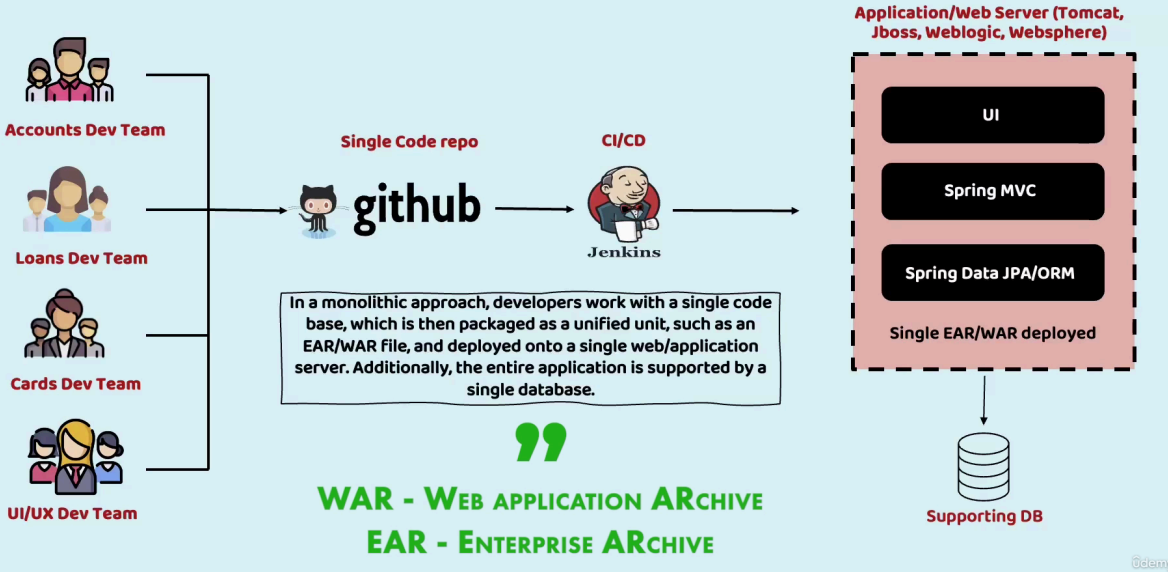
Develop Microservices with Java, Spring Boot, Spring Cloud, Docker, Kubernetes, Helm, Microservices Security

The Monolith

eazy bytes



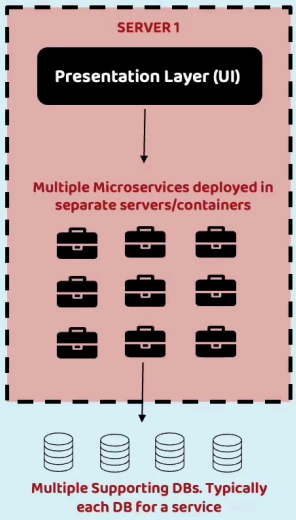
In a monolithic approach, developers work with a single code base, which is then packaged as a unified unit, such as an EAR/WAR file, and deployed onto a single web/application server. Additionally, the entire application is supported by a single database.

WAR - Web application ARchive

EAR - Enterprise ARchive

The GREAT MICROSERVICES

eazy bytes



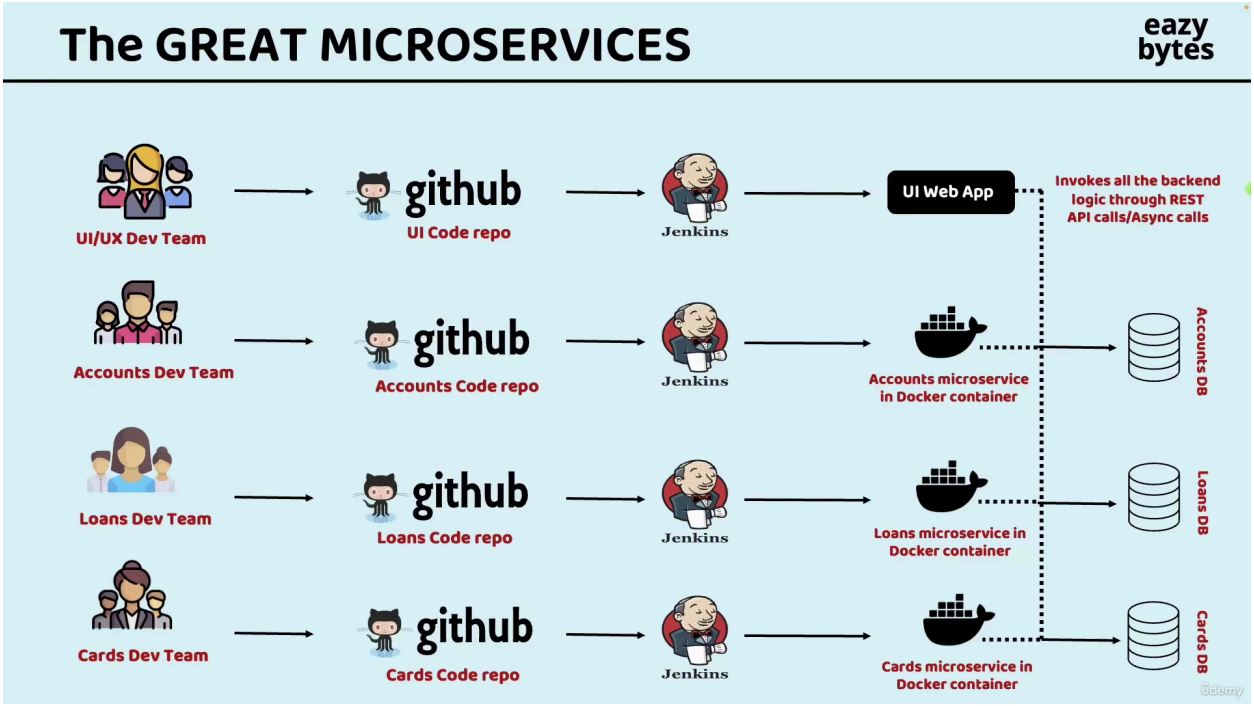
Microservices are independently releasable services that are modeled around a business domain. A service encapsulates functionality and makes it accessible to other services via networks—you construct a more complex system from these building blocks. One microservice might represent Accounts, another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

Pros

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

Cons

Complexity
Infrastructure overhead
Security concerns



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

Monolithic Application

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

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

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

🛒 Example: Online Shopping Application

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

This app has several features:

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

In a **monolithic architecture**:

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

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

Microservices Application

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

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

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

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

🛒 Example: Online Shopping Application

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

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

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

Each of these services is:

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

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

—---

Sample Dependencies

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

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

—----------

Spring Boot DevTools

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

Here's a breakdown to make it clear 👇

⚙️ What Spring Boot DevTools Does

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

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

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

🚫 Why Not Use It in Production

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

💡 Typical Usage

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

Spring Boot – schema.sql & data.sql Summary

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

- **Important Rule:**

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

spring.jpa.hibernate.ddl-auto=none

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

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

spring.jpa.hibernate.ddl-auto=create

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

- **Common ddl-auto Options:**

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

1. Development Environment

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

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

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

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

2. Testing Environment

- **Goal:** Consistency, repeatability.

- **Approaches:**

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

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

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

3. Production Environment

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

- **Recommended Approach:**

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

        - Versioned scripts (V1__create_tables.sql, etc.)

        - Track applied migrations and safely apply only new changes.

        - Ensures explicit, reviewable, and reversible schema changes.

- **Not Recommended:**

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

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

✅ **Key Principle:**

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

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

Springdoc OpenAPI

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

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

Getting Started

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

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

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

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

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

Run and Access the Docs

Once you start your Spring Boot app, open:

👉 **Swagger UI:**

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

👉 **OpenAPI JSON/YAML:**

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

Annotations

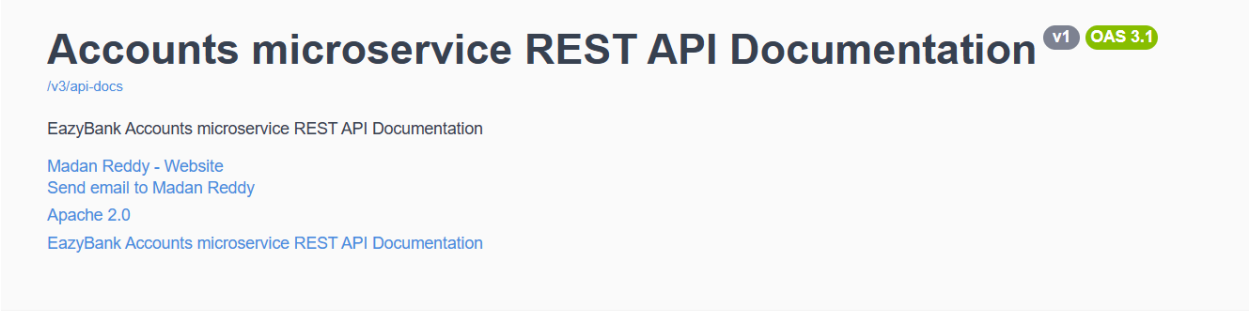
@OpenAPIDefinition

Where: In your AccountsApplication.java (main class)

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

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

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

}
```

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



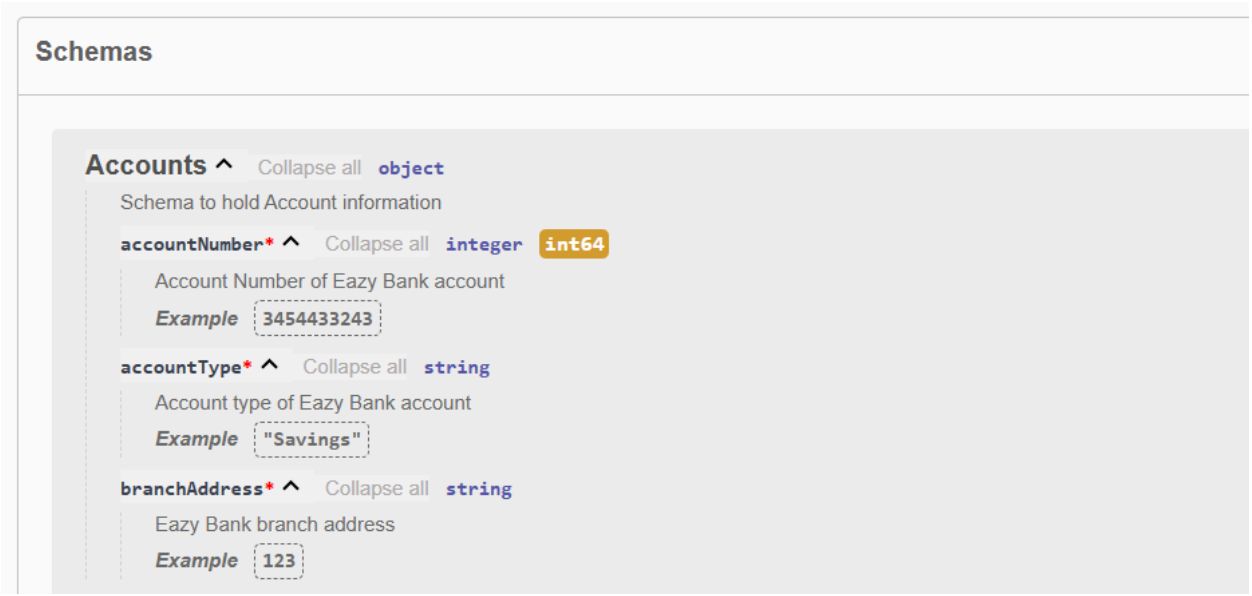
@Schema

Where: On your AccountsDto.java

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

package com.eazybytes.accounts.dto;

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

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

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

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

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



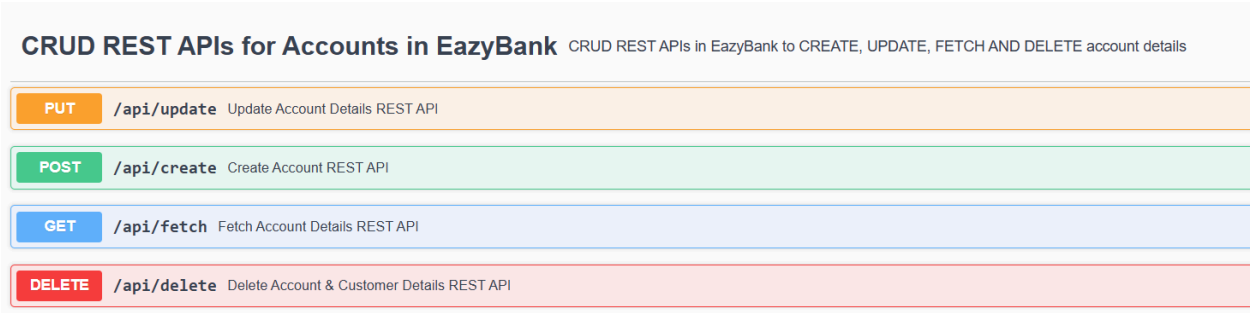
💡 **Helps with:**
In Swagger UI → displays field descriptions and sample values when showing request/response schemas.

@Tag

Where: On top of your controller.

Purpose:
Groups related endpoints into sections for better readability.

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



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

CRUD REST APIs for Accounts in EazyBank

and lists all related endpoints underneath.

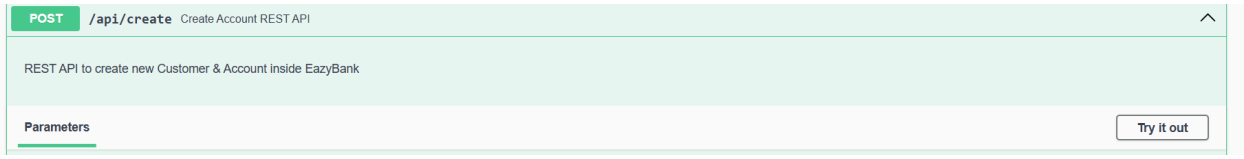
@Operation

Where: Above each controller method.

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

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

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



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

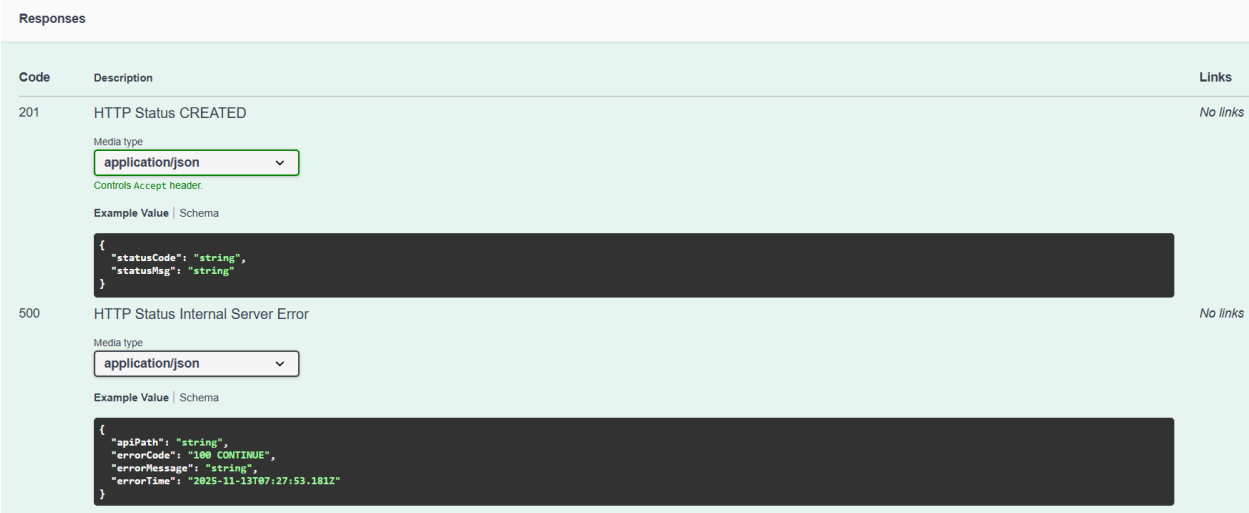
@ApiResponse and @ApiResponses

Where: Above each controller method.

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

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

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



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

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

@Parameter

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

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

In your code:

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

You could optionally add:

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

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

more refer:

Spring Boot validation annotations

Add dependency

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

Step 2 — Use @Valid in Controller

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

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

And also

```java
@Validated
public class AccountsController {

    private IAccountsService iAccountsService;
```

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

Step 3 — Annotate DTO Fields

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

A. Null and Empty Checks

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

🔢 B. Numeric Validations

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

🧾 C. String & Pattern Validations

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

📅 D. Date Validations

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

Example DTO with Validations

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


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

Exception Handling

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

extends ResponseEntityExceptionHandler

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

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

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

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

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

Generic Exception — handleGlobalException

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

Custom Exceptions — ResourceNotFoundException

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

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

Sample Project Structure

Recommended Spring Boot Project Structure

pgsql

Copy code

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

@MappedSuperclass in Spring Boot / JPA

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

🧩 Simple meaning:

It's like saying:

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

🧱 **Example**

```
import jakarta.persistence.MappedSuperclass;

import jakarta.persistence.Column;

import java.time.LocalDateTime;

@MappedSuperclass

public abstract class BaseEntity {

    @Column(name = "created_at")

    private LocalDateTime createdAt;

    @Column(name = "updated_at")

    private LocalDateTime updatedAt;

    // getters and setters

}
```

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

🧠 **Usage Example**

```
import jakarta.persistence.Entity;

import jakarta.persistence.Id;

