
Name
Type
Features
State
Ready
Unacked
Total
incoming
deliver / get
ack
/
send-communication.message
classic
D Args
running
0
0
0


RabbitMQ
RabbitMQ 4.2.1
Erlang 27.3.4.6
Refreshed 2025-12-04 12:46:11
Refresh every 5 seconds
Virtual host All
Cluster rabbit@d96db7efa4d5
User guest
Log out
Overview
Connections
Channels
Exchanges
Queues and Streams
Admin
Exchanges
All exchanges (9)
Pagination
Page 1 of 1 - Filter:
Regex ?
Displaying 9 items , page size up to: 100
Virtual host
Name
Type
Features
Message rate in
Message rate out
+/-
/ (AMQP default) direct D
/ amq.direct direct D
/ amq.fanout fanout D
/ amq.headers headers D
/ amq.match headers D
/ amq.rabbitmq.trace topic D I
/ amq.topic topic D
/ communication-sent topic D
/ send-communication topic D
Add a new exchange
HTTP API
Documentation
Tutorials
New releases
Commercial edition
Commercial support
Discussions
Discord
Plugins
GitHub


User guest
Log out
Overview
Connections
Channels
Exchanges
Queues and Streams
Admin
Exchange: send-communication
Overview
Message rates last minute ?
Currently idle
Details
Type topic
Features durable: true
Policy
Bindings
This exchange
To
Routing key
Arguments
send-communication.message
#
Unbind
Add binding from this exchange
To queue :
Routing key:
Arguments:
String
Bind

Steps to create bindings using Spring Cloud Stream

bytes

Below are the steps to create bindings using Spring Cloud Stream,

1. **Add the Stream related dependencies:** Add the maven dependencies spring-cloud-stream, spring-cloud-stream-binder-rabbit inside pom.xml of message service where we defined functions

2. **Add the stream binding and rabbitmq properties inside application.yml of message service**

We need to define input binding for each function accepting input data, and an output binding for each function returning output data. Each binding can have a logical name following the below convention. Unless you use partitions (for example, with Kafka), the <index> part of the name will always be 0. The <functionName> is computed from the value of the spring.cloud.function.definition property.

Input binding: <functionName> + -in- + <index>
Output binding: <functionName> + -out- + <index>

The binding names exist only in Spring Cloud Stream and RabbitMQ doesn't know about them. So to map between the Spring Cloud Stream binding and RabbitMQ, we need to define destination which will be the exchange inside the RabbitMQ. group is typically application name, so that all the instances of the application can point to same exchange and queue.

The queues will be created inside RabbitMQ based on the queue-naming strategy (<destination>.<group>) includes a parameter called consumer group.

```yaml
spring:
  application:
    name: message
  cloud:
    function:
      definition: email|sms
    stream:
      bindings:
        emailsms-in-0:
          destination: send-communication
          group: ${spring.application.name}
        emailsms-out-0:
          destination: communication-sent
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
    connection-timeout: 10s
```

Event producing and consuming in accounts microservice

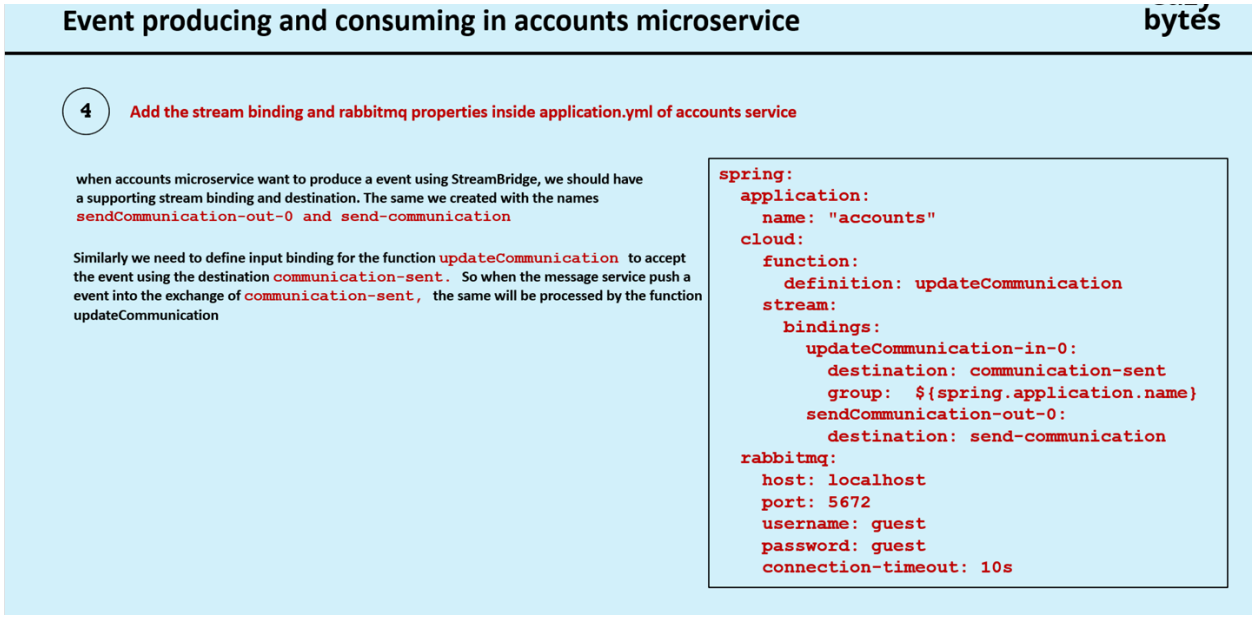
bytes

Below are the steps for event producing and consuming in accounts microservice

1. **Autowire StreamBridge class:** StreamBridge is a class inside Spring Cloud Stream which allows user to send data to an output binding. So to produce the event, autowire the StreamBridge class into the class from where you want to produce a event

2. **Use send() of StreamBridge to produce a event like shown below,**

```java
@Override
public void createAccount(CustomerDto customerDto) {
    Customer customer = CustomerMapper.mapToCustomer(customerDto, new Customer());
    Optional<Customer> optionalCustomer = customerRepository.findByMobileNumber(
            customerDto.getMobileNumber());
    if (optionalCustomer.isPresent()) {
        throw new CustomerAlreadyExistsException("Customer already registered with given mobileNumber "
                + customerDto.getMobileNumber());
    }
    Customer savedCustomer = customerRepository.save(customer);
    Accounts savedAccount = accountsRepository.save(createNewAccount(savedCustomer));
    sendCommunication(savedAccount, savedCustomer);
}

private void sendCommunication(Accounts account, Customer customer) {
    var accountsMsgDto = new AccountsMsgDto(account.getAccountNumber(), customer.getName(),
            customer.getEmail(), customer.getMobileNumber());
    log.info("Sending Communication request for the details: {}", accountsMsgDto);
    var result = streamBridge.send("sendCommunication-out-0", accountsMsgDto);
    log.info("Is the Communication request successfully processed ? : {}", result);
}
```



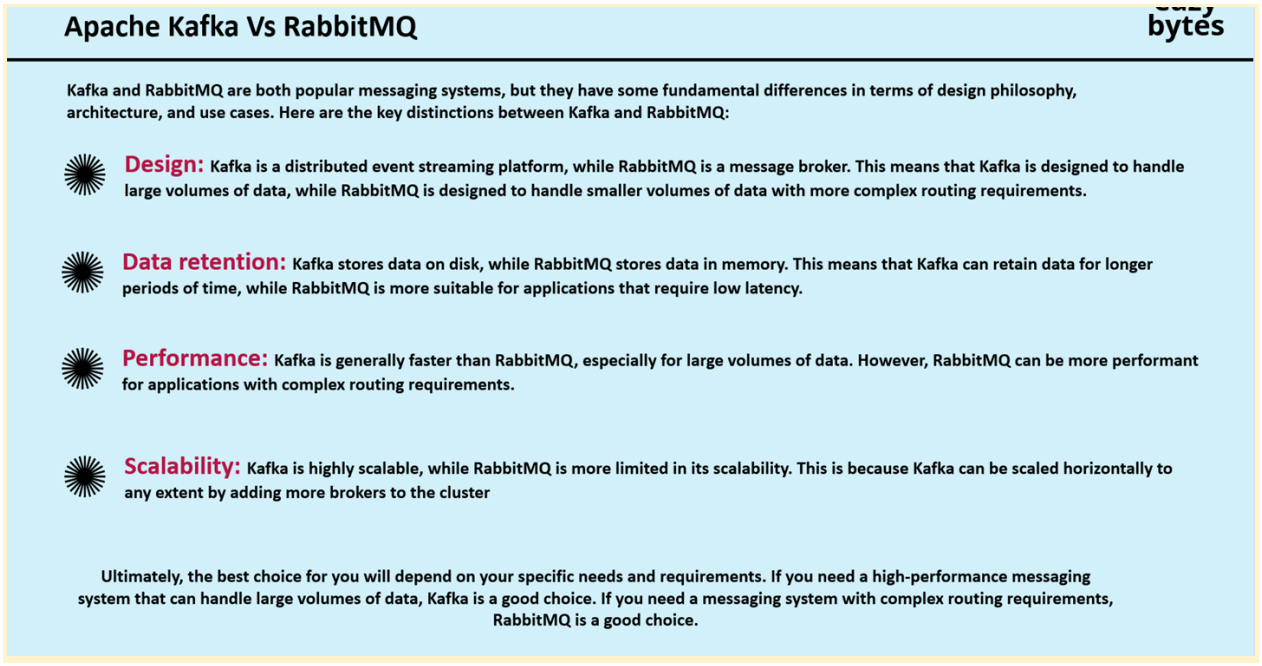
Event Driven Microservices using Kafka , spring cloud functions & stream

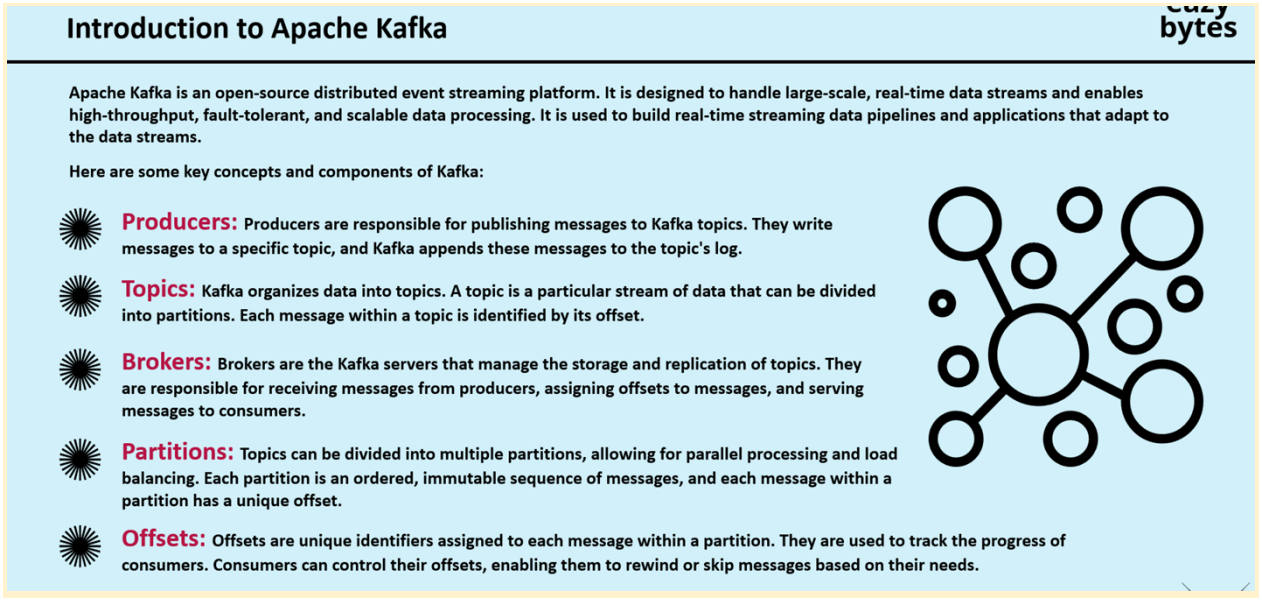
Apache **Kafka** is a **distributed event streaming platform** used for building **real-time data pipelines and streaming applications**. It allows systems to **publish (produce), subscribe (consume), store, and process** streams of events (messages) in a fault-tolerant and scalable way.

https://kafka.apache.org/ , Real-world example :

- **Netflix** uses Kafka to monitor user activity and optimize recommendations and video playback experiences for over 230 million subscribers.

- **Food delivery platforms** like Swiggy and Zomato use Kafka to handle order events and **track delivery partner locations instantly** for live map updates.
- **LinkedIn**, where Kafka originated, uses it for activity streams, operational metrics, and the newsfeed.
- **Uber** uses Kafka to exchange data between user and driver applications, compute demand, and Pricing is updated **on demand**.



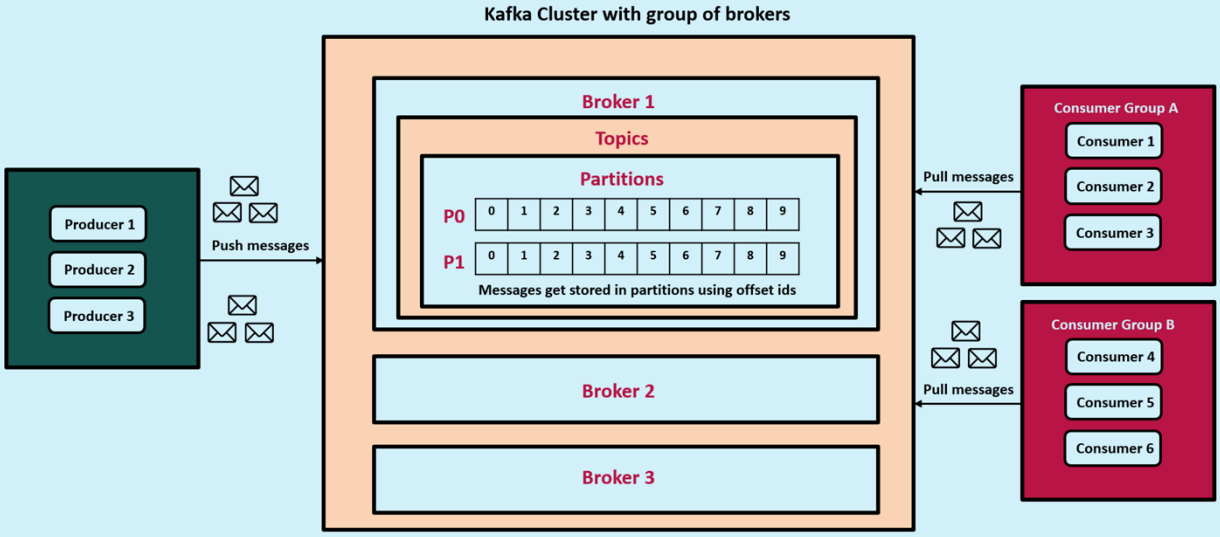


Replication: Kafka allows topics to be replicated across multiple brokers to ensure fault tolerance. Replication provides data redundancy, allowing for failover and high availability.

Consumers: Consumers read messages from Kafka topics. They subscribe to one or more topics and consume messages by reading from specific partitions within those topics. Each consumer maintains its offset to track its progress in the topic.

Consumer Groups: Consumers can be organized into consumer groups. Each message published to a topic is delivered to only one consumer within each group. This enables parallel processing of messages across multiple consumers.

Streams: Kafka Streams is a client library that enables stream processing within Kafka. It allows you to build applications that consume, transform, and produce data in real-time.



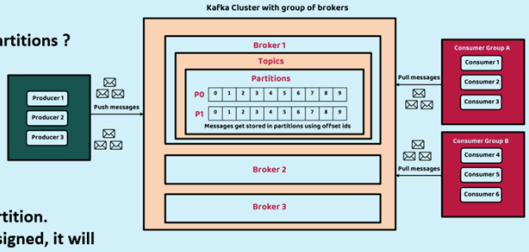
Introduction to Apache Kafka

bytés

A Kafka cluster can have any number of producers, consumers, brokers. For a production set up, atleast 3 brokers is recommended. This helps in maintaining replications, fault tolerant system etc.

