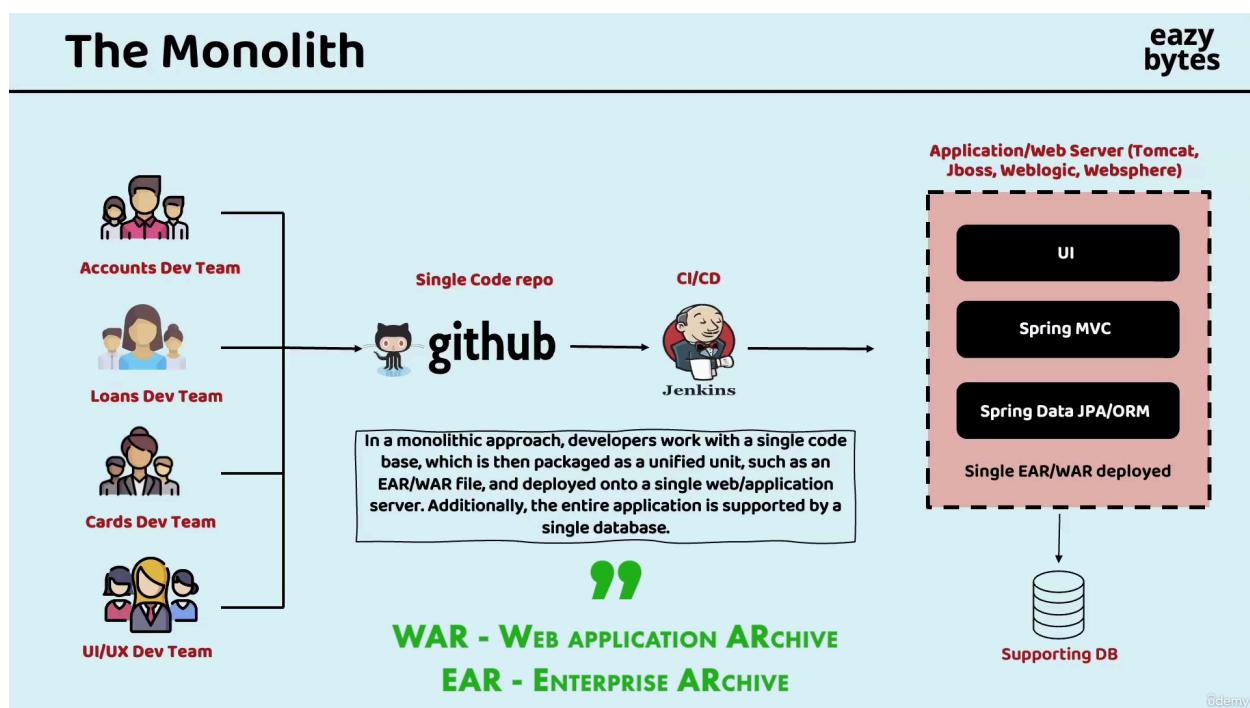


Develop Microservices with Java, Spring Boot, Spring Cloud, Docker, Kubernetes, Helm, Microservices Security



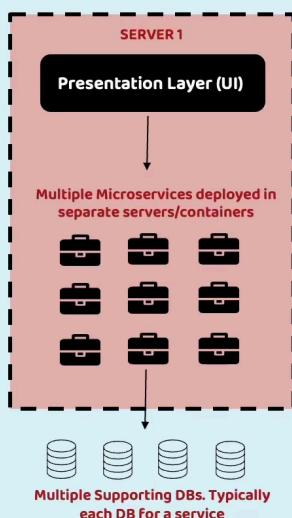
The Monolith

eazy bytes



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

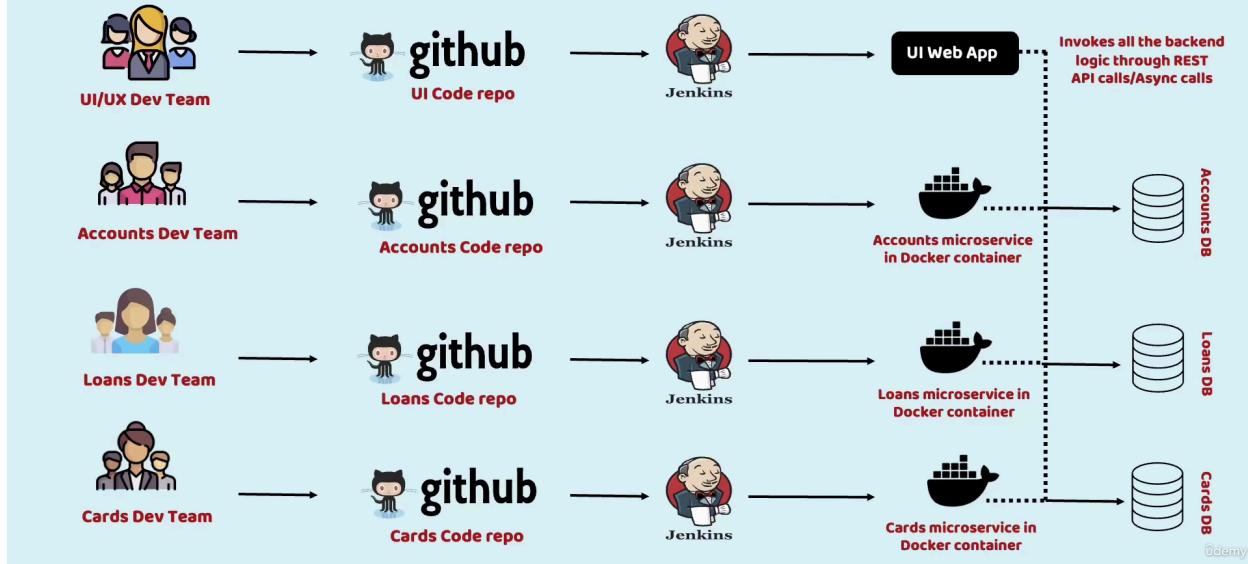
Cons

- Complexity
- Infrastructure overhead
- Security concerns

Odemy

The GREAT MICROSERVICES

eazy
bytes



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.

```

@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.

Accounts microservice REST API Documentation v1 OAS 3.1

/v3/api-docs

EazyBank Accounts microservice REST API Documentation

Madan Reddy - Website
Send email to Madan Reddy
Apache 2.0
EazyBank Accounts microservice REST API Documentation

@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;
}
```

Schemas

```
Accounts ^ Collapse all object
Schema to hold Account information

accountNumber* ^ Collapse all integer int64
Account Number of Eazy Bank account
Example 3454433243

accountType* ^ Collapse all string
Account type of Eazy Bank account
Example "Savings"

branchAddress* ^ Collapse all string
Eazy Bank branch address
Example 123
```

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 {
```

CRUD REST APIs for Accounts in EazyBank

CRUD REST APIs in EazyBank to CREATE, UPDATE, FETCH AND DELETE account details

PUT /api/update Update Account Details REST API

POST /api/create Create Account REST API

GET /api/fetch Fetch Account Details REST API

DELETE /api/delete Delete Account & Customer Details REST API



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:

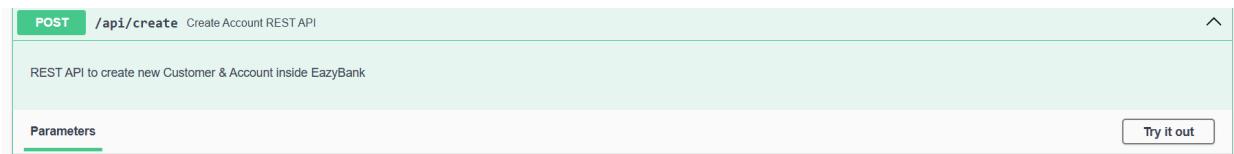
Describes what a specific endpoint does — its summary and details.

```
@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));
    }

```

Responses

Code	Description	Links
201	HTTP Status CREATED Media type <code>application/json</code> Controls Accept header. Example Value Schema <pre>{ "statusCode": "string", "statusMsg": "string" }</pre>	No links
500	HTTP Status Internal Server Error Media type <code>application/json</code> Example Value Schema <pre>{ "apiPath": "string", "errorCode": "100_CONTINUE", "errorMessage": "string", "errorTime": "2025-11-13T07:27:53.181Z" }</pre>	No links

💡 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

```

@Validated
public class AccountsController {

    private IAccountsService iAccountsService;

}

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

B. Numeric Validations

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

C. String & Pattern Validations

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

D. Date Validations

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

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

```

@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

```

@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

```

@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);
}

```

```

1 package com.eazybytes.accounts.exception;
2
3 import org.springframework.http.HttpStatus;[]
4
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

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

```

1 package com.eazybytes.accounts.constants;
2
3 public final class AccountsConstants {
4
5     private AccountsConstants() {
6         // restrict instantiation
7     }
8
9     public static final String SAVINGS = "Savings";
10    public static final String ADDRESS = "123 Main Street, New York";
11    public static final String STATUS_201 = "201";
12    public static final String MESSAGE_201 = "Account created successfully";
13    public static final String STATUS_200 = "200";
14    public static final String MESSAGE_200 = "Request processed successfully";
15    public static final String STATUS_417 = "417";
16    public static final String MESSAGE_417_UPDATE= "Update operation failed. Please try again or contact Dev team";
17    public static final String MESSAGE_417_DELETE= "Delete operation failed. Please try again or contact Dev team";
18    // public static final String STATUS_500 = "500";
19    // public static final String MESSAGE_500 = "An error occurred. Please try again or contact Dev team";
20
21 }
22

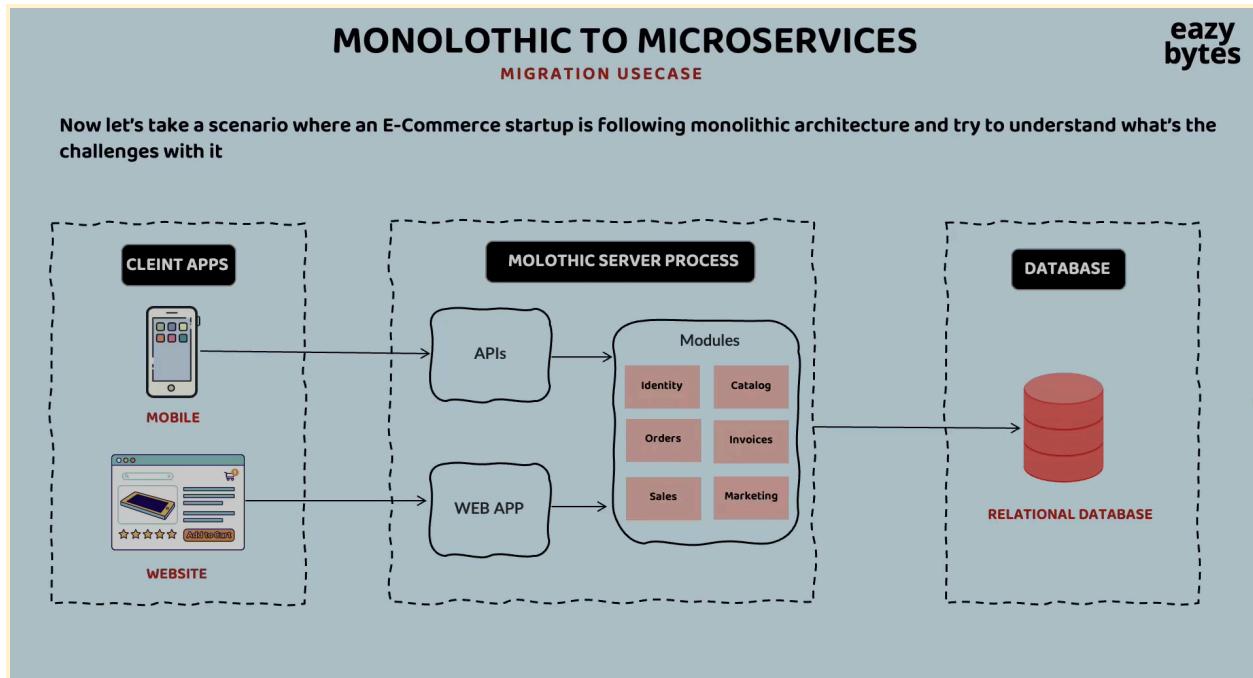
```

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)

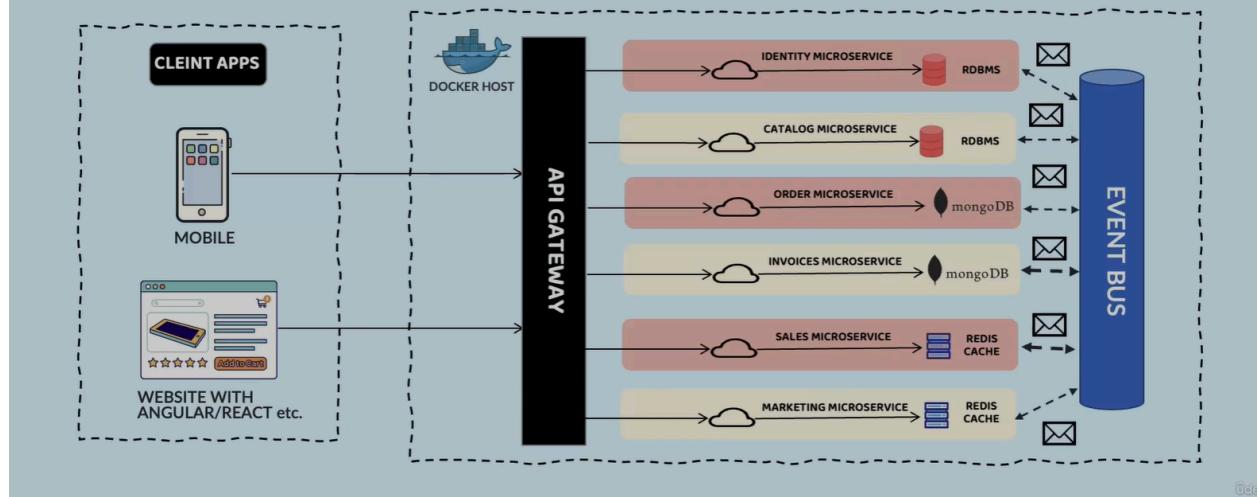
Once the system becomes large:

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)

So the Ecommerce company decided and adopted the below cloud-native design by leveraging Microservices architecture to make their life easy and less risk with the continuous changes.



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

eazy
bytes

The Strangler Fig Pattern is a software migration pattern used to gradually replace or refactor a legacy system with a new system, piece by piece, without disrupting the existing functionality. This pattern gets its name from the way a strangler fig plant grows around an existing tree, slowly replacing it until the original tree is no longer needed.

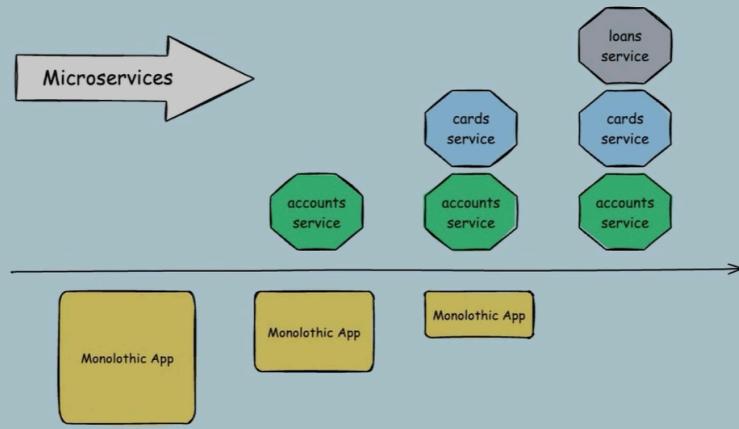
When to Use the Strangler Fig Pattern:

- When you need to modernize a large or complex legacy system.
- When you want to avoid the risk associated with a complete system rewrite or "big bang" migration.
- When the legacy system needs to remain operational during the transition to the new system.



The Strangler Fig Pattern facilitates the migration of a monolithic application to a modern microservices architecture by leveraging a Domain-Driven Design (DDD) approach.

The legacy monolith is carefully analyzed, broken down into distinct domains, and services are gradually rewritten using newer technologies. This incremental transformation ensures that each service is refactored independently, allowing for a smooth transition from the monolith to a fully microservices-based architecture while maintaining system functionality throughout the process.



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

DEPLOYMENT, PORTABILITY & SCALABILITY OF MICROSERVICES

eazy bytes

CHALLENGE 3

DEPLOYMENT

How do we deploy all the tiny 100s of microservices with less effort & cost?

PORTABILITY

How do we move our 100s of microservices across environments with less effort, configurations & cost?

SCALABILITY

How do we scale our applications based on the demand on the fly with minimum effort & cost?

To overcome the above challenges, we should **containerize** our microservices. Why? Containers offer a self-contained and isolated environment for applications, including all necessary dependencies. By containerizing an application, it becomes portable and can run seamlessly in any cloud environment. Containers enable unified management of applications regardless of the language or framework used.

Docker is an open source platform that "provides the ability to package and run an application in a loosely isolated environment called a container"

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:

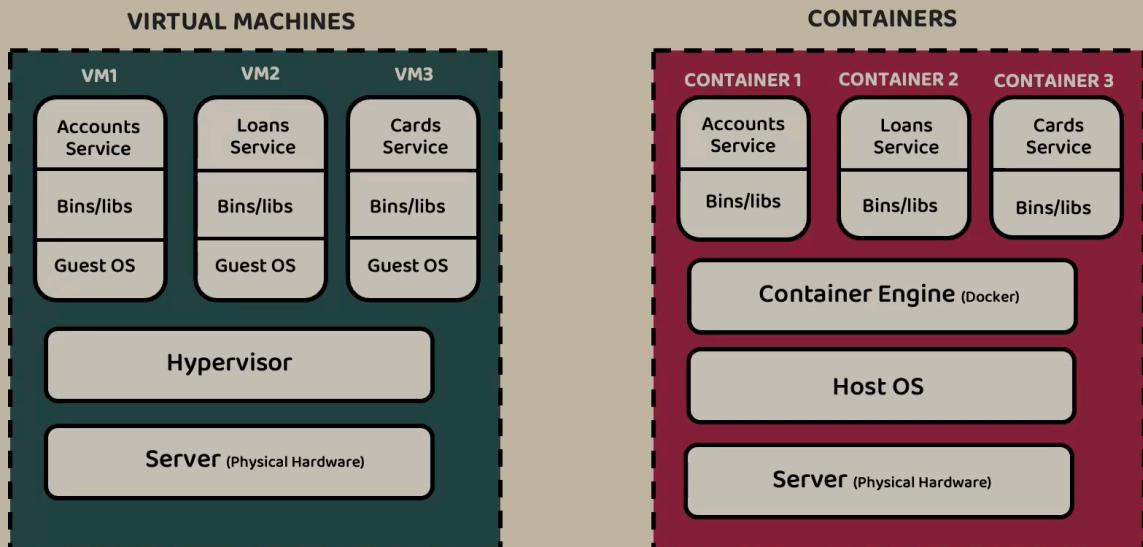
- Install JDK manually on the server
- Install Maven
- Install dependent libraries
- Configure environment variables
- Run JAR manually

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.

WHAT ARE CONTAINERS & HOW THEY ARE DIFFERENT FROM VMs ?



Main differences between virtual machines and containers. Containers don't need the Guest Os nor the hypervisor to assign resources; instead, they use the container engine.

WHAT ARE CONTAINERS & Docker ?

What is software containerization ?

Software containerization is an OS virtualization method that is used to deploy and run containers without using a virtual machine (VM). Containers can run on physical hardware, in the cloud, VMs, and across multiple OSs.

What is a container ?

A container is a loosely isolated environment that allows us to build and run software packages. These software packages include the code and all dependencies to run applications quickly and reliably on any computing environment. We call these packages as container images.

What is Docker ?

Docker is an open-source platform that enables developers to automate the deployment, scaling, and management of applications using containerization. Containers are lightweight, isolated environments that encapsulate an application along with its dependencies, libraries, and runtime components.

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

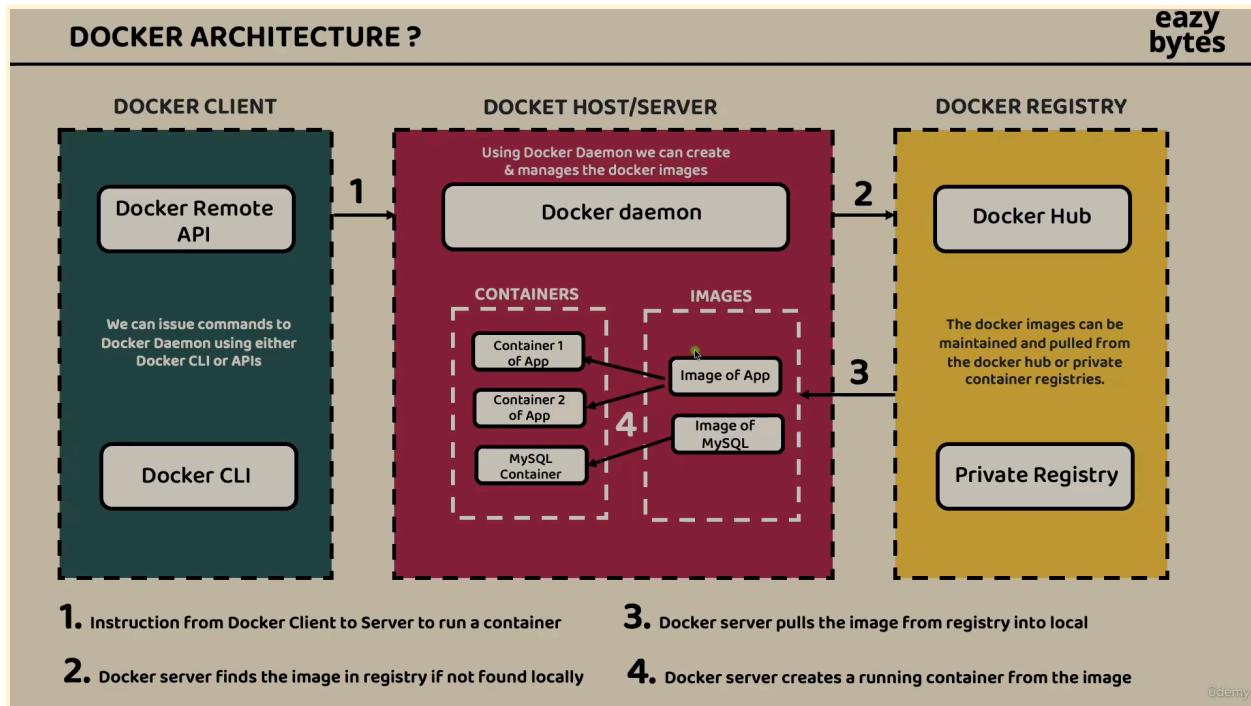
Works with Kubernetes for orchestration

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

GENERATE DOCKER IMAGES

To generate docker images from our existing microservices, we will explore the below three different commonly used approaches. We can choose one of them for the rest of the course

01 **Dockerfile -> accounts**
We need to write a dockerfile with the list of instructions which can be passed to Docker server to generate a docker image based on the given instructions

02 **Buildpacks -> loans**
Buildpacks (<https://buildpacks.io>), a project initiated by Heroku & Pivotal and now hosted by the CNCF. It simplifies containerization since with it, we don't need to write a low-level dockerfile.

03 **Google Jib -> cards**
Jib is an open-source Java tool maintained by Google for building Docker images of Java applications. It simplifies containerization since with it, we don't need to write a low-level dockerfile.



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="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 />
<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 ---
[INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/surefire/surefire-api/3.5.2/surefire-api-3.5.2.j
[INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/surefire/surefire-common/3.5.2/maven-suref
[INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/surefire/surefire-shared-utils/2.5.2/surefire-sha
```

```
[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 ---
[INFO] 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>
```