@Entity

public class User extends BaseEntity {

    @Id

    private Long id;
```

```
    private String username;
}
```

✅ The **user** table will have:

id

username

created_at

updated_at

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

💡 **When to Use**

Use @MappedSuperclass when you have:

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

—----------

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

The old way — using @Autowired

```
@RestController
public class CustomerController {

    @Autowired
```

```
    private CustomerService customerService;

    // endpoints

}
```

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

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

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

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

✅ Advantages:

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

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

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

What happens behind the scenes

When you use @AllArgsConstructor (from Lombok):

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

➡️ Lombok generates this constructor for you:

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

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

—----------

Package for constants/utility

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

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

Ex

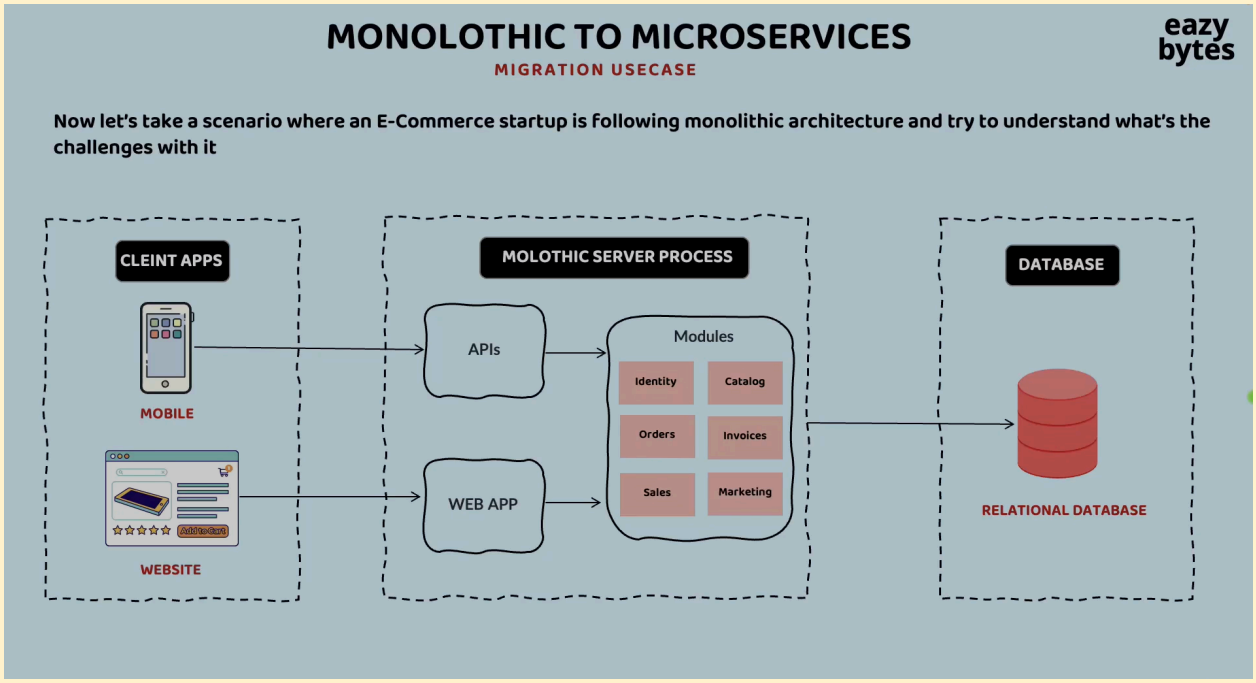
```java
package com.eazybytes.accounts.constants;

public final class AccountsConstants {

    private AccountsConstants() {
        // restrict instantiation
    }

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

}
```

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

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

Monolith → Microservices (Migration)

Monolithic Architecture (Before Migration)



- All modules like:
    - Identity

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

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

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

Advantages (initial days)

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

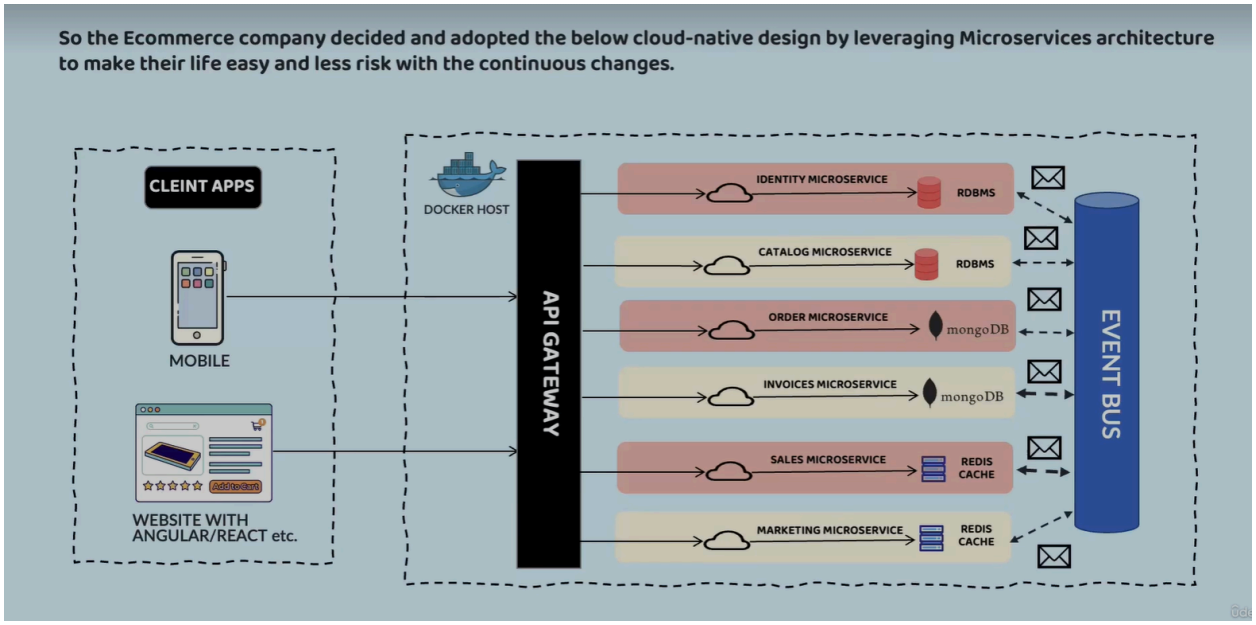
Problems after the app grows (from the second diagram)

Once the system becomes large:

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

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

Migration to Microservices (After Migration)



Key elements of the new design

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

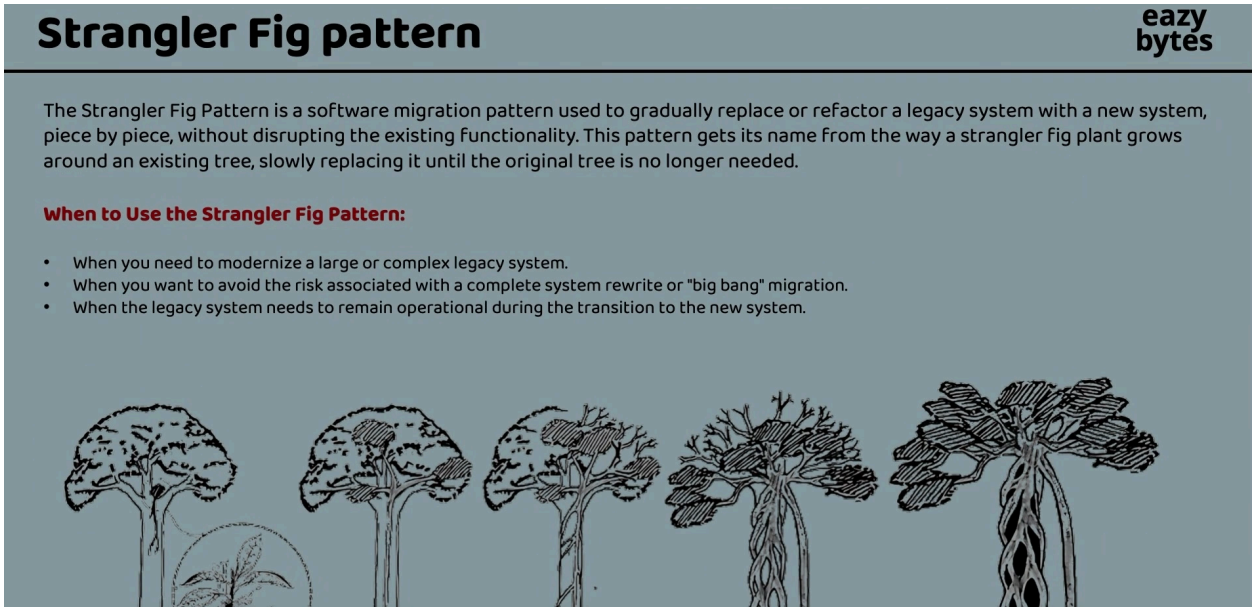
Each microservice:
✔ Has its own database
✔ Runs independently (usually Docker containers)
✔ Can be deployed independently

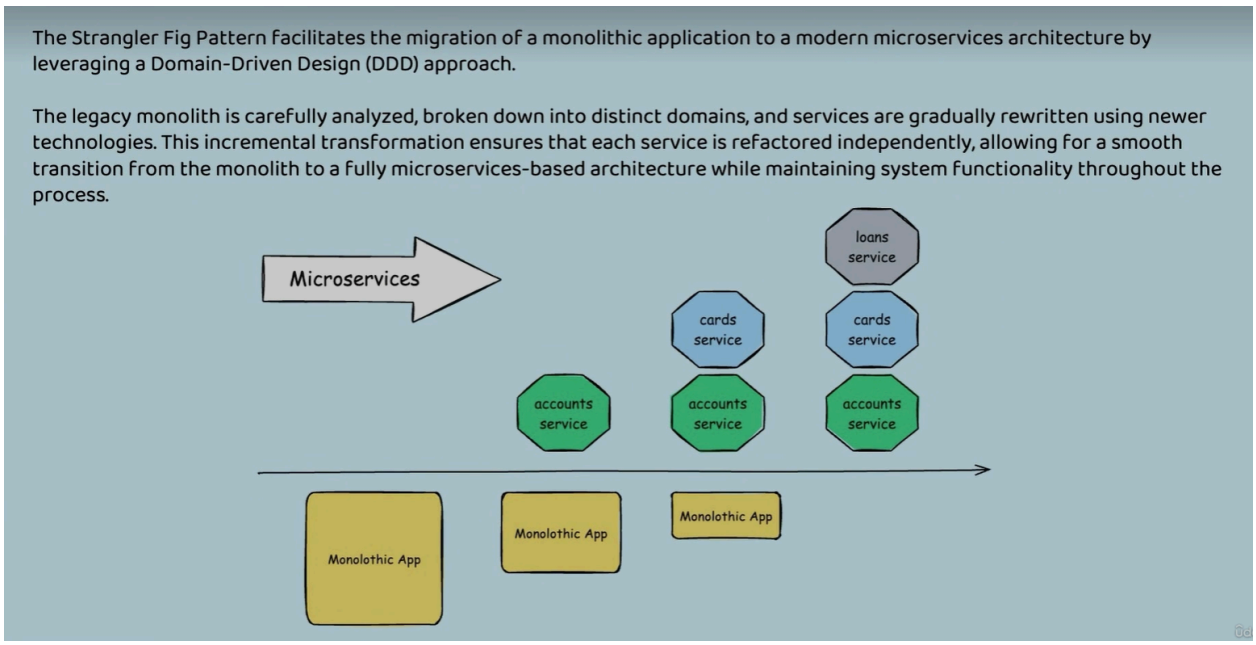
Event Bus

- Used for **async communication**

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

Strangler Fig Pattern



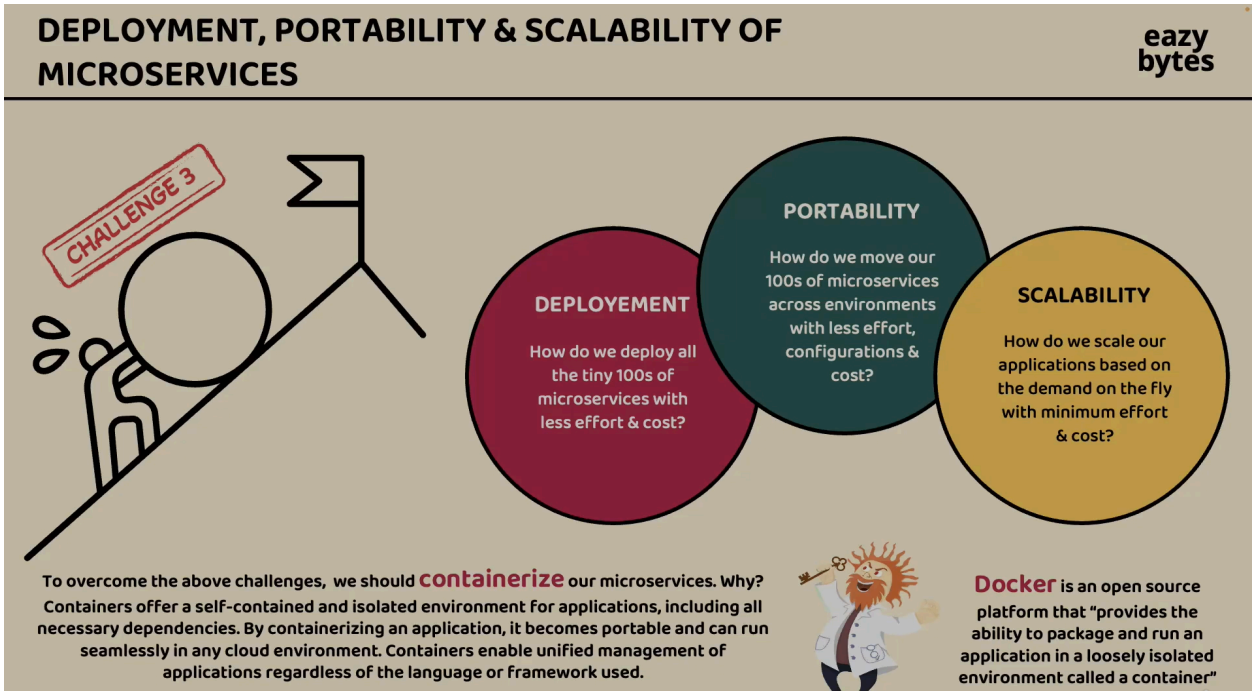


Strangler Fig Pattern migration:

✔ Identification
✔ Transformation
✔ Co-existence
✔ Elimination

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

Deployment ,portability and scaling of microservices



1. Deployment (Before vs After Containers)

🔴 **Before Containers**

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

❌ **Example**

To deploy a Payments service:

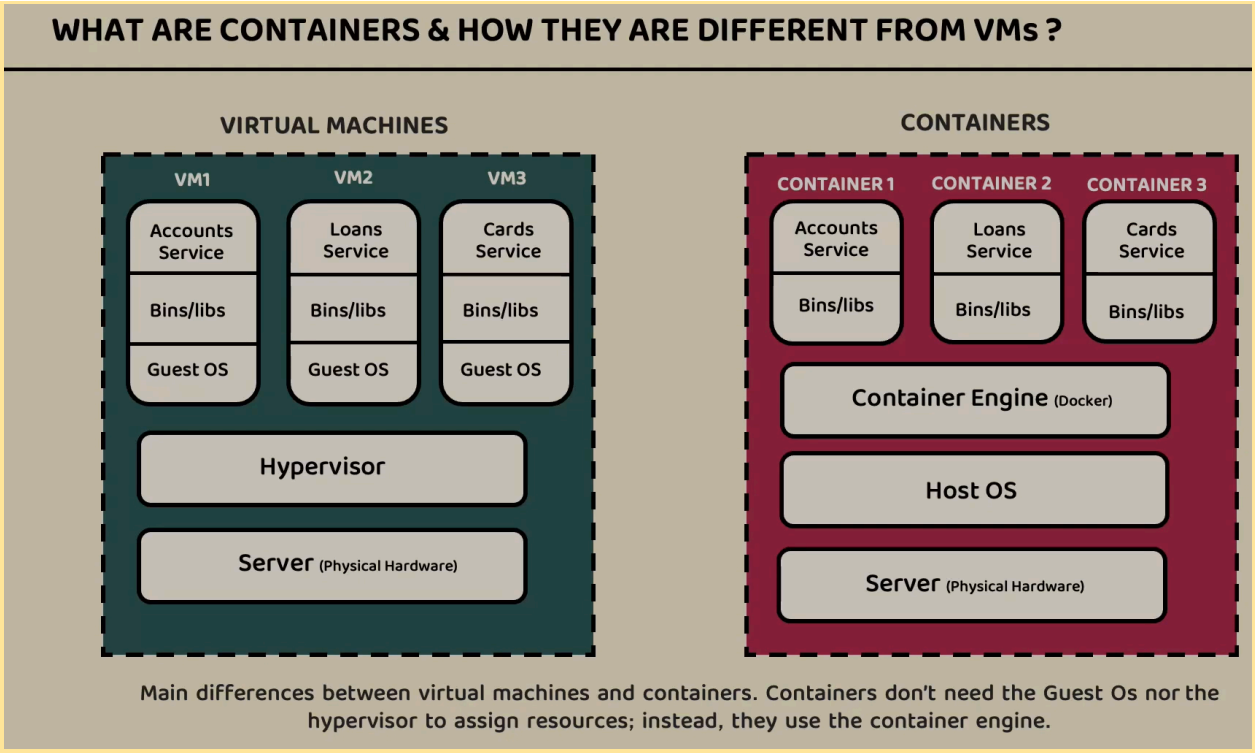
- Install JDK manually on the server
- Install Maven
- Install dependent libraries
- Configure environment variables
- Run JAR manually

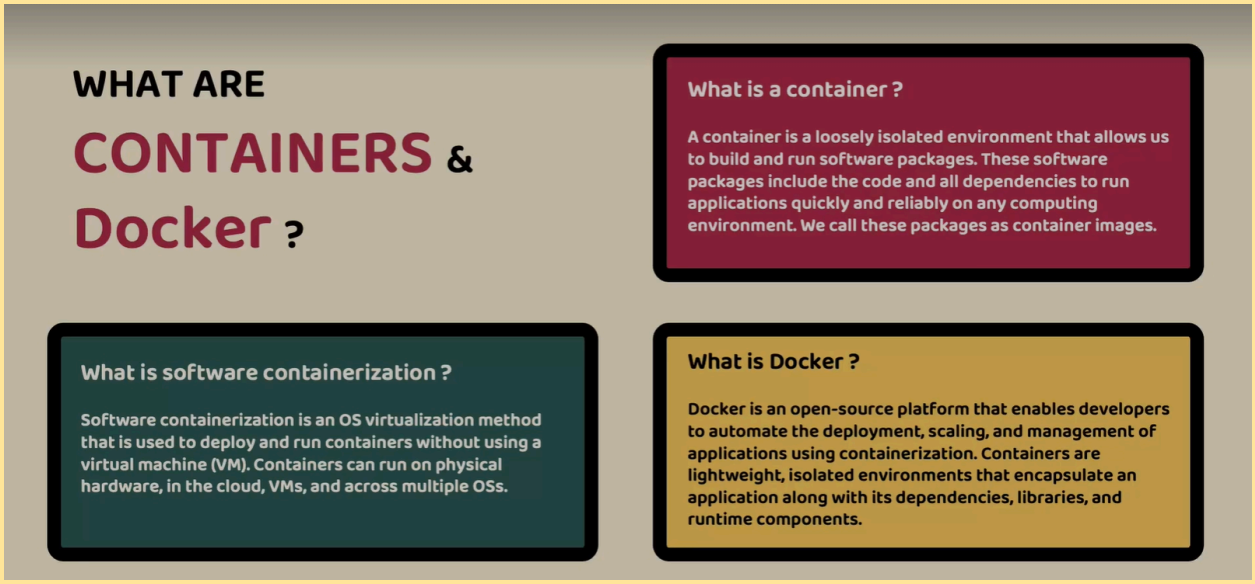
🟢 **After Containers**

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

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

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





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

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

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

Key Characteristics

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

✔ In Simple Terms

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

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

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

📌 Key Points

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

✔ Simple Meaning

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

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

Docker Engine → runs containers

Dockerfile → defines container blueprint

Docker Hub → stores and shares container images

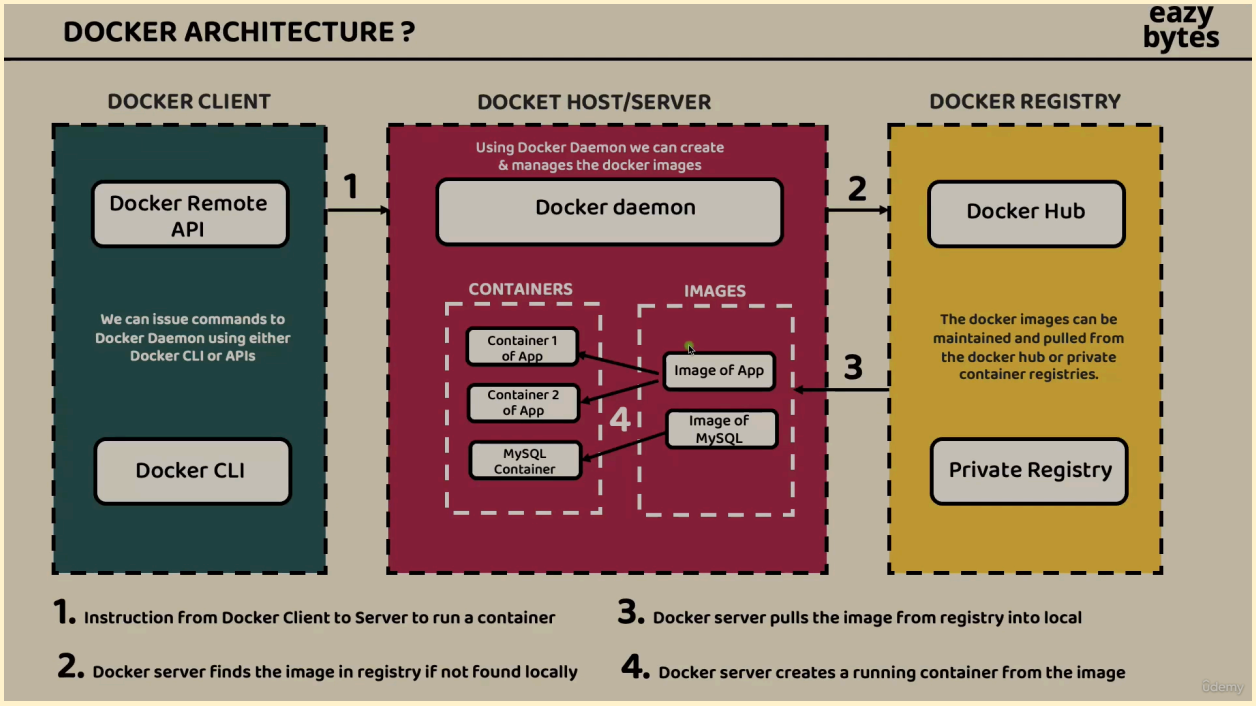
Works with Kubernetes for orchestration

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

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

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

Docker Architecture



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

When you run commands like:

docker build

docker pull

docker run

2. Docker Host:

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

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

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

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

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

Docker Hub (default)

Amazon ECR

GitHub Container Registry

Azure Container Registry

Google Artifact Registry

Private registry

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

Docker is open source platform

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

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

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

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

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

Docker CLI (Client)

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

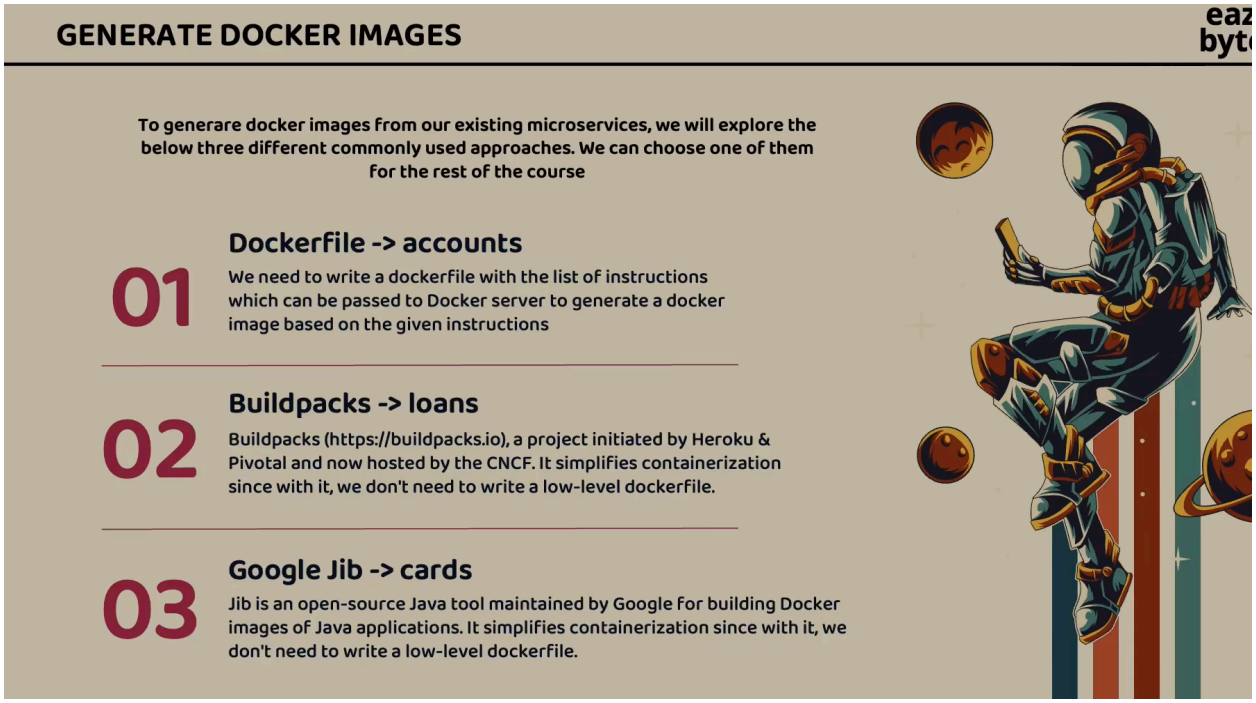
Docker Host (Server / Engine / Daemon)

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

Docker Registry

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

Generate Docker images



Spring Boot 2.x requires a minimum of Java 8

Spring Boot 3.x requires:

✔ Java 17 or higher

✔ Maven 3.6.3 or higher

Basic things before generating docker images

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

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

   <packaging>jar</packaging>

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

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

2. Add Spring Boot Plugin (if missing)

Spring Boot requires this plugin to create a runnable JAR:

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

Without this, the JAR may not start properly.

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

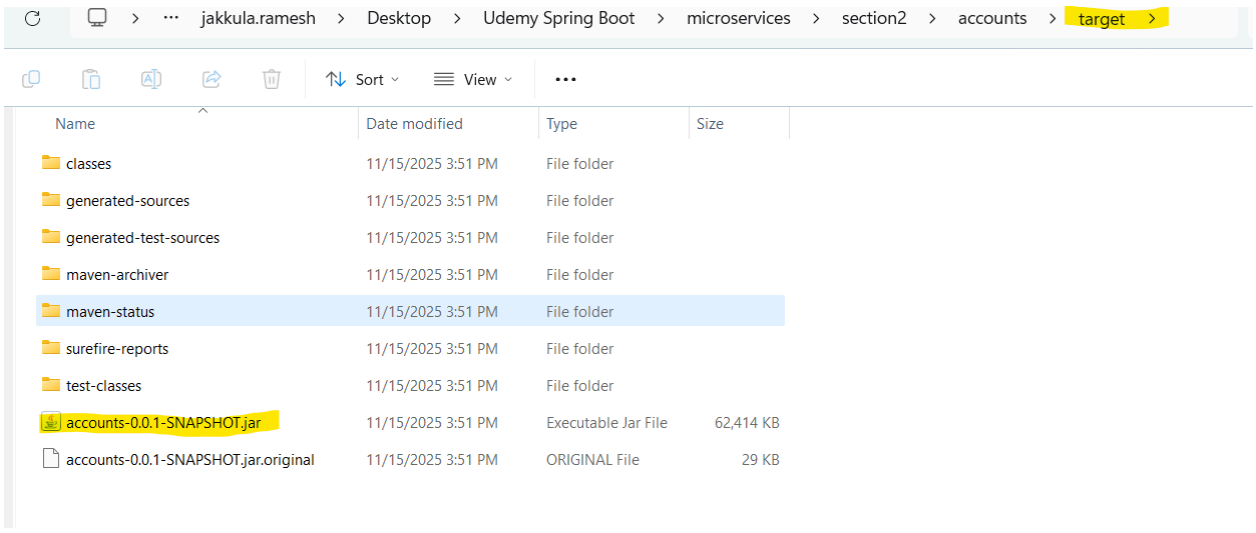
```
mvn clean install
```

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

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

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

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



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

i) Maven command to run Spring Boot

mvn spring-boot:run

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

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

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

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

ii) using the “.jar” file

run:

java -jar target/yourappname-version.jar

Example:

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

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

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

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

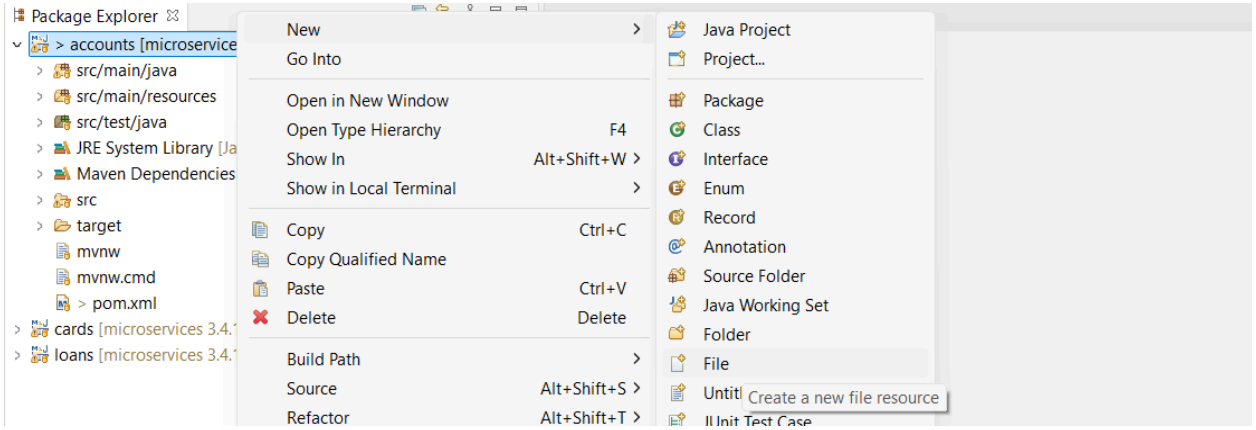
```
2025-11-15T16:00:18.129+05:30  WARN 24252 --- [  restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Th
erefore, database queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2025-11-15T16:00:18.926+05:30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