A Kafka broker can have any number of topics. Topic is a category under which producers can write and interested, authorized consumers can read data. For example, we can have topics like sendCommunication, dispatchOrder, purgeData etc.

Inside each topic, we can have any number of partitions. Why do we need partitions ? Since Kafka producers can handle enormous amount of data, it is not possible to store in a single server (broker). Therefore, a topic will be partitioned into multiple parts and distributed across multiple brokers, since Kafka is a distributed system. For example, we can store all customers data from a state, zipcode, region etc. inside a partition and the same can be replicated as per the configurations.



Offsets is a sequence id assigned to a message as they get stored inside a partition. The offset number starts from 0 and followed by 1,2,3.... Once offset id is assigned, it will never change. These are similar to sequence ids inside the DB tables.

By keeping track of offsets, Kafka provides reliability, fault tolerance, and flexibility to consumers. Consumers have fine-grained control over their progress, enabling them to manage message ordering, replay messages, ensure message delivery, and facilitate parallel processing.

Producer side story

eazy bytes

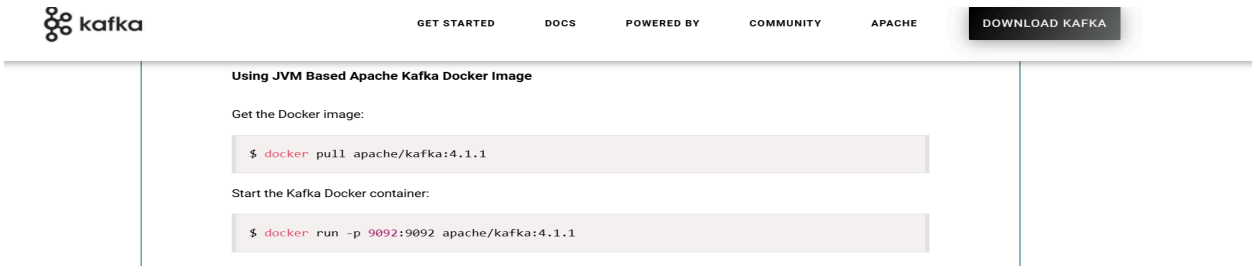
1. **Producer Configuration:** Before pushing a message into Kafka, a producer needs to be configured. This involves setting up properties such as the Kafka broker addresses, serialization format for messages, and other optional configurations like compression or batching.
2. **Topic Selection:** The producer needs to specify the topic to which it wants to push the message. Topics are predefined streams of data within Kafka. If the topic doesn't exist, it can be created dynamically, depending on the broker's configuration.
3. **Message Production:** The producer sends the message to Kafka by using the Kafka client library's API. The producer specifies the target topic and the serialized message. It may also provide a partition key (optional) to control which partition the message should be written to.
4. **Partition Assignment:** If a partition key is provided, Kafka uses it to determine the target partition for the message. If no partition key is provided, Kafka uses a round-robin or hashing algorithm to distribute messages evenly across partitions.
5. **Message Routing & offset assignment:** The producer sends the message to the appropriate Kafka broker based on the target topic and the partition assigned to the message. The broker receives the message and appends it to the log of the corresponding partition in a durable and ordered manner with the help of offset id.
6. **Message Replication:** Kafka ensures high availability and fault tolerance by replicating messages across multiple brokers. Once the message is written to the leader partition, Kafka asynchronously replicates it to other replicas of the partition.
7. **Acknowledgment and Error Handling:** The producer receives an acknowledgment from Kafka once the message is successfully written to the leader partition. The producer can handle any potential errors, retries, or failures based on the acknowledgment received. Depending on the acknowledgment mode configured, the producer may wait for acknowledgment from all replicas or just the leader replica.

Consumer side story

eazy bytes

1. **Consumer Group and Topic Subscription:** Consumers in Kafka are typically organized into consumer groups. Before reading messages, a consumer needs to join a consumer group and subscribe to one or more topics. This subscription specifies which topics the consumer wants to consume messages from.
2. **Partition Assignment:** Kafka assigns the partitions of the subscribed topics to the consumers within the consumer group. Each partition is consumed by only one consumer in the group. Kafka ensures a balanced distribution of partitions among consumers to achieve parallel processing.
3. **Offset Management:** Each consumer maintains its offset for each partition it consumes. Initially, the offset is set to the last committed offset or a specified starting offset. As the consumer reads messages, it updates its offset to keep track of the progress.
4. **Fetch Request:** The consumer sends fetch requests to the Kafka broker(s) it is connected to. The fetch request includes the topic, partition, and the offset from which the consumer wants to read messages. The request also specifies the maximum number of messages to be fetched in each request.
5. **Message Fetching:** Upon receiving the fetch request, the Kafka broker retrieves the requested messages from the corresponding partition's log. It sends the messages back to the consumer in a fetch response. The response contains the messages, their associated offsets, and metadata.
6. **Message Processing:** Once the consumer receives the messages, it processes them according to its application logic. This processing can involve transformations, aggregations, calculations, or any other operations based on the business requirements.
7. **Committing the Offset:** After successfully processing a batch of messages, the consumer needs to commit the offset to Kafka. This action signifies that the consumer has completed processing the messages up to that offset. Committing the offset ensures that the consumer's progress is persisted and can be resumed from that point in case of failure or restart.
8. **Polling Loop:** The consumer repeats the process of sending fetch requests, receiving messages, processing them, and committing the offset in a continuous loop. This loop allows the consumer to continuously consume and process new messages as they become available.

Installing kafka with the help of docker and copy the docker command under **Using JVM Based Apache Kafka Docker Image** option in kafka site https://kafka.apache.org/quickstart



```
C:\Users\jakkula.ramesh>docker run -p 9092:9092 apache/kafka:4.1.1
Unable to find image 'apache/kafka:4.1.1' locally
4.1.1: Pulling from apache/kafka
e4b07b2631b1: Pull complete
34074eb54496: Pull complete
4be6c1affd0e: Pull complete
```

As we know we developed a message service using spring cloud function and streams and used with rabbitmq previously.

To integrate with kafka , we just need to replace rabbitmq configuration with kafka, we already knew spring cloud function will take care of all other infra setup.

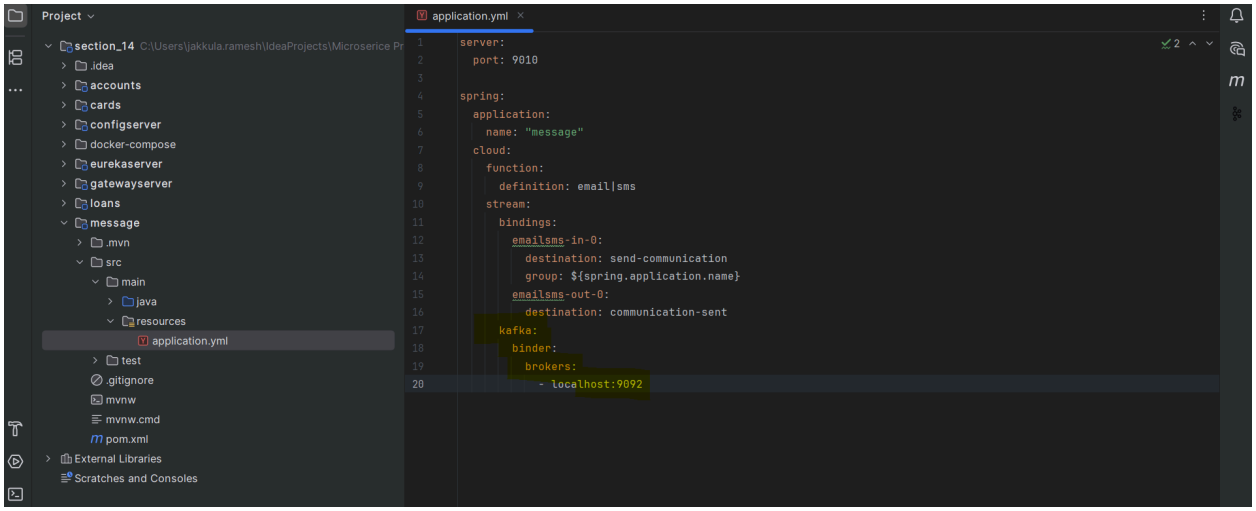
We just created business logic using functions once , now we can see we are just changing the environment using the same code.

Using kafka:

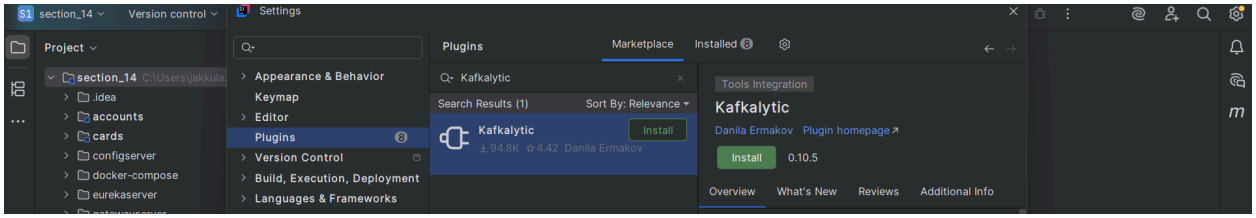
i) Install kafka and make sure it is running.