Name	Date modified	Type	Size
classes	11/15/2025 3:51 PM	File folder	
generated-sources	11/15/2025 3:51 PM	File folder	
generated-test-sources	11/15/2025 3:51 PM	File folder	
maven-archiver	11/15/2025 3:51 PM	File folder	
maven-status	11/15/2025 3:51 PM	File folder	
surefire-reports	11/15/2025 3:51 PM	File folder	
test-classes	11/15/2025 3:51 PM	File folder	
accounts-0.0.1-SNAPSHOT.jar	11/15/2025 3:51 PM	Executable Jar File	62,414 KB
accounts-0.0.1-SNAPSHOT.jar.original	11/15/2025 3:51 PM	ORIGINAL File	29 KB

4. We can run spring boot 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. Therefore, 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 available at 'jdbc:h2:mem:testdb'
2025-11-15T16:00:19.009+05:30  INFO 24252 --- [ restartedMain] o.s.b.d.a.OptionalReloadServer          : 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.w.e.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
2025-11-15T16:00:19.179+05:30  INFO 24252 --- [ restartedMain] c.e.accounts.AccountsApplication        : Started AccountsApplication in 10.129 seconds (process 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:

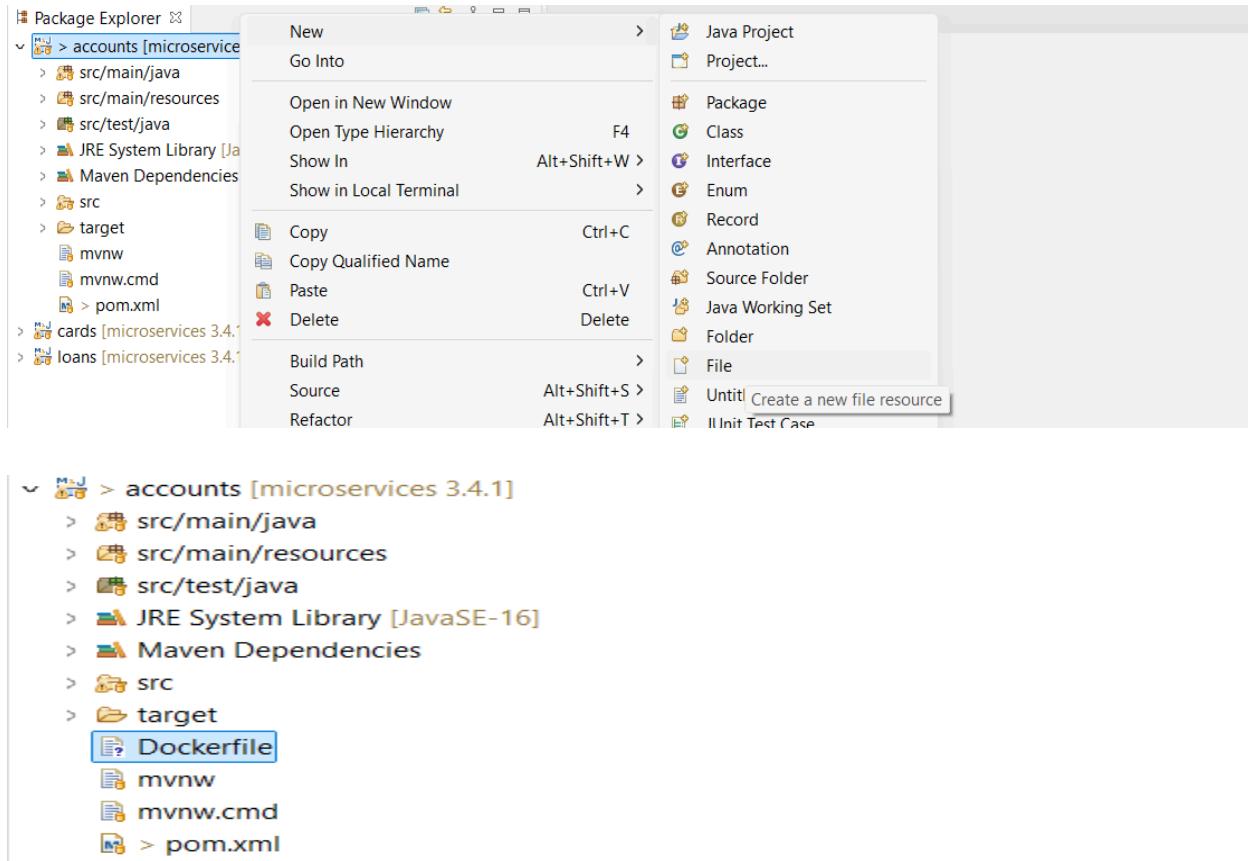
```
java -jar target/accounts-service-0.0.1-SNAPSHOT.jar
```

iii) In any IDE directly.

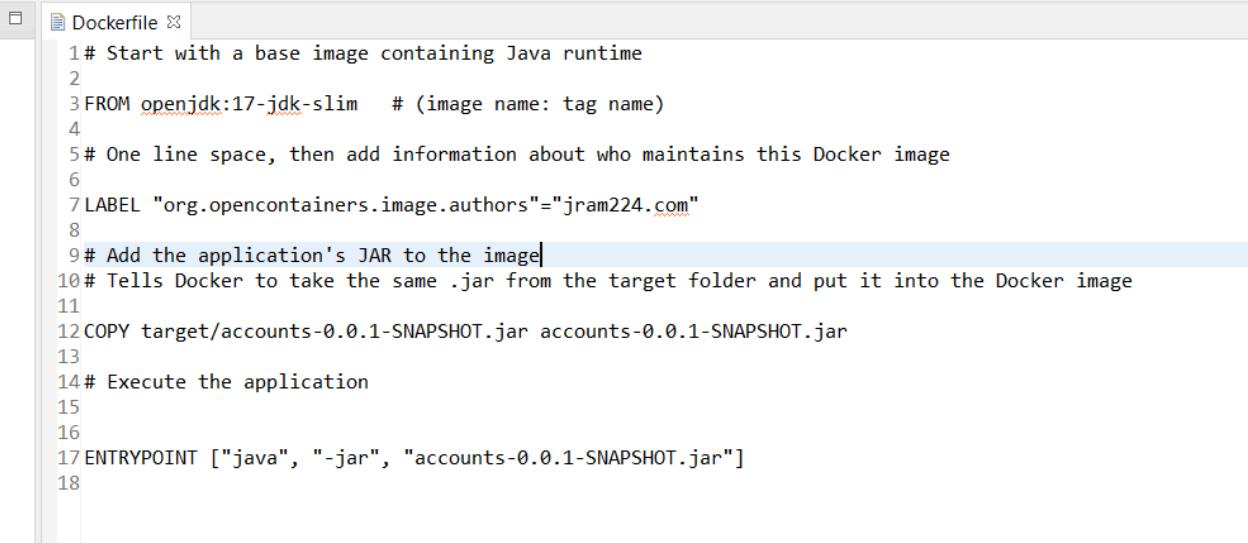
1) using Dockerfile

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.



```
1# Start with a base image containing Java runtime
2
3FROM openjdk:17-jdk-slim    # (image name: tag name)
4
5# One line space, then add information about who maintains this Docker image
6
7LABEL "org.opencontainers.image.authors"="jram224.com"
8
9# Add the application's JAR to the image
10# Tells Docker to take the same .jar from the target folder and put it into the Docker image
11
12COPY target/accounts-0.0.1-SNAPSHOT.jar accounts-0.0.1-SNAPSHOT.jar
13
14# Execute the application
15
16
17ENTRYPOINT ["java", "-jar", "accounts-0.0.1-SNAPSHOT.jar"]
18
```

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`.

- 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 .
```

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

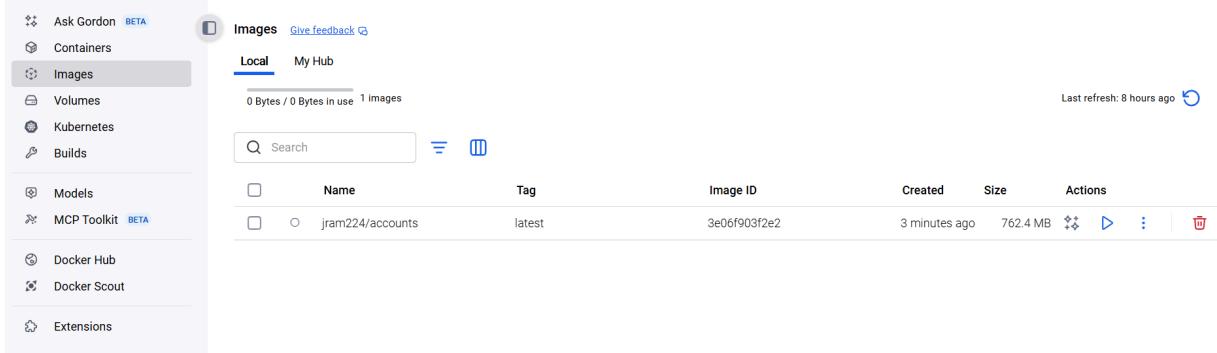
```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>docker build . -t jram224/accounts:latest
[+] Building 54.1s (7/7) FINISHED
--> [internal] load build definition from Dockerfile
--> => transferring dockerfile: 58B
--> [internal] load metadata for docker.io/library/openjdk:17.0.1-jdk-slim
--> [internal] load .dockerrcignore
--> => transferring context: 2B
--> [internal] load build context
--> => transferring context: 63.93MB
--> [1/2] FROM docker.io/library/openjdk:17.0.1-jdk-slim@sha256:fc5fa503124ba7021bbf8cb3718bf08791590d0aa2295c7cc551de65f9919290
--> => resolve docker.io/library/openjdk:17.0.1-jdk-slim@sha256:fc5fa503124ba7021bbf8cb3718bf08791590d0aa2295c7cc551de65f9919290
--> sha256:a2abf6c4d29d43a4bfb9f9fb769f524d0fb36a2edab49819c1bf3e76f40bf953ea 31.36MB / 31.36MB
--> sha256:2bbde5250315969db657b55bd8b2f5507fb659c0cf7f135edc84b684ffebab44a 1.58MB / 1.58MB
--> sha256:1343f138b677c0b1457cc7cb6310108df5388665281e0962273fb3492e52b86d 187.55MB / 187.55MB
--> => extracting sha256:a2abf6c4d29d43a4bfb9f9fb769f524d0fb36a2edab49819c1bf3e76f40bf953ea
--> => extracting sha256:2bbde5250315969db657b55bd8b2f5507fb659c0cf7f135edc84b684ffebab44a
--> => extracting sha256:1343f138b677c0b1457cc7cb6310108df5388665281e0962273fb3492e52b86d
--> [2/2] COPY target/accounts-0.0.1-SNAPSHOT.jar accounts-0.0.1-SNAPSHOT.jar
--> exporting to image
--> => exporting layers
--> => exporting manifest sha256:6e899f478c7381f3ca7ba73df37eac22f1476070db3c2f9738bb6b24c650e6ec
--> => exporting config sha256:b43fe172b9d3aad8104d642e2da94bc8787d310a17131541241eb8808+dfcb5d
--> => exporting attestation manifest sha256:53483db3ee3ff88510eb82eab124c7f9ab981e7b0475d4480a2aaaa9373af496
--> => exporting manifest list sha256:3e06f903f2e2681035ef1c3b308a779af2dee887ea5ab0902005dad0232b6b
--> => naming to docker.io/jram224/accounts:latest
--> => unpacking to docker.io/jram224/accounts:latest
```

- 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.



The screenshot shows the Docker Desktop interface with the 'Images' tab selected. On the left, there's a sidebar with various options like Ask Gordon, Containers, Images (which is highlighted), Volumes, Kubernetes, Builds, Models, MCP Toolkit, Docker Hub, Docker Scout, and Extensions. The main area is titled 'Images' with a 'Local' tab selected and a 'My Hub' tab. It shows '0 Bytes / 0 Bytes in use' and '1 images'. A search bar and filter icons are at the top. Below is a table with columns: Name, Tag, Image ID, Created, Size, and Actions. One row is visible: 'jram224/accounts' with 'latest' tag, Image ID '3e06f903f2e2', created '3 minutes ago', size '762.4 MB', and actions icons.

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:a7da989d53ee25f18b7810206b39580df30518043d1f34f1d514f665ba8026f4"
            ]
        }
    }
]
```

5. Run Docker Container Using the Image

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

```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>docker run -p 8080:8080 jram224/accounts:latest
:: Spring Boot ::          (v3.4.1)

2025-11-15T16:33:13.255Z INFO 1 --- [           main] c.e.accounts.AccountsApplication      : Starting AccountsApplication v0.0.1-SNAPSHOT using Java 17
.0.1 with PID 1 (/accounts-0.0.1-SNAPSHOT.jar started by root in /)
2025-11-15T16:33:13.258Z INFO 1 --- [           main] c.e.accounts.AccountsApplication      : No active profile set, falling back to 1 default profile:
"default"
2025-11-15T16:33:14.565Z INFO 1 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode
2025-11-15T16:33:14.613Z INFO 1 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 39 ms. Found 2
JPA repository interfaces.
2025-11-15T16:33:15.435Z INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2025-11-15T16:33:15.456Z INFO 1 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2025-11-15T16:33:15.457Z INFO 1 --- [           main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.34]
2025-11-15T16:33:15.505Z INFO 1 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[]   : Initializing Spring embedded WebApplicationContext
2025-11-15T16:33:15.506Z INFO 1 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 21
70 ms
2025-11-15T16:33:15.813Z INFO 1 --- [           main] com.zaxxer.hikari.HikariDataSource   : HikariPool-1 - Starting...
2025-11-15T16:33:16.070Z INFO 1 --- [           main] com.zaxxer.hikari.pool.HikariPool    : HikariPool-1 - Added connection conn0: url=jdbc:h2:mem:tes
tDB user:SA
2025-11-15T16:33:16.072Z INFO 1 --- [           main] com.zaxxer.hikari.HikariDataSource   : HikariPool-1 - Start completed.
2025-11-15T16:33:16.219Z INFO 1 --- [           main] o.hibernate.jpa.internal.util.LogHelper : HHH000294: Processing PersistenceUnitInfo [name: default]
2025-11-15T16:33:16.297Z INFO 1 --- [           main] org.hibernate.Version                : HHH000412: Hibernate ORM core version 6.6.4.Final
2025-11-15T16:33:16.345Z INFO 1 --- [           main] o.h.c.internal.RegionFactoryInitiator : HHH000026: Second-level cache disabled
2025-11-15T16:33:16.679Z INFO 1 --- [           main] o.s.o.j.p.SpringPersistenceUnitInfo  : No LoadTimeWeaver setup: ignoring JPA class transformer
2025-11-15T16:33:16.741Z WARN 1 --- [           main] org.hibernate.orm.deprecation       : HHH0000025: H2Dialect does not need to be specified explicitly using 'hibernate.dialect' (remove the property setting and it will be selected by default)
2025-11-15T16:33:16.769Z INFO 1 --- [           main] org.hibernate.orm.connections.Pooling : HHH10001005: Database info:
Database JDBC URL [Connecting through datasource 'HikariDataSource (HikariPool-1)']
Database driver: undefined/unknown
Remote Desktop Connection
```

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

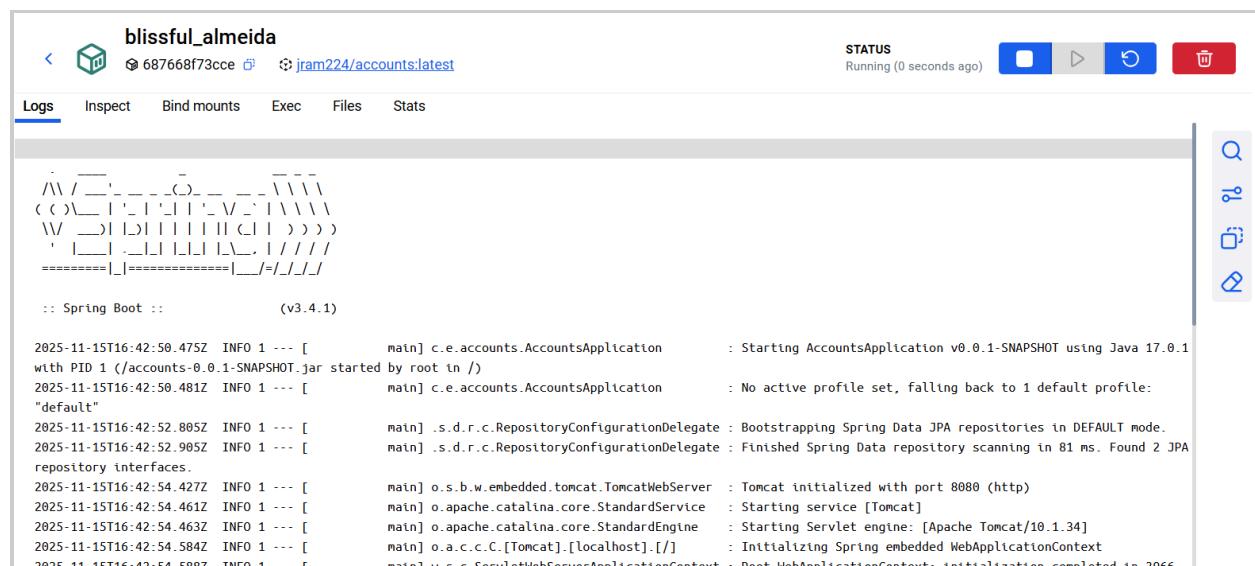
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>
```

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.



CRUD REST APIs for Cards in EazyBank CRUD REST APIs in EazyBank to CREATE, UPDATE, FETCH AND DELETE card details

PUT	/api/update	Update Card Details REST API
POST	/api/create	Create Card REST API
GET	/api/fetch	Fetch Card Details REST API
DELETE	/api/delete	Delete Card Details REST API

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
9a3c912e1692        jram224/accounts:latest   "java -jar accounts..."   10 minutes ago    Exited (143) 6 minutes ago
bdebfa29690f        jram224/accounts:latest   "java -jar accounts..."   15 minutes ago    Exited (130) 11 minutes ago

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

PORT MAPPING IN DOCKER

eazy bytes

What is port mapping or port forwarding or port publishing ?

By default, containers are connected to an isolated network within the Docker host. To access a container from your local network, you need to configure port mapping explicitly. For instance, when running the accounts Service application, we can provide the port mapping as an argument in the docker run command: `-p 8081:8080` (where the first value represents the external port and the second value represents the container port). Below diagram demonstrates the functionality of this configuration.

The diagram shows a user on the Local Network (yellow box) invoking `http://localhost:8081` to access Accounts related APIs. This request is mapped via port 8081 to the Docker Network (dark green box), where an Accounts service container is running at port 8080.

STEPS TO BE FOLLOWED

1) Run the maven command, "mvn clean install" from the location where pom.xml is present to generate a fat jar inside target folder

2) Write instructions to Docker inside a file with the name Dockerfile to generate a Docker image. Sample instructions are mentioned on the left hand side

3) Execute the docker command "docker build . -t eazybytes/accounts:s4" from the location where Dockerfile is present. This will generate the docker image based on the tag name provided

4) Execute the docker command "docker run -p 8080:8080 eazybytes/accounts:s4". This will start the docker container based on the docker image name and port mapping provided

Sample Dockerfile

```
#Start with a base image containing Java runtime
FROM openjdk:17-jdk-slim

#Information around who maintains the image
MAINTAINER eazybytes.com

# Add the application's jar to the container
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"]
```

Odemy

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**

```

79<build>
80  <plugins>
81    <plugin>
82      <groupId>org.springframework.boot</groupId>
83      <artifactId>spring-boot-maven-plugin</artifactId>
84      <configuration>
85        <image>
86          <name>jram224/${project.artifactId}:latest</name>
87        </image>
88      </configuration>
89    </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>`

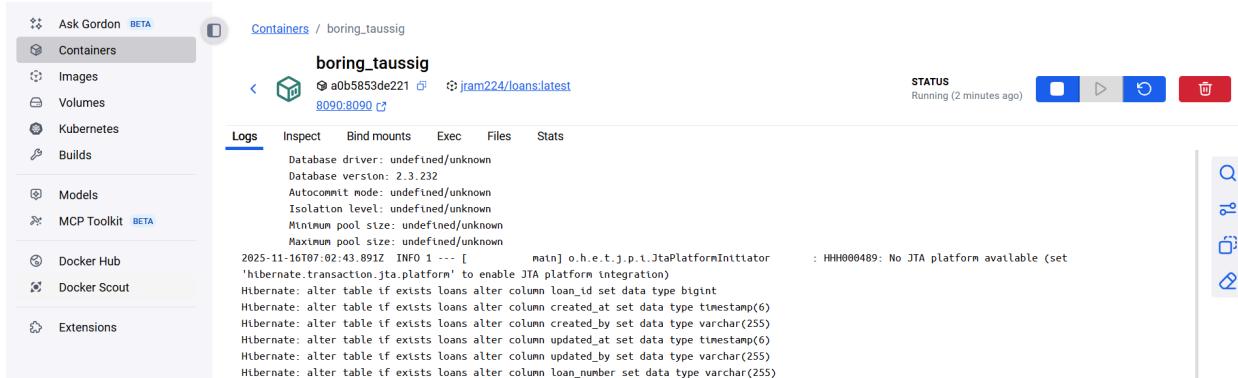
The “45 years ago” date shows up because the base image used by Buildpacks doesn’t have real timestamps, so Docker defaults to the Unix epoch (1970).

```
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
a0b5853de22195bb05137764a6fbfb5009ad11d359c87446193380671f93626c1

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	❖ 🛡 ⚡ 🗑



CRUD REST APIs for Cards in EazyBank

CRUD REST APIs in EazyBank to CREATE, UPDATE, FETCH AND DELETE card details

PUT	/api/update	Update Card Details REST API
POST	/api/create	Create Card REST API
GET	/api/fetch	Fetch Card Details REST API
DELETE	/api/delete	Delete Card Details REST API

Running a Spring Boot app as a container using Buildpacks

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 spring-boot:build-image" 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 8090:8090 eazybytes/loans: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>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <image>
          <name>eazybytes/${project.artifactId}:s4</name>
        </image>
      </configuration>
    </plugin>
  </plugins>
</build>

```

Cloud Native Buildpacks offer an alternative approach to Dockerfiles, prioritizing consistency, security, performance, and governance. With Buildpacks, developers can automatically generate production-ready OCI images from their application source code without the need to write a Dockerfile.