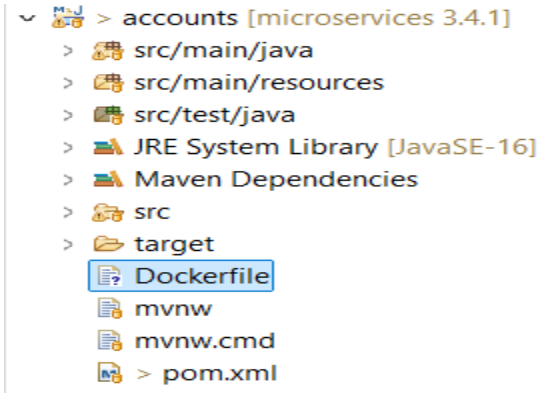
iii) In any IDE directly.

1) using Dockerfile

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

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





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

Dockerfile

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

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

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

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

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

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

# Execute the application


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

```

FROM openjdk:17-jdk-slim

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

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

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

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

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

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

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

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

   Make sure this is also the folder containing your Dockerfile.

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

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

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

Example:

docker build -t jram224/accounts:1.0 .

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

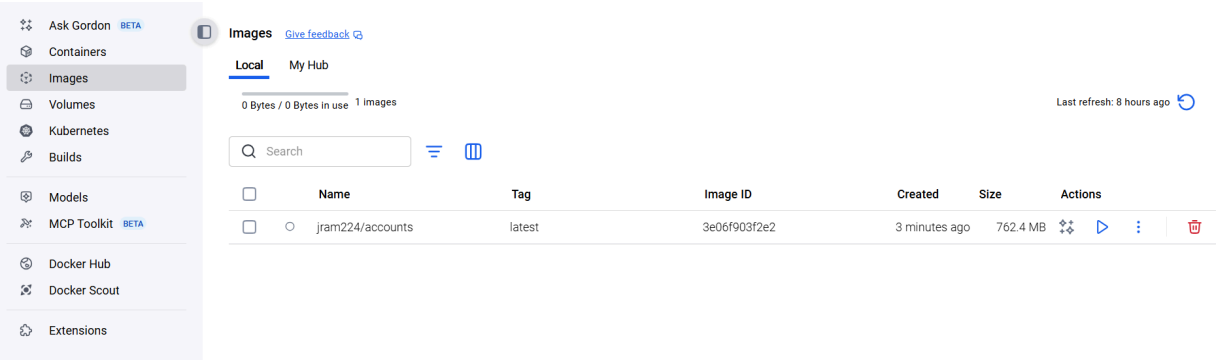
```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>docker build . -t jram224/accounts:latest
[+] Building 54.1s (7/7) FINISHED                                                                                  docker:desktop-linux
 => [internal] load build definition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