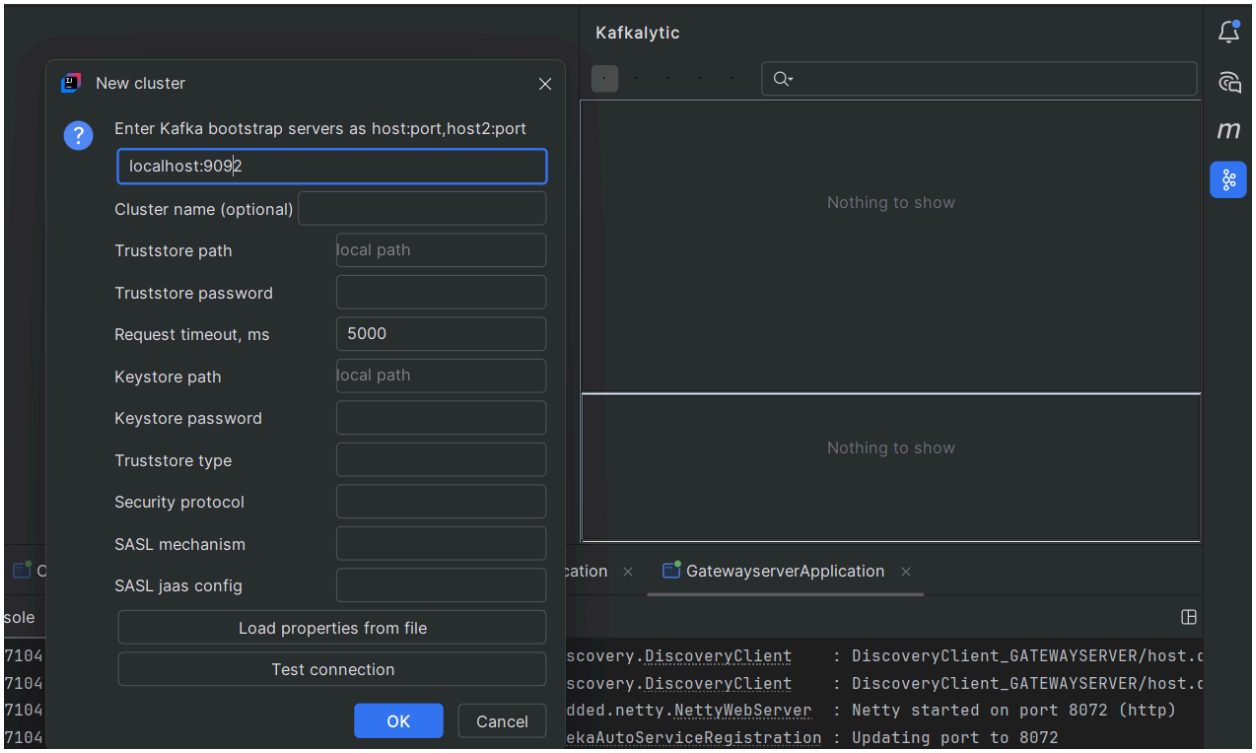
ii) Update the dependency related to kafka , kafka-specific configuration in application.yml in the microservices.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-binder-kafka</artifactId>
</dependency>
```

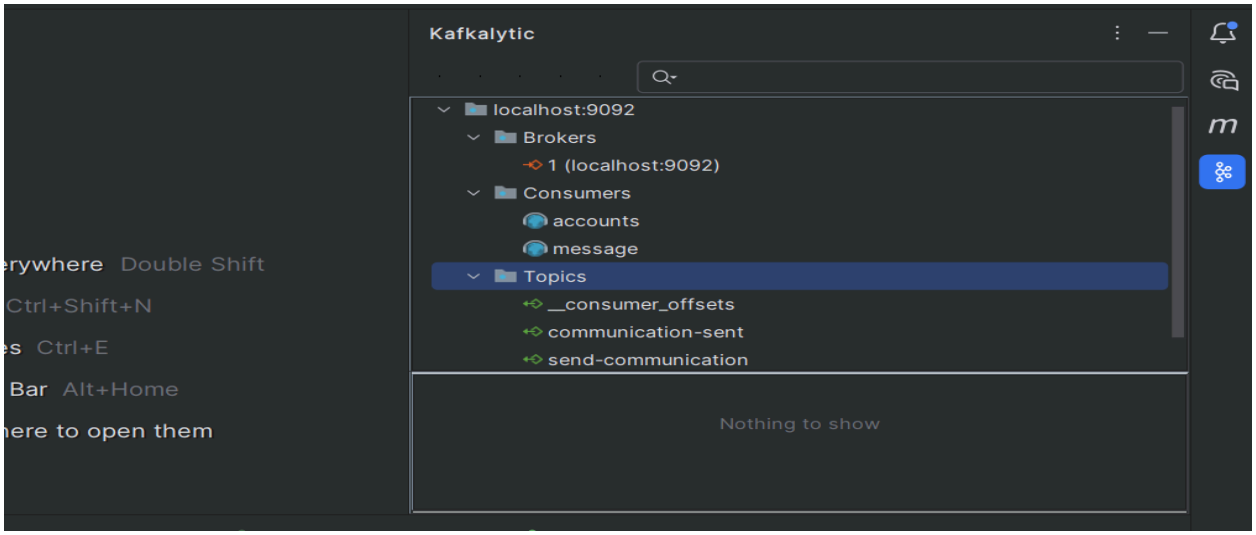


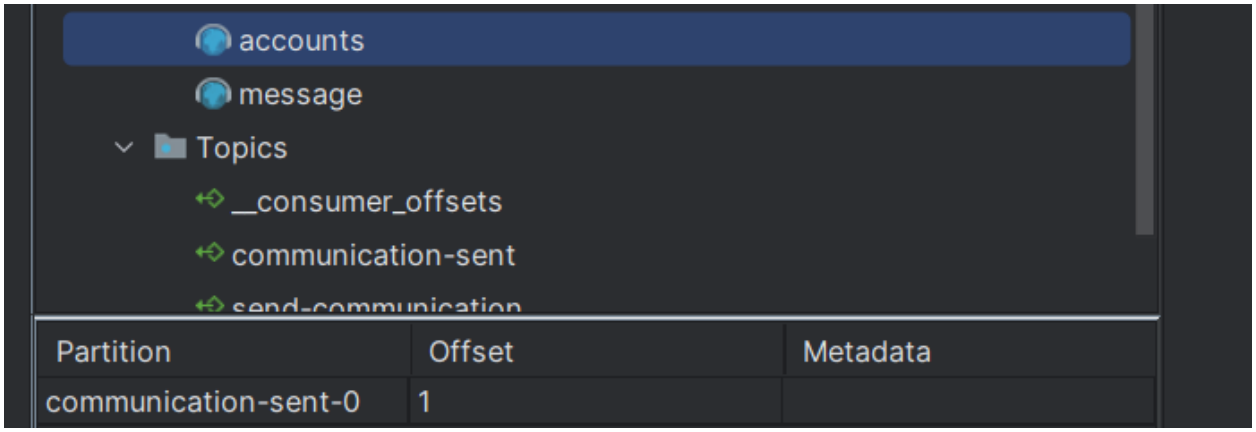
iii) **Install the *Kafkalytic* plugin** from the IntelliJ Marketplace to monitor Kafka topics, messages, offsets, etc.. **Add Kafka broker details** to *Kafkalytic* for connection.





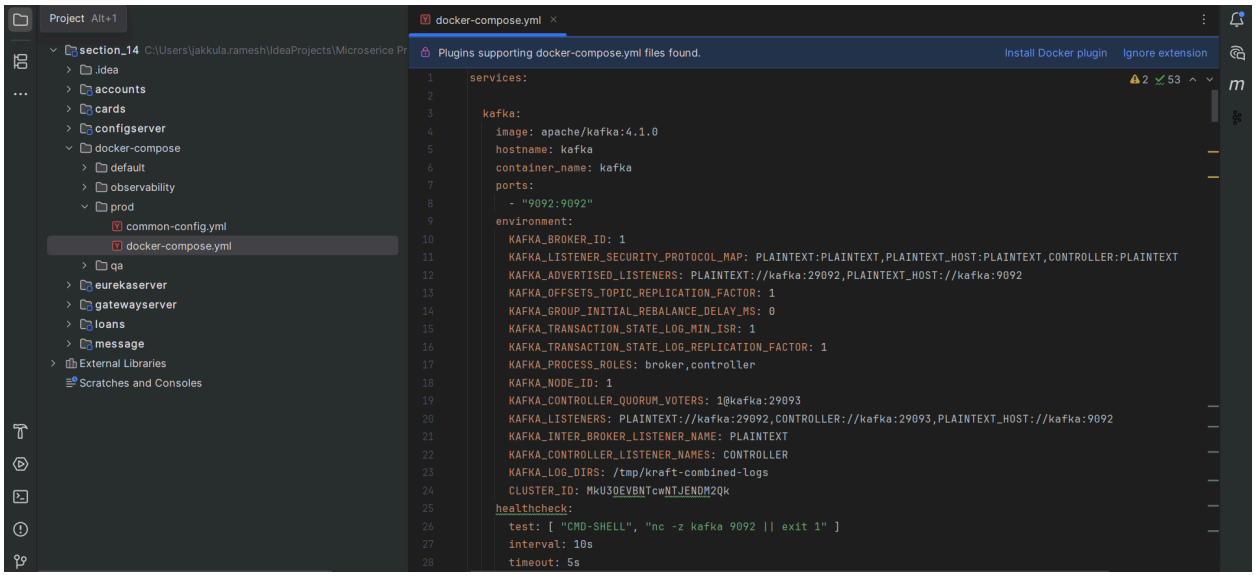
iv) **Run the microservices** and observe message flow in Kafka.

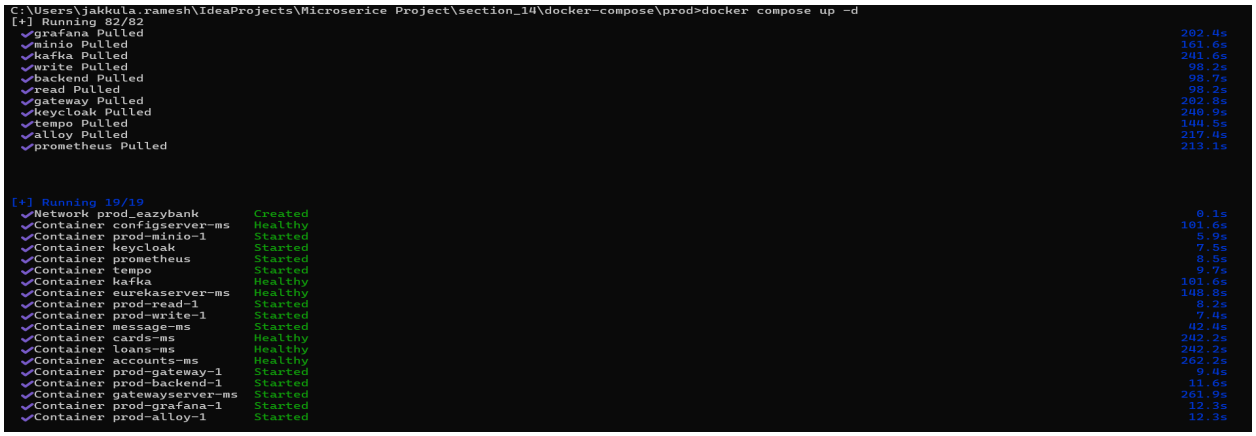


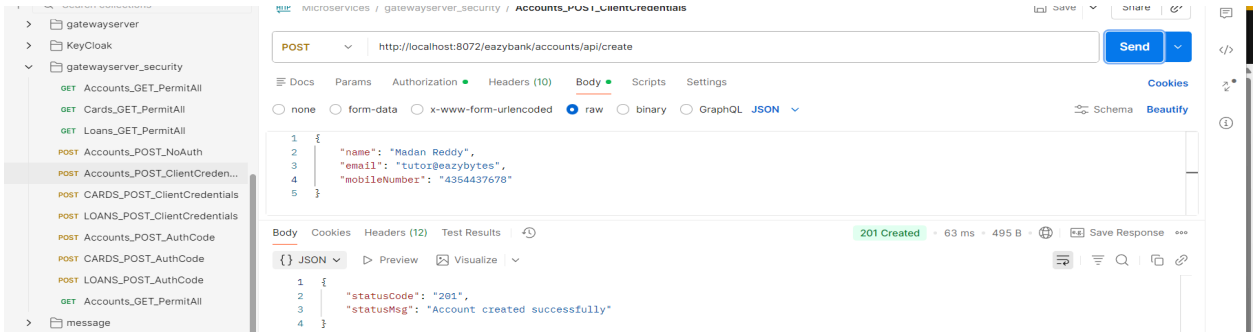


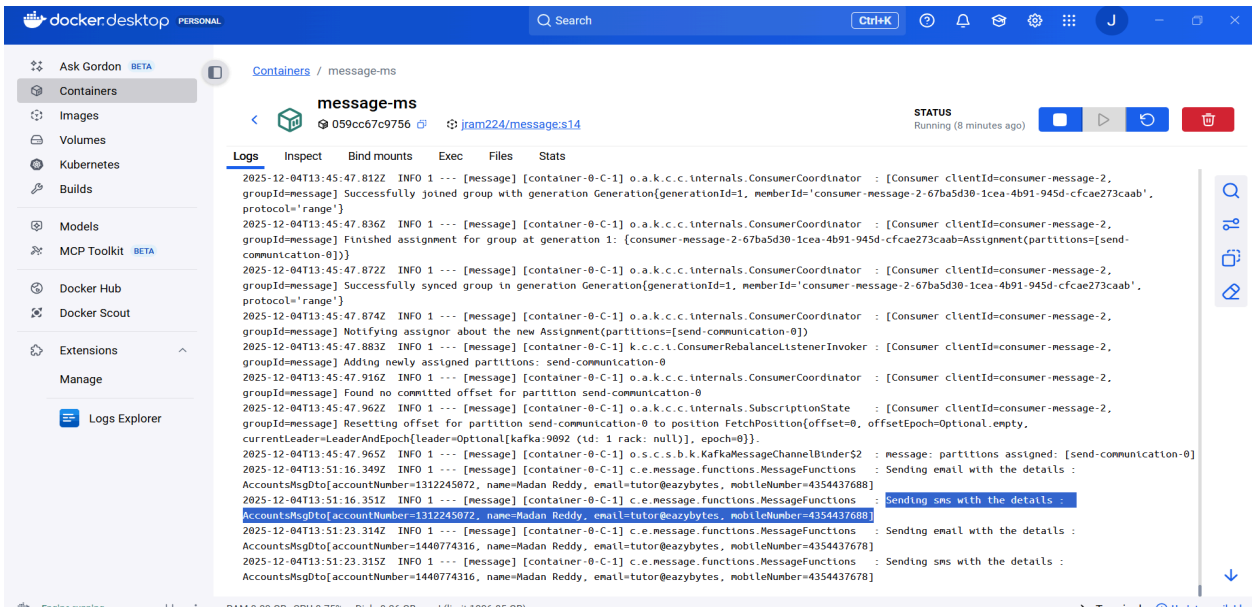
We can also test by generating docker images , docker-compose files.

https://developer.confluent.io/confluent-tutorials/kafka-on-docker/









Steps to use Apache Kafka in the place of RabbitMQ

Below are the steps to use Apache Kafka in the place of RabbitMQ,

1. **Add maven dependencies:** Add the maven dependency spring-cloud-stream-binder-kafka in the place of spring-cloud-stream-binder-rabbitmq dependency

2. **Add Kafka related properties inside the application.yml file of both accounts and message services**

```
spring:
  application:
    name: "accounts"
  cloud:
    function:
      definition: updateCommunication
    stream:
      bindings:
        updateCommunication-in-0:
          destination: communication-sent
          group:  ${spring.application.name}
        sendCommunication-out-0:
          destination: send-communication
      kafka:
        binder:
          brokers:
            - localhost:9092
```