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`.

```

80    ...
81        <plugin>
82            <groupId>com.google.cloud.tools</groupId>
83            <artifactId>jib-maven-plugin</artifactId>
84            <version>3.4.6</version>
85            <configuration>
86                <to>
87                    <image>jram224/${project.artifactId}:latest</image>
88                </to>
89            </configuration>
90        </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
```

The screenshot displays the Docker UI interface. On the left, a sidebar lists various sections: Ask Gordon (Beta), Containers (selected), Images, Volumes, Kubernetes, Builds, Models, MCP Toolkit (Beta), Docker Hub, Docker Scout, and Preferences. The main area shows a container named "admiring_maxwell" with the image ID "9e41a4d8005b6baaafff8dd69cc59f0082717ff8d9a426d01a11d7d26a706152" and port mapping "9000:9000". The "Logs" tab is selected, showing log entries related to database initialization and Hibernate schema creation. To the right, a status bar indicates "Running (54 seconds ago)". Below the UI, a browser window shows the Swagger UI for the "CRUD REST APIs for Cards in EazyBank" at the URL "http://localhost:9000 - Generated server url".

Running a Spring Boot app as a container using Google Jib

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>
```

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

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	○ Amazon ECS	○ OpenShift	○ Gradle	○ Terraform	
	○ CircleCI	○ CircleCI	○ Maven	○ GoReleaser	○ Skaffold	
	○ GitLab	○ GitLab	○ Google	○ Carvel kubectl	○ Tilt	
	○ Google	○ Google	...	○ ...	○ ...	
	○ Heroku	○ Tekton				
	○ Spring Boot	○ ...				
	○ Tekton					
	○ ...					
Governance	CNCF	Docker	Red Hat	Google	CNCF	
Best for Building...	○ Applications	○ Applications	○ Applications	○ Applications	○ Applications	
	○ Base Images	○ Base Images				
	○ OS Images	○ OS Images				

**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.**

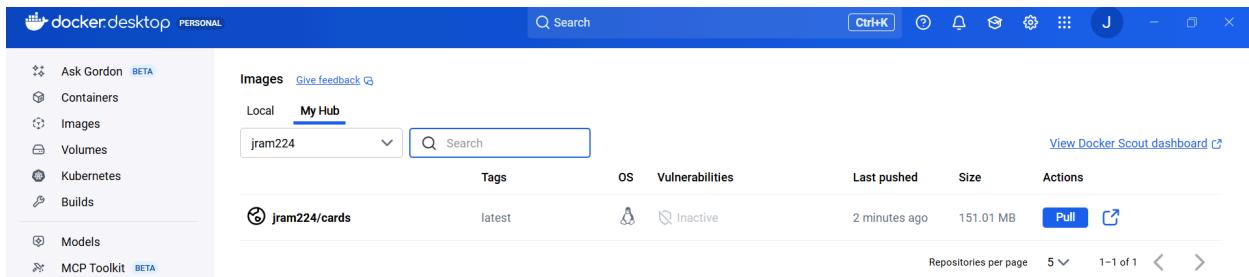
`docker image push docker.io/<imagename>:<tag>`

```
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:c3bc1b14dfd1324cbef6ca568e19ac9f5567b942b30485cb01c93cb41caf14f 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
687668f73cce   jram224/accounts:latest  "java -jar accounts-..."  35 hours ago  Exited (143)  35 hours ago
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   4cd99dc3034  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.

1 Start all services

```
docker compose up
```

Runs all containers defined in **docker-compose.yml**.

To run in the background:

```
docker compose up -d
```

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

```
docker compose down
```

Equivalent to:

- stop containers
- remove containers
- remove networks

3 Start existing containers (without recreating)

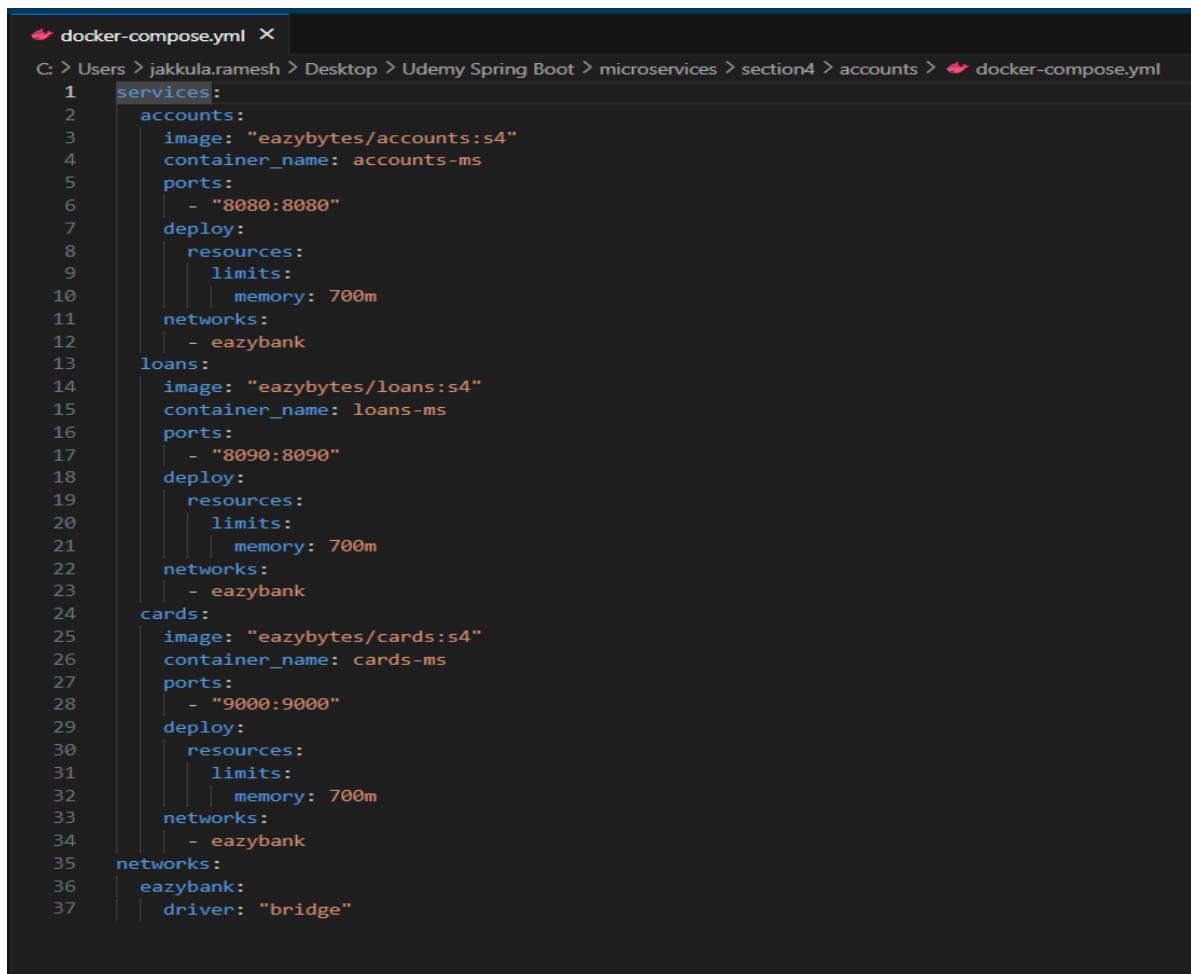
```
docker compose start
```

Start containers that are already created but stopped.

④ Stop running containers (but do not remove them)

docker compose stop

Stops services but does *not* remove containers.



```
docker-compose.yml
C: > Users > jakkula.ramesh > Desktop > Udemy Spring Boot > microservices > section4 > accounts > docker-compose.yml
1  services:
2    accounts:
3      image: "eazybytes/accounts:s4"
4      container_name: accounts-ms
5      ports:
6        - "8080:8080"
7      deploy:
8        resources:
9          limits:
10            memory: 700m
11      networks:
12        - eazybank
13    loans:
14      image: "eazybytes/loans:s4"
15      container_name: loans-ms
16      ports:
17        - "8090:8090"
18      deploy:
19        resources:
20          limits:
21            memory: 700m
22      networks:
23        - eazybank
24    cards:
25      image: "eazybytes/cards:s4"
26      container_name: cards-ms
27      ports:
28        - "9000:9000"
29      deploy:
30        resources:
31          limits:
32            memory: 700m
33      networks:
34        - eazybank
35    networks:
36      eazybank:
37        driver: "bridge"
```

IMPORTANT DOCKER COMMANDS

eazy
bytes

01	docker images To list all the docker images present in the Docker server	06	docker ps To show all running containers	11	docker container stop [container-id] To stop one or more running containers
02	docker image inspect [image-id] To display detailed image information for a given image id	07	docker ps -a To show all containers including running and stopped	12	docker container kill [container-id] To kill one or more running containers instantly
03	docker image rm [image-id] To remove one or more images for a given image ids	08	docker container start [container-id] To start one or more stopped containers	13	docker container restart [container-id] To restart one or more containers
04	docker build . -t [image-name] To generate a docker image based on a Dockerfile	09	docker container pause [container-id] To pause all processes within one or more containers	14	docker container inspect [container-id] To inspect all the details for a given container id
05	docker run -p [hostport]:[containerport] [image_name] To start a docker container based on a given image	10	docker container unpause [container-id] To resume/unpause all processes within one or more containers	15	docker container logs [container-id] To fetch the logs of a given container id

Üder

16	docker container logs -f [container-id] To follow log output of a given container id	21	docker image prune To remove all unused images	26	docker logout To login out from docker hub container registry
17	docker rm [container-id] To remove one or more containers based on container ids	22	docker container stats To show all containers statistics like CPU, memory, I/O usage	27	docker history [image-name] Displays the intermediate layers and commands that were executed when building the image
18	docker container prune To remove all stopped containers	23	Docker system prune Remove stopped containers, dangling images, and unused networks, volumes, and cache	28	docker exec -it [container-id] sh To open a shell inside a running container and execute commands
19	docker image push [container_registry/username:tag] To push an image from a container registry	24	docker rmi [image-id] To remove one or more images based on image ids	29	docker compose up To create and start containers based on given docker compose file
20	docker image pull [container_registry/username:tag] To pull an image from a container registry	25	docker login -u [username] To login in to docker hub container registry	30	docker compose down To stop and remove containers for services defined in the Compose File

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.

What are cloud native applications ?

eazy bytes



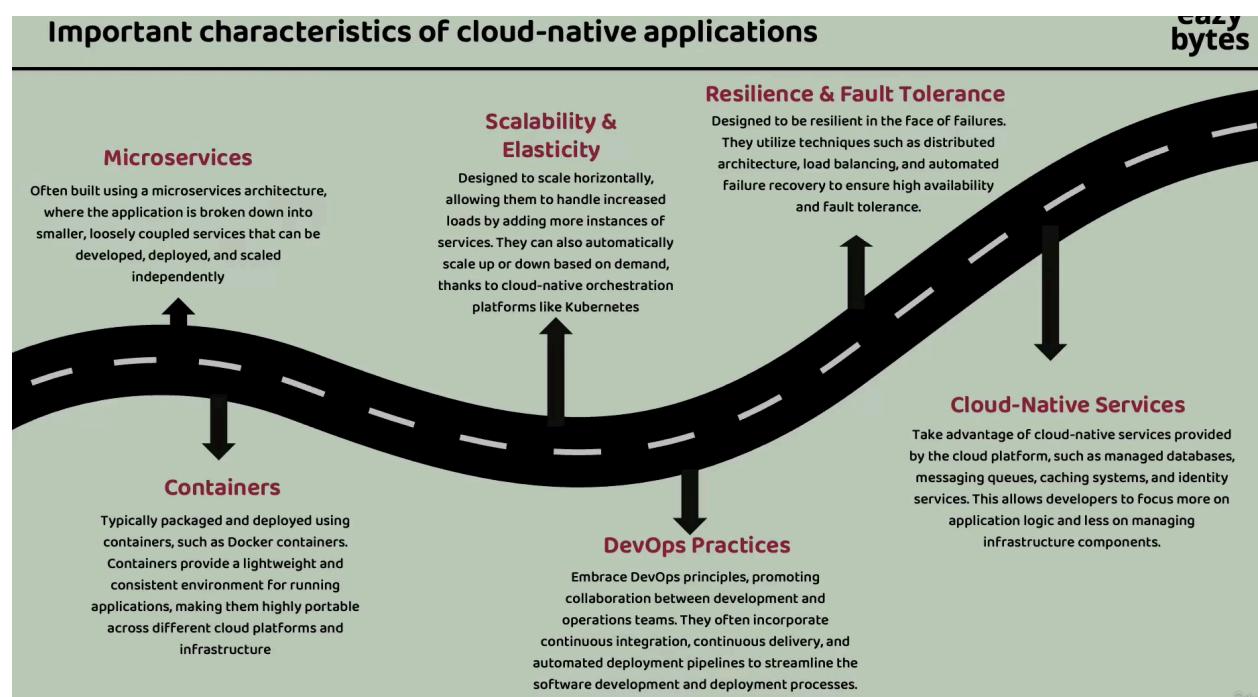
The layman definition

Cloud-native applications are software applications designed specifically to leverage cloud computing principles and take full advantage of cloud-native technologies and services. These applications are built and optimized to run in cloud environments, utilizing the scalability, elasticity, and flexibility offered by the cloud.

The Cloud Native Computing Foundation (CNCF) definition

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.



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.

2 Containers (Docker)

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

3 Orchestration (Kubernetes)

- Manages container deployment, scaling, and recovery automatically.

4 DevOps + CI/CD

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

5 Scalability & Elasticity

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

6 Resilience & Fault Tolerance

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

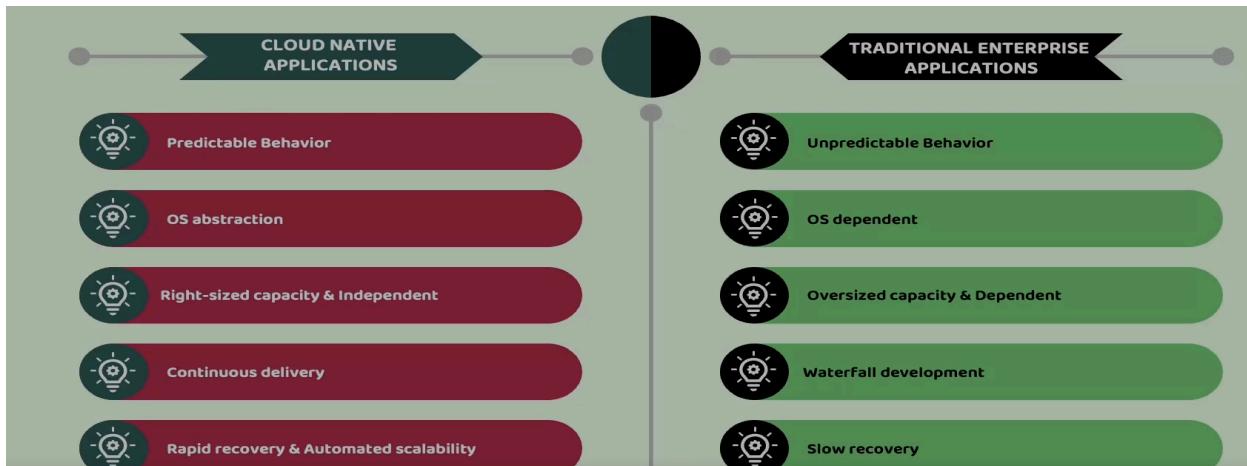
7 API-Driven

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

8 Cloud Managed Services

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

Cloud-Native vs Traditional Applications

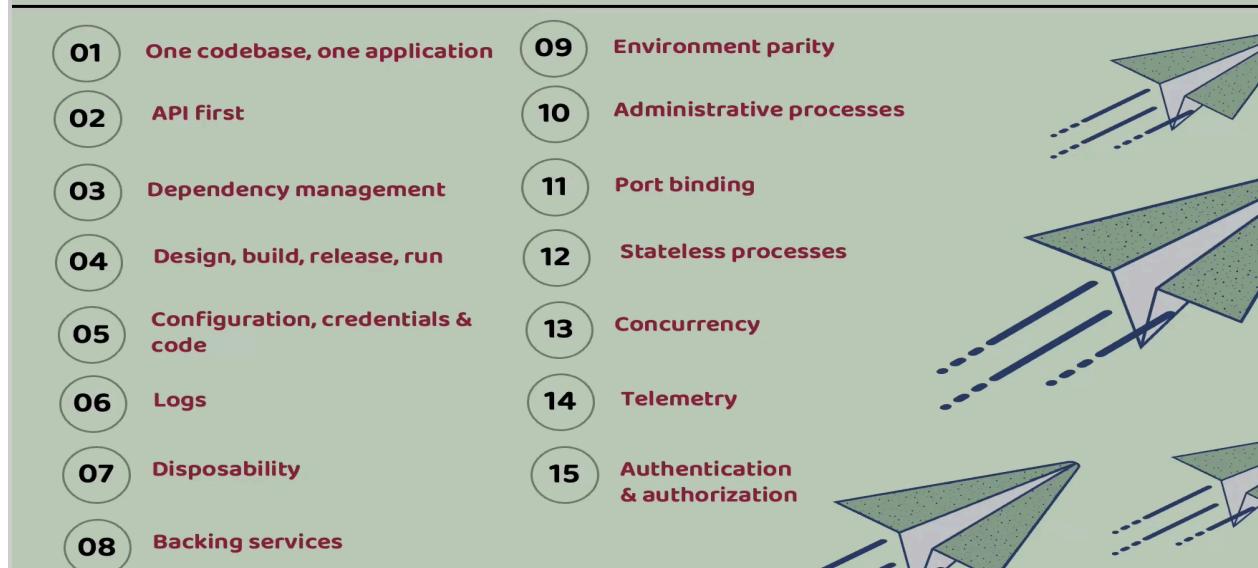


Cloud-Native vs Traditional Applications

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

15-Factor methodology

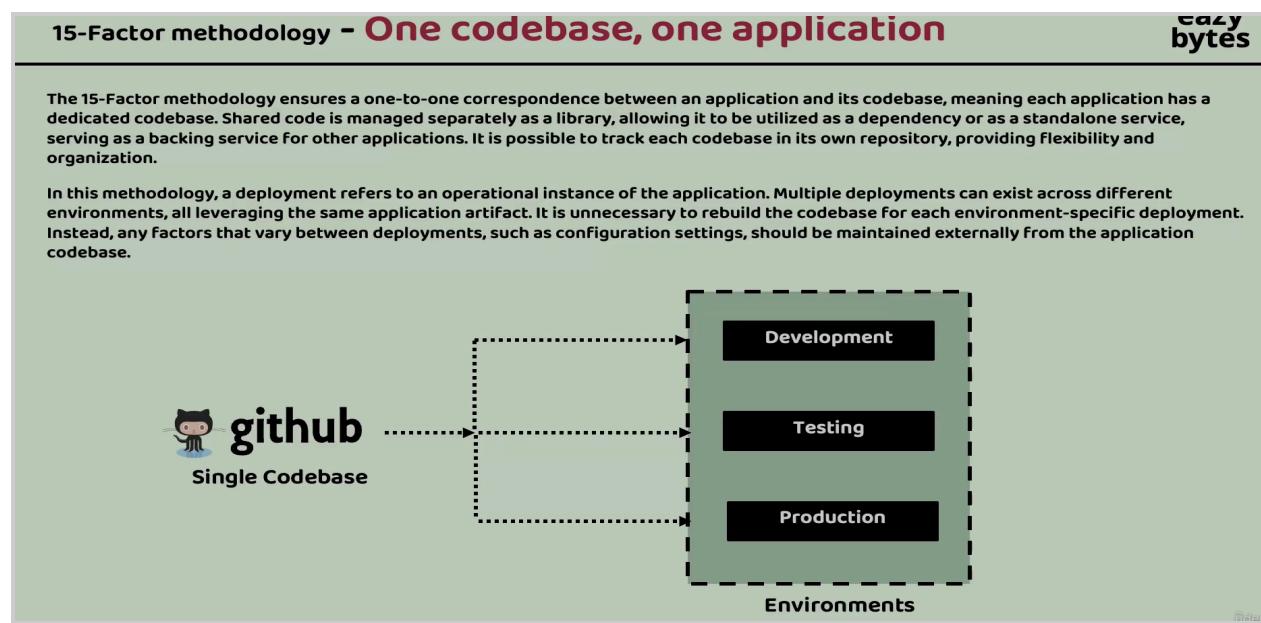


① One Codebase, One Application

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



② 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

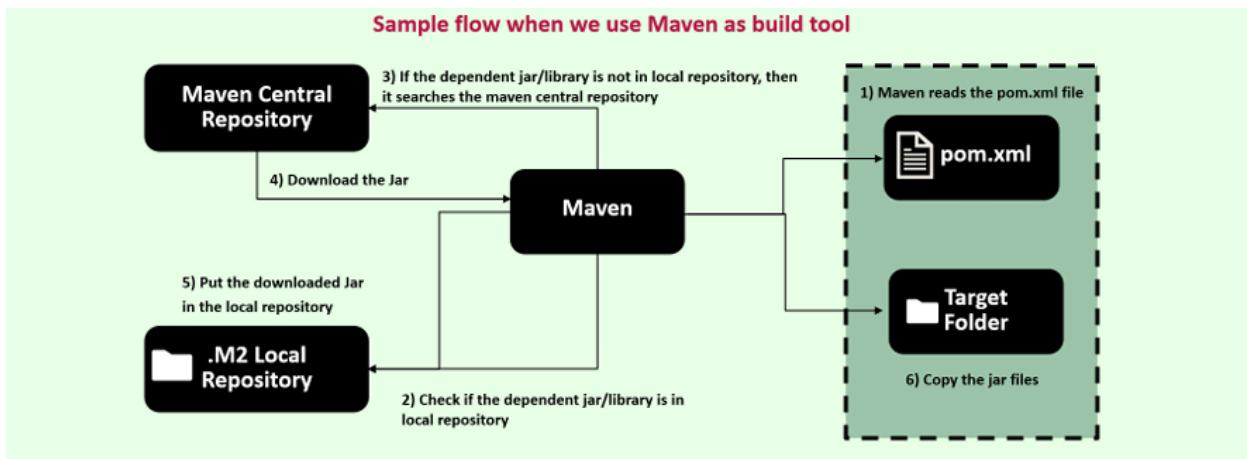
③ Dependency Management

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

Key Points:

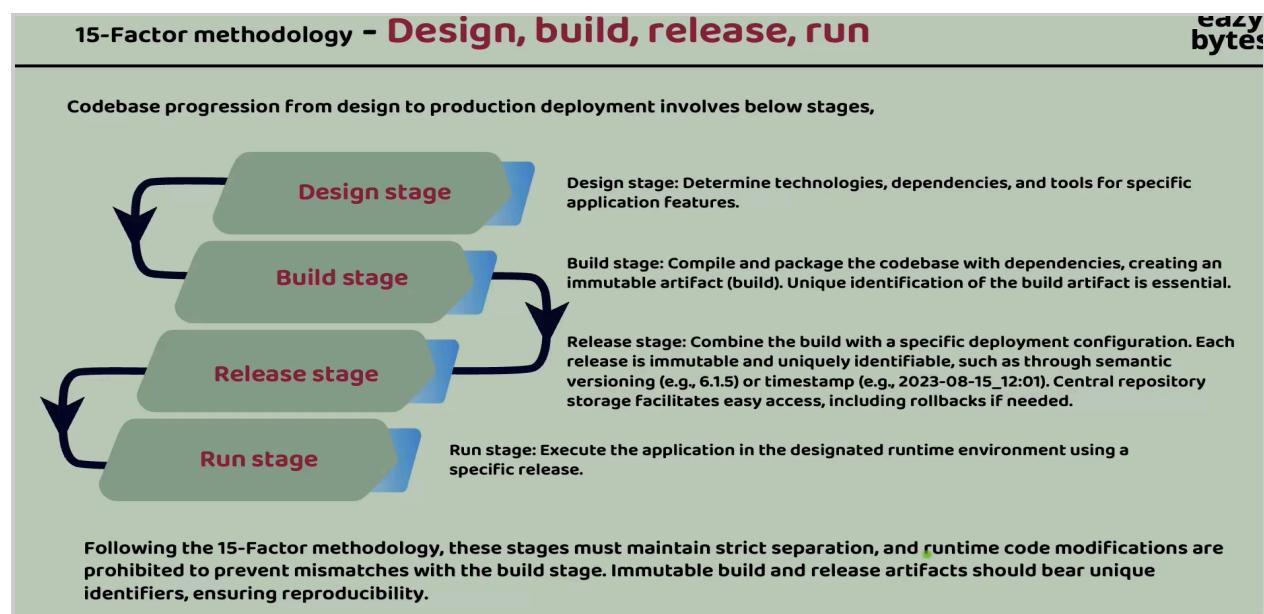
- No implicit/hidden dependencies
- Use dependency managers (Maven, Gradle, npm, etc.)
- Dependency manager will:
 - Check local repo
 - If missing → download from central repo
 - Store in cache
- Ensures consistency and reproducibility

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)

15-Factor methodology - Configuration, credentials & code

eazy bytes

According to the 15-Factor methodology, configuration encompasses all elements prone to change between deployments. It emphasizes the ability to modify application configuration independently, without code changes or the need to rebuild the application.

Configuration may include resource handles for backing services (e.g., databases, messaging systems), credentials for accessing third-party APIs, and feature flags. It is essential to evaluate whether any confidential or environment-specific information would be at risk if the codebase were exposed publicly. This assessment ensures proper externalization of configuration.

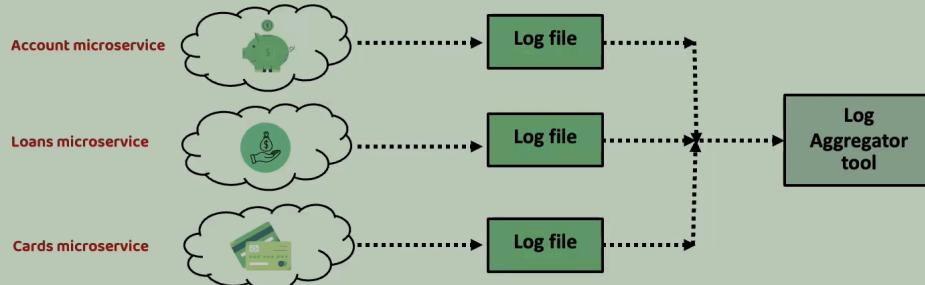
To comply with this principle, configuration should not be embedded within the code or tracked in the same codebase, except for default configuration, which can be bundled with the application. Other configurations can still be managed using separate files, but they should be stored in a distinct repository.

The methodology recommends utilizing environment variables to store configuration. This enables deploying the same application in different environments while adapting its behavior based on the specific environment's configuration.

The diagram illustrates the 15-Factor methodology's approach to configuration. On the left, a GitHub logo with the text "Single Codebase" is shown. A dashed arrow points from this to a central box labeled "Configurations", which contains three sub-boxes: "Dev Config", "Testing Config", and "Prod Config". Another dashed arrow points from the "Configurations" box to a second central box labeled "Environments", which contains three sub-boxes: "Development", "Testing", and "Production". A green dot is positioned near the bottom right of the "Environments" box.

6 Logs – Treat Logs as Event Streams

In a cloud-native application, log routing and storage are not the application's concern. Instead, applications should direct their logs to the standard output, treating them as sequentially ordered events based on time. The responsibility of log storage and rotation is now shifted to an external tool, known as a log aggregator. This tool retrieves, gathers, and provides access to the logs for inspection purposes.



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

15-Factor methodology - Backing services **bytes**

Backing services refer to external resources that an application relies on to provide its functionality. These resources can include databases, message brokers, caching systems, SMTP servers, FTP servers, or RESTful web services. By treating these services as attached resources, you can modify or replace them without needing to make changes to the application code.

Consider the usage of databases throughout the software development life cycle. Typically, different databases are used in different stages such as development, testing, and production. By treating the database as an attached resource, you can easily switch to a different service depending on the environment. This attachment is achieved through resource binding, which involves providing necessary information like a URL, username, and password for connecting to the database.

In the below example, we can see that a local DB can be swapped easily to a third-party DB like AWS DB with out any code changes,

```
graph LR; MicroserviceApp[Microservice App] -- "URL" --> LocalDB[Local DB]; MicroserviceApp -- "URL" --> AWSDB[AWS DB]; MicroserviceApp -- "URL" --> AWS_S3[AWS S3];
```

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

Environment parity aims to minimize differences between various environments & avoiding costly shortcuts. Here, the adoption of containers can greatly contribute by promoting the same execution environment.

There are three gaps that this factor addresses:



Time gap: The time it takes for a code change to be deployed can be significant. The methodology encourages automation and continuous deployment to reduce the time between code development and production deployment.



People gap: Developers create applications, while operators handle their deployment in production. To bridge this gap, a DevOps culture promotes collaboration between developers and operators, fostering the "you build it, you run it" philosophy.



Tools gap: Handling of backing services differs across environments. For instance, developers might use the H2 database locally but PostgreSQL in production. To achieve environment parity, it is recommended to use the same type and version of backing services across all environments.

10 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

11 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.

12 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.

13 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.

15-Factor methodology - Concurrency

Scalability is not solely achieved by creating stateless applications. While statelessness is important, scalability also requires the ability to serve a larger number of users. This means that applications should support concurrent processing to handle multiple users simultaneously.

According to the 15-Factor methodology, processes play a crucial role in application design. These processes should be horizontally scalable, distributing the workload across multiple processes on different machines. This concurrency is only feasible when applications are stateless. In Java Virtual Machine (JVM) applications, concurrency is typically managed through the use of multiple threads, which are available from thread pools.

Vertical Scalability	
 Virtual Machine 2 GB RAM 2 CPU Before	 Virtual Machine 4 GB RAM 4 CPU After

Horizontal Scalability		
 Virtual Machine 2 GB RAM 2 CPU Before	 Virtual Machine 2 GB RAM 2 CPU After	 Virtual Machine 2 GB RAM 2 CPU After

14 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

15 Authentication & Authorization (Secure by Default)

Every cloud-native app must:

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

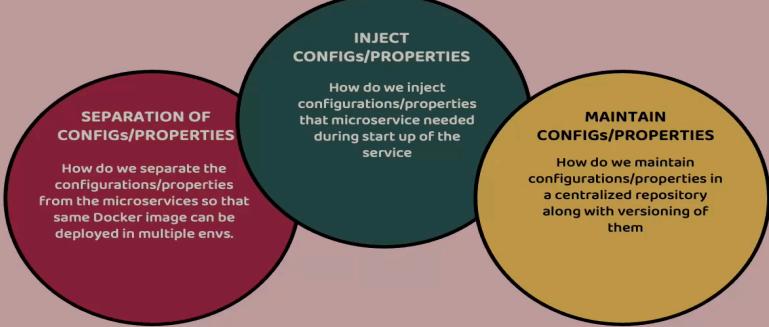
Common methods:

- OAuth2
 - JWT
 - OpenID
-

Configuration Management in Microservices

CONFIGURATION MANAGEMENT IN MICROSERVICES

eazy bytes



There are multiple solutions available in Spring Boot ecosystem to handle this challenge. Below are the solutions. Let's try to identify which one suites for microservices

- 1) Configuring Spring Boot with properties and profiles
- 2) Applying external configuration with Spring Boot
- 3) Implementing a configuration server with Spring Cloud Config Server

GeekyML

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:

HOW TO READ PROPERTIES IN SPRINGBOOT APPS

eazy bytes

In Spring Boot, there are multiple approaches to reading properties. Below are the most commonly used approaches,

Using @Value Annotation

(1)

You can use the @Value annotation to directly inject property values into your beans. This approach is suitable for injecting individual properties into specific Fields. For example:

```
@Value("${property.name}")
private String propertyName;
```

Using Environment

(2)

The Environment interface provides methods to access properties from the application's environment. You can autowire the Environment bean and use its methods to retrieve property values. This approach is more flexible and allows accessing properties programmatically. For example:

```
@Autowired
private Environment environment;

public void getProperty() {
    String propertyName =
        environment.getProperty("property.name");
}
```

Using @ConfigurationProperties

(3)

Recommended approach as it avoids hard coding the property keys

The @ConfigurationProperties annotation enables binding of entire groups of properties to a bean. You define a configuration class with annotated Fields matching the properties, and Spring Boot automatically maps the properties to the corresponding fields.

```
@ConfigurationProperties("prefix")
public class MyConfig {
    private String property;
}

// getters and setters
```

In this case, properties with the prefix "prefix" will be mapped to the Fields of the MyConfig class.

Basic ways:

Using @value : which @Value → reads single values

Values which we need inject from application.properties or yml

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

```
@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;
```

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

Code	Details
200	<p>Response body</p> <pre>1</pre>

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.

```
accounts:
  message: "Welcome to EazyBank accounts related local APIs "
  contactDetails:
    name: "John Doe - Developer"
    email: "john@eazibank.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*.

```
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:

```
19    app:  
20        name: Accounts Service  
21        version: 1.0.0  
22        build: 2025  
23        author: Ram
```

```
6     @ConfigurationProperties(prefix = "app")  no usages  
7     @Data  
8     public class AppConfigProperties {  
9         private String name;  no usages  
10        private String version;  no usages  
11        private String build;  no usages  
12        private String author;  no usages  
13    }  
14 }
```

```
@EnableConfigurationProperties(AppConfigProperties.class)  
> public class AccountsApplication {  
  
>>     public static void main(String[] args) { SpringApplication.run(AccountsApplication.class, args); }  
}
```

```
189     @GetMapping("account_details")  no usages  
190     public ResponseEntity<AppConfigProperties> getAppDetails() {  
191         return ResponseEntity.status(HttpStatus.OK).body(appConfigProperties);  
192     }  
193 }
```

```
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."

The slide has a green header with the title "Profiles". The main content area is divided into two sections: a green sidebar and a pink main area.

Green Sidebar Content:

- Spring provides a great tool for grouping configuration properties into so-called profiles(dev, qa, prod) allowing us to activate a bunch of configurations based on the active profile.**
- Profiles are perfect for setting up our application for different environments, but they're also being used in another use cases like Bean creation based on a profile etc.**
- So basically a profile can influence the application properties loaded and beans which are loaded into the Spring context.**

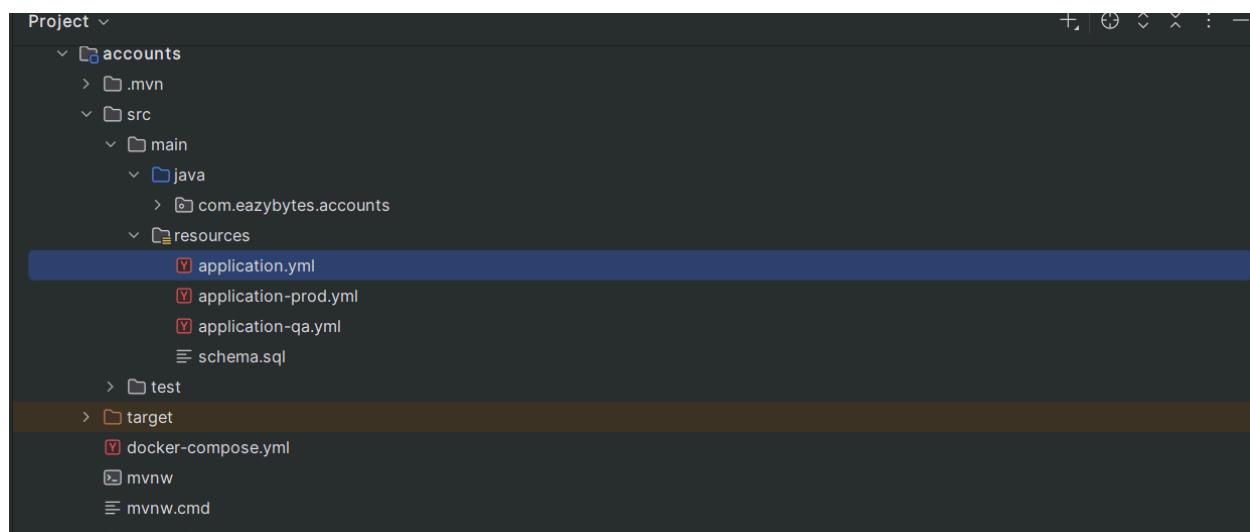
Pink Main Area Content:

- The default profile is always active. Spring Boot loads all properties in `application.properties` into the default profile.**
- We can create another profiles by creating property files like below,**
`application_prod.properties -----> for prod profile`
`application_qa.properties -----> for QA profile`
- We can activate a specific profile using `spring.profiles.active` property like below,**
`spring.profiles.active=prod`

Important Note: An important point to consider is that once an application is built and packaged, it should not be modified. If any configuration changes are required, such as updating credentials or database handles, they should be made externally.

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`

`application_qa`

<pre>build: version: "3.0" app: name: Accounts Dev Service version: 1.0.0 build: 2025 author: RamDev</pre>	<pre>build: version: "2.0" app: name: Accounts QA Service version: 1.0.0 build: 2025 author: RamQA</pre>
---	---

`Application_prod`

<pre>build: version: "1.0" app: name: Accounts Prod Service version: 1.0.0 build: 2025 author: RamProd</pre>

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`

<pre>17 config: 18 activate: 19 on-profile: "prod"</pre>	<pre>17 config: 18 activate: 19 on-profile: "qa"</pre>
--	--

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:

The screenshot shows a terminal window with two parts. On the left, lines 17 and 18 of a configuration file are shown. Line 17 contains the word "profiles:" and line 18 contains "active: "qa"". On the right, a JSON response is displayed with a status of 200 and a header "Response body". The JSON object contains the following properties: "name": "Accounts QA Service", "version": "1.0.0", "build": "2025", and "author": "RamQA".

```
200
Response body
{
  "name": "Accounts QA Service",
  "version": "1.0.0",
  "build": "2025",
  "author": "RamQA"
}

17      profiles:
18      active: "qa"
```

Or

The screenshot shows a terminal window with two parts. On the left, lines 17 and 18 of a configuration file are shown. Line 17 contains the word "profiles:" and line 18 contains "active: "prod"". On the right, a JSON response is displayed with a status of 200 and a header "Response body". The JSON object contains the following properties: "name": "Accounts Prod Service", "version": "1.0.0", "build": "2025", and "author": "RamProd".

```
200
Response body
{
  "name": "Accounts Prod Service",
  "version": "1.0.0",
  "build": "2025",
  "author": "RamProd"
}

17      profiles:
18      active: "prod"
```

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
```

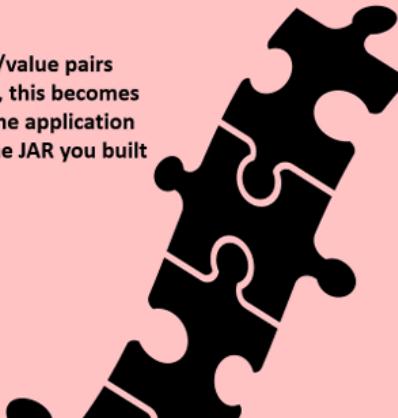
How to externalize configurations using command-line arguments ?

eazy
bytes

Spring Boot automatically converts command-line arguments into key/value pairs and adds them to the Environment object. In a production application, this becomes the property source with the highest precedence. You can customize the application configuration by specifying command-line arguments when running the JAR you built earlier.

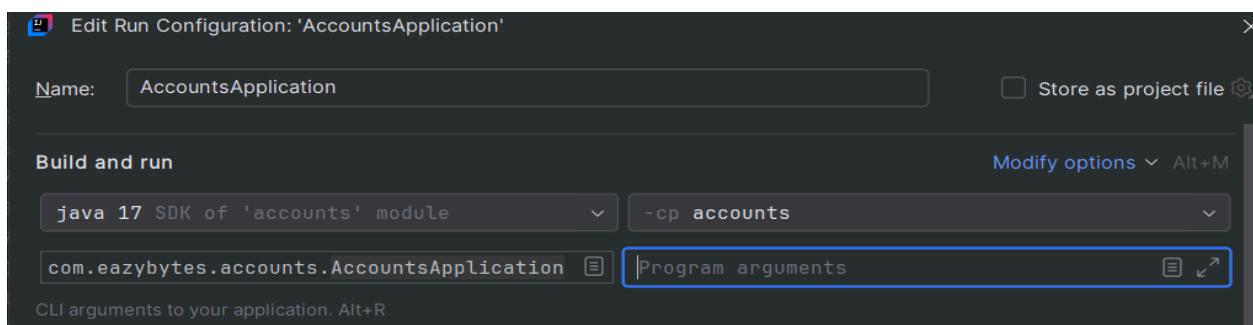
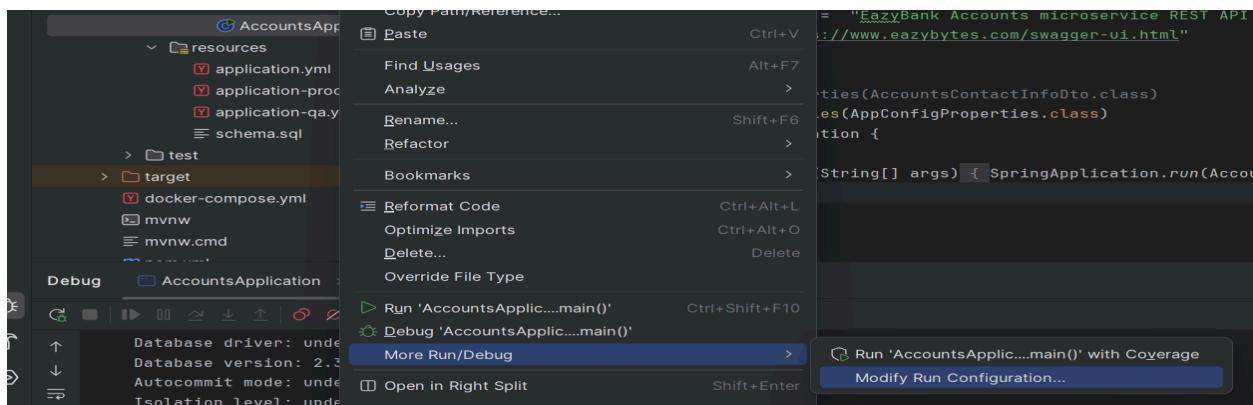
```
java -jar accounts-service-0.0.1-SNAPSHOT.jar --build.version="1.1"
```

The command-line argument follows the same naming convention as the corresponding Spring property, with the familiar `--` prefix for CLI arguments.