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

   docker images

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

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



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

Inspect detailed image info

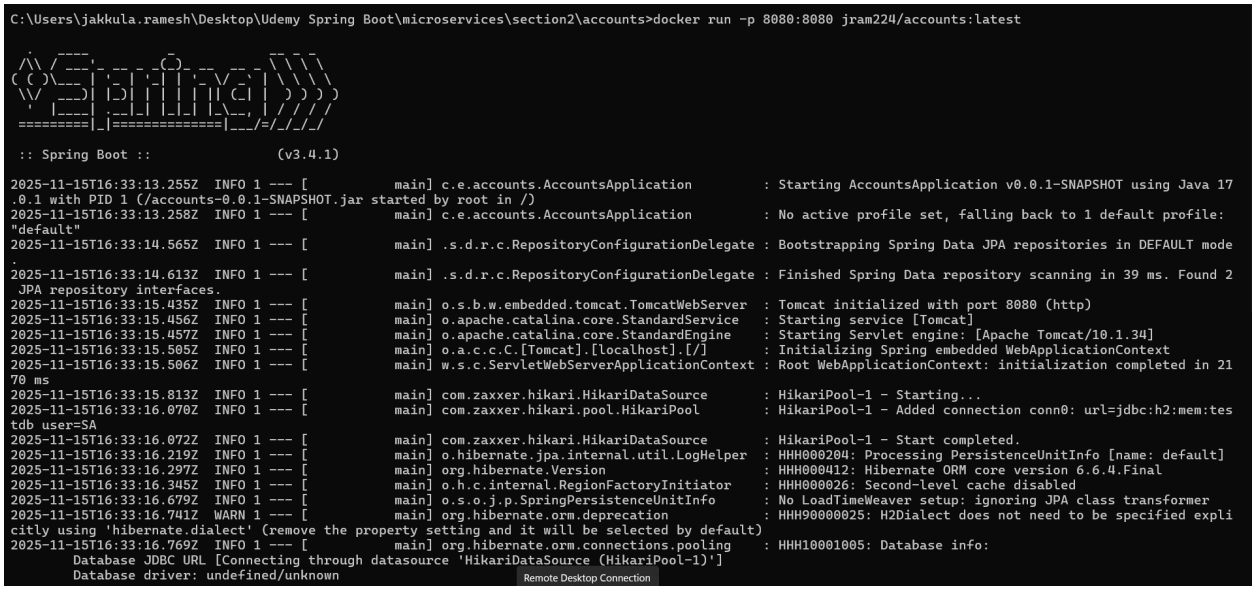
docker image inspect <image-id>

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

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

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

5. Run Docker Container Using the Image

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



Step-by-Step Explanation

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

Running Docker Container in Detached Mode

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

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

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

just add -d to your normal docker run command

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

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

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

We can check running containers with:

docker ps

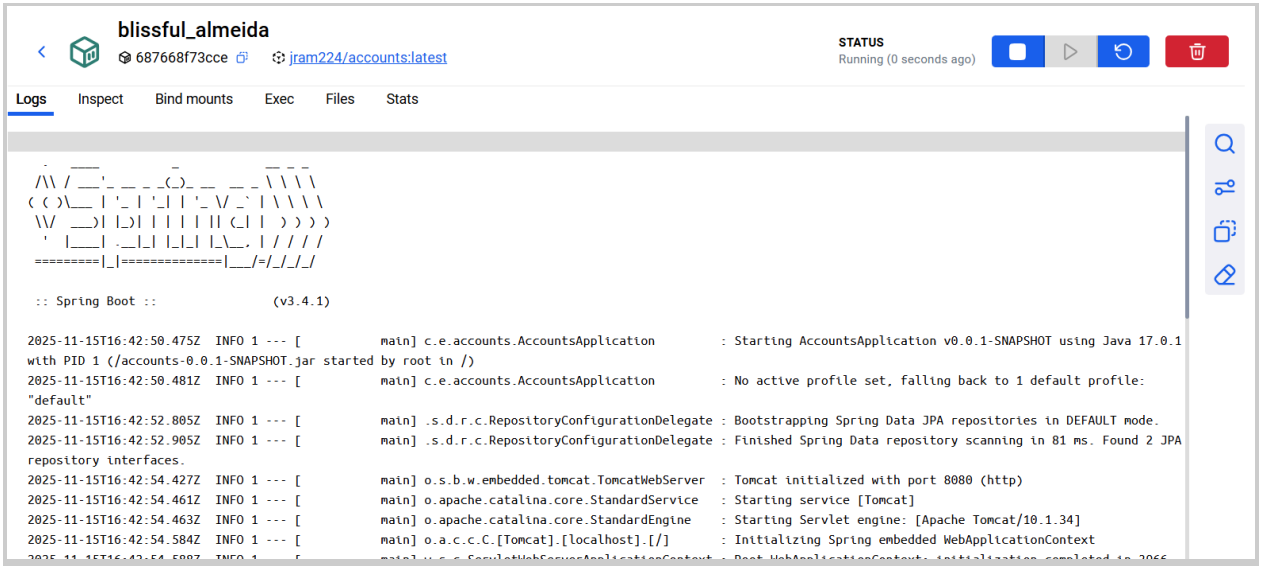
```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\accounts>docker ps
CONTAINER ID   IMAGE                    COMMAND                  CREATED         STATUS         PORTS                                       NAMES
9a3c912e1692   jram224/accounts:latest   "java -jar accounts-…"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

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

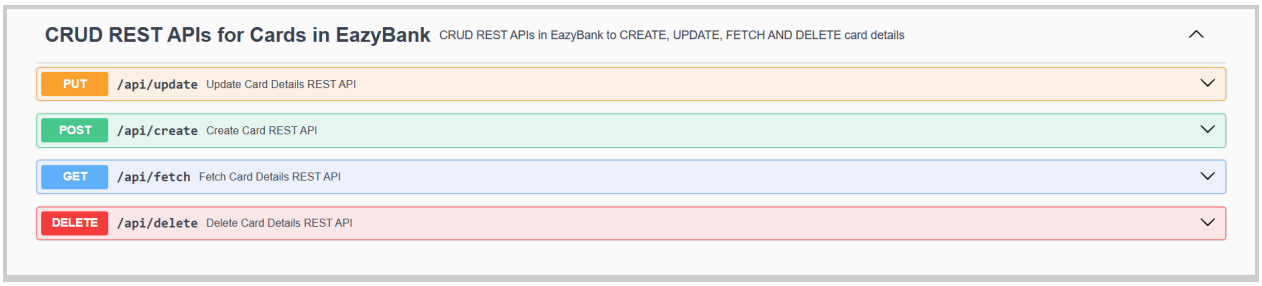
And stop them with:

docker stop <container_id>

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

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

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

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





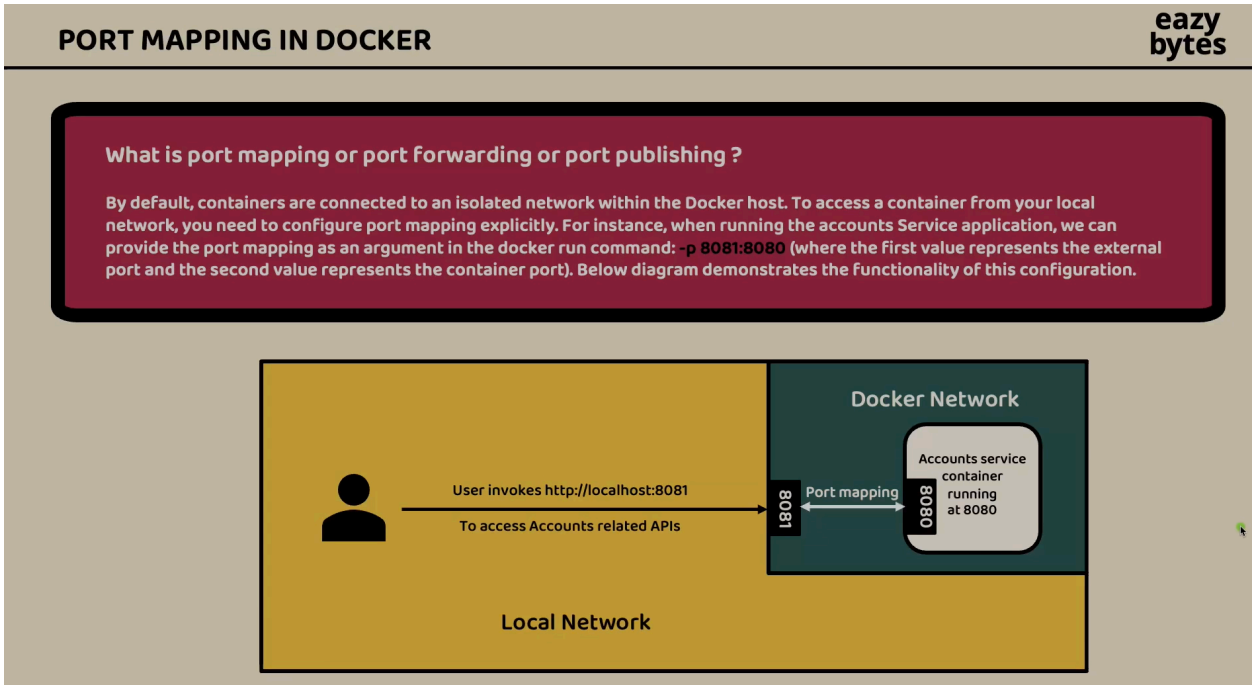
docker run → always creates a **new container** from the image.

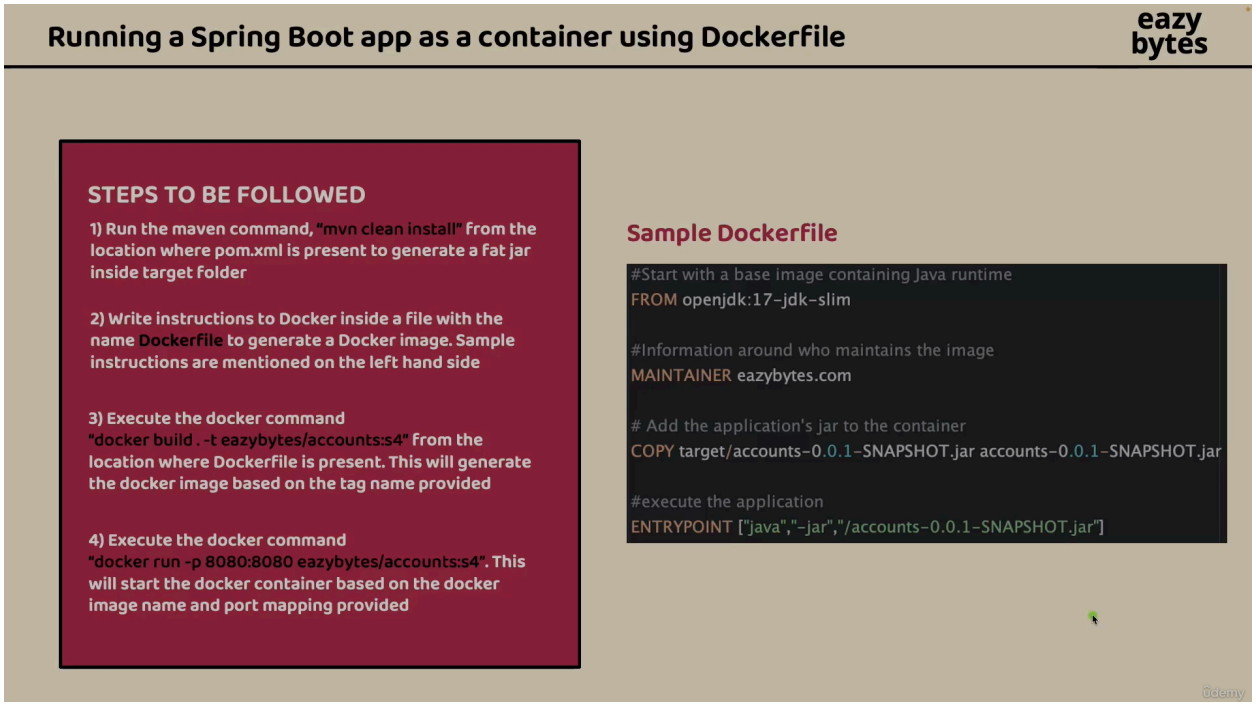
docker ps -a → lists all containers (running or stopped).

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

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

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

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

Finally





Challenges with Dockerfile Approach

Steep Learning Curve

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

Maintenance & Updates

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

Not Developer-Focused

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

Error-prone

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

Alternatives

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

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

2) using Buildpacks

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

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

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

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

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

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

Run the Buildpack Docker image command

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

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

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

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

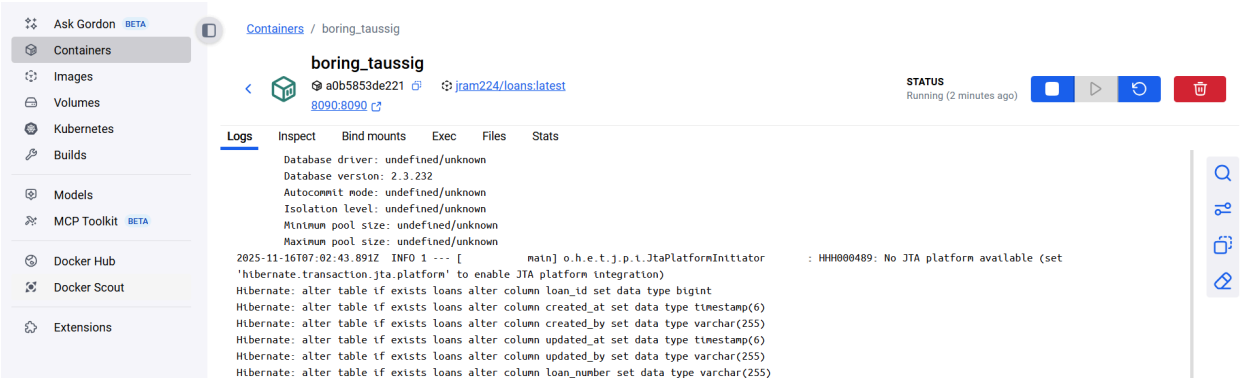
iii) Run Docker Container Using the Image

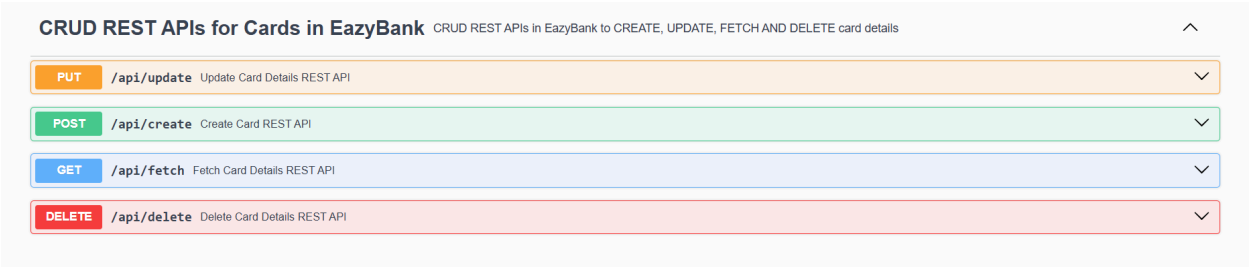
docker run -d -p <host-port>:<container-port> <image-name>:<tag>

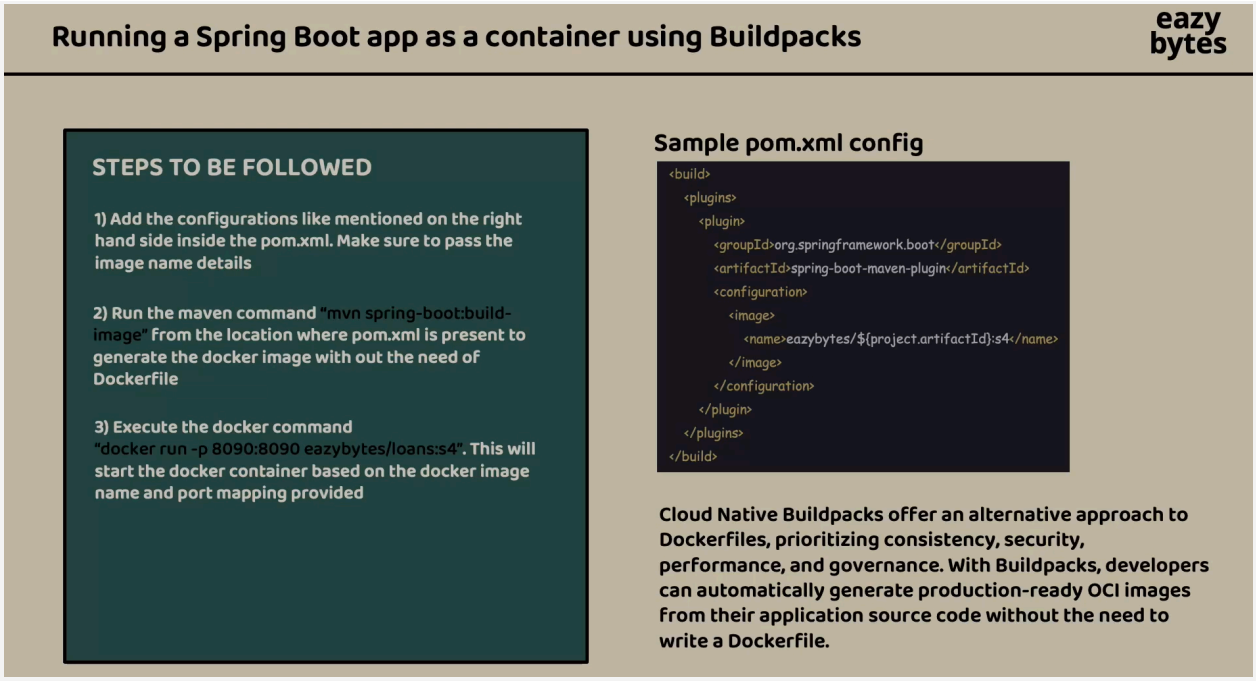
The “45 years ago” date shows up because the base image used by Buildpacks doesn’t have real timestamps, so Docker defaults to the Unix epoch (1970).

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

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

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

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







2) using Google Jib

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

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

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

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

Run the Buildpack Docker image command