```
spring:
  application:
    name: message
  cloud:
    function:
      definition: email|sms
    stream:
      bindings:
        emailsms-in-0:
          destination: send-communication
          group: ${spring.application.name}
        emailsms-out-0:
          destination: communication-sent
      kafka:
        binder:
          brokers:
            - localhost:9092
```

Before Kubernetes

Before Kubernetes, we used **Docker/Docker Compose** to run containers on a single server (like Docker Desktop). This approach worked for small applications but became difficult as the number of containers increased. We faced challenges like:

- No auto-scaling
- No self-healing if a container failed
- No built-in load balancing
- Hard to deploy and manage many containers

Kubernetes (or K8s)

Kubernetes is an open-source **container orchestration platform** used to **automate the deployment, scaling, monitoring and management of containerized applications**.

Kubernetes is often called **K8s,** which means the **'8' represents the eight letters between 'K'** and **'s'."**

(**orchestration** means to organize, manage, and control multiple things so they work together smoothly. Here, organizing and controlling multiple containers so they run smoothly together.)

Kubernetes helps you run many containers (like Docker containers) on multiple machines, monitors them, replaces failed ones, and keeps everything running smoothly automatically.

Why was Kubernetes created?

When applications run in containers, managing one or two is easy. But managing **hundreds or thousands** becomes difficult. Kubernetes solves this by:

✔ Automatically starting or stopping containers

✔ Restarting containers if they fail

✔ Distributing load across machines

✔ Scaling up when traffic increases and scaling down when needed

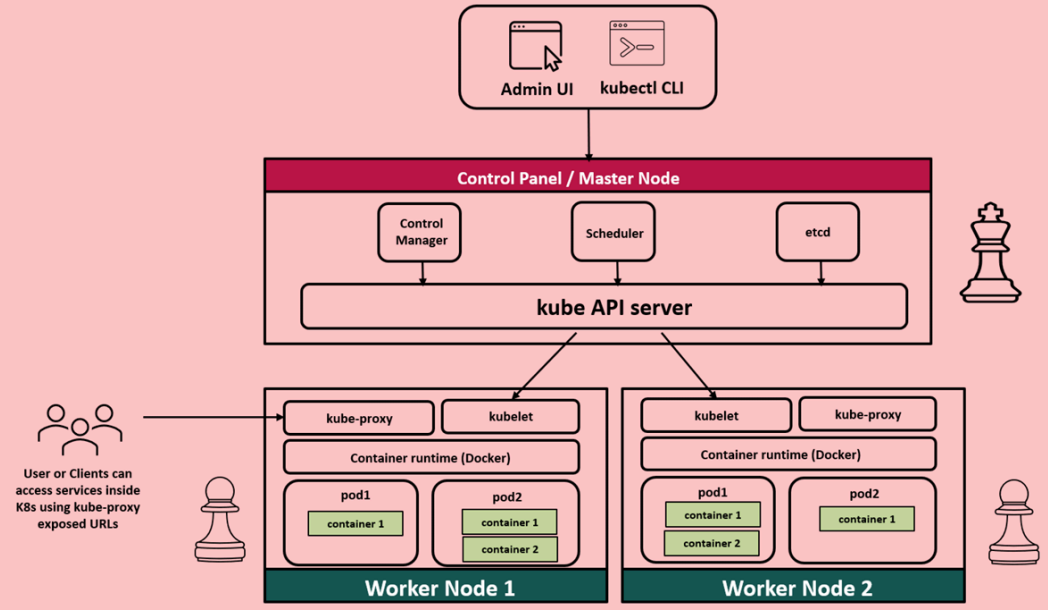
✔ Rolling out new versions of applications without downtime

A **Kubernetes Cluster** = Master/Control Plane + Worker Nodes

The full setup that combines both and manages everything end-to-end.

What Kubernetes Does	Meaning
Deploy containers	Runs application containers on available servers (nodes)
Load Balancing	Routes traffic evenly so no container is overloaded
Auto Scaling	Automatically increases/decreases containers based on demand
Self-Healing	Restarts or reschedules failed containers automatically
Rolling Updates	Deploys new versions with **zero downtime**

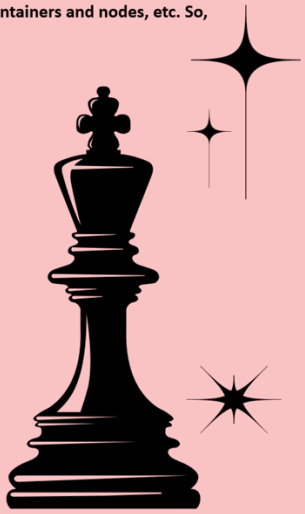
KUBERNETES INTERNAL ARCHITECTURE



Components of Control Panel (Master Node)

The master node is responsible for managing an entire cluster. It monitors the health check of all the nodes in the cluster, stores members' information regarding different nodes, plans the containers that are scheduled to certain worker nodes, monitors containers and nodes, etc. So, when a worker node fails, the master moves the workload from the failed node to another healthy worker node.

Below are the details of the four basic components present inside the control panel,



API server - The API server is the primary interface for interacting with the Kubernetes cluster. It exposes the Kubernetes API, which allows users and other components to communicate with the cluster. All administrative operations and control commands are sent to the API server, which then processes and validates them.

Scheduler - The scheduler is responsible for placing Pods onto available nodes in the cluster. It takes into account factors like resource requirements, affinity, anti-affinity, and other constraints to make intelligent decisions about which node to assign a Pod to. The scheduler continuously monitors the cluster and ensures that Pods are distributed optimally.

Controller manager - The controller manager maintains the cluster. It handles node failures, replicates components, maintains the correct number of pods, etc. It constantly tries to keep the system in the desired state by comparing it with the current state of the system.

etcd - etcd is a distributed key-value store that serves as the cluster's primary data store. It stores the configuration data and the desired state of the system, including information about Pods, Services, ReplicationControllers, and more. The API server interacts with etcd to read and write cluster data.

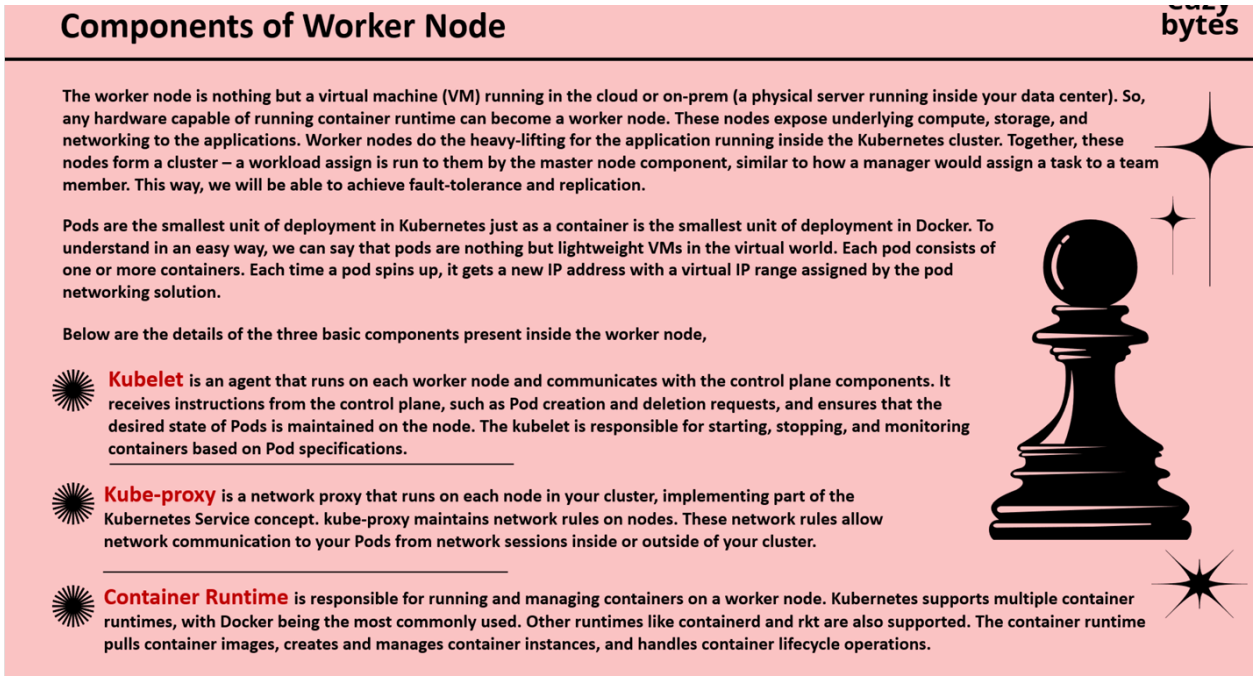
1️⃣ Control Plane (Master Node)

This is the **brain of Kubernetes**.
It decides what should run, when, where, how many copies, ensures everything is healthy, etc.

Core components:

Component	Function (Simple Explanation)
API Server (kube-apiserver)	Entry point for all commands (kubectl, UIs, automation). Acts like a receptionist.
Scheduler (kube-scheduler)	Decides *which node* runs a pod based on CPU, RAM, taints, affinity etc.
Controller Manager (kube-controller-manager)	Ensures desired state = actual state. Restarts failed pods, scales replicas, updates nodes etc.
etcd	A distributed database where the entire cluster config & state is stored. The **source of truth**.

📌 If Control Plane fails → cluster cannot make decisions, but already-running workloads may continue.



2️⃣ Worker Nodes

These are the machines where containers actually run.

Main components:

Component	Function
kubelet	Talks to API server, runs containers assigned to its node, reports status & health.
Container Runtime (Docker / containerd / CRI-O)	Starts & runs containers.
kube-proxy	Handles networking & load balancing inside the cluster. Routes traffic to proper pod IP.

We mainly interact with Kubernetes in **two ways**:

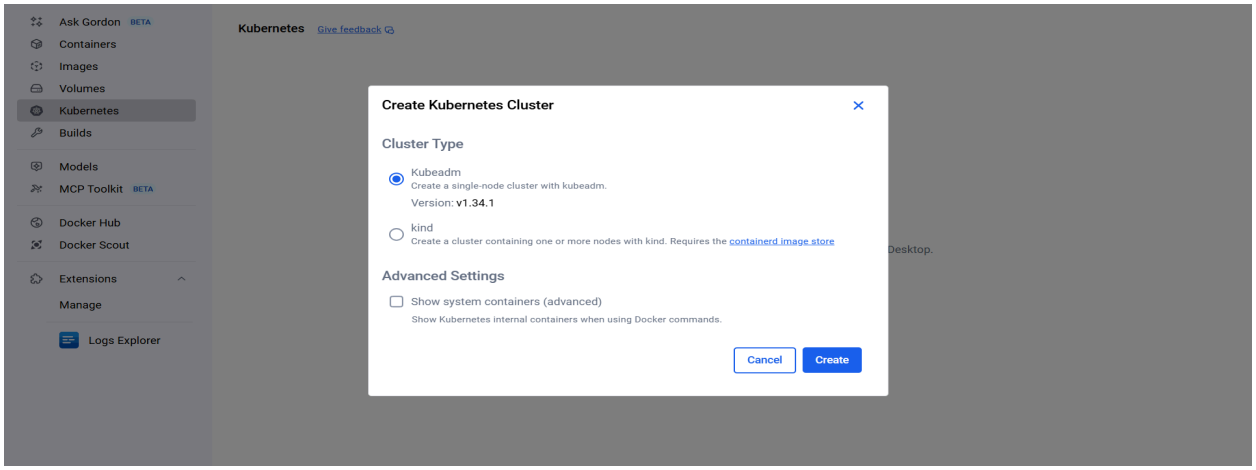
1) kubectl CLI (Command-line tool) : This is the primary way developers/administrators give instructions to Kubernetes.

2) Admin UI / Kubernetes Dashboard : A web-based UI (also called **Kubernetes Dashboard**) to manage and monitor the cluster visually.

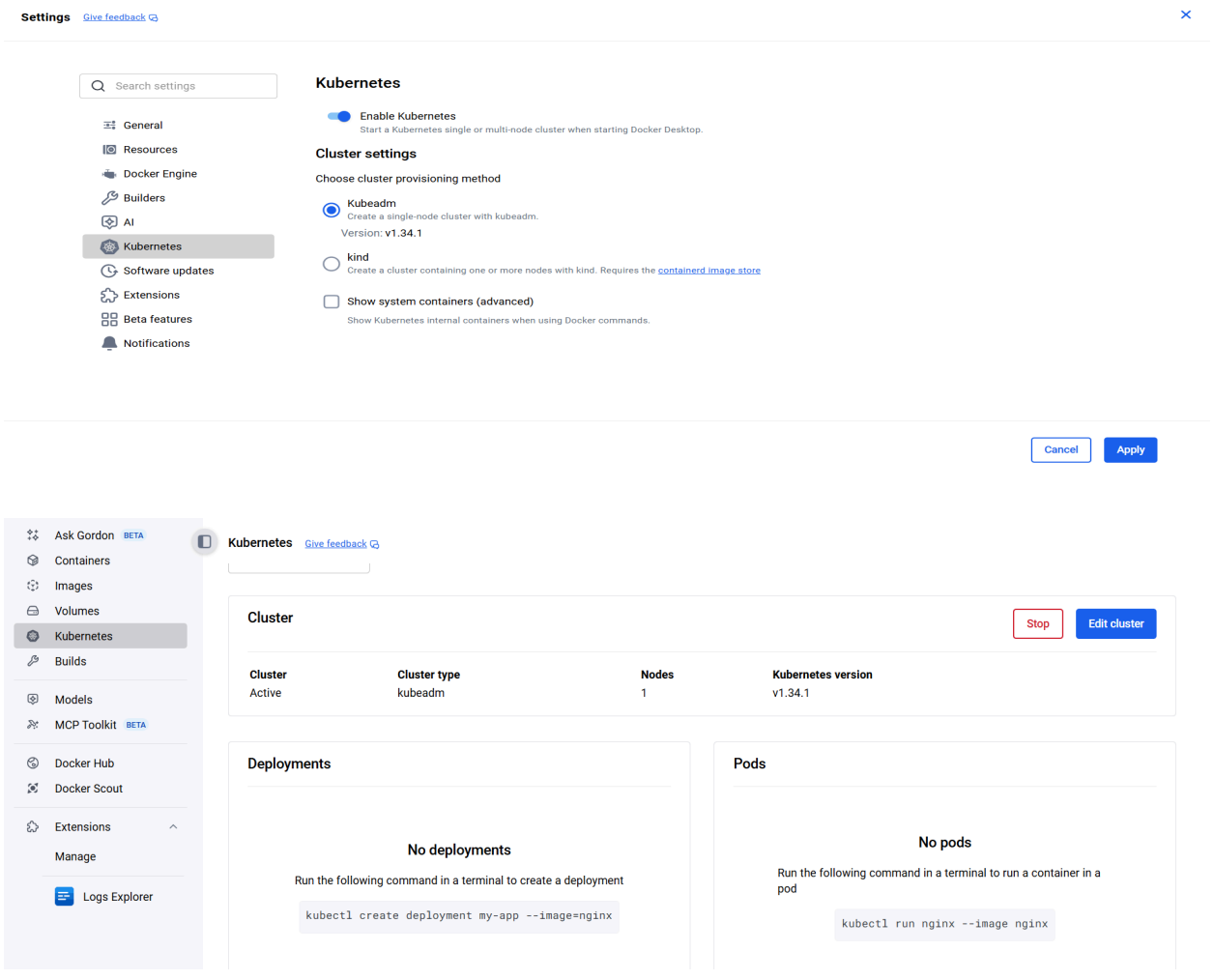
Set up local **Kubernetes Cluster** using the Docker desktop:

To set up **Kubernetes** in the local using Docker desktop :
https://docs.docker.com/desktop/use-desktop/kubernetes/

We can create directly from docker dashboard **Kubernetes** option in UI or from the settings we can enable it.



or



Interacting with **kubectl CLI (Command-line tool):**

```
C:\Users\jakkula.ramesh>kubectl version
Client Version: v1.34.1
Kustomize Version: v5.7.1
Server Version: v1.34.1

C:\Users\jakkula.ramesh>kubectl config get-contexts
CURRENT   NAME             CLUSTER          AUTHINFO         NAMESPACE
*         docker-desktop   docker-desktop   docker-desktop

C:\Users\jakkula.ramesh>kubectl get nodes
NAME             STATUS   ROLES           AGE     VERSION
docker-desktop   Ready    control-plane   4m10s   v1.34.1

C:\Users\jakkula.ramesh>kubectl config get-clusters
NAME
docker-desktop

C:\Users\jakkula.ramesh>
```

Interacting with **Admin UI :**

To set up Kubernetes UI :
https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/

Before that we need to install Helm.

Helm is a package manager for Kubernetes that simplifies installing, upgrading, and managing applications. Like Node for JS , Heml is for Kubernetes . To install Heml : https://helm.sh/ , https://helm.sh/docs/intro/install

```
C:\Users\jakkula.ramesh>choco
Chocolatey v2.3.0
Please run 'choco -?' or 'choco <command> -?' for help menu.

C:\Users\jakkula.ramesh>
```

```
C:\Windows\System32>choco install kubernetes-helm
Chocolatey v2.3.0
Installing the following packages:
kubernetes-helm
By installing, you accept licenses for the packages.
Downloading package from source 'https://community.chocolatey.org/api/v2/'

kubernetes-helm v4.0.0 [Approved]
kubernetes-helm package files install completed. Performing other installation steps.
The package kubernetes-helm wants to run 'chocolateyInstall.ps1'.
Note: If you don't run this script, the installation will fail.
Note: To confirm automatically next time, use '-y' or consider:
choco feature enable -n allowGlobalConfirmation
Do you want to run the script?([Y]es/[A]ll - yes to all/[N]o/[P]rint): Y

Downloading kubernetes-helm 64 bit
  from 'https://get.helm.sh/helm-v4.0.0-windows-amd64.zip'
Progress: 100% - Completed download of C:\Users\jakkula.ramesh\AppData\Local\Temp\chocolatey\kubernetes-helm\4.0.0\helm-v4.0.0-windows-amd64.zip (19.5 MB).
Download of helm-v4.0.0-windows-amd64.zip (19.5 MB) completed.
Hashes match.
Extracting C:\Users\jakkula.ramesh\AppData\Local\Temp\chocolatey\kubernetes-helm\4.0.0\helm-v4.0.0-windows-amd64.zip to C:\ProgramData\chocolatey\lib\kubernetes-helm\tools...
C:\ProgramData\chocolatey\lib\kubernetes-helm\tools
 ShimGen has successfully created a shim for helm.exe
 The install of kubernetes-helm was successful.
  Deployed to 'C:\ProgramData\chocolatey\lib\kubernetes-helm\tools'

Chocolatey installed 1/1 packages.
 See the log for details (C:\ProgramData\chocolatey\logs\chocolatey.log).

C:\Windows\System32>
```

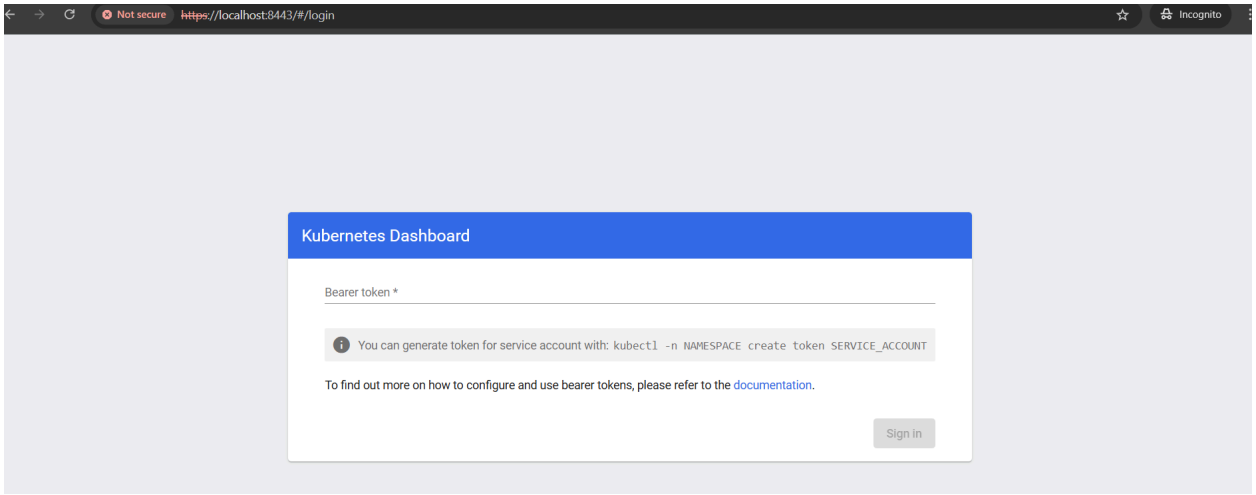
Helm installed

```
C:\Users\jakkula.ramesh>helm version
version.BuildInfo{Version:"v4.0.0", GitCommit:"99cd1964357c793351be481d55abbe21c6b2f4ec", GitTreeState:"clean", GoVersion:"go1.25.3", KubeClientVersion:"v1.
34"}

C:\Users\jakkula.ramesh>helm repo add kubernetes-dashboard https://kubernetes.github.io/dashboard/
"kubernetes-dashboard" has been added to your repositories

C:\Users\jakkula.ramesh>helm upgrade --install kubernetes-dashboard kubernetes-dashboard/kubernetes-dashboard --create-namespace --namespace kubernetes-dash
board
Release "kubernetes-dashboard" does not exist. Installing it now.
level=WARN msg="unable to find exact version; falling back to closest available version" chart=kubernetes-dashboard requested="" selected=7.14.0
NAME: kubernetes-dashboard
LAST DEPLOYED: Fri Dec  5 12:32:50 2025
NAMESPACE: kubernetes-dashboard
STATUS: deployed
REVISION: 1
DESCRIPTION: Install complete
TEST SUITE: None
NOTES:
**********************************************************************************************
*** PLEASE BE PATIENT: Kubernetes Dashboard may need a few minutes to get up and become ready ***
**********************************************************************************************

Congratulations! You have just installed Kubernetes Dashboard in your cluster.

To access Dashboard run:
  kubectl -n kubernetes-dashboard port-forward svc/kubernetes-dashboard-kong-proxy 8443:443

NOTE: In case port-forward command does not work, make sure that kong service name is correct.
      Check the services in Kubernetes Dashboard namespace using:
        kubectl -n kubernetes-dashboard get svc

Dashboard will be available at:
  https://localhost:8443

C:\Users\jakkula.ramesh>
```

```
Congratulations! You have just installed Kubernetes Dashboard in your cluster.