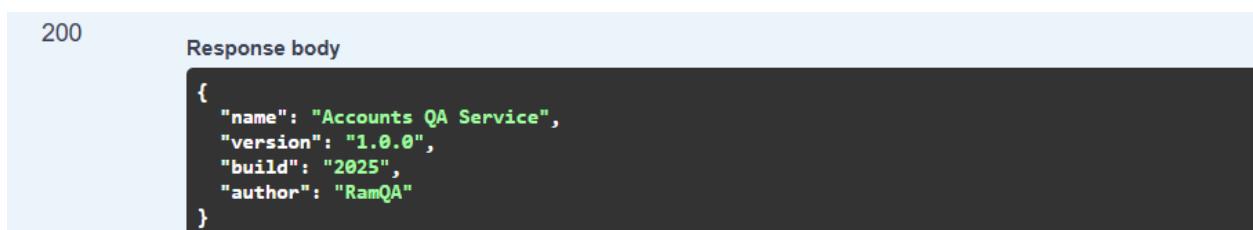
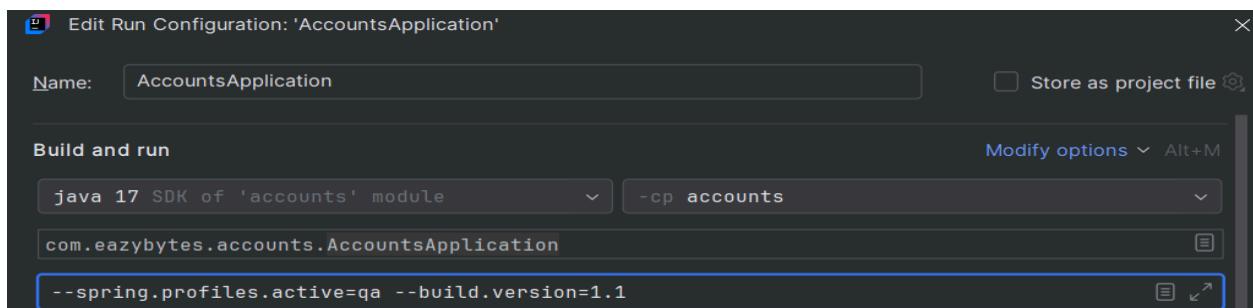


We can also provide command-line arguments directly in the IDE's run configuration, apart from just from the command line.”

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
```

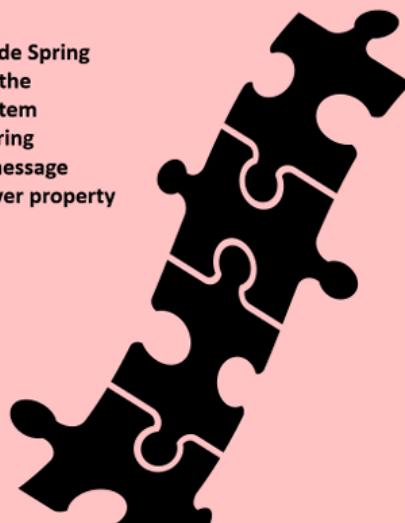
How to externalized configurations using JVM system properties ?

eazy
bytes

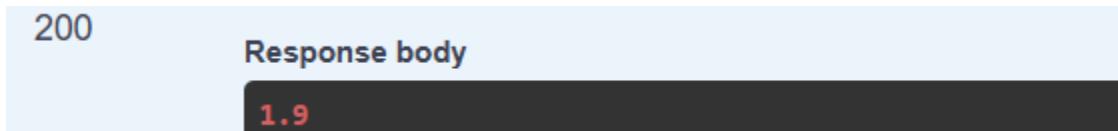
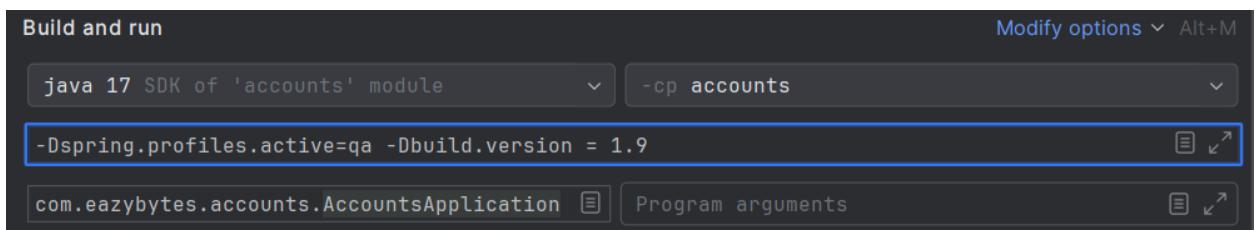
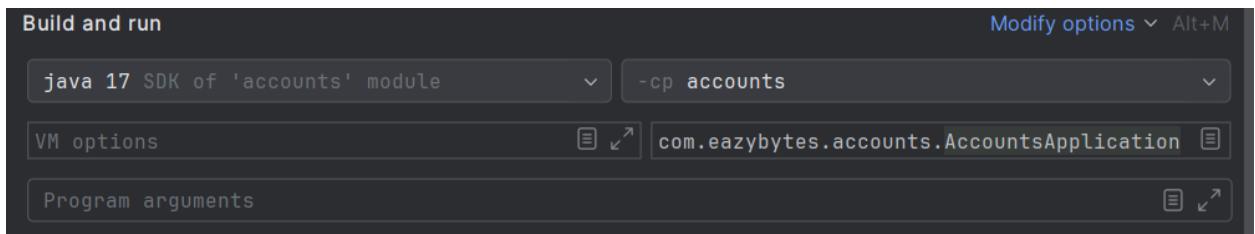
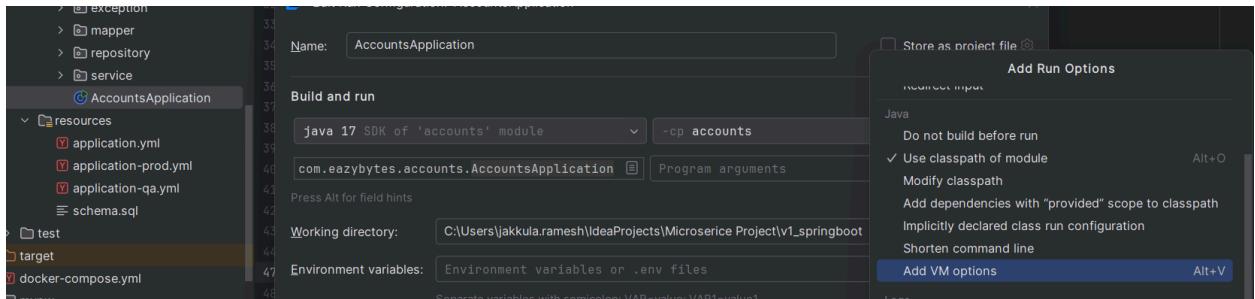
JVM system properties, similar to command-line arguments, can override Spring properties with a lower priority. This approach allows for externalizing the configuration without the need to rebuild the JAR artifact. The JVM system property follows the same naming convention as the corresponding Spring property, prefixed with **-D** for JVM arguments. In the application, the message defined as a JVM system property will be utilized, taking precedence over property files.

```
java -Dbuild.version="1.2" -jar accounts-service-0.0.1-SNAPSHOT.jar
```

In the scenario where both a JVM system property and a command-line argument are specified, the precedence rules dictate that Spring will prioritize the value provided as a command-line argument. This means that the value specified through the CLI will be utilized by the application, taking precedence over the JVM properties.



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.

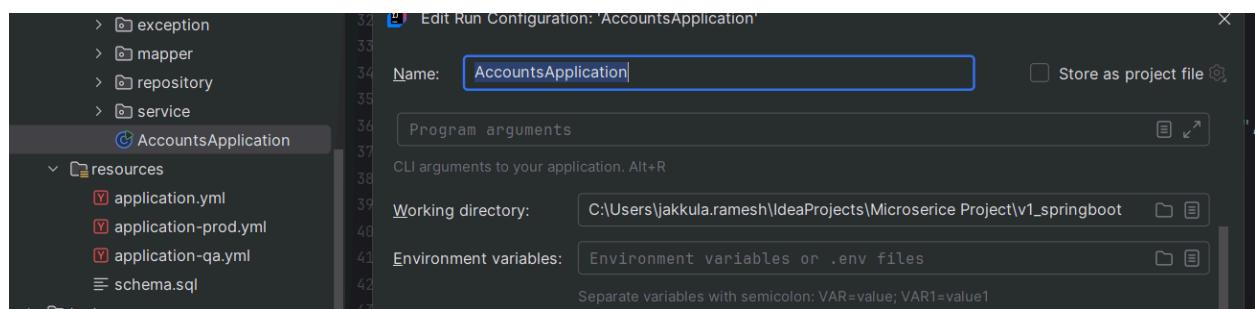
✗ Once you close CMD, those values disappear.

To map an environment variable to a Spring property:

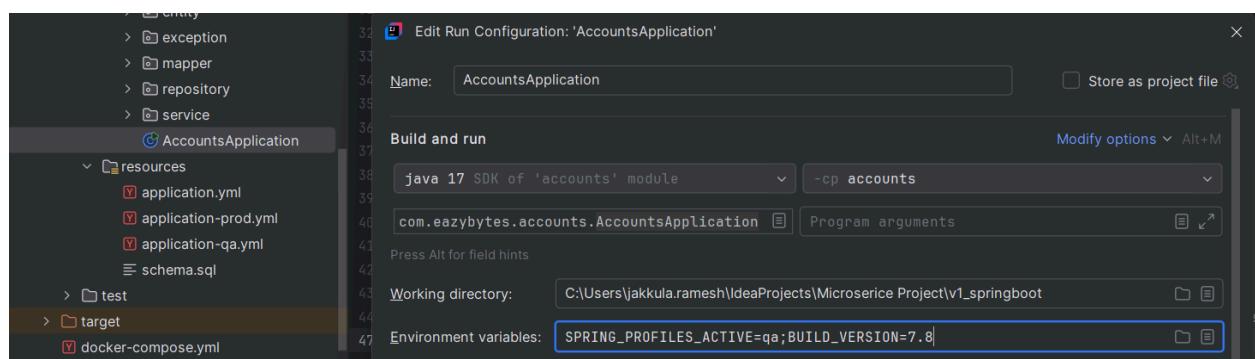
- Convert all letters to **uppercase**
- Replace dots (.) or dashes (-) with **underscores** (_)

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 ; .



200

Response body

7.8

This screenshot shows a successful API response with a status code of 200. The response body contains the number 7.8, indicating the version or build number.

200

Response body

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

This screenshot shows a successful API response with a status code of 200. The response body is a JSON object containing service metadata: name, version, build, and author.

How to externalized configurations using environment variables ?

eazy
bytes

Environment variables are widely used for externalized configuration as they offer portability across different operating systems, as they are universally supported. Most programming languages, including Java, provide mechanisms to access environment variables, such as the `System.getenv()` method.

To map a Spring property key to an environment variable, you need to convert all letters to uppercase and replace any dots or dashes with underscores. Spring Boot will handle this mapping correctly internally. For example, an environment variable named `BUILD_VERSION` will be recognized as the property `build.version`. This feature is known as relaxed binding.

Windows

```
env:BUILD_VERSION="1.3"; java -jar accounts-service-0.0.1-SNAPSHOT.jar
```

Linux based OS

```
BUILD_VERSION="1.3" java -jar accounts-service-0.0.1-SNAPSHOT.jar
```



Most-Useful Priority

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

1 CLI arguments, JVM properties, and environment variables are effective ways to externalize configuration and maintain the immutability of the application build. However, using these approaches often involves executing separate commands and manually setting up the application, which can introduce potential errors during deployment.

2 Given that configuration data evolves and requires changes, similar to application code, what strategies should be employed to store, track revisions and audit the configuration used in a release?

3 In scenarios where environment variables lack granular access control features, how can you effectively control access to configuration data?

4 When the number of application instances grows, handling configuration in a distributed manner for each instance becomes challenging. How can such challenges be overcome?



5 Considering that neither Spring Boot properties nor environment variables support configuration encryption, how should secrets be managed securely?

6 After modifying configuration data, how can you ensure that the application can read it at runtime without necessitating a complete restart?

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.

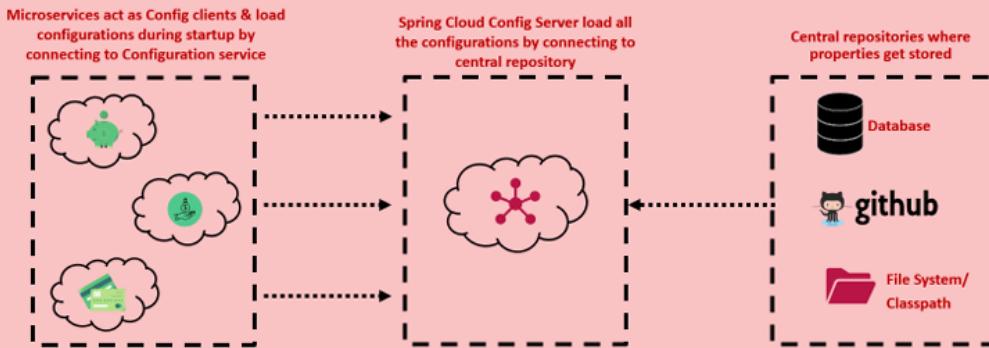
Spring Cloud Config

eazy bytes

A centralized configuration server with Spring Cloud Config can overcome all the drawbacks that we discussed in the previous slide. Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system. With the Config Server you have a central place to manage external properties for applications across all environments.

Centralized configuration revolves around two core elements:

- A data store designed to handle configuration data, ensuring durability, version management, and potentially access control.
- A server that oversees the configuration data within the data store, facilitating its management and distribution to multiple applications.

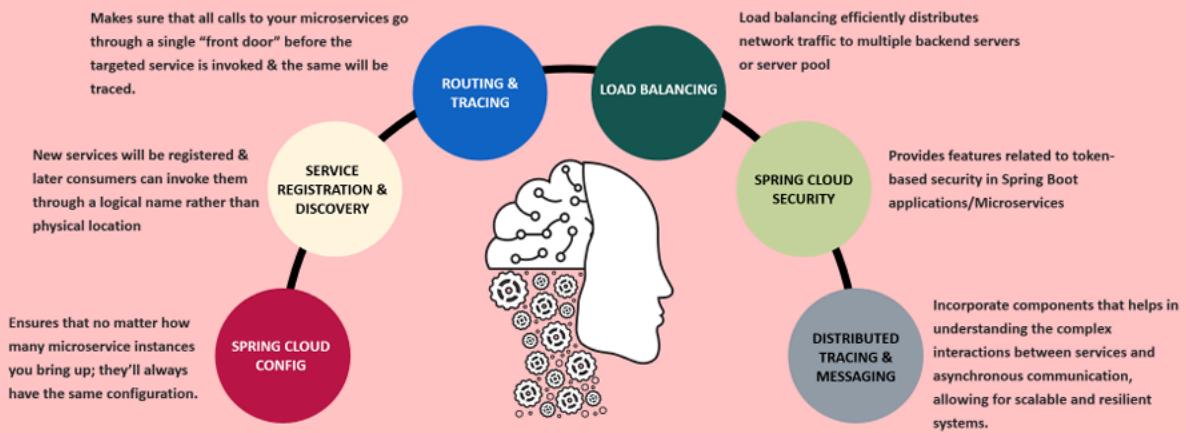


WHAT IS SPRING CLOUD?

USING SPRING CLOUD FOR MICROSERVICES DEVELOPMENT

eazy bytes

Spring Cloud provides frameworks for developers to quickly build some of the common patterns of Microservices



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

Config Server SPRING CLOUD CONFIG

Central management for configuration via Git, SVN, or HashiCorp Vault.



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.



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).

```
1   server:
2     port: "8071"
3   spring:
4     application:
5       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.

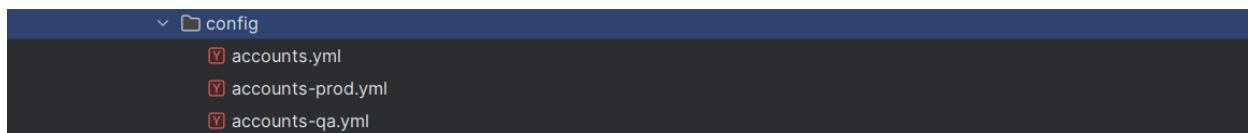
So, if your microservice has:

`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.searchLocations=classpath:/config`

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

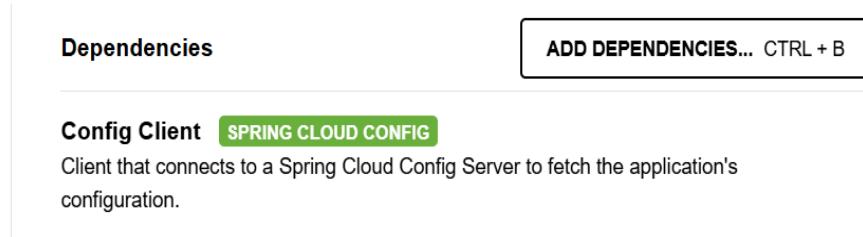
```
1  server:
2    port: "8071"
3  spring:
4    application:
5      name: "springcloudconfigserver"
6    profiles:
7      active: native
8    cloud:
9      config:
10     server:
11       native:
12         search_LOCATIONS: "classpath:/config"
13
```

- **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.



```
3  spring:
4    application:
5      name: "accounts"
6    profiles:
7      active: "prod"
8    config:
9      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).

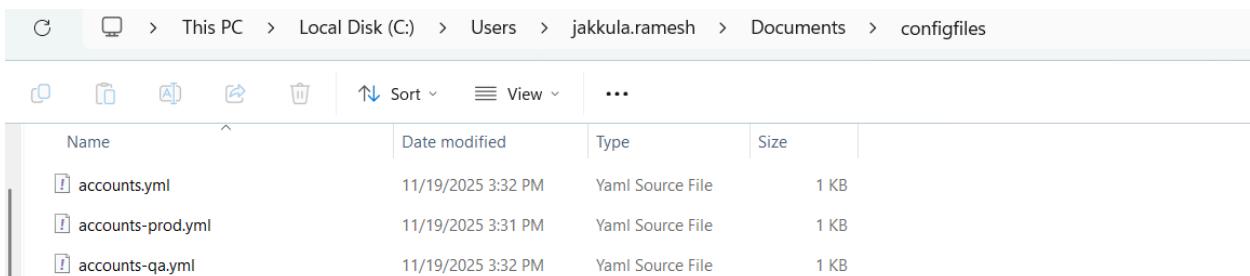


The **classpath approach**, where config files are kept inside `src/main/resources/config`, is suitable only for small or demo projects because **it is not secure—anyone who has access to the Config Server's codebase can view all configuration details**. It also **requires rebuilding and redeploying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

```
springcloudconfigserver...\application.yml
1 server:
2   port: "8071"
3 spring:
4   application:
5     name: "springcloudconfigserver"
6   profiles:
7     active: native
8   cloud:
9     config:
10    server:
11      native:
12        searchLocations: "file:///C:/Users/jakkula.namesh/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**.

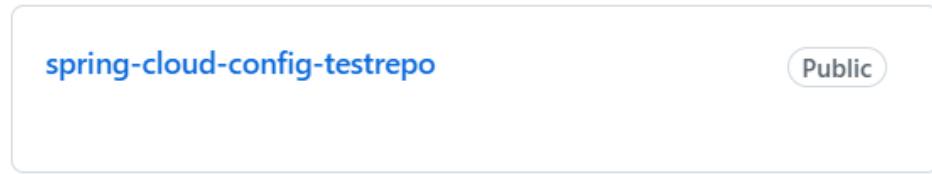
Ex 3: Reading Configurations from the GitHub Repository

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

A screenshot of a GitHub commit history. The commit was made by 'Rameshdhoni' at 1be3157, 1 minute ago. The commit message is 'Add files via upload'. Three files were added via upload: 'accounts-prod.yml', 'accounts-qa.yml', and 'accounts.yml', all added 1 minute ago.

- Configure Spring Cloud Config Server to use that GitHub repo

A screenshot of a code editor showing the 'application.yml' configuration file. The file contains the following YAML code:

```
server:
  port: "8071"
spring:
  application:
    name: "springcloudconfigserver"
  profiles:
    active: git
  cloud:
    config:
      server:
        git:
          uri: https://github.com/Rameshdhoni/spring-cloud-config-testrepo.git
          default-label: main
          timeout: 5
          clone-on-start: true
          force-pull: true
```

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.

```
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

```
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:

AQBsd908asda87asd...