```
mvn compile jib:dockerBuild
```

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

No Dockerfile is needed.

Jib automatically detects your Java version from the project.

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

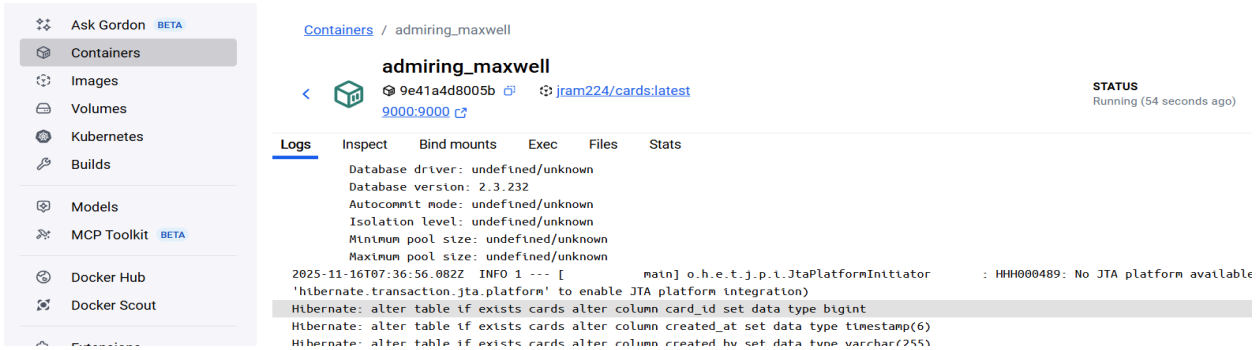
```
C:\Users\jakkula.ramesh\Desktop\Udemy Spring Boot\microservices\section2\cards>mvn compile jib:dockerBuild
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------< com.eazybytes:cards >-------------------------
[INFO] Building cards 0.0.1-SNAPSHOT
[INFO] --------------------------------[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