To access Dashboard run:
  kubectl -n kubernetes-dashboard port-forward svc/kubernetes-dashboard-kong-proxy 8443:443

NOTE: In case port-forward command does not work, make sure that kong service name is correct.
      Check the services in Kubernetes Dashboard namespace using:
        kubectl -n kubernetes-dashboard get svc

Dashboard will be available at:
  https://localhost:8443
```

```
C:\Users\jakkula.ramesh> kubectl -n kubernetes-dashboard port-forward svc/kubernetes-dashboard-kong-proxy 8443:443
Forwarding from 127.0.0.1:8443 -> 8443
Forwarding from [::1]:8443 -> 8443
```

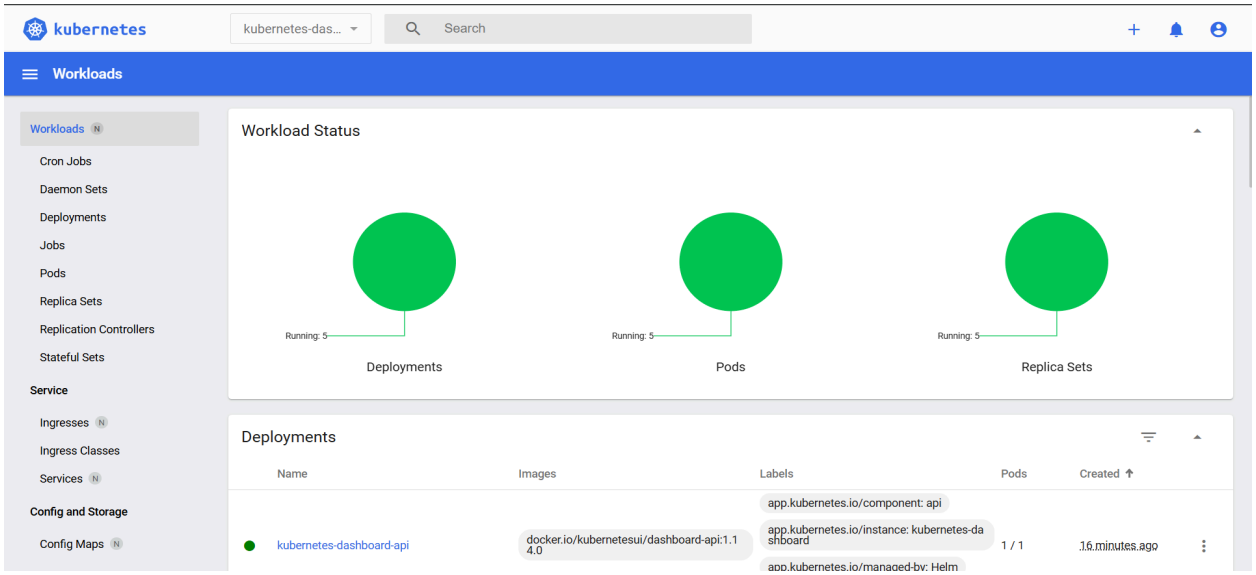
Hit localhost:8443



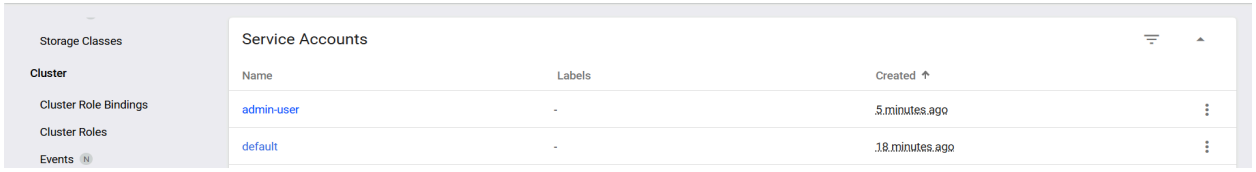
To get a token to login to Kubernetes UI , we should create a user first.
https://github.com/kubernetes/dashboard/blob/master/docs/user/access-control/creating-sample-user.md

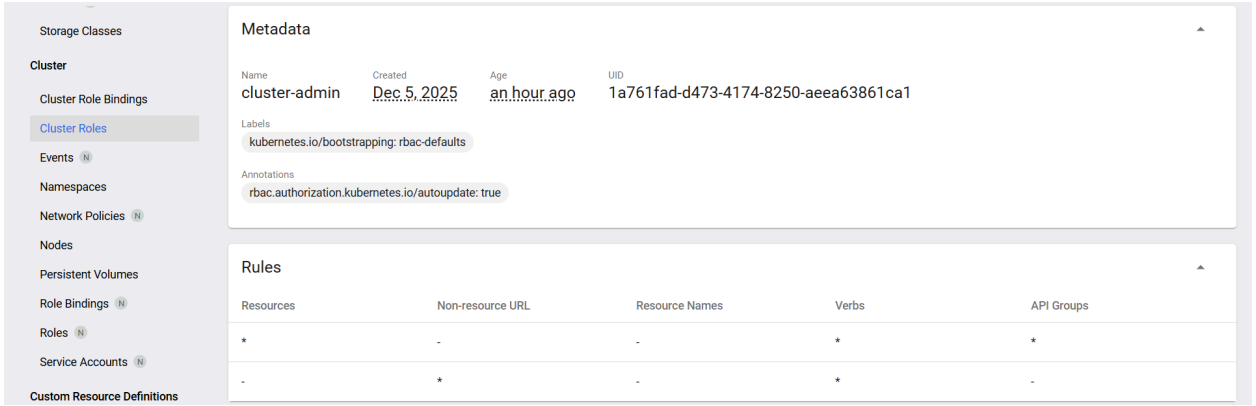
Created respective .yml files mentioned above url get the token.

```
Microsoft Windows [Version 10.0.26100.7171]
(c) Microsoft Corporation. All rights reserved.

C:\Users\jakkula.ramesh\Desktop\K8s>kubectl apply -f dashboard-adminuser.yaml
serviceaccount/admin-user created

C:\Users\jakkula.ramesh\Desktop\K8s>kubectl apply -f dashboard-rolebinding.yaml
clusterrolebinding.rbac.authorization.k8s.io/admin-user created

C:\Users\jakkula.ramesh\Desktop\K8s>kubectl -n kubernetes-dashboard create token admin-user
eyJhbGciOiJSUzI1NiIsImtpZCI6ImpmdXgyWEFHaklRdUx3MzNfUllBOGhOLUFyUWdrOHFnUUNJSUY0Z0EyNXcifQ.eyJhdWQiOlsiaHR0cHM6Ly9rdWJlcm5ldGVzLmRlZmF1bHQuc3ZjLmNsdXN0ZXIub
G9jYWwiXSwiZXhwIjoxNzY0OTIyNzU1LCJpYXQiOjE3NjQ5MTkxNTUsImlzcyI6Imh0dHBzOi8va3ViZXJuZXRlcy5kZWZhdWx0LnN2Yy5jbHVzdGVyLmxvY2FsIiwianRpIjoiYTY1NmJkNzYtYTU4NC00M
Dc1LWE5NDYtODJmZjcxYjUwZWY2Iiwia3ViZXJuZXRlcy5pbyI6eyJuYW1lc3BhY2UiOiJrdWJlcm5ldGVzLWRhc2hib2FyZCIsInNlcnZpY2VhY2NvdW50Ijp7Im5hbWUiOiJhZG1pbi11c2VyIiwidWlkI
joiZjcyNGJkZjItYmQyNi00OTZlLTgzYWMtMGIxNzg1YjYwZDAzIn19LCJuYmYiOjE3NjQ5MTkxNTUsInN1YiI6InN5c3RlbTpzZXJ2aWNlYWNjb3VudDprdWJlcm5ldGVzLWRhc2hib2FyZDphZG1pbi11c
2VyIn0.cRCcKSZdEFlTMfg0TMG428tEx6RQxsqW5-ksST1xBzjTlVQniudxyncIYFq63ZVVSfTDD6onQh3XMbQYWDJzDEQMEcM_kgy_Tf4asoCZLz2woYe904YG-f7CB_LkYT5r9KmupFDtBAbNairFnKY1N
tO8zE84a3lc82yjeWHgztdvJYghzMp3sbPyjaHQIDVQqeQWYrxpXeSmdG5ZumVkUHqaIn4G6-2hwHVdsVgnV5ojJXTq_qpvQLQd1arjdrY_eGCjse65uCQt68cRrKW8ZrNIcK2CklrSxaAGfl5NG8Sj0_-1y
EyHCCIk4ogMj_fd1uAYzQOXsxPqxq9Ton_IYg

C:\Users\jakkula.ramesh\Desktop\K8s>
```



We can see created account also under service accounts





Deploy microservice into Kubernetes Clusters :

https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

Step-1 : Create Kubernetes Manifest/YAML files (Deployment + Service)

To deploy a microservice into Kubernetes, we first need a **YAML configuration file** for respective microservices.
These YAML/manifest files tell Kubernetes **what to create and how to run it**.

A single YAML file may contain multiple Kubernetes yaml files separated by `---`.

kind → Type of Kubernetes object (Deployment, Service, ConfigMap, etc.)
replicas → Number of pod instances to run
labels (app/app) → Helps Kubernetes identify & group pods
container image → Docker image to run the app

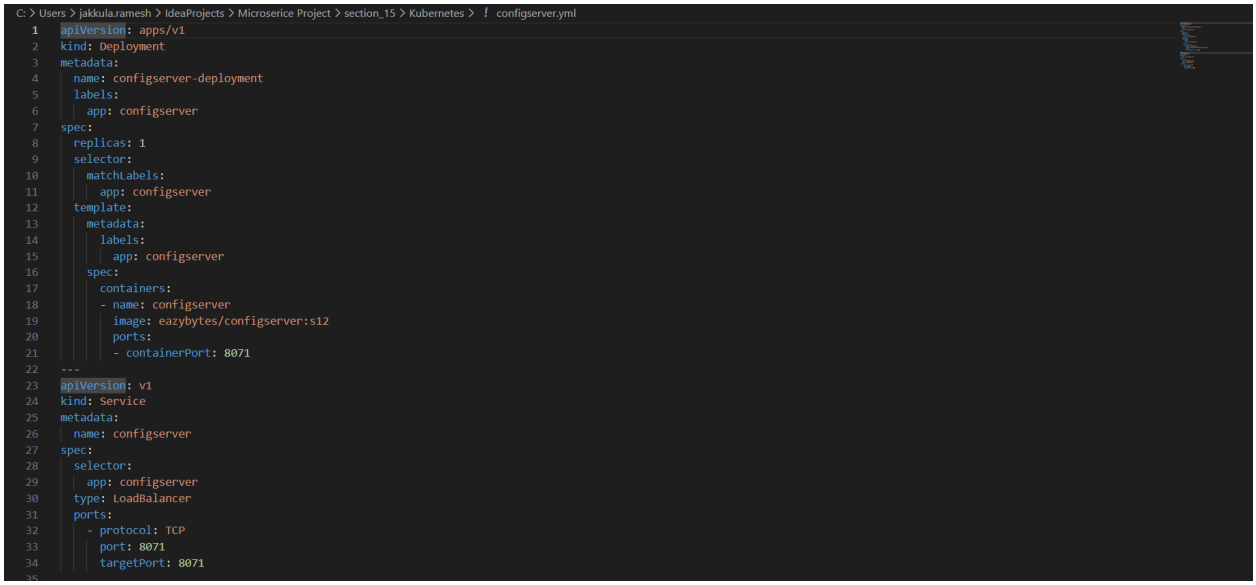
- If you don't specify any registry, Kubernetes assumes **docker.io (Docker Hub)**
- If the image is in AWS/GCP/Azure/private registry, the **full image path must be mentioned**

Next, we add a **Service** object in the same YAML to expose the microservice to users.
Inside Service, we define:

ports → Which port application runs on

type → How the service is exposed

EX:



Used for **running the application pod(s)**

- apiVersion: apps/v1 → API version used for creating a Deployment
- kind: Deployment → Tells Kubernetes we are deploying an application
- name: configserver-deployment → Deployment name
- labels: app=configserver → Helps Kubernetes identify/group resources
- replicas: 1 → Only one pod instance will run
- selector.matchLabels.app=configserver → Deployment manages pods with this label
- template: → Pod blueprint created by deployment
- template.labels.app=configserver → Label inside pod matching the selector
- containers: → Pod will contain this container
- name: configserver → Container name
- image: eazybytes/configserver:s12 → Image used to create container
  (No registry mentioned → default = docker.io)
- containerPort: 8071 → Application listens inside container on port 8071

Used for **accessing the Pod from outside or inside cluster**

- apiVersion: v1 → API version for Service resource
- kind: Service → Exposes Pod(s) to network
- name: configserver → Service name
- selector.app=configserver → Connects Service to Pods with same label
  (**this is the common part used in both Deployment & Service**)
- type: LoadBalancer → Exposes app externally (mainly for cloud)
- port: 8071 → External port clients will hit
- targetPort: 8071 → The port inside container receiving traffic

Step-2: Use Kubernetes commands to check & apply, /deploying to clusters

Before deploying

```
Microsoft Windows [Version 10.0.26100.7171]
(c) Microsoft Corporation. All rights reserved.

C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl get deployment
No resources found in default namespace.

C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl get pods
No resources found in default namespace.

C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl get replicaset
No resources found in default namespace.

C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl get services
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   3h47m
```

After deploying

```
C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl apply -f configserver.yml
deployment.apps/configserver-deployment created
service/configserver created

C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl get deployment
NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
configserver-deployment   0/1     1            0           25s

C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl get pods
NAME                                     READY   STATUS              RESTARTS   AGE
configserver-deployment-5767c559c7-ghphq   0/1     ContainerCreating   0          31s

C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl get services
NAME           TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
configserver   LoadBalancer   10.106.99.46   localhost     8071:32015/TCP   36s
kubernetes     ClusterIP      10.96.0.1      <none>        443/TCP          3h50m

C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl get replicaset
NAME                                DESIRED   CURRENT   READY   AGE
configserver-deployment-5767c559c7   1         1         1       50s