3. Store encrypted values in GitHub repo

In your Git repo config file:

db:

```
password: "{cipher}AQBs..."
```

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

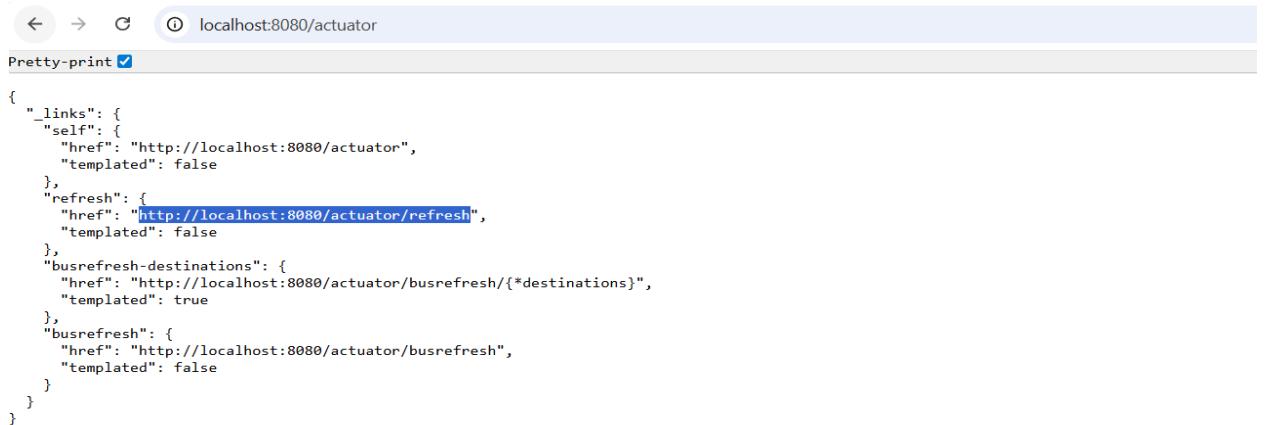
```

23     management:
24         endpoints:
25             web:
26                 exposure:
27                     include: refresh
28

```

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

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



A screenshot of a browser window showing the JSON response of the `/actuator` endpoint. The URL in the address bar is `localhost:8080/actuator`. The response is a well-structured JSON object with various links for management, refresh, and bus refresh operations.

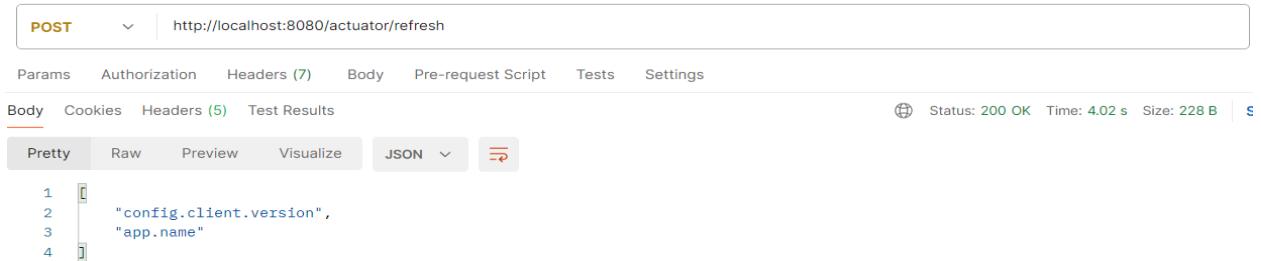
```

{
  "_links": {
    "self": {
      "href": "http://localhost:8080/actuator",
      "templated": false
    },
    "refresh": {
      "href": "http://localhost:8080/actuator/refresh",
      "templated": false
    },
    "busrefresh-destinations": {
      "href": "http://localhost:8080/actuator/busrefresh/{*destinations}",
      "templated": true
    },
    "busrefresh": {
      "href": "http://localhost:8080/actuator/busrefresh",
      "templated": false
    }
  }
}

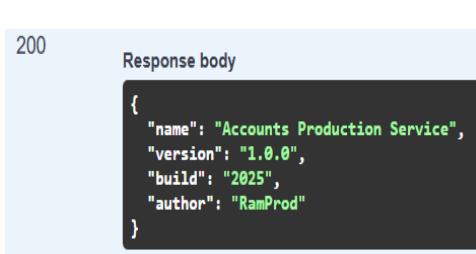
```

Then invoke refresh endpoint mentioned in refresh

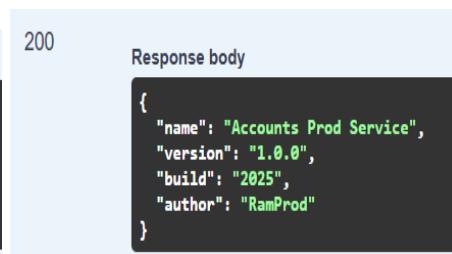
POST <http://localhost:8080/actuator/refresh>



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,

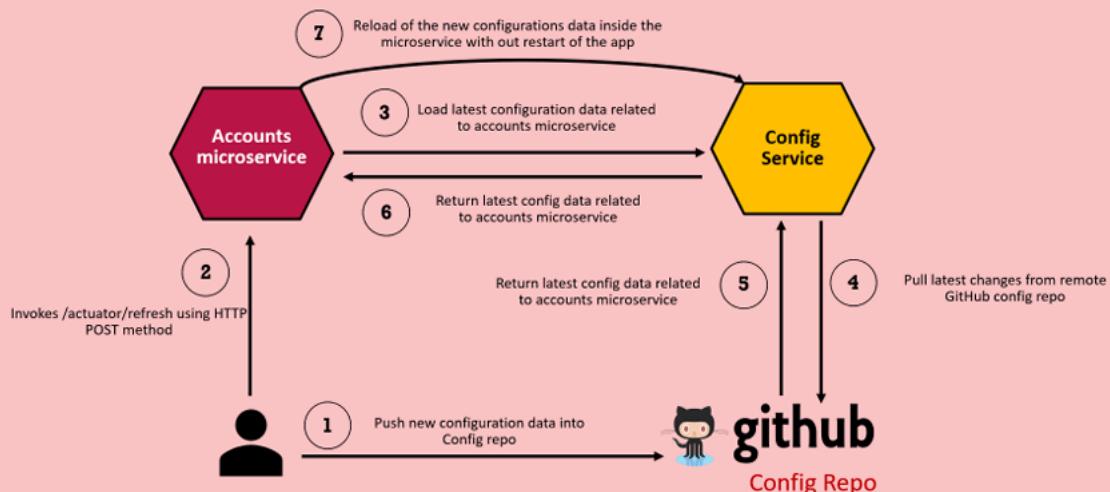
- 1 Add actuator dependency in the Config Client services: Add Spring Boot Actuator dependency inside pom.xml of the individual microservices like accounts, loans, cards to expose the /refresh endpoint

- 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%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
2b60085445fe: Download complete
ed8d255a715b: 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

```
55 <dependency>
56   <groupId>org.springframework.cloud</groupId>
57   <artifactId>spring-cloud-starter-bus-amqp</artifactId>
58 </dependency>
```

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

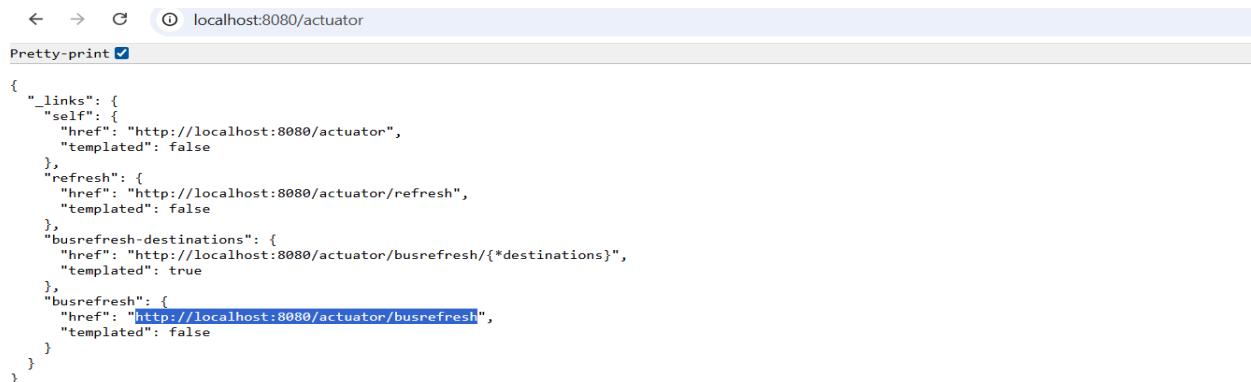
If we add *, it allows all actuator endpoints. If we specify particular endpoints, then only those endpoints will be allowed.

```
rabbitmq:
  host: localhost
  port: 5672
  username: guest
  password: guest
management:
  endpoints:
    web:
      exposure:
        include: refresh, bus-refresh
      show: true
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

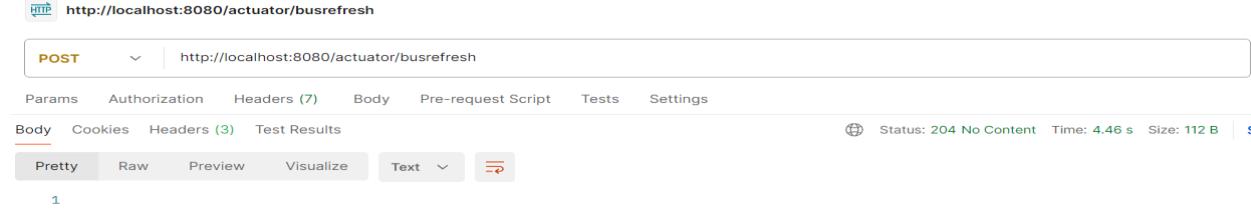
4. Trigger Config Refresh

After updating configuration in Spring Cloud Config repo, call:

All microservices connected to RabbitMQ will reload their configuration automatically.



```
{ "_links": { "self": { "href": "http://localhost:8080/actuator", "templated": false }, "refresh": { "href": "http://localhost:8080/actuator/refresh", "templated": false }, "busrefresh-destinations": { "href": "http://localhost:8080/actuator/busrefresh/{*destinations}", "templated": true }, "busrefresh": { "href": "http://localhost:8080/actuator/busrefresh", "templated": false } } }
```

POST http://localhost:8080/actuator/busrefresh

Status: 204 No Content Time: 4.46 s Size: 112 B

Response body:

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

Refresh configurations at runtime using Spring Cloud Bus

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.

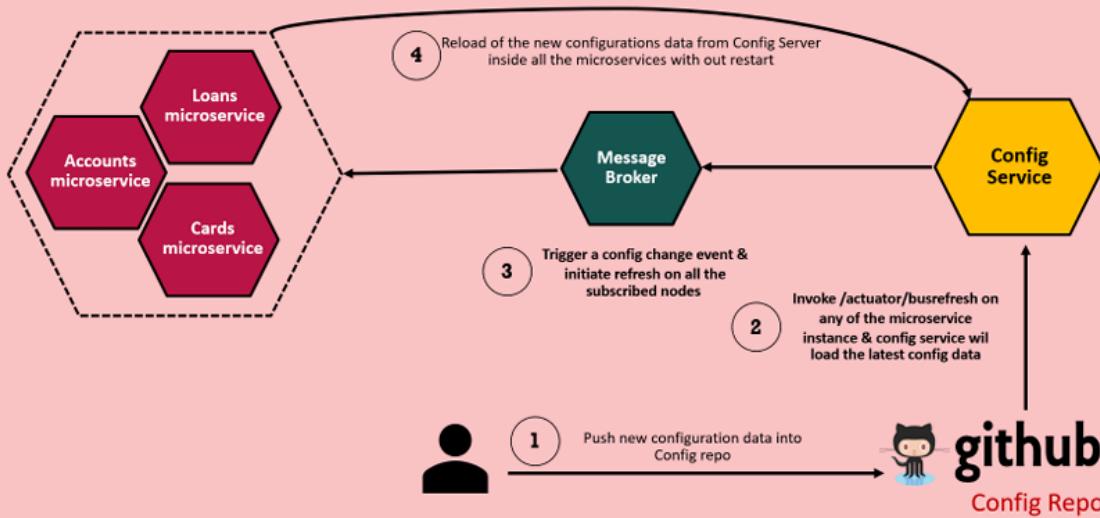
Below are the steps to follow,

- 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

Refresh configurations at runtime using Spring Cloud Bus

eazy
byte



Though this approach reduces manual work to a great extent, but still there is a single manual step involved which is invoking the /actuator/busrefresh on any of the microservice instances. Let's see how we can avoid and completely automate the process.

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
-

```

    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

The screenshot shows the GitHub repository settings page for 'spring-cloud-config-testrepo'. The 'Webhooks' tab is selected. On the left, there's a sidebar with options like General, Access, Collaborators, and Moderation options. The main area shows a form for adding a webhook:

- Webhooks / Add webhook**
- Payload URL ***: `https://example.com/postreceive`
- Content type ***: `application/x-www-form-urlencoded`
- Secret**: (A text input field)

- 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.

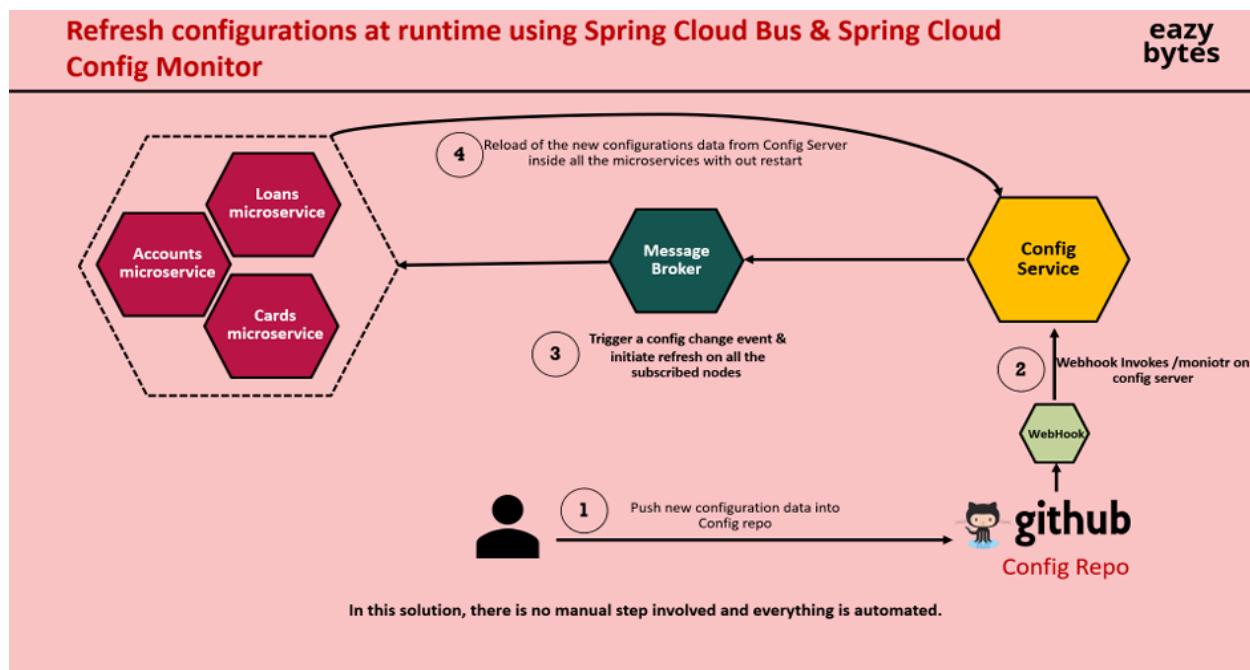
Refresh configurations at runtime using Spring Cloud Bus & Spring Cloud Config Monitor

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.



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)

Liveness and Readiness probes

eazy bytes

A **liveness** probe sends a signal that the container or application is either alive (passing) or dead (failing). If the container is alive, then no action is required because the current state is good. If the container is dead, then an attempt should be made to heal the application by restarting it.

In simple words, liveness answers a true-or-false question: "Is this container alive?"

A **readiness** probe used to know whether the container or app being probed is ready to start receiving network traffic. If your container enters a state where it is still alive but cannot handle incoming network traffic (a common scenario during startup), you want the readiness probe to fail. So that, traffic will not be sent to a container which isn't ready for it.

If someone prematurely send network traffic to the container, it could cause the load balancer (or router) to return a 502 error to the client and terminate the request. The client would get a "connection refused" error message.

In simple words, readiness answers a true-or-false question: "Is this container ready to receive network traffic ?"

Inside Spring Boot apps, actuator gathers the "Liveness" and "Readiness" information from the ApplicationAvailability interface and uses that information in dedicated health indicators: LivenessStateHealthIndicator and ReadinessStateHealthIndicator. These indicators are shown on the global health endpoint ("`/actuator/health`"). They are also exposed as separate HTTP Probes by using health groups: "`/actuator/health/liveness`" and "`/actuator/health/readiness`"

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
```

accounts	11/15/2025 3:14 PM	File folder
cards	11/15/2025 3:14 PM	File folder
configserver	11/15/2025 3:14 PM	File folder
docker-compose	11/15/2025 3:14 PM	File folder
loans	11/15/2025 3:14 PM	File folder

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.

Name	Date modified	Type	Size
default	11/15/2025 3:14 PM	File folder	
prod	11/15/2025 3:14 PM	File folder	
qa	11/15/2025 3:14 PM	File folder	

Name	Date modified	Type	Size
common-config.yml	11/15/2025 3:14 PM	Yaml Source File	1 KB
docker-compose.yml	11/15/2025 3:14 PM	Yaml Source File	2 KB

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

1 RabbitMQ 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
<code>test</code>	Command run inside container to check health
<code>interval: 10s</code>	Run test every 10 seconds
<code>timeout: 5s</code>	Fail test if no response in 5 seconds
<code>retries: 10</code>	Mark container unhealthy after 10 failures
<code>start_period: 5s</code>	Wait 5 seconds before starting checks

Extends:

extends:

file: common-config.yml

service: network-deploy-service

Meaning:

- shared network config

2 Config Server

configserver:

image: "eazybytes/configserver:s6"

container_name: configserver-ms

ports:

- "8071:8071"

depends_on:

rabbit:

condition: service_healthy

Explanation:

Setting	Meaning
<code>depends_on.condition: service_healthy</code>	Start only after RabbitMQ is healthy
<code>ports: 8071</code>	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
<code>curl ... /readiness</code>	Ensures Spring Boot says "ready"
<code>grep UP</code>	Only ready if UP
<code>start_period: 10s</code>	Wait for Spring Boot startup

Extends:

extends:

file: common-config.yml

service: microservice-base-config

Meaning:

- memory limit
- RabbitMQ host
- network config

3 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 NAMES STATUS PORTS
3446dc1094a eazybytes/accounts:s6 "java -cp @/app/jib_..." 54 seconds ago Up Less than a second accounts-ms
49df7df49ca0 eazybytes/loans:s6 "java -cp @/app/jib_..." 54 seconds ago Up Less than a second loans-ms
78ae22774607 eazybytes/cards:s6 "java -cp @/app/jib_..." 54 seconds ago Up Less than a second cards-ms
18e243b04b24 eazybytes/configserver:s6 "java -cp @/app/jib_..." 54 seconds ago Up 42 seconds (healthy) configserver-ms
74e6d30d3735 rabbitmq:3.12-management "docker-entrypoint.s..." 54 seconds ago Up 53 seconds (healthy) 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/>

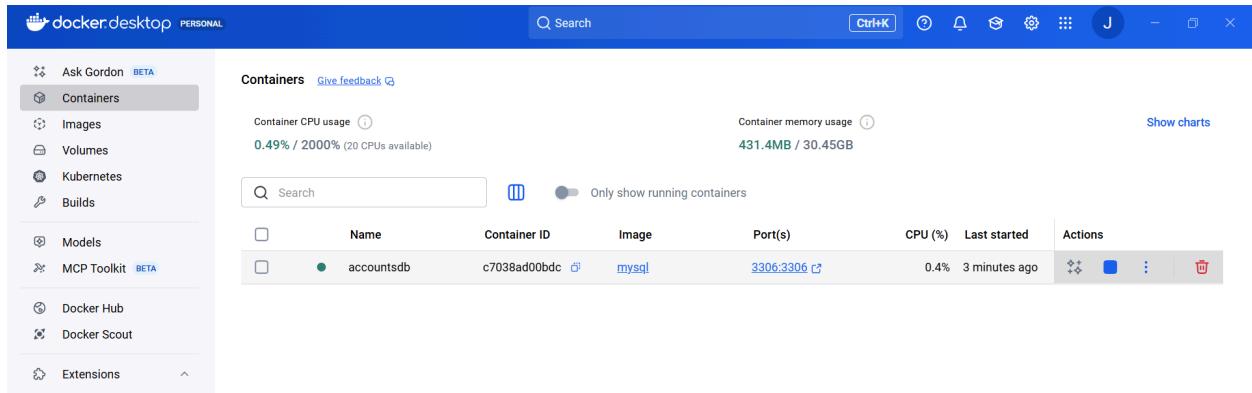
```
C:\Users\jakkula.ramesh>docker run -p 3306:3306 --name accountsdb -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=accountsdb -d mysql
Unable to find image 'mysql:latest' locally
latest: Pulling from library/mysql
480d01bd7a6a: Pull complete
5a3f7744d0e7: Pull complete
494c372d15c3: Pull complete
21aa606d8d58: Pull complete
dceef8f7340c: Pull complete
834e15e3ed24: Pull complete
c276de9b5571: Pull complete
0cd145fbb449: Pull complete
f5f78fcfd9ccb: Pull complete
023a182c62a0: Pull complete
Digest: sha256:569c4128dfa625ac2ac62cdd8af588a3a6a60a049d1a8d8f0fac95880ecdbbe5
Status: Downloaded newer image for mysql:latest
c7038ad00bdc121c7cc590acd8e53deb15a6ad50b6c979ffc7b862e7b6b646e7

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

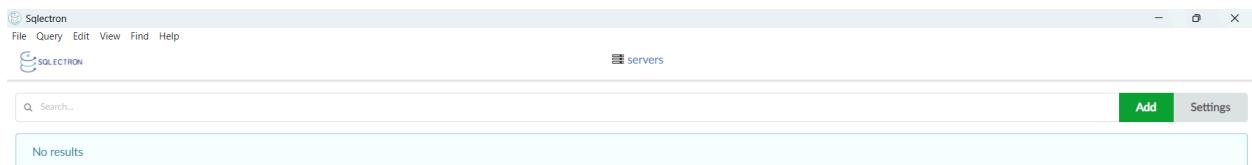
- **--name accountsdb**
Assign a custom name to the MySQL container so you can easily start, stop, or reference it.
- **-e MYSQL_ROOT_PASSWORD=root**
Sets the root user password inside the MySQL server.
(You can replace `root` with a more secure password.)
- **-e MYSQL_DATABASE=accountsdb**
Automatically 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.