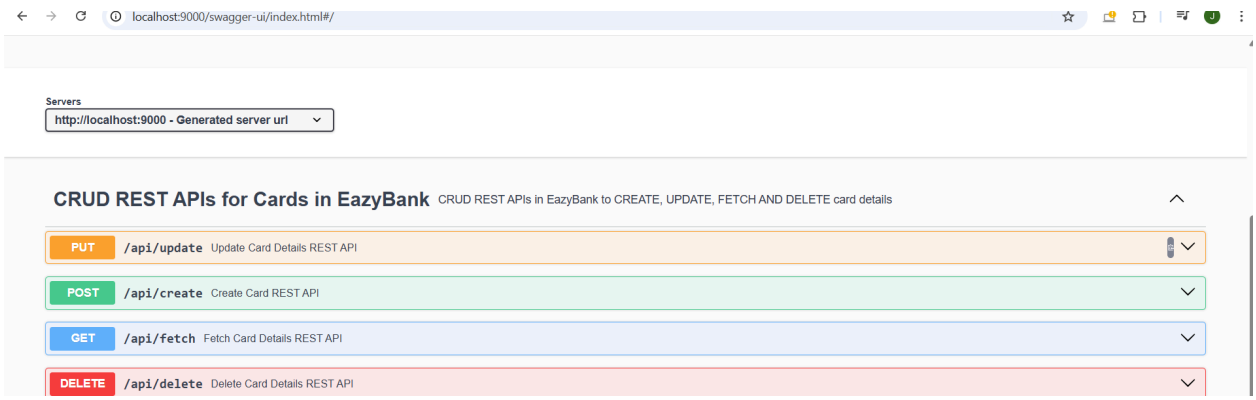
```
[INFO] Using credentials from Docker config (C:\Users\jakkula.ramesh\.docker\config.json) for eclipse-temurin:17-jre
[INFO] Using base image with digest: sha256:75ab7d1b4b18483e9245342cbee253b558952c1def5c1c18956196330a01683e
[INFO]
[INFO] Container entrypoint set to [java, -cp, @/app/jib-classpath-file,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

iii) Run Docker Container Using the Image

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

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

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





Running a Spring Boot app as a container using Google Jib

eazy bytes

STEPS TO BE FOLLOWED

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

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

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

Sample pom.xml config

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

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

Comparison of the Dockerfile, Buildpacks, and Jib:

https://buildpacks.io/features/

Comparison

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

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

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

Examples of remote image repositories:

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

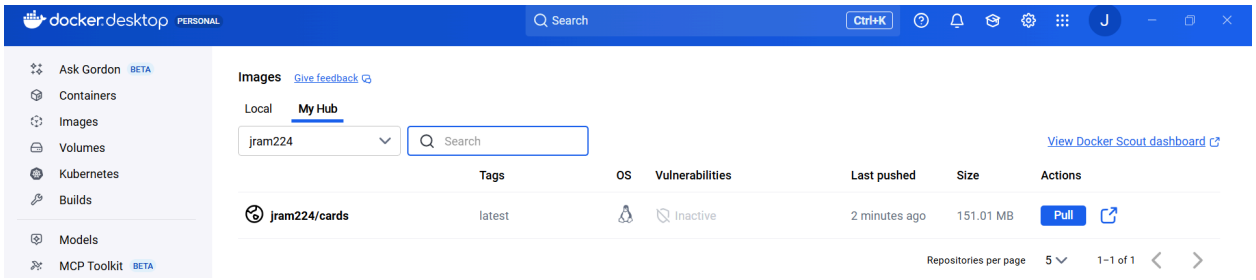
How to Push Docker Images to Docker Hub

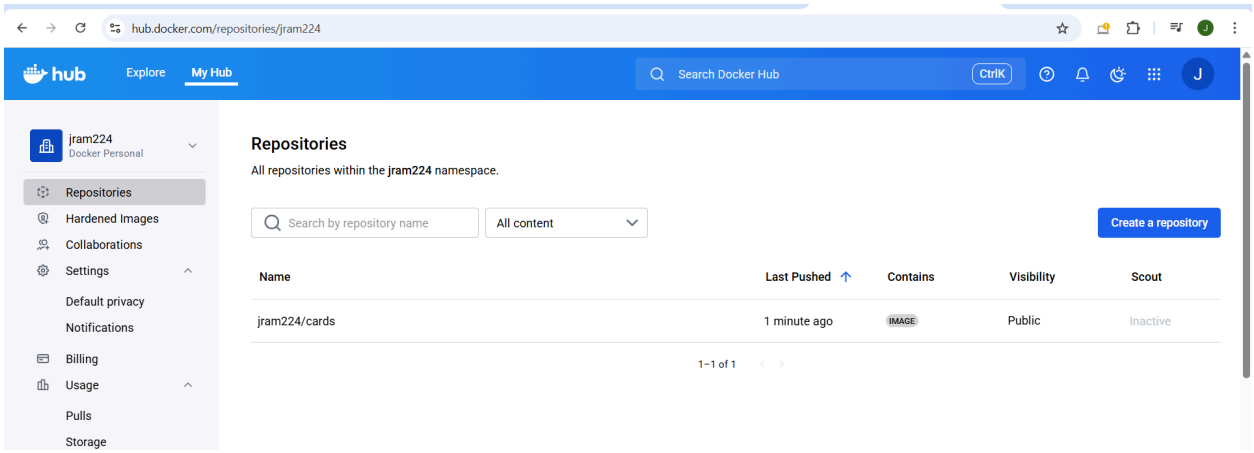
Since you are already **logged in to Docker Desktop** on your local machine, Because of this existing authentication, **you can push images to Docker Hub without entering your username and password again**.

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

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

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

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

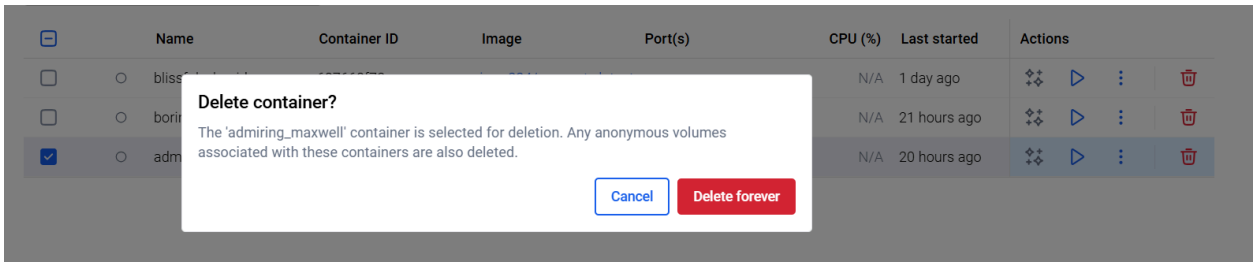
We can see either docker hub site or Hub option in docker desktop.

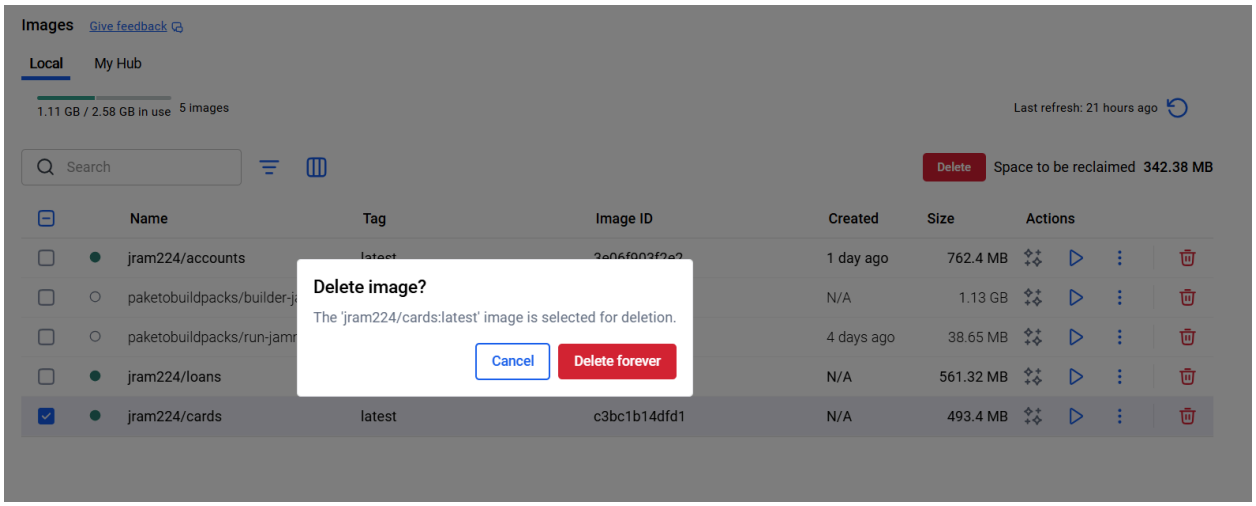




We can pull Docker images from a remote Docker registry (Docker Hub, ECR, ACR, etc.)also.

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






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

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

Pull
docker pull <imagename>:<tag>

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

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

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

docker run <options> <image-name>

But this approach has **major disadvantages**:

❌ 1. Running multiple applications becomes painful

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

docker run ...

docker run ...

docker run ...

for each application manually.

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

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

docker run ...

docker run ...

docker run ...

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

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

You must manually provide:

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

This becomes messy.

Solution: Use Docker Compose

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

Docker Compose

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

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

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

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

Without Compose

Run 6 commands manually:

docker run service1

`docker run service2`

`docker run service3`

`docker run instance1`

`docker run instance2`

With Compose

Run **one** command:

`docker compose up`

If you want, I can also show:

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

1️⃣ Start all services

`docker compose up`

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

To run in the background:

`docker compose up -d`

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

`docker compose down`

Equivalent to:

- stop containers
- remove containers
- remove networks

3️⃣ Start existing containers (without recreating)

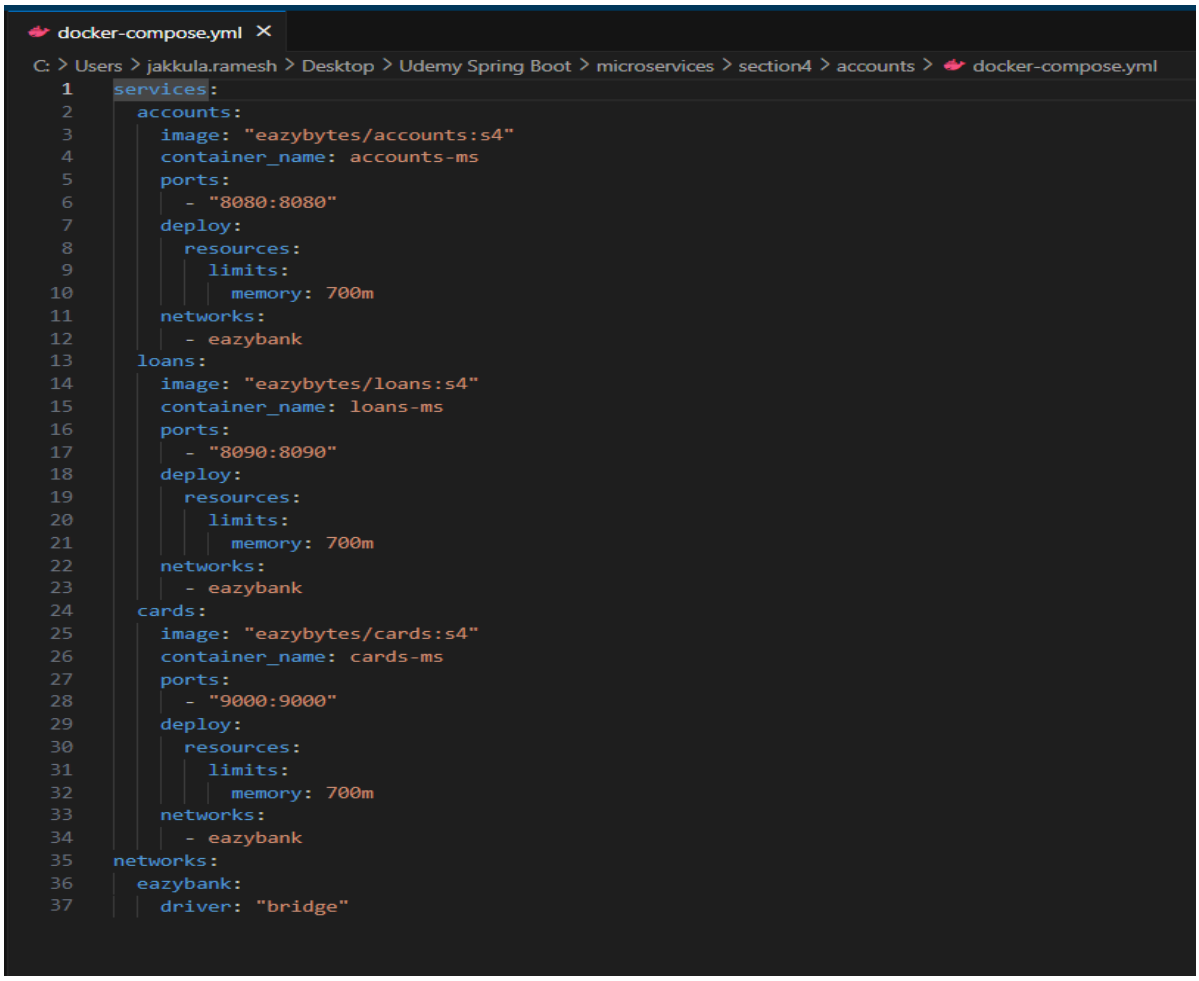
docker compose start

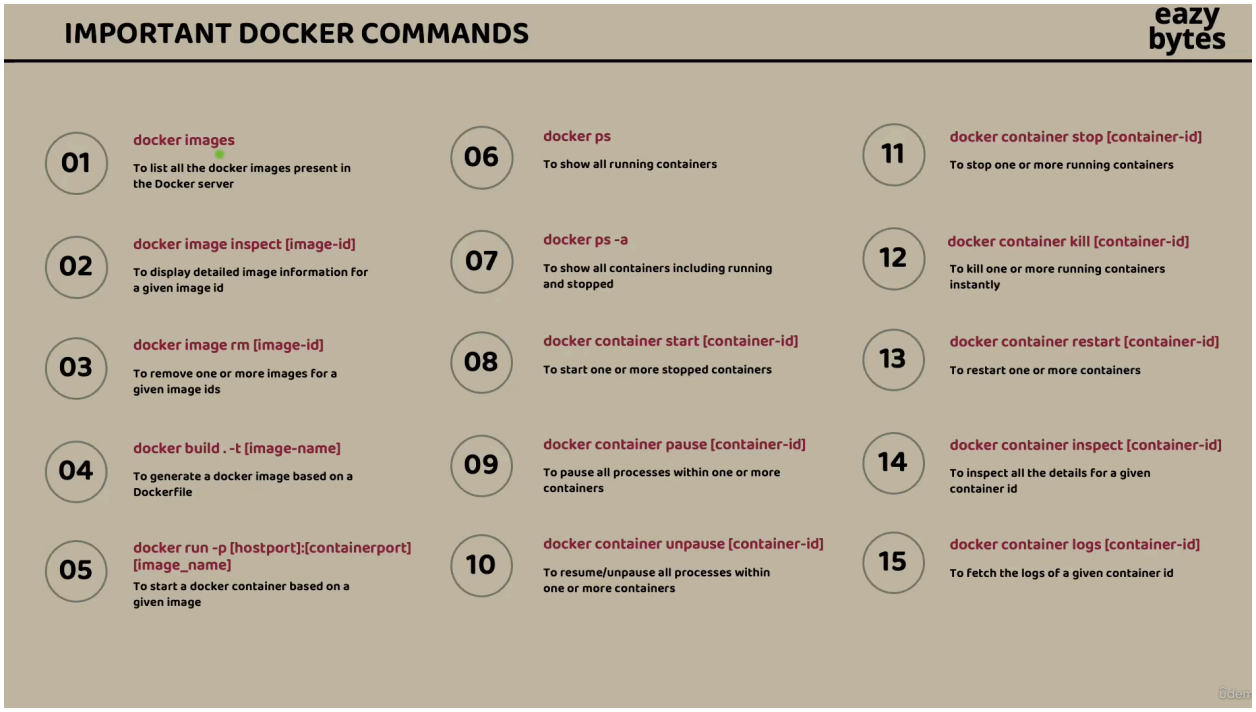
Start containers that are already created but stopped.

4️⃣ Stop running containers (but do not remove them)

docker compose stop

Stops services but does *not* remove containers.



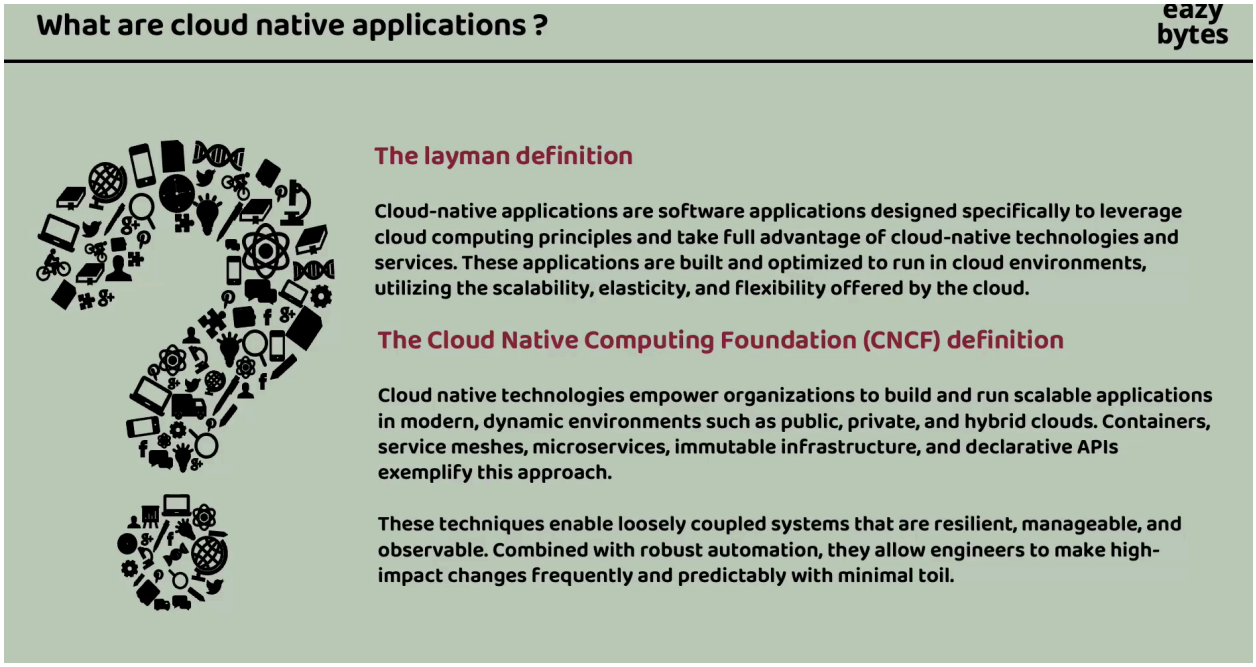


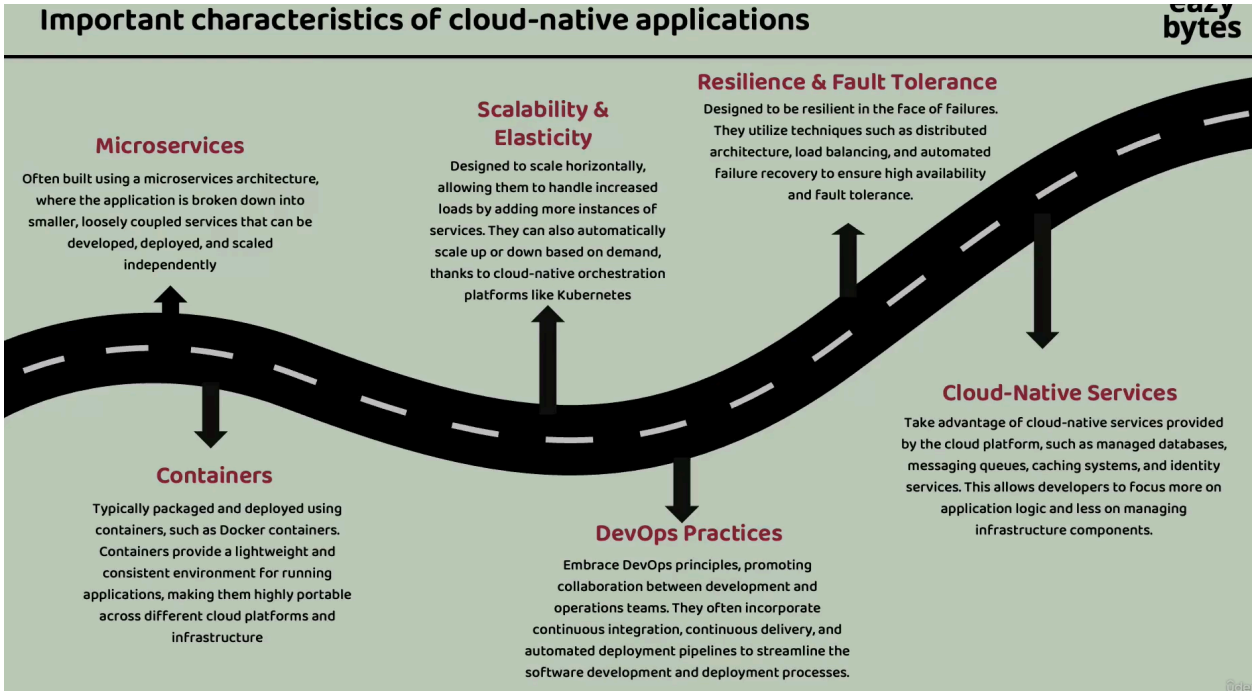
Cloud-Native Application

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

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

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





Key Characteristics of Cloud-Native Applications

1 Microservices Architecture

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

2️⃣ Containers (Docker)

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

3️⃣ Orchestration (Kubernetes)