C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>
```

1. Deploy the microservice : kubectl apply -f configserver.yml

2. Check Deployment status : kubectl get deployment

Shows replicas, availability & rollout progress.

3. Check Pods status : kubectl get pods

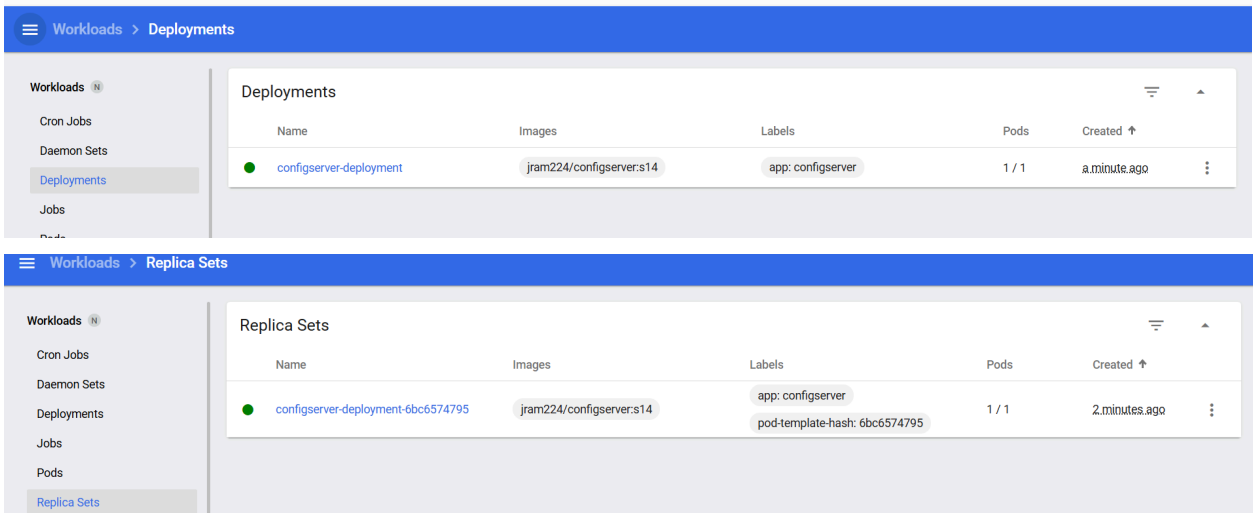
Verifies Pods created by Deployment.

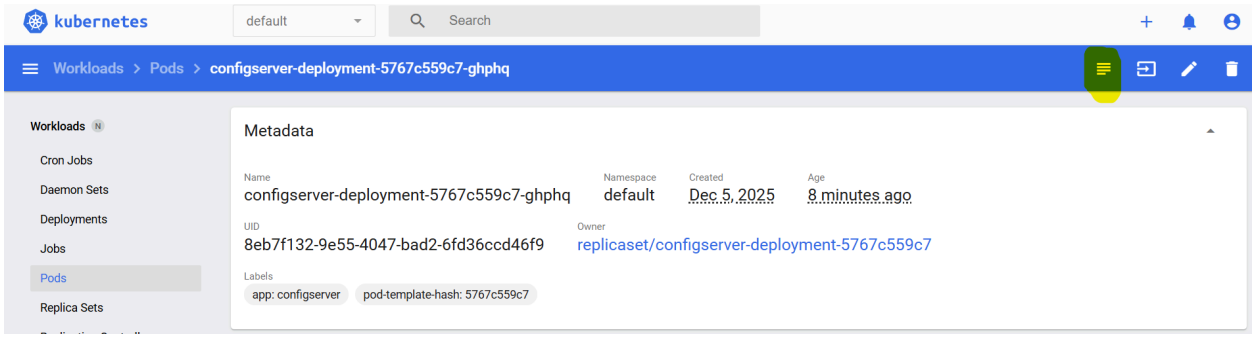
4. Check Service status : kubectl get service

Shows port, type (ClusterIP/NodePort/LoadBalancer) & cluster IP.

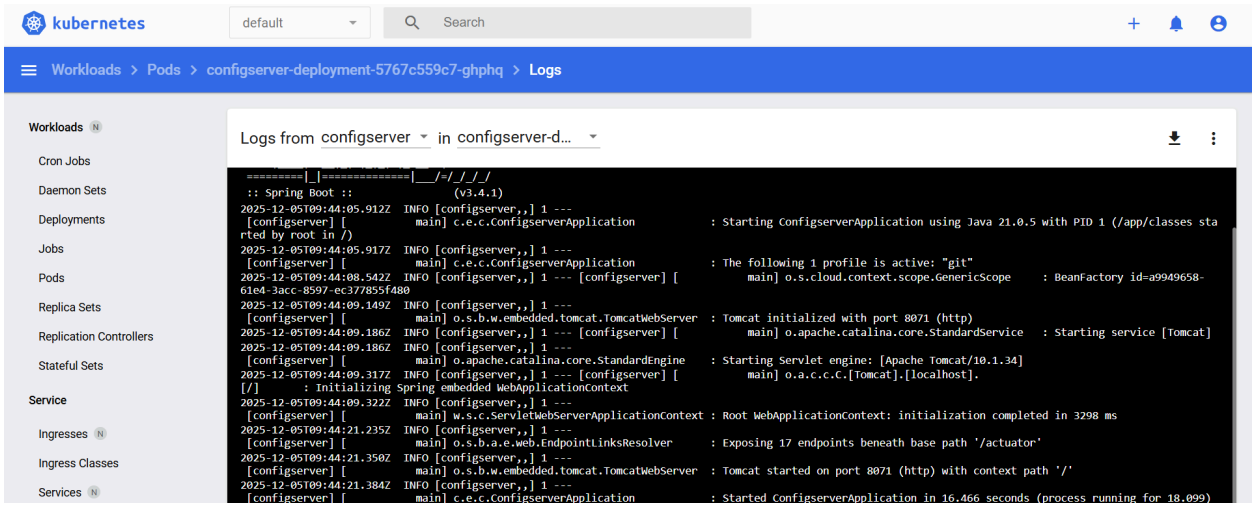
5. Check ReplicaSet created behind the Deployment : kubectl get replicaset

We can see that the docker is deployed .





We can see the running log of services of config server when click on above option docker



ConfigMaps

In Docker Compose, we pass environment variables inside the service section (environment:).

When moving to **Kubernetes ,** we should not hard-code these values inside Deployment YAML.

Instead, the correct & standard approach is to store them in:

✔ ConfigMap → for non-secret environment configs

(profile, eureka URL, port, hostnames etc.)

A ConfigMap is an API object used to **store non-confidential data in key-value pairs**. Pods can consume ConfigMaps as **environment variables, command-line arguments, or as configuration files in a volume**.

A ConfigMap allows you to decouple environment-specific configuration from your container images, so that your applications are easily portable.

https://kubernetes.io/docs/concepts/configuration/configmap/

✔ Secret → for passwords, credentials, tokens

https://kubernetes.io/docs/concepts/configuration/secret/

Then Deployment will **read values from ConfigMap/Secret** the same way Docker Compose containers read from the environment:.

A Kubernetes ConfigMap is an essential Kubernetes resource used to store configuration data separately from the application code.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: eazybank-configmap
data:
  SPRING_PROFILES_ACTIVE: prod
```

The apiVersion and kind fields are required for all Kubernetes objects. When creating a config map inside K8s, the kind should be "ConfigMap"

The metadata field contains the name of the ConfigMap and other metadata about the object.

The data field is where the key-value pairs are stored. The keys can be any alphanumeric string, and the values can be strings, numbers, or binary data.

Example mapping from Docker-Compose → Kubernetes

Docker-Compose Example:

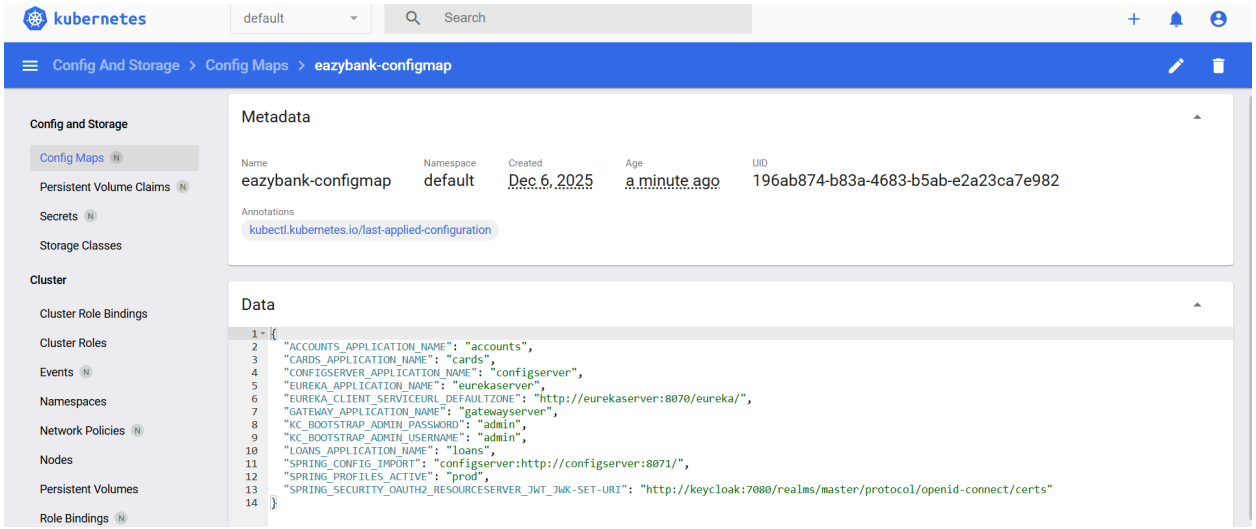
```yaml
services:
  configserver:
    image: eazybytes/configserver:s12
    environment:
      - SPRING_PROFILES_ACTIVE=dev
      - EUREKA_SERVER_URL=http://eureka:8761/eureka
```

Kubernetes Equivalent → Using ConfigMap

1️⃣ Create ConfigMap

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: configserver-config
data:
  SPRING_PROFILES_ACTIVE: "dev"
  EUREKA_SERVER_URL: "http://eureka:8761/eureka"
```

```
C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl apply -f  configmaps.yaml
configmap/eazybank-configmap created

C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>
```



We will create Kubernetes YAML files for all microservices:

Keycloak, ConfigServer, EurekaServer, Accounts, Loans, Cards, Gateway

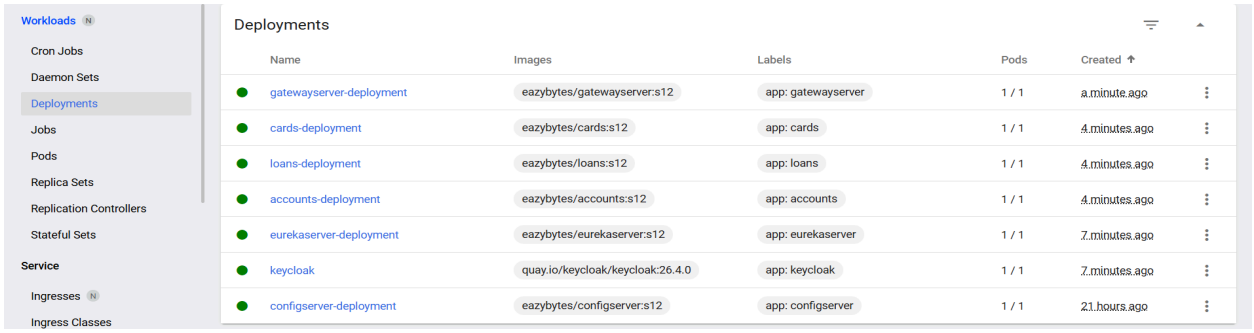
and inject common values from ConfigMaps.

The code snippet which will use common values from ConfigMaps.

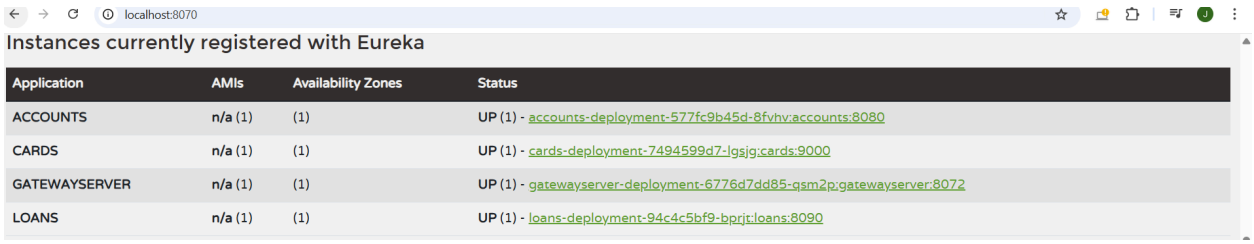
```yaml
        valueFrom:
          configMapKeyRef:
            name: eazybank-configmap
            key: SPRING_PROFILES_ACTIVE
      - name: SPRING_CONFIG_IMPORT
        valueFrom:
          configMapKeyRef:
            name: eazybank-configmap
            key: SPRING_CONFIG_IMPORT
      - name: EUREKA_CLIENT_SERVICEURL_DEFAULTZONE
        valueFrom:
          configMapKeyRef:
            name: eazybank-configmap
            key: EUREKA_CLIENT_SERVICEURL_DEFAULTZONE
```

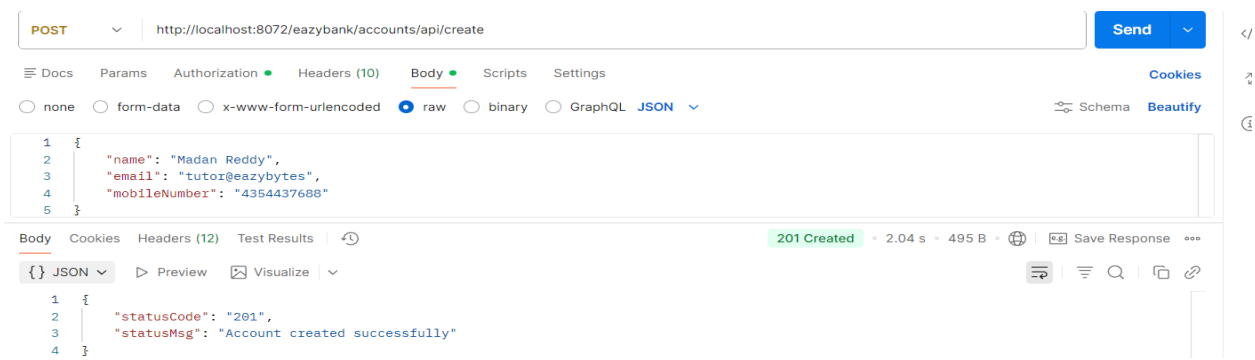
Start all .yml in order to see the output. Sample image

```
C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl apply -f 1_keycloak.yml
deployment.apps/keycloak created
service/keycloak created

C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl apply -f 2_configmaps.yaml
configmap/eazybank-configmap unchanged

C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl apply -f 3_configserver.yml
deployment.apps/configserver-deployment configured
service/configserver unchanged
```



We can also see in the eureka all services are running.





Kubernetes manifest file to deploy a Container

bytes

In Kubernetes, a Deployment is a high-level resource used to manage the deployment and scaling of containerized applications. It provides a declarative way to define and maintain the desired state of your application. When you create a Deployment, Kubernetes ensures that the specified number of replicas of your application are running and automatically handles scaling, rolling updates, and rollbacks.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: accounts-deployment
  labels:
    app: accounts
spec:
  replicas: 1
  selector:
    matchLabels:
      app: accounts
  template:
    metadata:
      labels:
        app: accounts
    spec:
      containers:
      - name: accounts
        image: eazybytes/accounts:latest
        ports:
        - containerPort: 8080
        env:
        - name: SPRING_PROFILES_ACTIVE
          valueFrom:
            configMapKeyRef:
              name: eazybank-configmap
              key: SPRING_PROFILES_ACTIVE
```

The apiVersion and kind fields are required for all Kubernetes objects. For deployment manifest file, the kind should be “Deployment”

Metadata: The metadata section contains information about the Deployment, such as its name and labels. The Deployment is named "accounts-deployment", and it has the label "app: accounts".

Spec: The spec section defines the desired state of the Deployment.

Replicas: The replicas field is set to 1, indicating that only one replica of the container should be running at any given time.

Selector: The selector field is used to select the pods controlled by this Deployment. In this case, it's using the label "app: accounts" to identify the pods.

Kubernetes manifest file to deploy a Container

bytes

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: accounts-deployment
  labels:
    app: accounts
spec:
  replicas: 1
  selector:
    matchLabels:
      app: accounts
  template:
    metadata:
      labels:
        app: accounts
    spec:
      containers:
      - name: accounts
        image: eazybytes/accounts:latest
        ports:
        - containerPort: 8080
        env:
        - name: SPRING_PROFILES_ACTIVE
          valueFrom:
            configMapKeyRef:
              name: eazybank-configmap
              key: SPRING_PROFILES_ACTIVE
```

Template: The template section specifies the pod template that will be used to create pods for this Deployment.

Metadata: The metadata section inside the template defines the labels for the pods. The pod will have the label "app: accounts".

Spec: The spec section inside the template specifies the details of the pod's specification.

Containers: The containers field lists the containers that should be part of the pod. In this case, there is one container named "accounts" based on the "eazybytes/accounts:latest" image.

Ports: The ports section exposes port 8080 of the container.

Environment Variables: The env section sets an environment variable named "SPRING_PROFILES_ACTIVE" and assigns it a value from a ConfigMap.

valueFrom: The valueFrom field allows you to reference data from a ConfigMap. In this case, it is using configMapKeyRef to fetch the value of the "SPRING_PROFILES_ACTIVE" key from the ConfigMap named "eazybank-configmap".

Kubernetes manifest file to create a service

bytes

In Kubernetes, a Service is an essential resource that provides network connectivity to a set of pods. It acts as a stable endpoint for accessing and load balancing traffic across multiple replicas of a pod. Services abstract away the underlying network details, allowing pods to be more dynamic and scalable without affecting how clients access them.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: accounts-service
spec:
  selector:
    app: accounts
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

The apiVersion and kind fields are required for all Kubernetes objects. For service manifest file, the kind should be “Service”

Metadata: The metadata section contains information about the Service, such as its name.

Spec: The spec section defines the desired state of the Service.

Selector: The selector field is used to select the pods that the Service will route traffic to. In this case, it uses the label "app: accounts" to select the pods controlled by the Deployment with the same label.

Type: The type field specifies the type of Service. In this case, it is set to "ClusterIP," which means that the Service will be accessible only from within the cluster.


```yaml
apiVersion: v1
kind: Service
metadata:
  name: accounts-service
spec:
  selector:
    app: accounts
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```


Ports: The ports section defines the ports that the Service should listen on and forward traffic to.