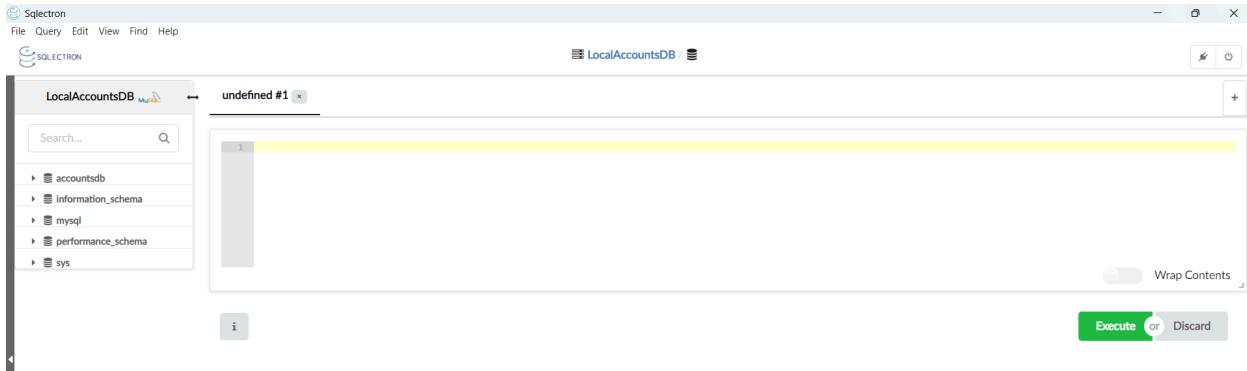
We need to connect with **Sqlectron** as connections to see the UI of the database.



Server Information

Connection Test
Successfully connected ×

Name <input type="text" value="LocalAccountsDB"/>	Database Type MySQL <input type="checkbox"/> SSL		
Server Address <input type="text" value="localhost"/> <input type="text" value="3306"/> <input type="text" value="Domain"/> <input style="float: right; border: none; width: 20px; height: 20px; background: #e0f2e0;" type="text" value="Unix socket path"/>			
User <input type="text" value="root"/>	Password <input type="password" value="*****"/>	Initial Database/Keyspace <input type="text" value="Database"/>	Initial Schema <input type="text" value="Schema"/>
URI <input style="width: 100%; height: 40px; border: none;" type="text"/>			



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.

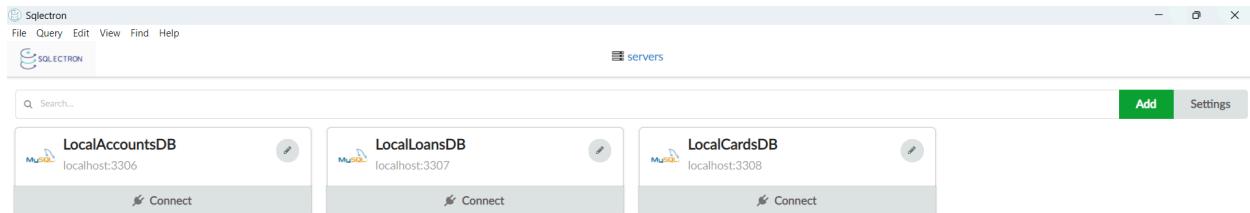
Only the **host ports** are different (3306, 3308, 3309).

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
<input type="checkbox"/>	accountsdb	c7038ad00bdc	mysql	3306:3306 ↕	0.43%	21 minutes ago	
<input type="checkbox"/>	loansdb	6c2a58266d01 ↗	mysql	3307:3306 ↕	0.29%	2 minutes ago	
<input type="checkbox"/>	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?

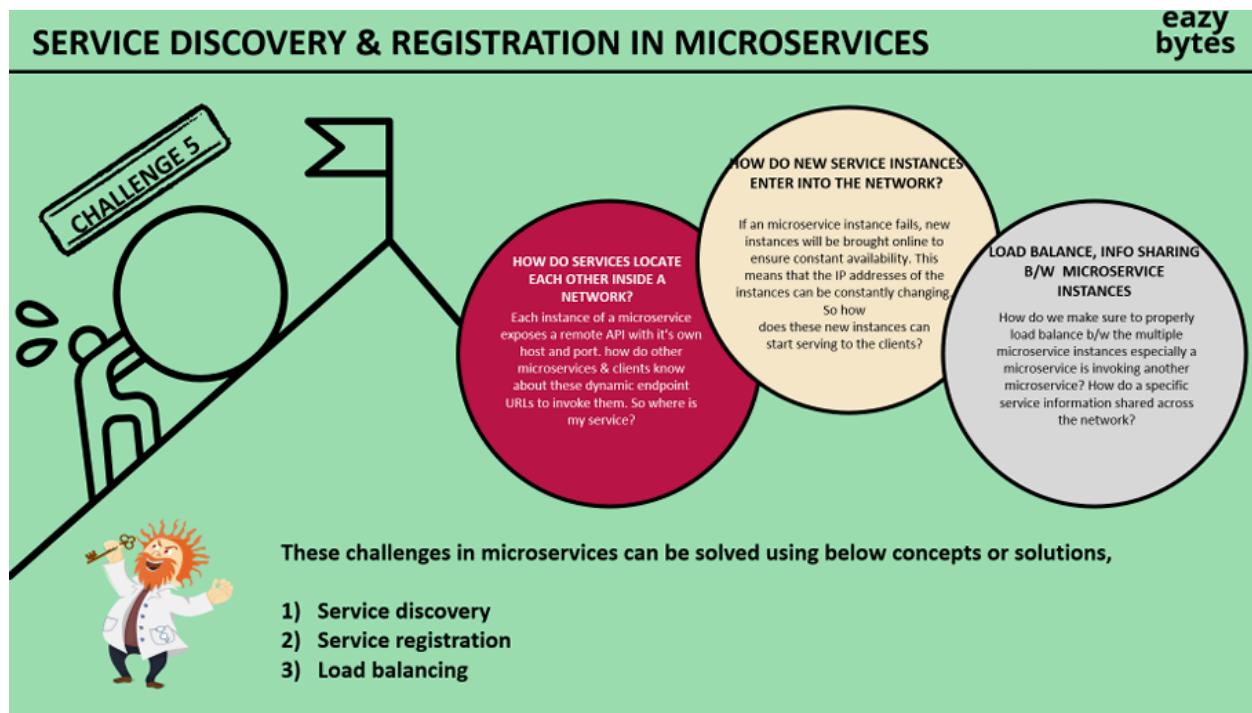
iii) How do we load balance between multiple instances?

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 there 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.

```

    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka-server</artifactId>
    </dependency>

```

The screenshot shows the project structure on the left and the `application.yml` file content on the right. The project structure includes a `eurekaserver` module with `src/main/java` and `src/resources` directories. The `application.yml` file contains the following configuration:

```

spring:
  application:
    name: "eurekasher"
  config:
    import: "optional:configserver:http://localhost:8071/"
  management:
    endpoints:
      web:
        exposure:
          include: "*"
    health:
      readiness-state:
        enabled: true
      liveness-state:
        enabled: true
    endpoint:
      health:
        probes:
          enabled: true

```

```

server:
  port: 8070
eureka:
  instance:
    hostname: localhost
    client:
      fetchRegistry: false
      registerWithEureka: false
      serviceUrl:
        defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

```

`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.

Environment	test	Current time	2025-11-22T19:38:52 +0530
Data center	default	Uptime	00:05
		Lease expiration enabled	false
		Renews threshold	1
		Renews (last min)	0

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

Steps to build Eureka Server

eazy
bytes

Below are the steps to build a Eureka Server application using Spring Cloud Netflix's Eureka,

- 1 **Set up a new Spring Boot 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-starter-netflix-eureka-server` maven dependency.

- 2 **Configure the properties:** In the application properties or YAML file, add the following configurations,

```
server:  
  port: 8070  
  
eureka:  
  instance:  
    hostname: localhost  
  client:  
    fetchRegistry: false  
    registerWithEureka: false  
    serviceUrl:  
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```

- 3 **Add the Eureka Server annotation:** In the main class of your project, annotate it with `@EnableEurekaServer`. This annotation configures the application to act as a Eureka Server.

- 4 **Build and run the Eureka Server:** Build your project and run it as a Spring Boot application. Open a web browser and navigate to <http://localhost:8070>. You should see the Eureka Server dashboard, which displays information about registered service instances.

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

Add Eureka Client dependency in Each microservice



A screenshot of a code editor showing a snippet of XML code. The code defines a dependency for the Spring Cloud Netflix Eureka Client. It includes the group ID (org.springframework.cloud), artifact ID (spring-cloud-starter-netflix-eureka-client), and the closing tag for the dependency element. The code is pasted into a code block with line numbers 71 through 74 visible on the left.

```
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.

```

00 --- [accounts] [ restartedMain] com.netflix.discovery.DiscoveryClient : Registered Applications size is zero : true
00 --- [accounts] [ restartedMain] com.netflix.discovery.DiscoveryClient : Application version is '-1: true'
00 --- [accounts] [ restartedMain] com.netflix.discovery.DiscoveryClient : Getting all instance registry info from the eureka server
00 --- [accounts] [ restartedMain] com.netflix.discovery.DiscoveryClient : The response status is 200
00 --- [accounts] [ restartedMain] c.n.discovery.InstanceInfoReplicator : Starting heartbeat executor: renew interval is: 30
00 --- [accounts] [ restartedMain] c.n.discovery.InstanceInfoReplicator : InstanceInfoReplicator onDemand update allowed rate per min is 4
00 --- [accounts] [ restartedMain] o.s.c.n.e.s.EurekaServiceRegistry : Discovery Client initialized at timestamp 1763822450829 with initial instances count: 0
00 --- [accounts] [ restartedMain] com.netflix.discovery.DiscoveryClient : Registering application ACCOUNTS with eureka with status UP
00 --- [accounts] [ foReplicator->d] com.netflix.discovery.DiscoveryClient : Saw Local status change event StatusChangeEvent [timestamp=1763822450839, current=UP, previous=S
00 --- [accounts] [ foReplicator->d] com.netflix.discovery.DiscoveryClient : DiscoveryClient_ACCOUNTS/host.docker.internal:accounts:8080: registering service...
00 --- [accounts] [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
00 --- [accounts] [ restartedMain] s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to 8080
00 --- [accounts] [ foReplicator->d] com.netflix.discovery.DiscoveryClient : DiscoveryClient_ACCOUNTS/host.docker.internal:accounts:8080 - registration status: 204
00 --- [accounts] [ restartedMain] c.e.accounts.AccountsApplication : Started AccountsApplication in 37.705 seconds (process running for 40.222)
00 --- [accounts] [ reshExecutor->d] com.netflix.discovery.DiscoveryClient : Disable delta property : false
00 --- [accounts] [ reshExecutor->d] com.netflix.discovery.DiscoveryClient : Single vip registry refresh property : null
00 --- [accounts] [ reshExecutor->d] com.netflix.discovery.DiscoveryClient : Force full registry fetch : false
00 --- [accounts] [ reshExecutor->d] com.netflix.discovery.DiscoveryClient : Application is null : false

```

Configure application.yml to register with Eureka Server

```

management:
  endpoints:
    web:
      exposure:
        include: "*"
  endpoint:
    shutdown:
      access: unrestricted
  info:
    env:
      enabled: true
  eureka:
    instance:
      preferIpAddress: true
    client:
      fetchRegistry: true
      registerWithEureka: true
      serviceUrl:
        defaultZone: http://localhost:8070/eureka/
  info:
    app:
      name: "accounts"
      description: "Easy Bank Accounts Application"
      version: "1.0.0"

```

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)

The screenshot shows the Spring Eureka dashboard at localhost:8070. The top navigation bar includes links for HOME and LAST 1000 SINCE STARTUP. The main section, titled 'System Status', displays various configuration parameters:

Environment	test	Current time	2025-11-22T21:06:05 +0530
Data center	default	Uptime	00:56
		Lease expiration enabled	false
		Renews threshold	6
		Renews (last min)	3

A red warning message at the bottom states: "EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE."

The 'DS Replicas' section lists instances currently registered with Eureka:

Application	AMIs	Availability Zones	Status
ACCOUNTS	n/a (1)	(1)	UP (1) - host.docker.internal:accounts:8080
CARDS	n/a (1)	(1)	UP (1) - host.docker.internal:cards:9000
LOANS	n/a (1)	(1)	UP (1) - host.docker.internal:loans:8090

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

i) RestTemplate

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

- 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

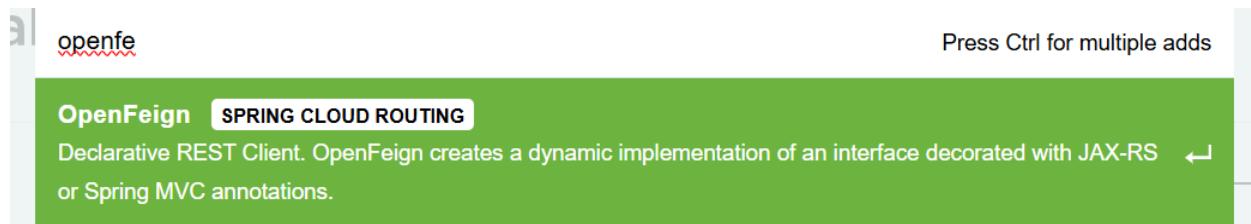
2) Asynchronous Communication: Asynchronous methods like message brokers (e.g., [RabbitMQ](#), [Kafka](#)) or event streaming, where a service publishes a message without waiting for an immediate reply.
No waiting for a response.

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:



```

34   <dependency>
35     <groupId>org.springframework.cloud</groupId>
36     <artifactId>spring-cloud-starter-openfeign</artifactId>
37   </dependency>

```

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

```

@EnableFeignClients
public class AccountsApplication {

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

```

Step 3: Make Sure Services Are Registered in Eureka

Each microservice must have:

```

yaml
copy code

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

```

@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.

```
@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

EUREKA SELF-PRESERVATION TO AVOID TRAPS IN NETWORK

by

- Configurations which will directly or indirectly impact self-preservation behavior of eureka
 - ✓ `eureka.instance.lease-renewal-interval-in-seconds = 30`
Indicates the frequency the client sends heartbeats to server to indicate that it is still alive
 - ✓ `eureka.instance.expiration-duration-in-seconds = 90`
Indicates the duration the server waits since it received the last heartbeat before it can evict an instance
 - ✓ `eureka.server.eviction-interval-timer-in-ms = 60 * 1000`
A scheduler(EvictionTask) is run at this frequency which will evict instances from the registry if the lease of the instances are expired as configured by lease-expiration-duration-in-seconds. It will also check whether the system has reached self-preservation mode (by comparing actual and expected heartbeats) before evicting.
 - ✓ `eureka.server.renewal-percent-threshold = 0.85`
This value is used to calculate the expected % of heartbeats per minute eureka is expecting.
 - ✓ `eureka.server.renewal-threshold-update-interval-ms = 15 * 60 * 1000`
A scheduler is run at this frequency which calculates the expected heartbeats per minute
 - ✓ `eureka.server.enable-self-preservation = true`
By default self-preservation mode is enabled but if you need to disable it you can change it to 'false'

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.

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

ROUTING, CROSS CUTTING CONCERNS IN MICROSERVICES

eazy
bytes



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)

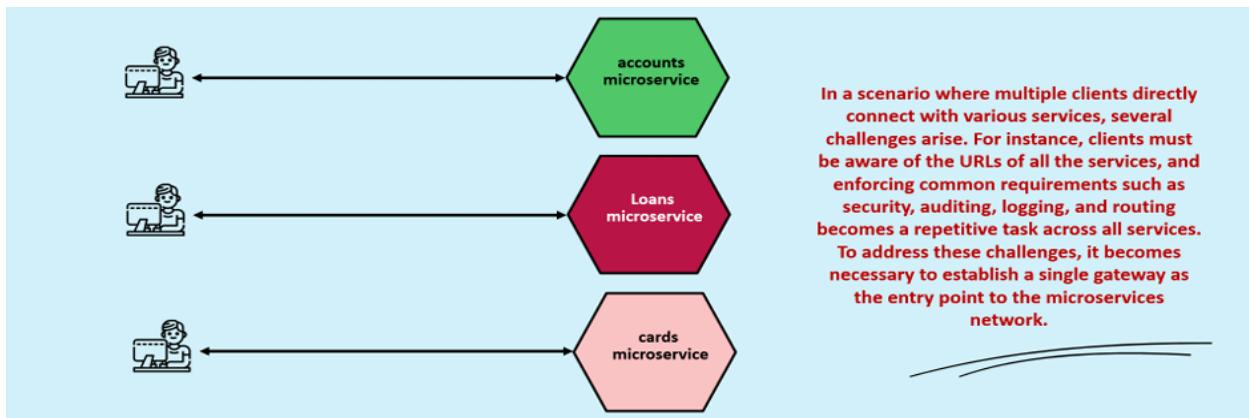
Client → User Service (<http://localhost:8081/users>)

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:

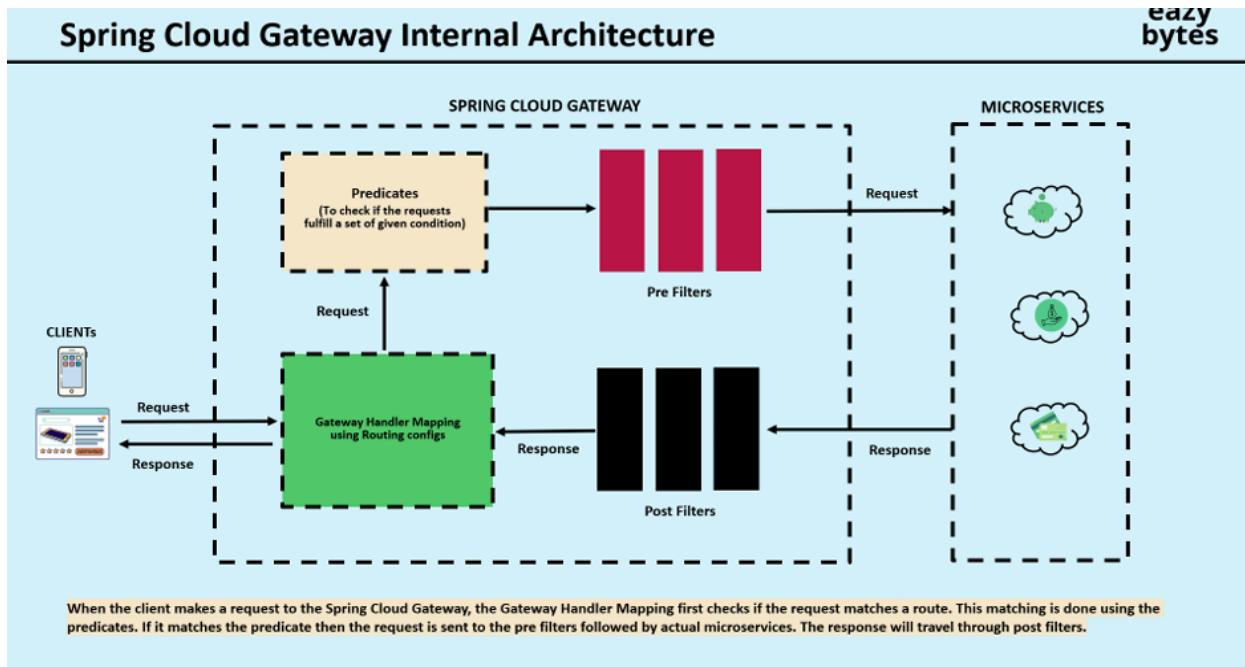
Open-source API Gateways

- 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).

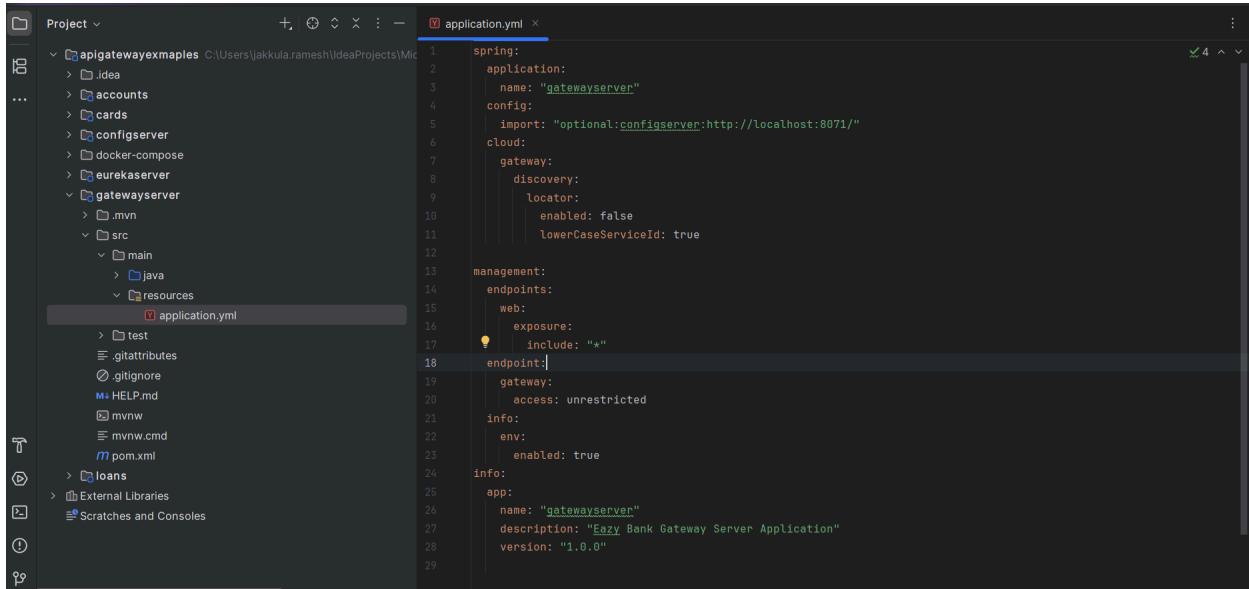
Spring Boot Actuator : For health, metrics, refresh endpoint.

Spring Boot DevTools : For auto restart during development.

Project	Language	Dependencies
<input type="radio"/> Gradle - Groovy <input type="radio"/> Gradle - Kotlin	<input checked="" type="radio"/> Java <input type="radio"/> Kotlin <input type="radio"/> Groovy	ADD DEPENDENCIES... CTRL + B
Reactive Gateway SPRING CLOUD ROUTING Provides a simple, yet effective way to route to APIs in reactive applications. Provides cross-cutting concerns to those APIs such as security, monitoring/metrics, and resiliency.		
Config Client SPRING CLOUD CONFIG Client that connects to a Spring Cloud Config Server to fetch the application's configuration.		
Eureka Discovery Client SPRING CLOUD DISCOVERY A REST based service for locating services for the purpose of load balancing and failover of middle-tier servers.		
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.		
Spring Boot DevTools DEVELOPER TOOLS Provides fast application restarts, LiveReload, and configurations for enhanced development experience.		
Project <input type="radio"/> Gradle - Groovy <input type="radio"/> Gradle - Kotlin <input checked="" type="radio"/> Maven Spring Boot <input type="radio"/> 4.0.1 (SNAPSHOT) <input type="radio"/> 4.0.0 <input type="radio"/> 3.5.9 (SNAPSHOT) <input checked="" type="radio"/> 3.5.8 <input type="radio"/> 3.4.13 (SNAPSHOT) <input type="radio"/> 3.4.12 Project Metadata Group: com.gatewayserver Artifact: gatewayserver Name: gatewayserver Description: Gateway Server project for Spring Boot! Package name: com.gatewayserver.gatewayserver Packaging: <input checked="" type="radio"/> Jar <input type="radio"/> War Configuration: <input checked="" type="radio"/> Properties <input type="radio"/> YAML Java: <input type="radio"/> 25 <input type="radio"/> 21 <input checked="" type="radio"/> 17		

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

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



The screenshot shows the IntelliJ IDEA interface with the project tree on the left and the code editor on the right. The code editor displays the `application.yml` file. The file content is as follows:

```
spring:
  application:
    | name: "gatewayserver"
  config:
    | import: "optional:configserver:http://localhost:8071/"
  cloud:
    gateway:
      discovery:
        locator:
          enabled: false
          lowerCaseServiceId: true
    management:
      endpoints:
        web:
          exposure:
            include: "*"
        endpoint:
          gateway:
            access: unrestricted
      info:
        env:
          enabled: true
      app:
        name: "gatewayserver"
        description: "Eazy Bank Gateway Server Application"
        version: "1.0.0"
```

management:

Expose all actuator endpoints

```
13   management:
14     endpoints:
15       web:
16         exposure:
17           include: "*"
```

management.endpoint.gateway.access: unrestricted

Allows access to sensitive actuator gateway endpoints

like `/actuator/gateway/routes`, `/actuator/gateway/globalfilters`. Otherwise, they are restricted.

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

spring.cloud.gateway.discovery.locator

```
1  spring:
2    application:
3      name: "gatewayserver"
4    config:
5      import: "optional:configserver:http://localhost:8071/"
6    cloud:
7      gateway:
8        discovery:
9          locator:
10            enabled: false
11            lowerCaseServiceId: true
```

enabled: false

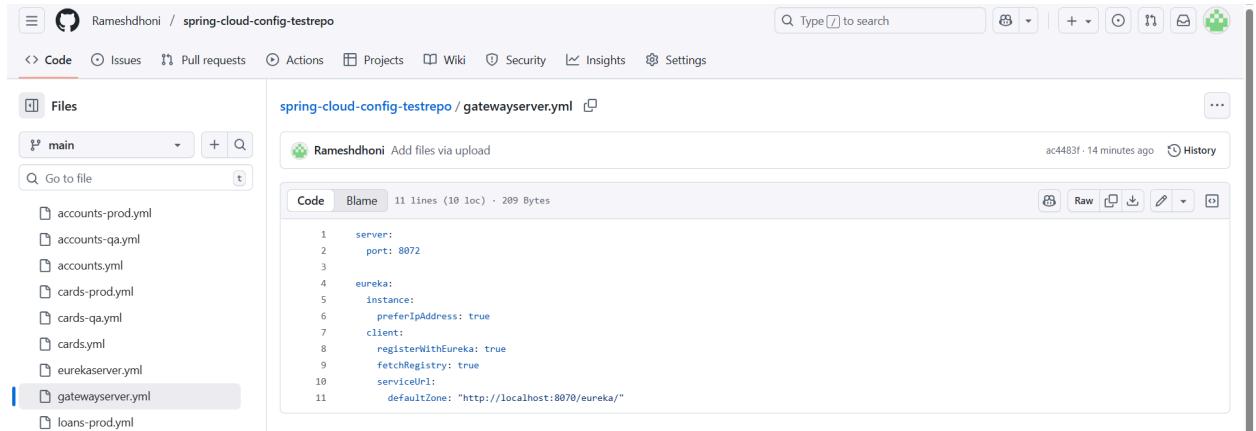
- If *true*, Gateway automatically creates routes for each Eureka service.
- If *false*, you will write routes manually in YAML.

lowerCaseServiceId: true

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

ACCOUNTS → accounts

LOANS → loans



```
1   server:
2     port: 8072
3
4   eureka:
5     instance:
6       preferIpAddress: true
7     client:
8       registerWithEureka: true
9       fetchRegistry: true
10      serviceUrl:
11        defaultZone: "http://localhost:8070/eureka/"
```

Steps to create Spring Cloud Gateway

Lazy
bytes

Below are the steps to make a microservice application to register and act as a Eureka client,

- 1 Set up a new Spring Boot 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-starter-gateway**, **spring-cloud-starter-config** & **spring-cloud-starter-netflix-eureka-client** maven dependencies.

- 2 Configure the properties: In the application properties or YAML file, add the following configurations. Make routing configurations using **RouteLocatorBuilder**

```
eureka:  
  instance:  
    preferIpAddress: true  
  client:  
    registerWithEureka: true  
    fetchRegistry: true  
    serviceUrl:  
      defaultZone: http://localhost:8070/eureka/  
  spring:  
    cloud:  
      gateway:  
        discovery:  
          locator:  
            enabled: true  
            lowerCaseServiceId: true
```

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

The screenshot shows the Eureka dashboard at localhost:8070. It displays the following sections:

- Data center**: Shows basic statistics for the default data center.

Data center	default	Uptime	00:22
		Lease expiration enabled	false
		Renews threshold	8
		Renews (last min)	7

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.
- DS Replicas**: Shows instances currently registered with Eureka.

Application	AMIs	Availability Zones	Status
ACCOUNTS	n/a (1)	(1)	UP (1) - host.docker.internal:accounts:8080
CARDS	n/a (1)	(1)	UP (1) - host.docker.internal:cards:9000
GATEWAYSERVER	n/a (1)	(1)	UP (1) - host.docker.internal:gatewayserver:8072
LOANS	n/a (1)	(1)	UP (1) - host.docker.internal:loans:8090
- General Info**: Shows system-level information.

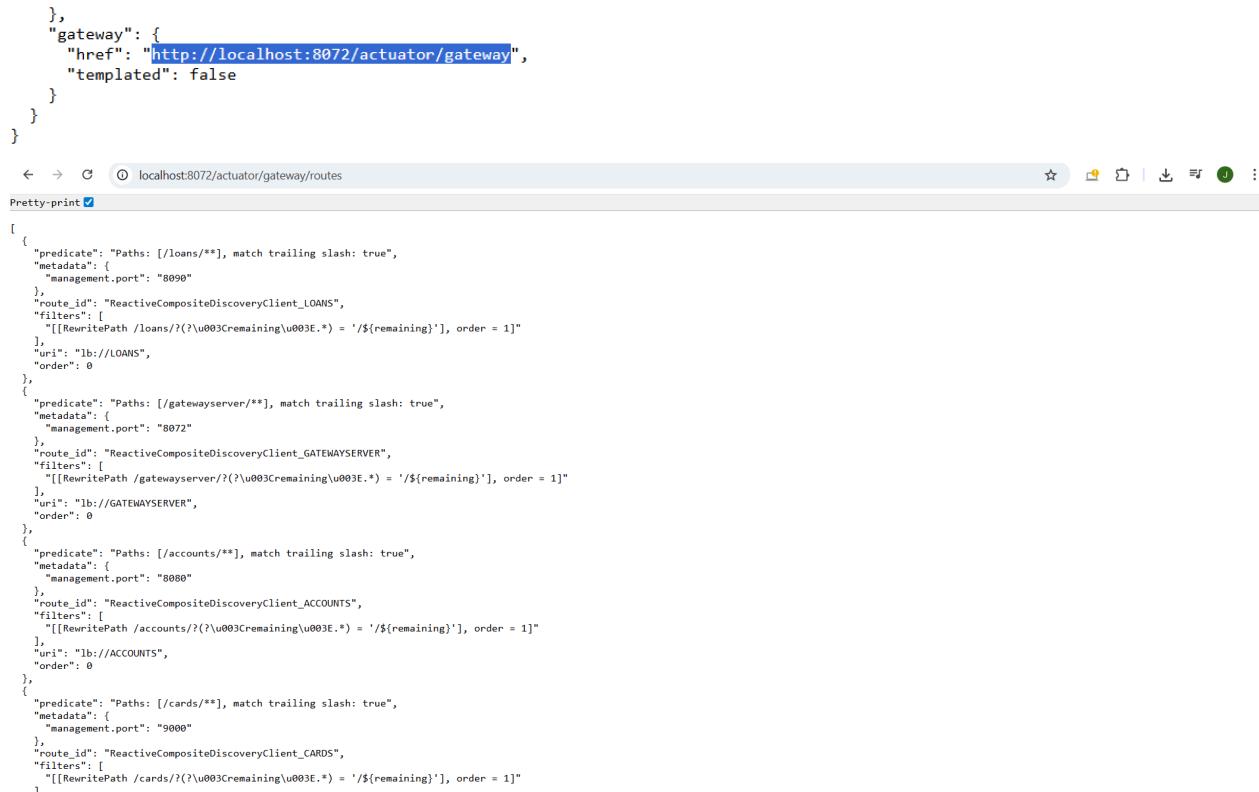
Name	Value
total-avail-memory	112mb

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 .

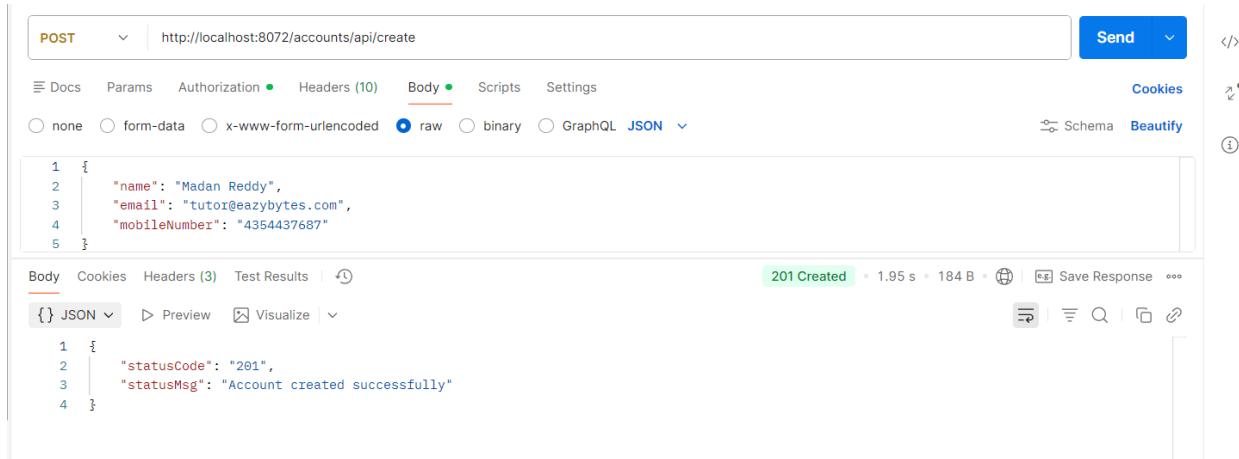


```
{  
  "gateway": {  
    "href": "http://localhost:8072/actuator/gateway",  
    "templated": false  
  }  
}  
  
[  
  {  
    "predicate": "Paths: [/loans/**], match trailing slash: true",  
    "metadata": {  
      "management.port": "8090"  
    },  
    "route_id": "ReactiveCompositeDiscoveryClient_LOANS",  
    "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:



The screenshot shows a Postman request to `http://localhost:8072/accounts/api/create`. The request method is `POST`. The request body is a JSON object with fields: `"name": "Madan Reddy"`, `"email": "tutor@eazybytes.com"`, and `"mobileNumber": "4354437687"`. The response status is `201 Created`, and the response body is `{"statusCode": "201", "statusMsg": "Account created successfully"}`.

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/functional-java-routes-api.html>

```
spring:  
  cloud:  
    gateway:  
      routes:  
        - id: accounts_route  
          uri: lb://ACCOUNTS  
          predicates:  
            - Path=/bank/accounts/**  
          filters:  
            - RewritePath=/bank/accounts/(?<segment>.*), /${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

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

What happens?

Step	Meaning
Path match	When request URL starts with <code>/eazybank/accounts/**</code>
Rewrite	Remove <code>/eazybank/accounts</code> 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/(?<segment>.*)", "/${segment}")
                .addResponseHeader("X-Response-Time", new Date().toString()))
            .uri("lb://ACCOUNTS"))
        .route(p -> p
            .path("/eazybank/loans/**")
            .filters(f -> f.rewritePath("/eazybank/loans/(?<segment>.*)", "/${segment}")
                .addResponseHeader("X-Response-Time", new Date().toString()))
            .uri("lb://LOANS"))
        .route(p -> p
            .path("/eazybank/cards/**")
            .filters(f -> f.rewritePath("/eazybank/cards/(?<segment>.*)", "/${segment}")
                .addResponseHeader("X-Response-Time", new Date().toString()))
            .uri("lb://CARDS")).build();
}
```

4

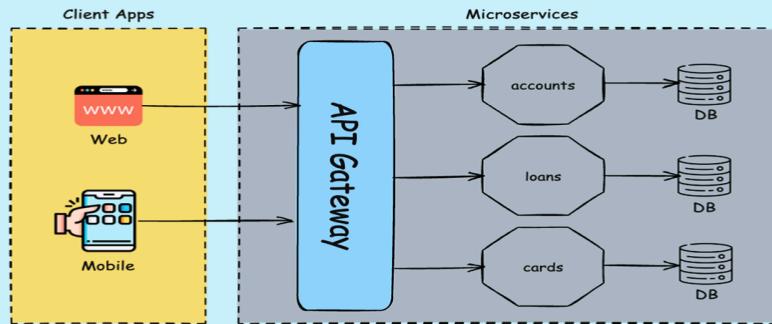
Build and run the application: Build your project and run it as a Spring Boot application. Invokes the APIs using <http://localhost:8072> which is the gateway path.

Design Patterns around api gateway

API Gateway Pattern

eazy bytes

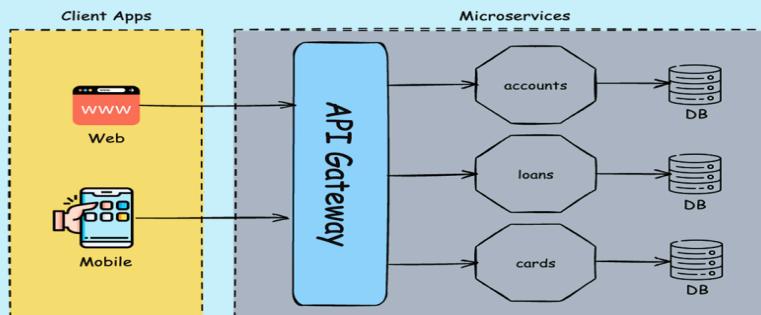
The API Gateway Pattern is a critical architectural component in microservices design, offering a unified entry point for multiple microservices. It acts as a gateway between the external clients (e.g., web apps, mobile apps) and the internal microservices, helping streamline communication, security,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



Gateway Routing pattern

eazy bytes

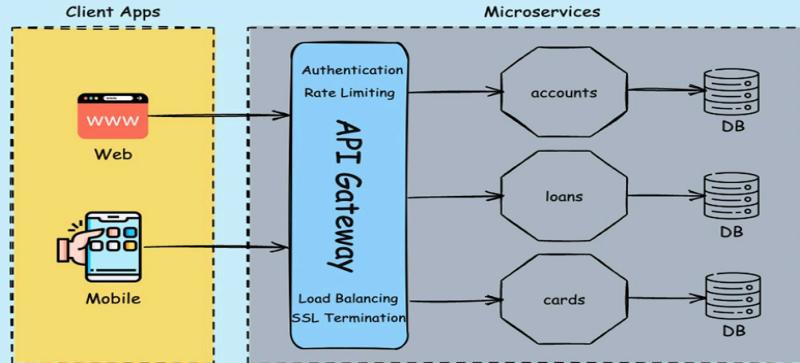
The Gateway Routing pattern is a design pattern used in microservices architectures where an API Gateway routes incoming client requests to the appropriate backend microservices based on various factors like the URL, headers, or request parameters.



Gateway offloading Pattern

eazy bytes

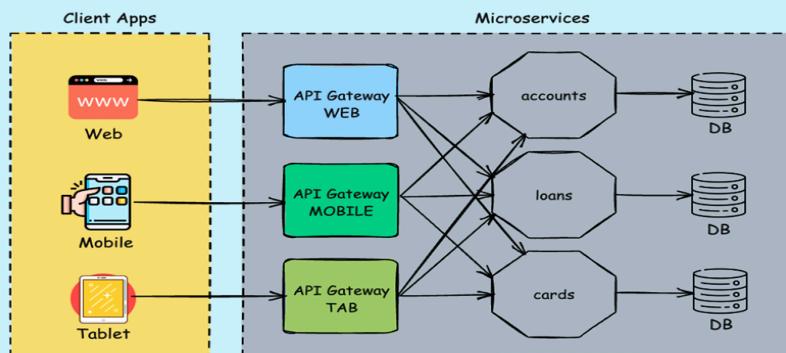
The Gateway Offloading Pattern is an architectural pattern used in microservices to offload certain cross-cutting concerns—such as security, caching, rate limiting, and monitoring—from individual microservices to the API Gateway. This pattern helps centralize and simplify the implementation of these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



Backend For Frontend (BFF) Pattern

eazy bytes

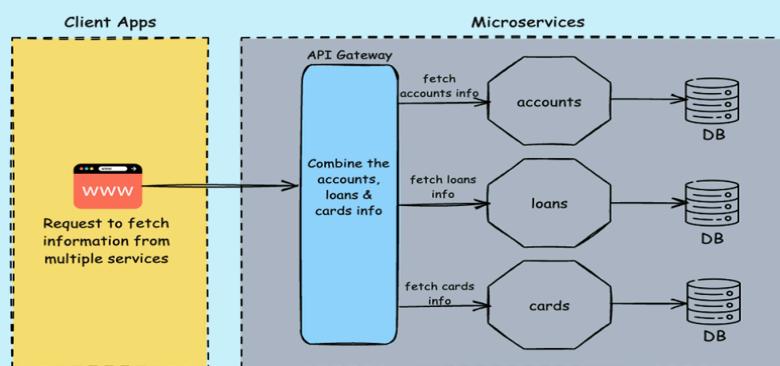
The Backend For Frontend (BFF) Pattern is a design pattern used in microservices architectures where a separate backend service is created for each client type (e.g., web, mobile, tablet). Each frontend (client) has its own specialized backend to optimize communication between the frontend and the microservices,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

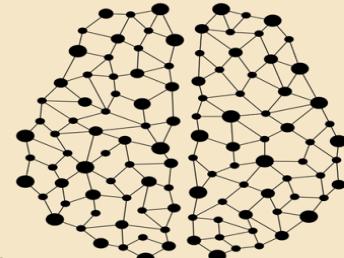
In the Java ecosystem, **Hystrix**, a fault-tolerance library developed by Netflix, was once widely used for implementing these resilience patterns. However, Hystrix entered **maintenance mode in 2018** and is no longer actively developed.

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

Resilience4j is a lightweight fault tolerance library designed for functional programming. It offers the following patterns for increasing fault tolerance due to network problems or failure of any of the multiple services:

- Circuit breaker** - Used to stop making requests when a service invoked is failing
- Fallback** - Alternative paths to failing requests
- Retry** - Used to make retries when a service has temporarily failed
- Rate limit** - Limits the number of calls that a service receives in a time
- Bulkhead** - Limits the number of outgoing concurrent requests to a service to avoid overloading



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.