- Manages container deployment, scaling, and recovery automatically.

4️⃣ DevOps + CI/CD

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

5️⃣ Scalability & Elasticity

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

6️⃣ Resilience & Fault Tolerance

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

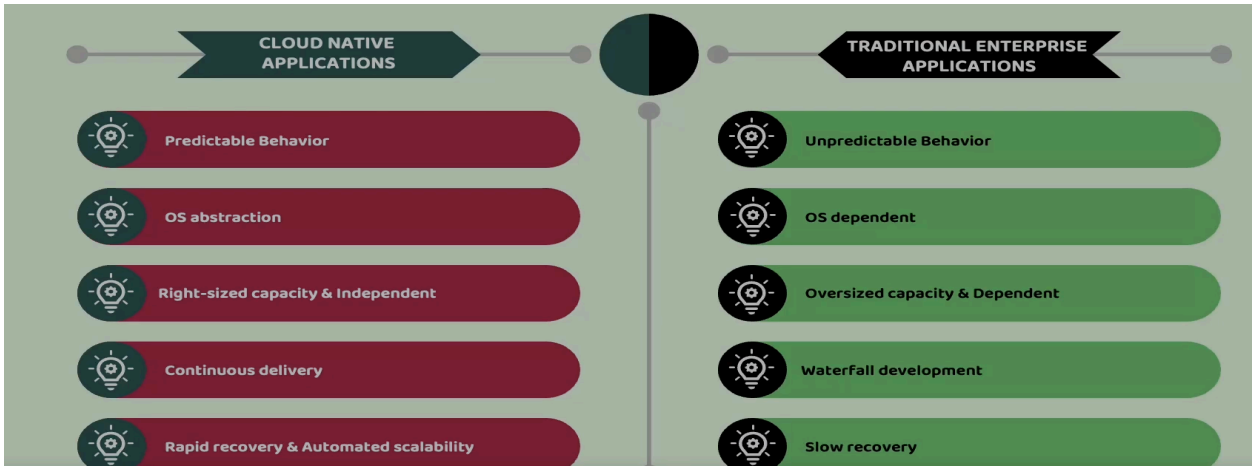
7️⃣ API-Driven

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

8️⃣ Cloud Managed Services

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

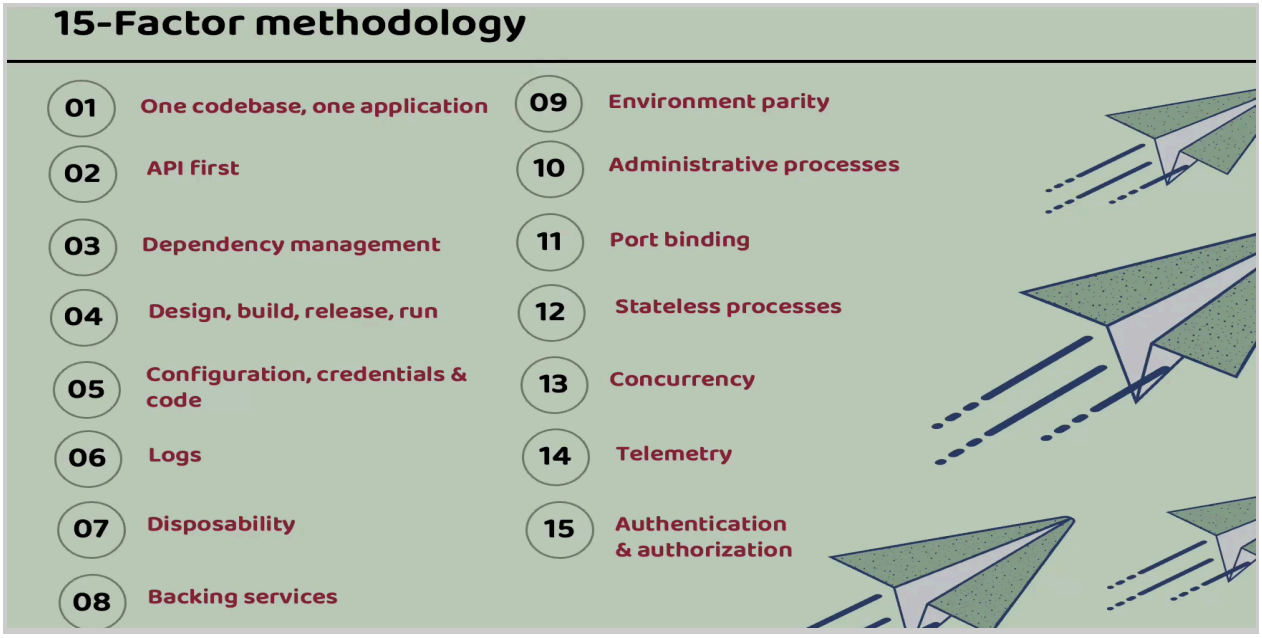
Cloud-Native vs Traditional Applications



📌 Cloud-Native vs Traditional Applications

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

Core Development Principles of Cloud-Native Applications

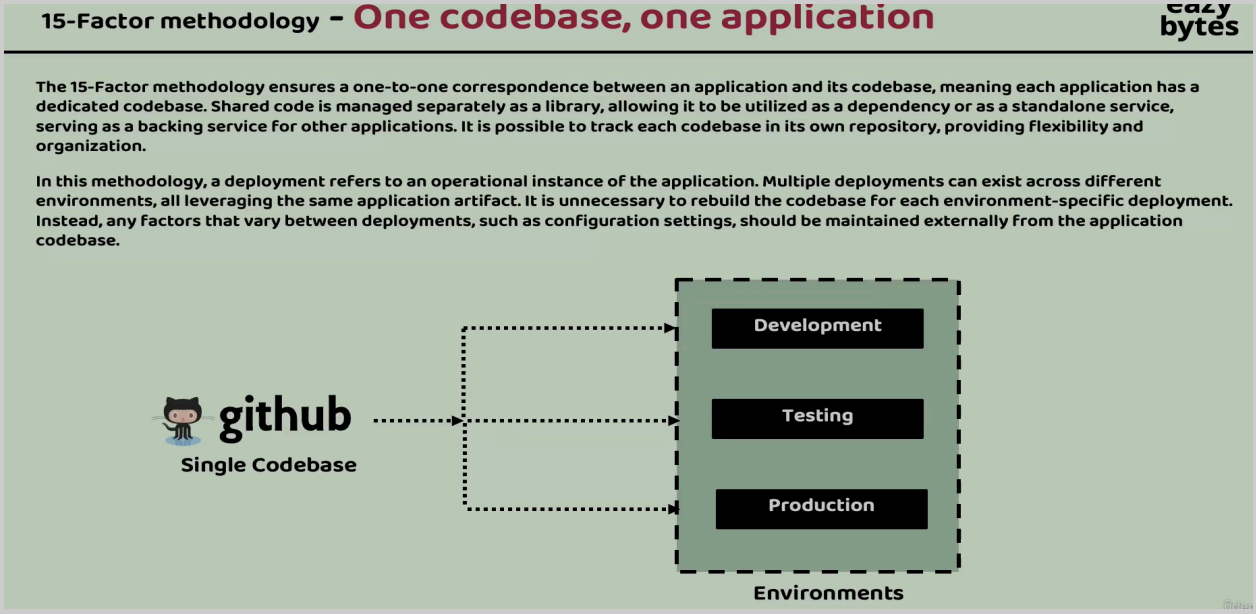


1️⃣ One Codebase, One Application

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

Key Points:

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



2️⃣ API First

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

Key Points:

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

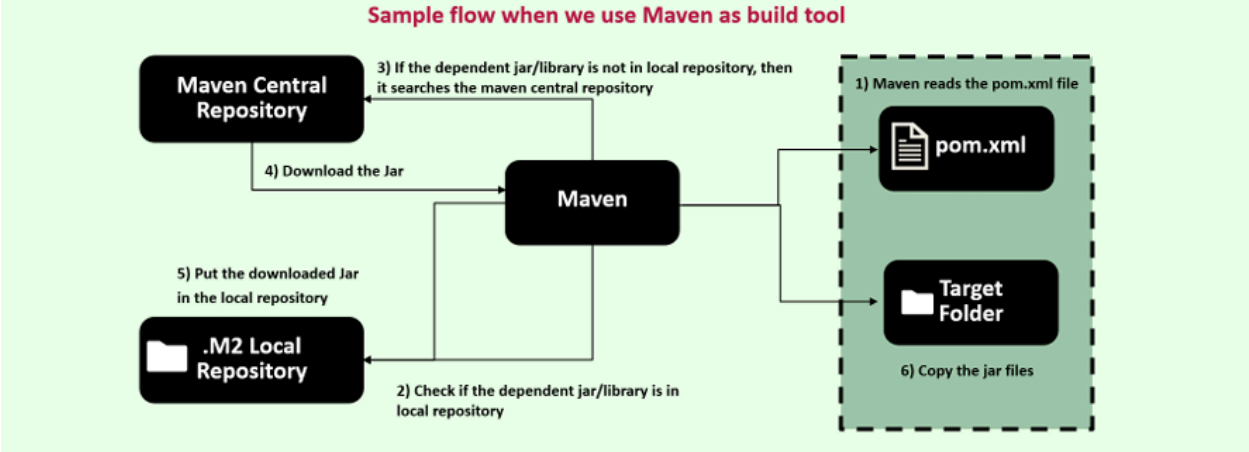
3️⃣ Dependency Management

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

Key Points:

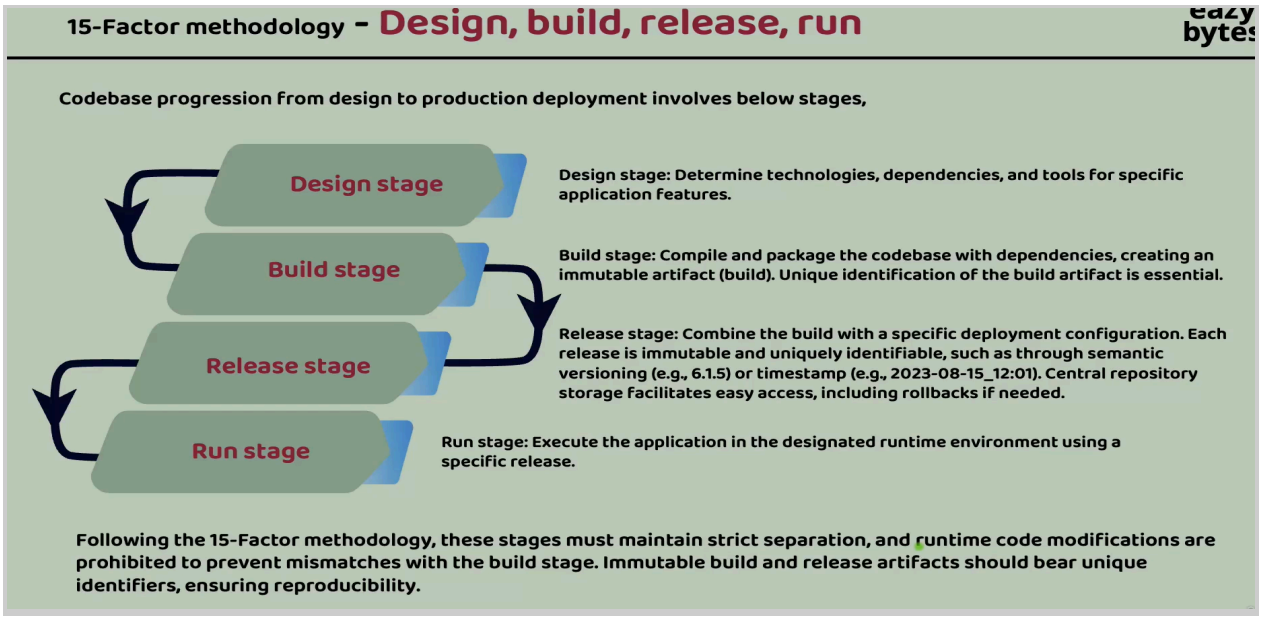
- No implicit/hidden dependencies
- Use dependency managers (Maven, Gradle, npm, etc.)
- Dependency manager will:
    - Check local repo
    - If missing → download from central repo
    - Store in cache
- Ensures consistency and reproducibility

Java Example Flow (Maven):



4 Design, Build, Release, Run

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



Key Points:

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

5 Configuration, Credentials & Code (Separation of Concerns)

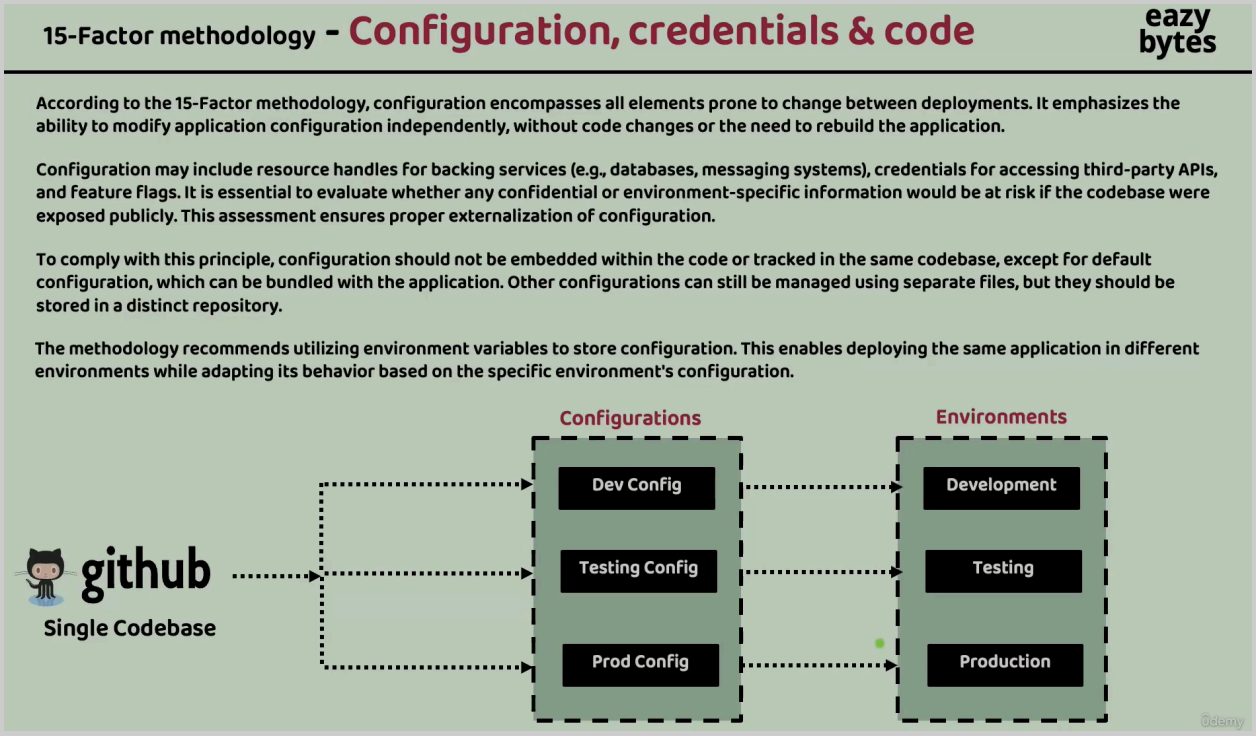
Configuration must be **externalized**, not hardcoded inside the application.

Examples of configs to externalize:

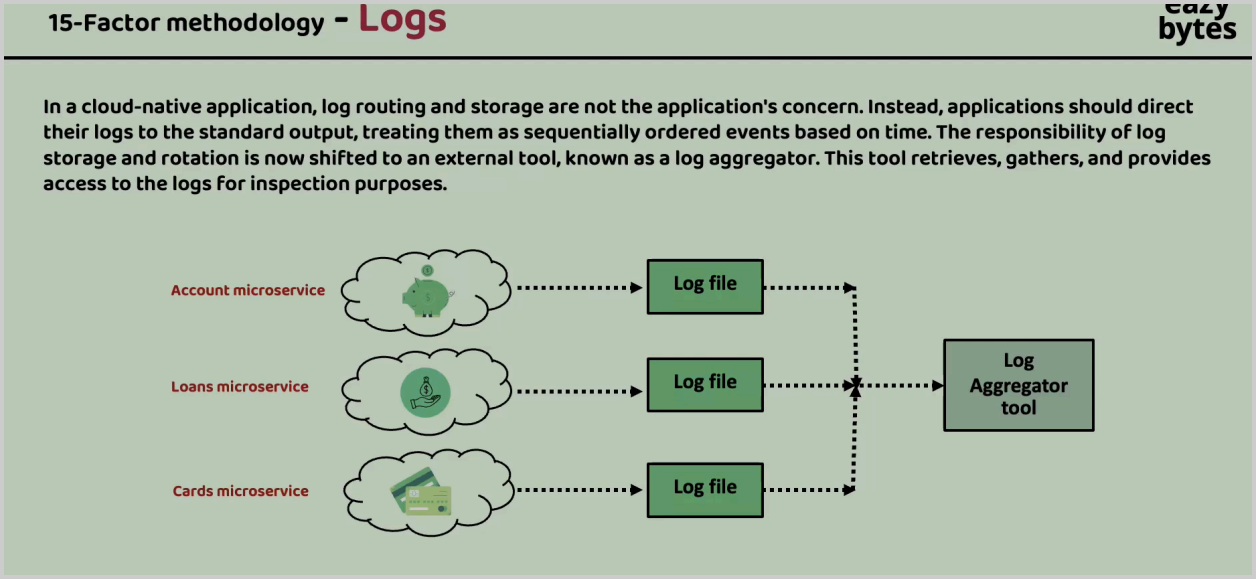
- Database URLs
- Credentials / API keys
- Environment variables
- Feature flags

Why?

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



6 Logs – Treat Logs as Event Streams



7 Disposability – Fast Startup & Graceful Shutdown

Applications must be:

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

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

Practices:

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

8 Backing Services – Treat Services as Attached Resources

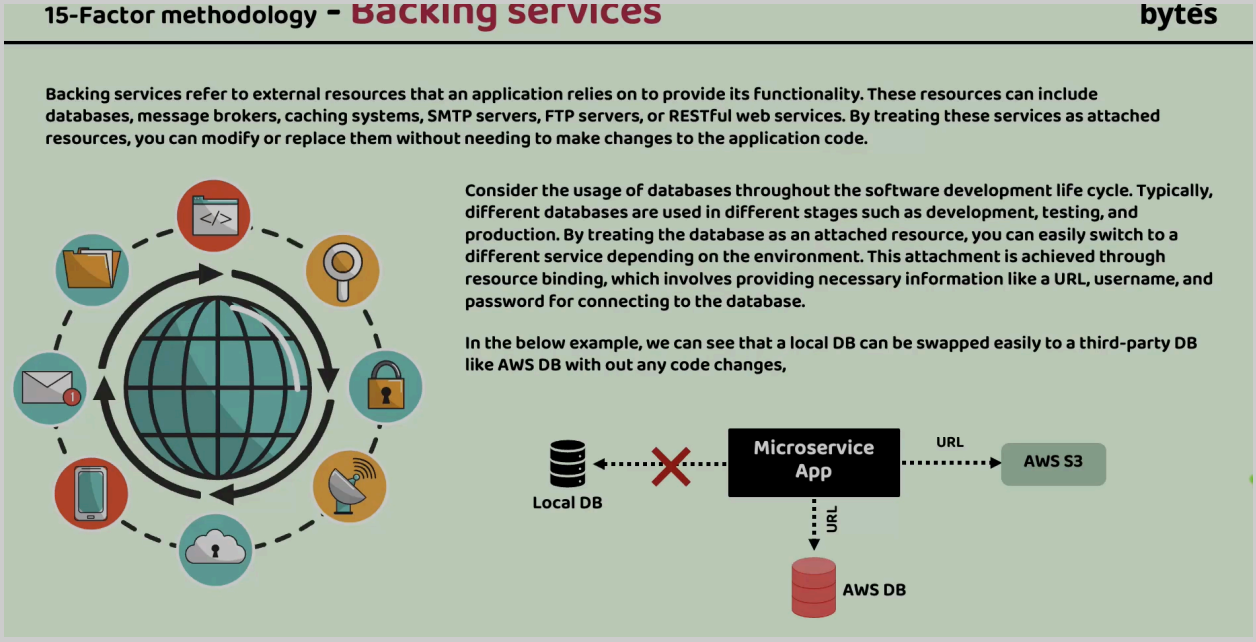
Resources like:

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

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

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

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



9 Environment Parity

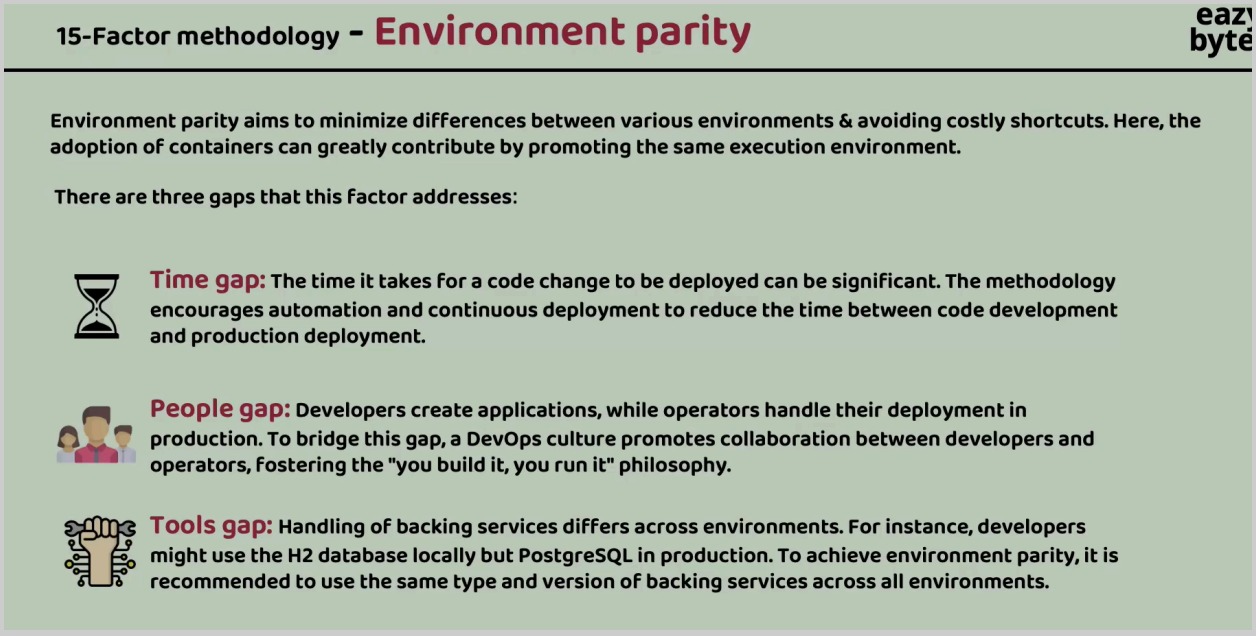
Keep Dev, Test, QA, and Prod environments **as similar as possible**.

Why?

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

Achieved by:

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



🔟 Admin Processes

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

Examples:

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

These should run:

- As standalone jobs
- Separate from the main app container

1️⃣1️⃣ Port Binding

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

Instead of relying on external servers like:

- Apache
- Nginx
- Tomcat (traditional WAR)

Cloud-native apps:

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

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

1️⃣2️⃣ Stateless Processes

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

- Database
- Redis
- Message queues
- Object storage

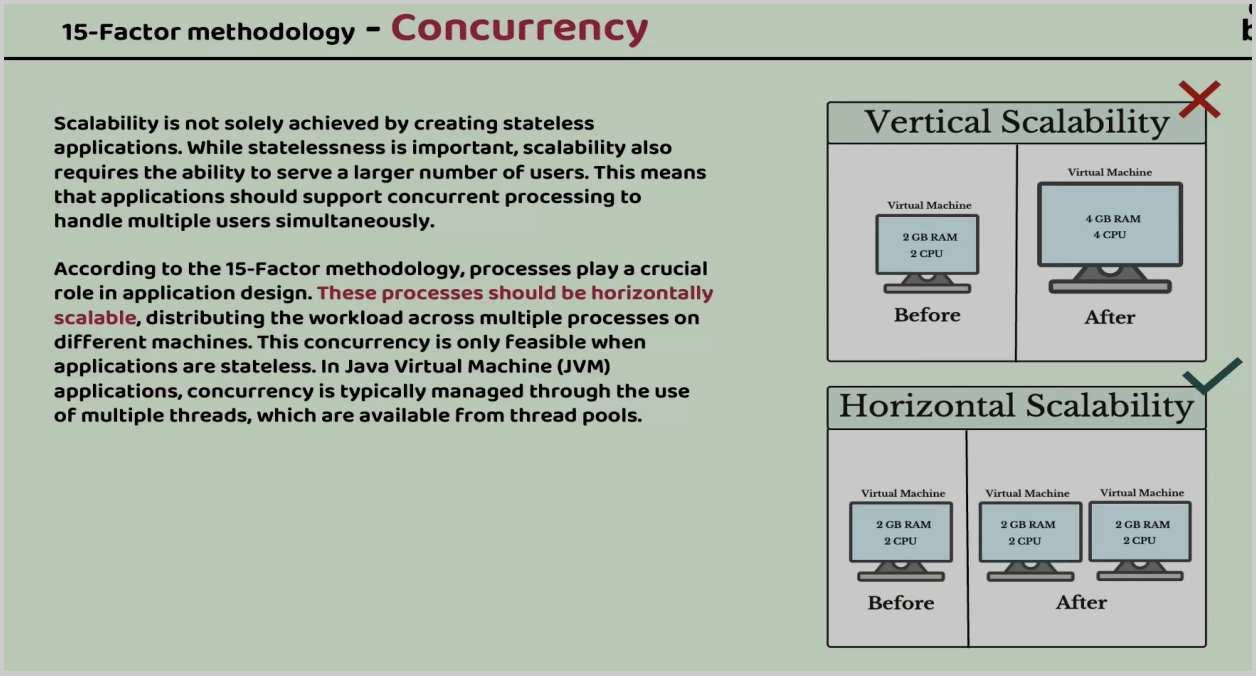
Reason:
Cloud-native platforms frequently kill and recreate containers. If your state is inside the container → it's lost.

1️⃣3️⃣ Concurrency – Scale Out via Processes

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

Cloud-native apps scale **horizontally**.

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



1️⃣4️⃣ Telemetry

Applications must expose:

- Metrics

- Traces
- Health checks

Why?
To allow monitoring tools like:

- Prometheus
- Grafana
- ELK
- Jaeger

to give system visibility.

Telemetry enables:

- Auto-scaling
- Alerting
- Diagnosis
- Performance insights

1️⃣5️⃣ Authentication & Authorization (Secure by Default)

Every cloud-native app must:

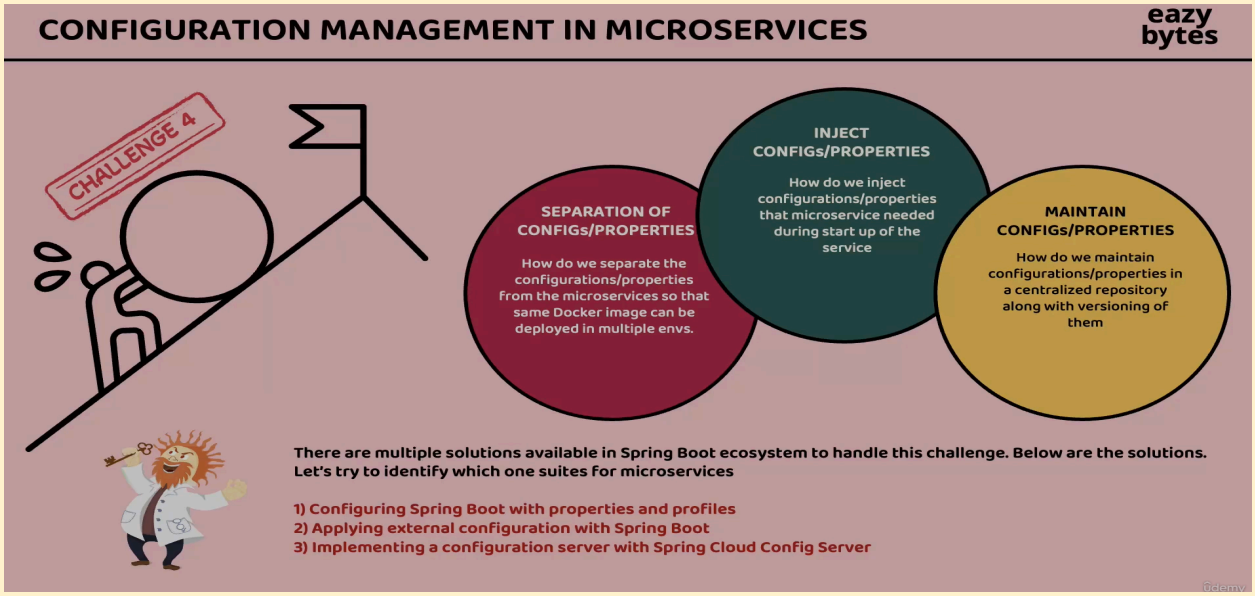
- Authenticate users
- Authorize access
- Secure data in transit and at rest

Common methods:

- OAuth2
- JWT
- OpenID

—----------

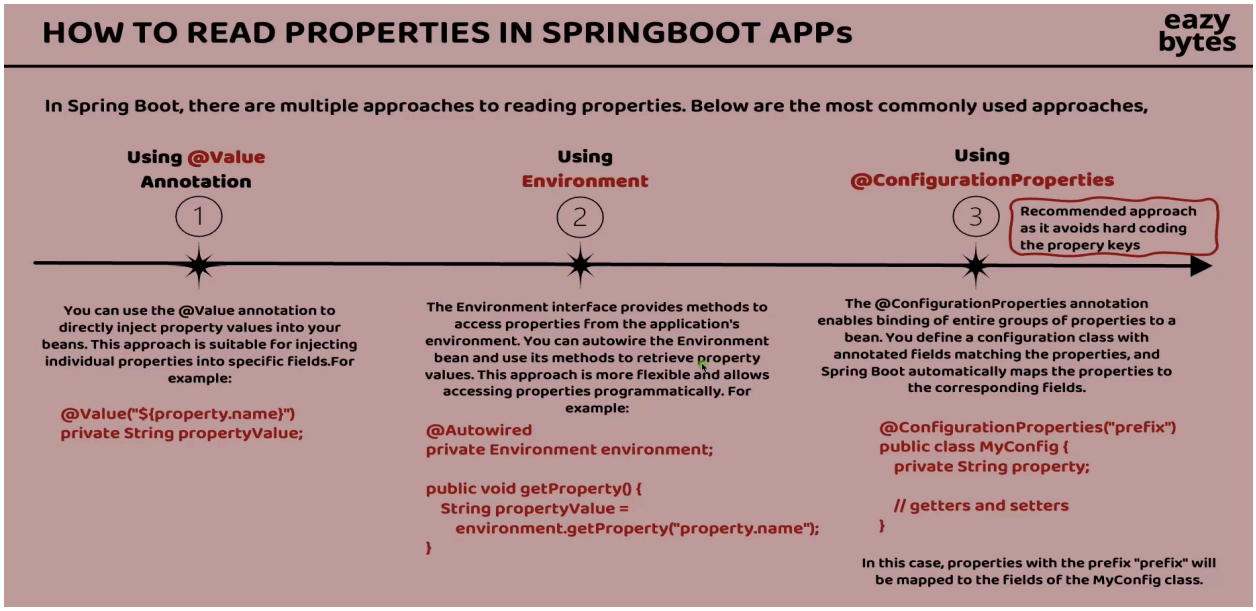
Configuration Management in Microservices



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

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

Read properties in spring boot:



Basic ways:

Using @value : which @Value → reads single values

Values which we need inject from application.propeties or yml

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