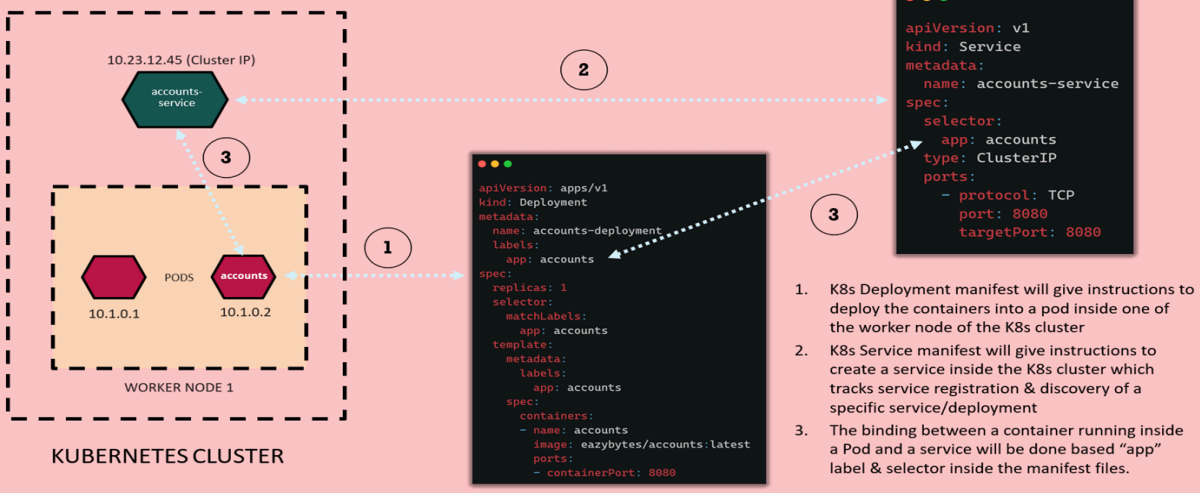
protocol: The protocol field specifies the protocol used for the service port. In this case, it's TCP.

port: The port field is the port number on which the Service will listen for incoming traffic.

targetPort: The targetPort field is the port number on the pods to which the incoming traffic will be forwarded.

How K8s deployment & services tied together

bytes



1. K8s Deployment manifest will give instructions to deploy the containers into a pod inside one of the worker node of the K8s cluster
2. K8s Service manifest will give instructions to create a service inside the K8s cluster which tracks service registration & discovery of a specific service/deployment
3. The binding between a container running inside a Pod and a service will be done based “app” label & selector inside the manifest files.

Self-Healing in Kubernetes

Kubernetes has **self-healing capability**, meaning it automatically monitors and recovers failed or unhealthy application components **without manual intervention**.

Kubernetes self-heals by continuously monitoring resources and automatically restoring the system back to the desired state when failures occur — no manual restart needed.

What Kubernetes Automatically Does

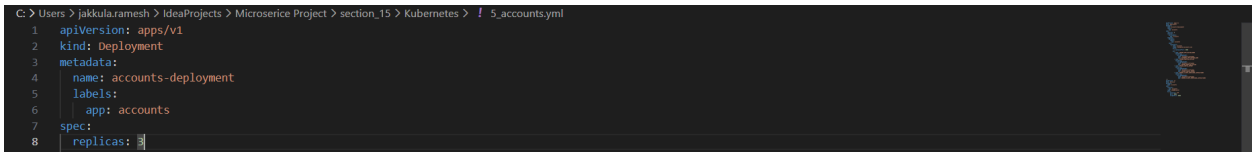
Issue Detected	Kubernetes Action
Pod crashes or stops responding	Restarts the container
Node becomes unavailable	Reschedules Pods on a different node
Pod is deleted accidentally	Creates a new Pod from ReplicaSet
Health check fails	Recreates or replaces unhealthy Pod
Desired replicas ≠ running replicas	Auto-creates missing Pods

Because Kubernetes uses the **desired state vs actual state** mechanism.

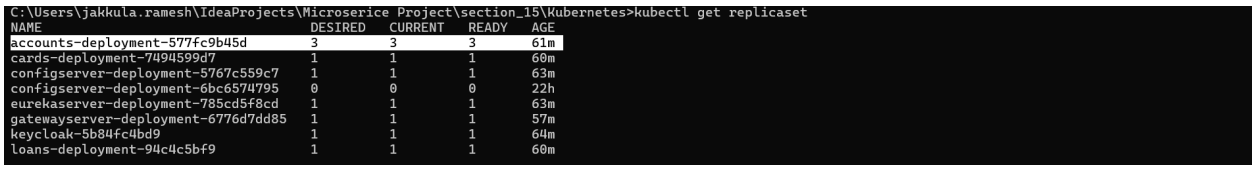
Ex: if we declare , we mentioned replicas: 3 for accounts service.

```
replicas: 3
```

This is the **desired state**.



We can see below desired state 3 and running ones are 3.



We can see below the running 3 pods for account service.

```
C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl get pods
NAME                                  READY   STATUS    RESTARTS   AGE
accounts-deployment-577fc9b45d-4ggt7  1/1     Running   0          2m46s
accounts-deployment-577fc9b45d-8fvhv  1/1     Running   0          62m
accounts-deployment-577fc9b45d-bl62r  1/1     Running   0          2m46s
cards-deployment-7494599d7-lgsjg      1/1     Running   0          62m
configserver-deployment-5767c559c7-h2hms  1/1  Running  0         65m
eurekaserver-deployment-785cd5f8cd-hh2q4  1/1  Running  0         64m
gatewayserver-deployment-6776d7dd85-qsm2p  1/1  Running  0        58m
keycloak-5b84fc4bd9-sdcxm             1/1     Running   0          65m
loans-deployment-94c4c5bf9-bprjt      1/1     Running   0          62m
```

Now , I will delete one pod and see how it self heals by Kubernetes automatically .

```
C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl delete pod accounts-deployment-577fc9b45d-bl62r
pod "accounts-deployment-577fc9b45d-bl62r" deleted from default namespace
```

If only 2 pods are running, Kubernetes detects mismatch and automatically starts another one to maintain 3. We can 3 even though we delete one pod means it self healed and created according to desired state value .

View events to confirm self-healing

kubectl get events --sort-by=.metadata.creationTimestamp

```
8m49s       Normal    Killing                 pod/accounts-deployment-577fc9b45d-bl62r        Stopping container accounts
8m49s       Normal    Scheduled               pod/accounts-deployment-577fc9b45d-mlbll        Successfully assigned default/accounts-deployment-577f
c9b45d-mlbll to docker-desktop
8m49s       Normal    SuccessfulCreate        replicaset/accounts-deployment-577fc9b45d       Created pod: accounts-deployment-577fc9b45d-mlbll
8m48s       Normal    Pulled                  pod/accounts-deployment-577fc9b45d-mlbll        Container image "eazybytes/accounts:s12" already prese
nt on machine
8m48s       Normal    Created                 pod/accounts-deployment-577fc9b45d-mlbll        Created container: accounts
8m48s       Normal    Started                 pod/accounts-deployment-577fc9b45d-mlbll        Started container accounts
```

```
C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl get replicaset
NAME                                  DESIRED   CURRENT   READY   AGE
accounts-deployment-577fc9b45d        3         3         3       68m
cards-deployment-7494599d7            1         1         1       67m
configserver-deployment-5767c559c7    1         1         1       70m
configserver-deployment-6bc6574795    0         0         0       22h
eurekaserver-deployment-785cd5f8cd    1         1         1       70m
gatewayserver-deployment-6776d7dd85   1         1         1       64m
keycloak-5b84fc4bd9                   1         1         1       71m
loans-deployment-94c4c5bf9            1         1         1       68m
```

```
C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl get pods
NAME                                  READY   STATUS    RESTARTS   AGE
accounts-deployment-577fc9b45d-4ggt7  1/1     Running   0          8m46s
accounts-deployment-577fc9b45d-8fvhv  1/1     Running   0          68m
accounts-deployment-577fc9b45d-mlbll  1/1     Running   0          20s
cards-deployment-7494599d7-lgsjg      1/1     Running   0          68m
configserver-deployment-5767c559c7-h2hms  1/1  Running  0         71m
eurekaserver-deployment-785cd5f8cd-hh2q4  1/1  Running  0         70m
gatewayserver-deployment-6776d7dd85-qsm2p  1/1  Running  0        64m
keycloak-5b84fc4bd9-sdcxm             1/1     Running   0          71m
loans-deployment-94c4c5bf9-bprjt      1/1     Running   0          68m
```

Automatic Rollbacks & Rollouts in Kubernetes

A **rollout** is the process of updating an application to a new version without downtime.
Kubernetes replaces old pods gradually with new ones using **Rolling Updates**.

You trigger a rollout when you:

- update container image tag
- change environment variables
- modify deployment YAML

- scale replicas

Example : update container image tag

To view detailed information about a specific Pod, we use:

kubectl describe pod <pod-name>

```
C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl get pods
NAME                                     READY   STATUS    RESTARTS   AGE
accounts-deployment-577fc9b45d-8fvhv     1/1     Running   0          132m
cards-deployment-7494599d7-lgsjg         1/1     Running   0          131m
configserver-deployment-6bc6574795-r8jl4 1/1     Running   0          101s
eurekaserver-deployment-785cd5f8cd-hh2q4 1/1     Running   0          134m
gatewayserver-deployment-6776d7dd85-qsm2p 1/1     Running   0          128m
keycloak-5b84fc4bd9-sdcxm                1/1     Running   0          135m
loans-deployment-94c4c5bf9-bprjt         1/1     Running   0          132m

C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl describe pod gatewayserver-deployment-6776d7dd85-qsm2p
Name:             gatewayserver-deployment-6776d7dd85-qsm2p
Namespace:        default
Priority:         0
Service Account:  default
Node:             docker-desktop/192.168.65.3
Start Time:       Sat, 06 Dec 2025 13:01:31 +0530
Labels:           app=gatewayserver
                  pod-template-hash=6776d7dd85
Annotations:      <none>
Status:           Running
IP:               10.1.0.19
IPs:
  IP:           10.1.0.19
Controlled By:  ReplicaSet/gatewayserver-deployment-6776d7dd85
Containers:
  gatewayserver:
    Container ID:   docker://2cb8a422e6f955a293f97d808a34c672adb0e6eff2b1903f9835ffcfa173295a
    Image:          eazybytes/gatewayserver:s12
    Image ID:       docker-pullable://eazybytes/gatewayserver@sha256:819584c0b8c0088758468c4aa128927ffb3850df971c84646fe9adc81af390e6
    Port:           8072/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Sat, 06 Dec 2025 13:02:08 +0530
```

Updating tag name from s12 to s11
update deployment using:

kubectl set image deployment/<deployment-name> <container-name>=<image-name>:<tag>

kubectl set image - Command used to update container images

deployment/<deployment-name> - Name of the Deployment you are modifying

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: gatewayserver-deployment
```

<container-name> - The container inside the pod whose image you want to change

```
  spec:
    containers:
    - name: gatewayserver
      image: eazybytes/gatewayserver:s12
```

<image-name>:<tag> - New Docker image & version tag to apply

```
C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl set image deployment/gatewayserver-deployment gatewayserver=eazybytes
/gatewayserver:s11
deployment.apps/gatewayserver-deployment image updated
```

Its updated

```
C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl describe pod gatewayserver-deployment-66c868cbdd-cz9kt
Name:             gatewayserver-deployment-66c868cbdd-cz9kt
Namespace:        default
Priority:         0
Service Account:  default
Node:             docker-desktop/192.168.65.3
Start Time:       Sat, 06 Dec 2025 15:12:39 +0530
Labels:           app=gatewayserver
                  pod-template-hash=66c868cbdd
Annotations:      <none>
Status:           Running
IP:               10.1.0.25
IPs:
  IP:           10.1.0.25
Controlled By:  ReplicaSet/gatewayserver-deployment-66c868cbdd
Containers:
  gatewayserver:
    Container ID:   docker://41dcaf0298bbc82f7fa949316c2a3afabac4633eb6b39f7139685f96c51a263a
    Image:          eazybytes/gatewayserver:s11
    Image ID:       docker-pullable://eazybytes/gatewayserver@sha256:8009fadd63368dc12ce072f6d60ca82fa6ec6d87c6e2963ff0be537fa25dfc56
```

Rollback

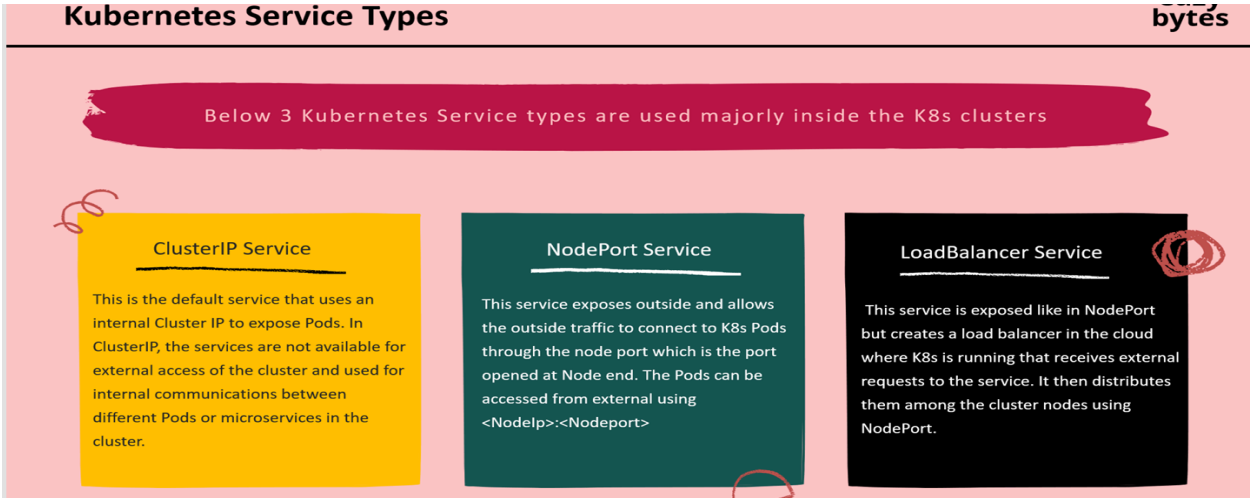
If the new version of your application is **unhealthy**, **fails readiness/liveness checks**, or crashes.

Kubernetes can **roll back to the previous working version automatically**.

```
C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl rollout history deployment gatewayserver-deployment
deployment.apps/gatewayserver-deployment
REVISION  CHANGE-CAUSE
1         <none>
2         <none>


C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl rollout undo deployment/gatewayserver-deployment --to-revision=1
deployment.apps/gatewayserver-deployment rolled back
```

```
C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl get pods
NAME                                     READY   STATUS    RESTARTS   AGE
accounts-deployment-577fc9b45d-8fvhv     1/1     Running   0          143m
cards-deployment-7494599d7-lgsjg         1/1     Running   0          143m
configserver-deployment-6bc6574795-r8jl4   1/1     Running   0          12m
eurekaserver-deployment-785cd5f8cd-hh2q4   1/1     Running   0          145m
gatewayserver-deployment-6776d7dd85-mzg2z   1/1     Running   0          50s
keycloak-5b84fc4bd9-sdcxm                1/1     Running   0          146m
loans-deployment-94c4c5bf9-bprjt         1/1     Running   0          143m

C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl describe pod gatewayserver-deployment-6776d7dd85-mzg2z
Name:             gatewayserver-deployment-6776d7dd85-mzg2z
Namespace:        default
Priority:         0
Service Account:  default
Node:             docker-desktop/192.168.65.3
Start Time:       Sat, 06 Dec 2025 15:20:07 +0530
Labels:           app=gatewayserver
                  pod-template-hash=6776d7dd85
Annotations:      <none>
Status:           Running
IP:               10.1.0.26
IPs:
  IP:           10.1.0.26
Controlled By:  ReplicaSet/gatewayserver-deployment-6776d7dd85
Containers:
  gatewayserver:
    Container ID:   docker://dc531aa352866e147cf9e6d4034ada34fba955e46f24f9c55a8fb88df32e2be3
    Image:          eazybytes/gatewayserver:s12
```

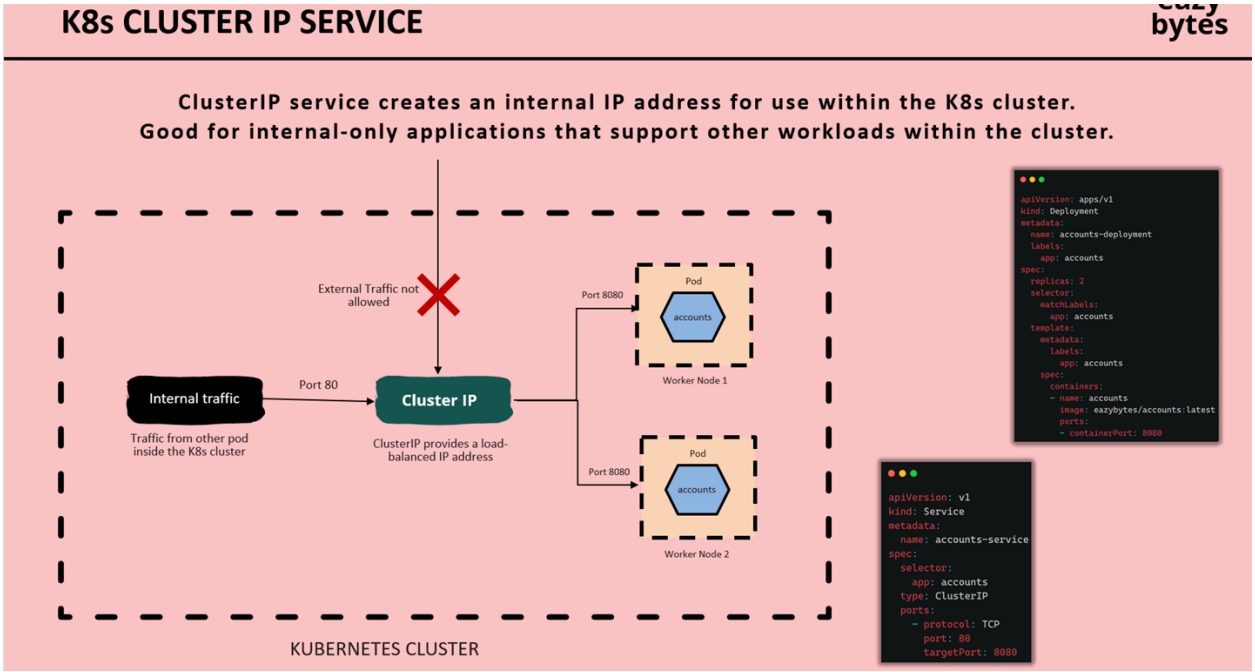
Kubernetes Service Types



1 ClusterIP (Default)

- Accessible only inside cluster
- Suitable for internal communication between microservices
- No external access

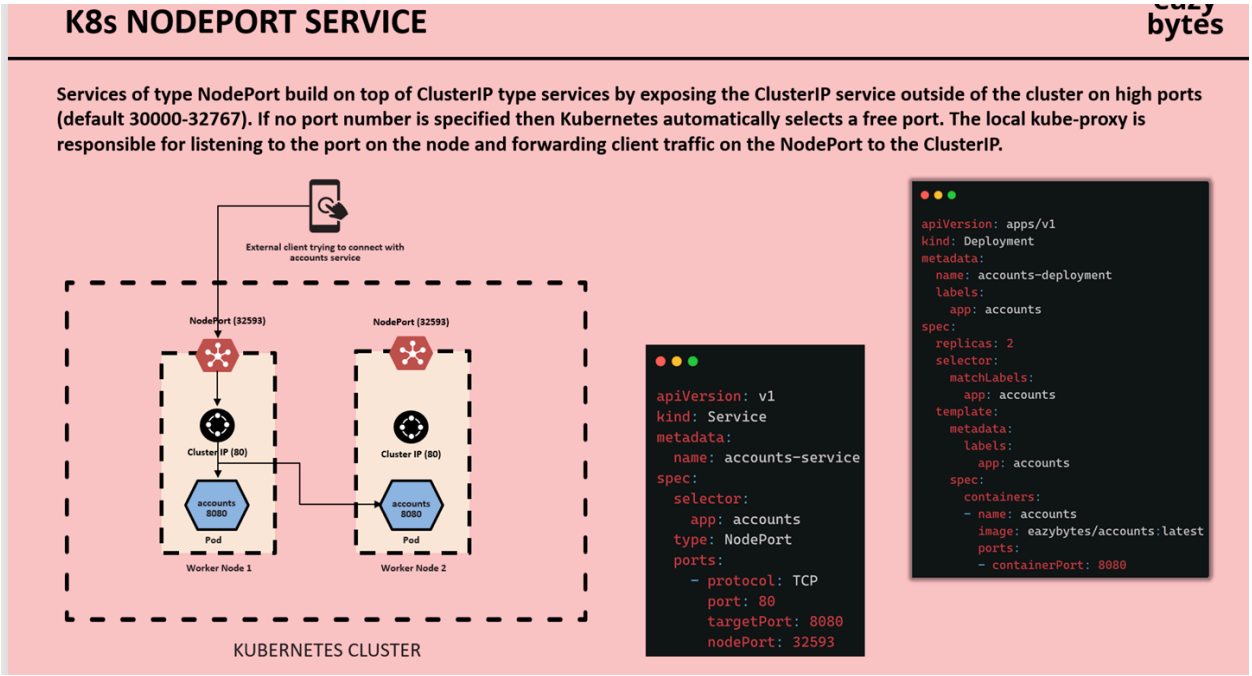
type: ClusterIP



2 NodePort

- Exposes service on a **port of every worker node**
- You access it using:
  NodeIP:NodePort
- Port range: **30000–32767**

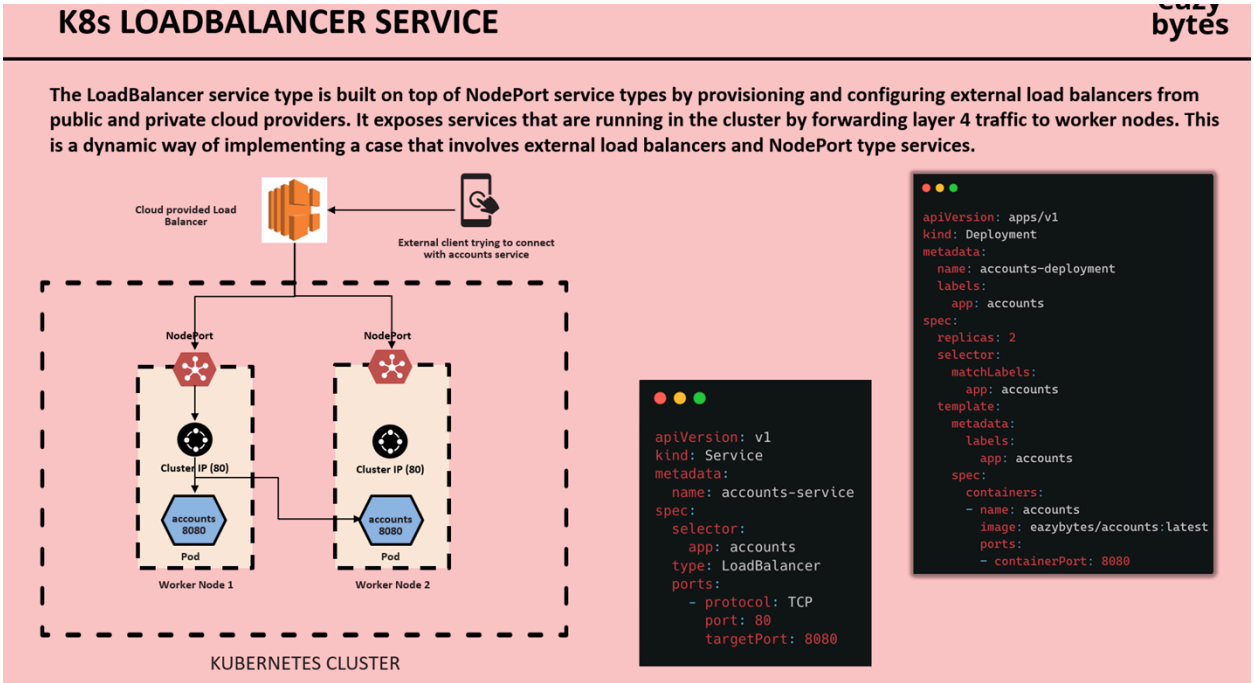
type: NodePort



3 LoadBalancer

- Exposes the service to **external world**
- Cloud provider creates a public IP + Load Balancer
- Internally uses NodePort + ClusterIP

Use Case: Production, publicly accessible services(APIs, UI applications, Gateway)



```
C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl get services
NAME           TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
accounts       LoadBalancer   10.98.178.91     localhost     8080:30101/TCP   179m
cards          LoadBalancer   10.111.45.202    localhost     9000:31082/TCP   179m
configserver   LoadBalancer   10.106.74.249    localhost     8071:31890/TCP   24h
eurekaserver   LoadBalancer   10.108.87.6      localhost     8070:31637/TCP   3h1m
gatewayserver  LoadBalancer   10.107.43.126    localhost     8072:31252/TCP   175m
keycloak       LoadBalancer   10.97.168.22     localhost     7080:31654/TCP   3h2m
kubernetes     ClusterIP      10.96.0.1        <none>        443/TCP          28h
loans          LoadBalancer   10.105.175.174   localhost     8090:30592/TCP   179m
```

TYPE = **LoadBalancer** : You are exposing the service externally (outside cluster).

EXTERNAL-IP = localhost Because you are running locally (Minikube, Kind, Docker Desktop), it maps externally to localhost instead of public cloud IP if we use any cloud services.

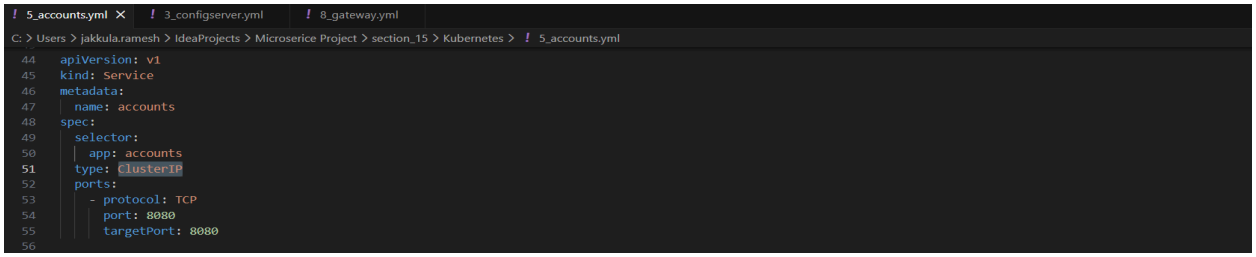
8080:30101/TCP format , 8080 → externally accessible port (LoadBalancer)

30101 → underlying NodePort used internally.

We can see it allowing external clients to use apis.

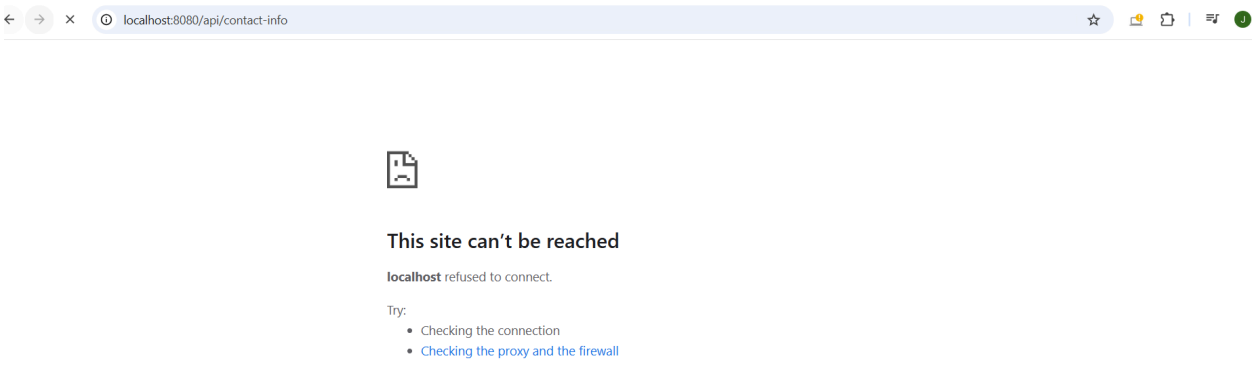
ClusterIP Example

We can see Cluster examples.

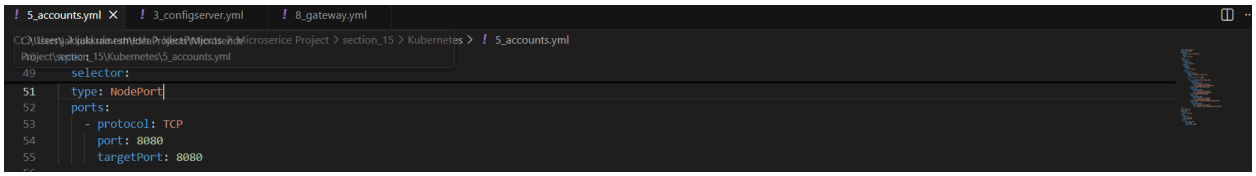


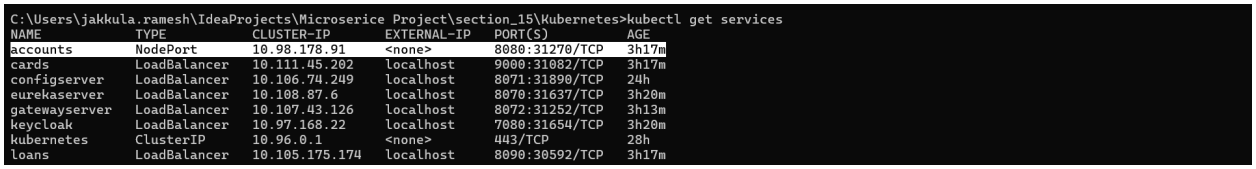

```
C:\Users\jakkula.ramesh\IdeaProjects\Microserice Project\section_15\Kubernetes>kubectl get services
NAME           TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
accounts       ClusterIP      10.98.178.91     <none>        8080/TCP         3h8m
cards          LoadBalancer   10.111.45.202    localhost     9000:31082/TCP   3h8m
configserver   LoadBalancer   10.106.74.249    localhost     8071:31890/TCP   24h
eurekaserver   LoadBalancer   10.108.87.6      localhost     8070:31637/TCP   3h11m
gatewayserver  LoadBalancer   10.107.43.126    localhost     8072:31252/TCP   3h4m
keycloak       LoadBalancer   10.97.168.22     localhost     7080:31654/TCP   3h11m
kubernetes     ClusterIP      10.96.0.1        <none>        443/TCP          28h
loans          LoadBalancer   10.105.175.174   localhost     8090:30592/TCP   3h8m
```

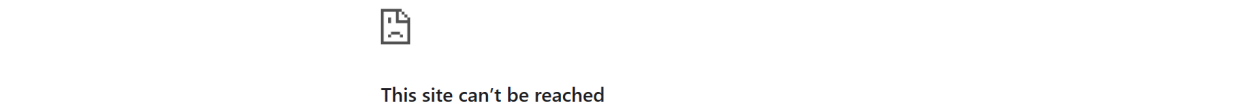
see for ClusterIP External is none. We can see it is not allowing any external api calls.



NodePort Example







When we tried internal ip menas with nodeport ip

We should use LoadBalancer only when we need the service to be accessible externally.

For internal service-to-service communication, **ClusterIP is preferred** to keep traffic internal and secure, and avoid exposing services publicly.

This is better than using NodePort, since NodePort exposes each service on a high port of every node and is less secure and scalable.

❌ Problems With Manually Creating Manifest Files

- Creating **individual manifest (.yaml) files for each microservice** is difficult when the application grows.
- Deploying each YAML using kubectl apply -f <file> becomes slow and time-consuming.
- For every environment (DEV, QA, PROD), we must **edit values in all files**, which is repetitive and error-prone.
- Deleting or updating multiple YAML files also takes more manual effort.
- Managing large-scale microservices becomes complex and inefficient.

✔ Solution → Helm

Helm is a **package manager for Kubernetes**, similar to how Maven manages Java dependencies or apt/yum manage OS packages.

Helm