```java
@Validated
public class AccountsController {

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

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

Code	Details
200	**Response body** 1

Not Suitable for Multiple or Grouped Properties

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

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

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

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

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

Your class becomes messy and unmaintainable.

Using Environment Interface:

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

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

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

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

We can see we got the location

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

Ex 2: for know java version

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

Using @ConfigurationProperties Annotation:

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

Step 1. Add Properties in application.yml

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

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

Step 2. Create POJO Using property values

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

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

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

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

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

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

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

Key Points

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

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

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

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

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

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

}
```

Step 4. Use It in Controller or Service

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

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

200

Response body


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


Ex2:

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

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

}
```

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

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

}
```

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

200

Response body


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


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

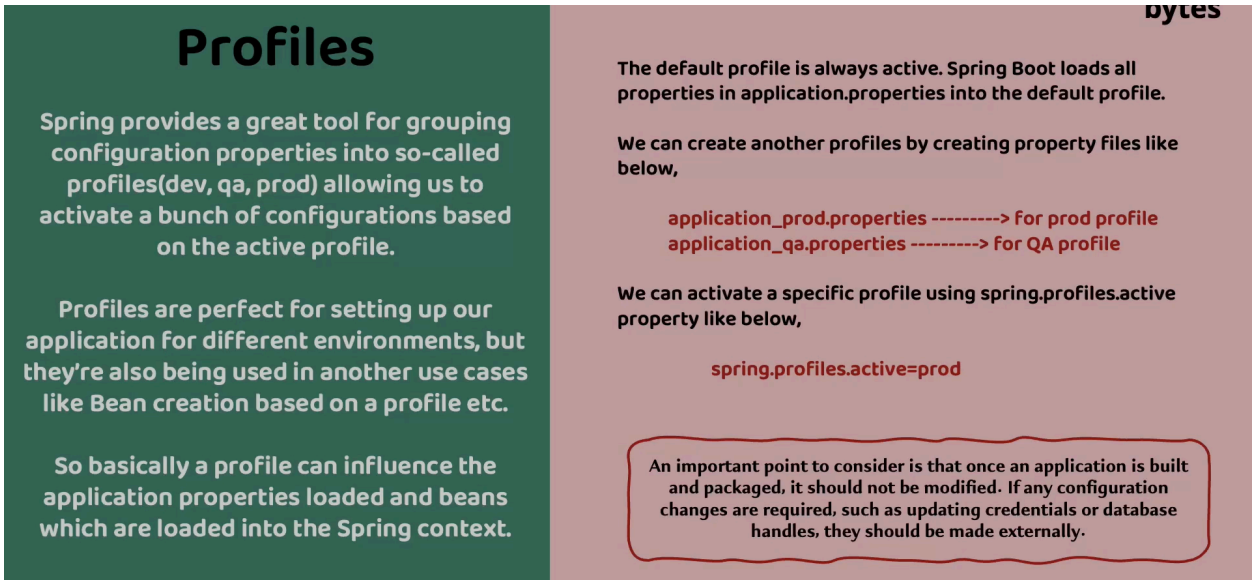
this is where **Spring Profiles** come into play

Spring Profiles

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

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

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



Example

We have microservice with different configurations for different environments.

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

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

application.yml

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

application_qa

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

Application_prod

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

Inside each profile-specific file, you use:

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

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

spring.config.activate.on-profile

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

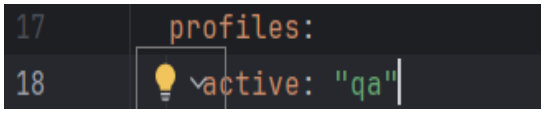
```yaml
17  config:
18    activate:
19      on-profile: "qa"
```

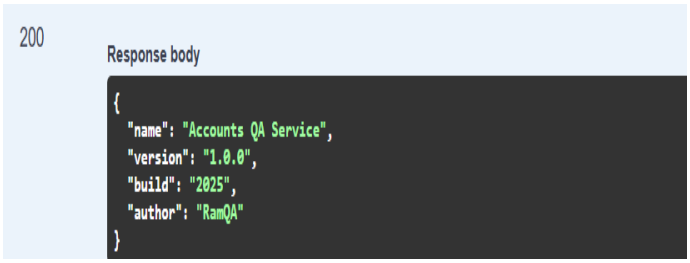
Then we should activate the environment (**spring.profiles.active=dev/prod/qa**) in the default file application.properties/application.yml file.

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

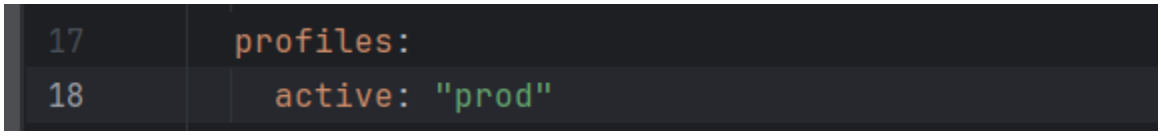
Because of this built-in mechanism, **you do NOT need to manually import or load profile files**

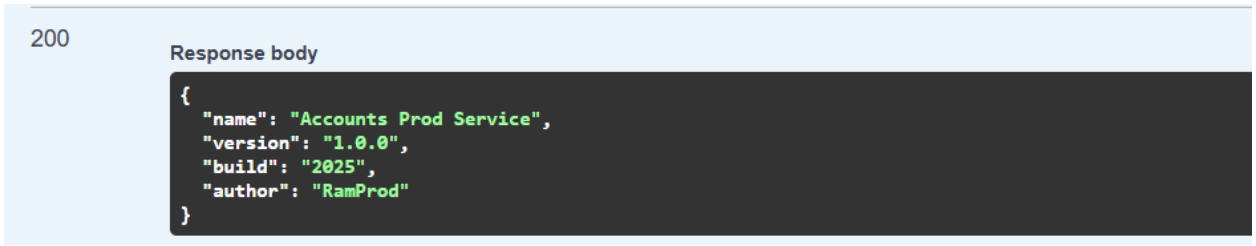
Then it loads the file that matches the active profile:





Or





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

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

Spring Boot allows overriding properties at startup using:

1. **Command-line arguments**

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

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

Externalize the configuration using Command-line arguments

Command-line arguments (highest precedence)

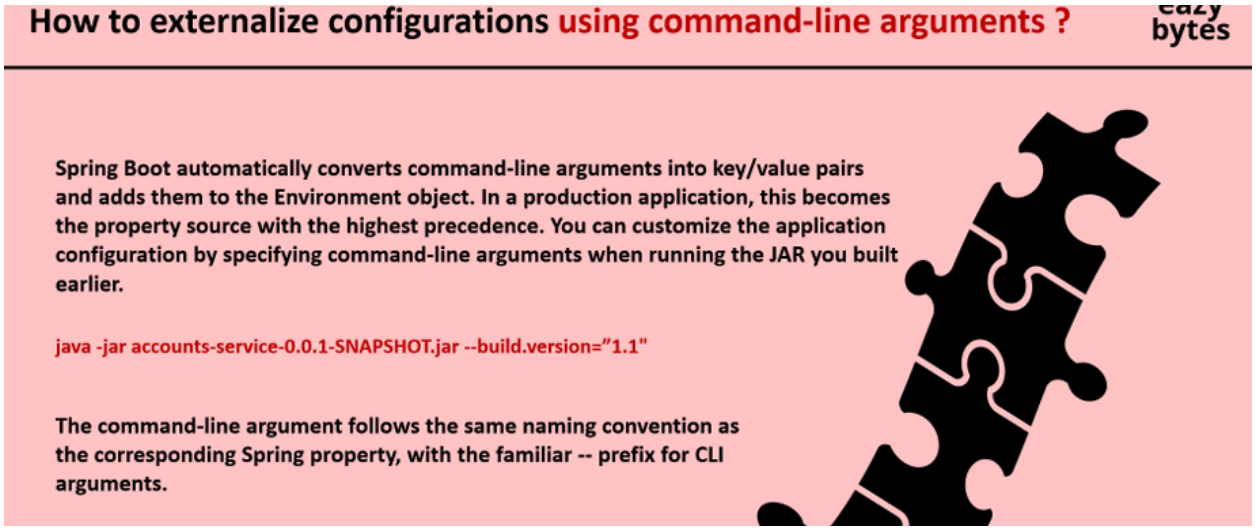
java -jar app.jar --key=value

Notes

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

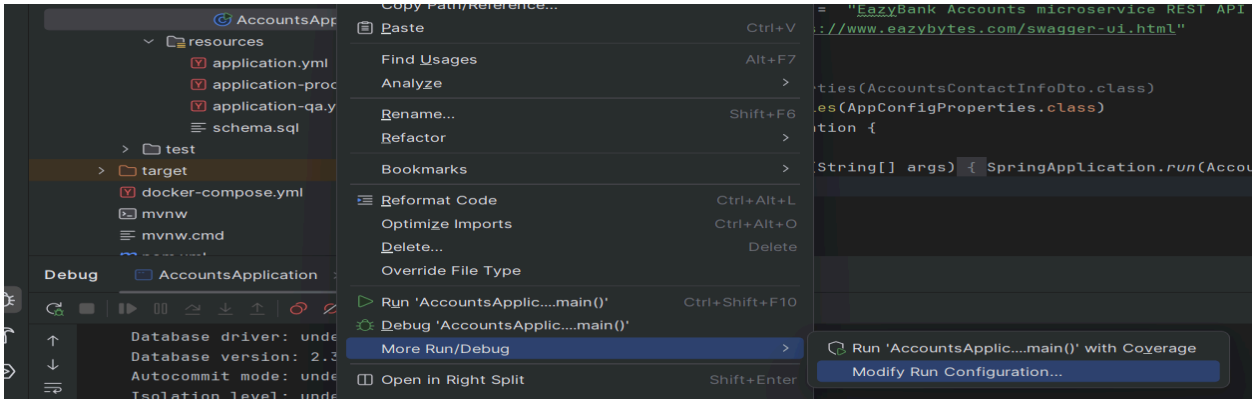
Note: Each property starts with --, Arguments are separated by spaces

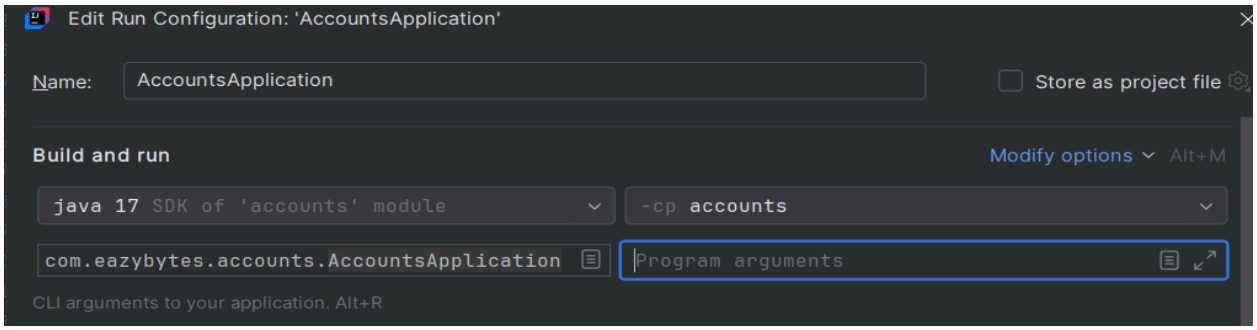
java -jar app.jar --spring.profiles.active=qa --server.port=8085



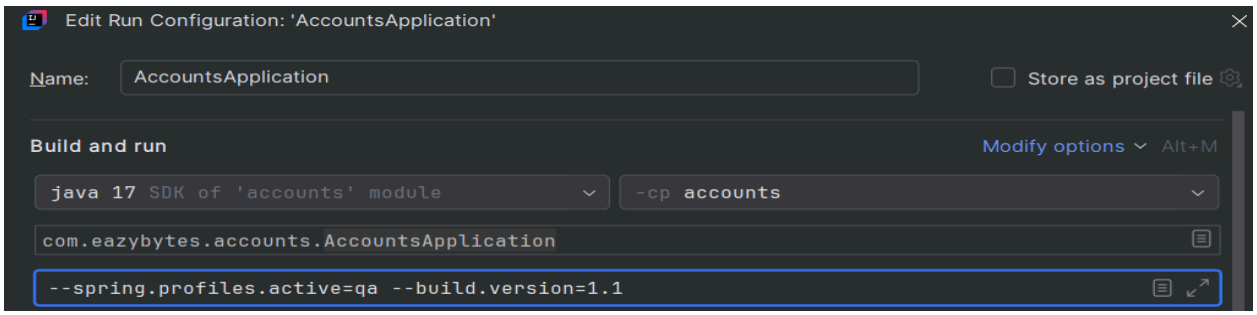
We can **also provide command-line arguments directly in the IDE**'s run configuration, **apart from just from the command line.**"

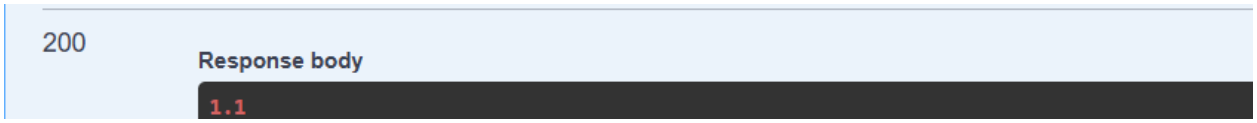
Go to the main class, right-click on it, and select *Modify Run Configuration*."

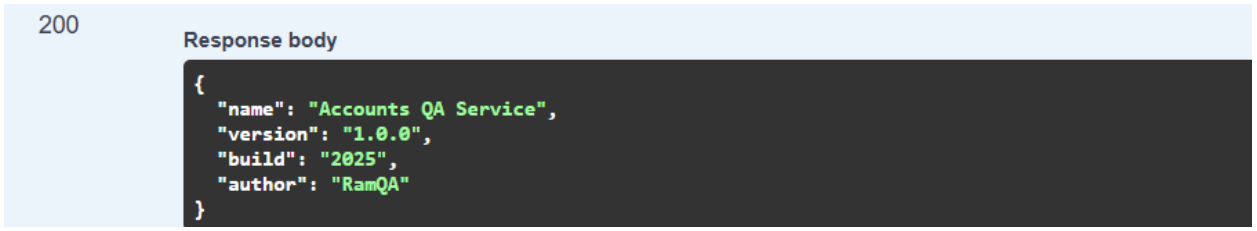




Provide in program arguments , I just providing two Arguments are separated by spaces







Externalize the configuration using JVM System properties

JVM system properties

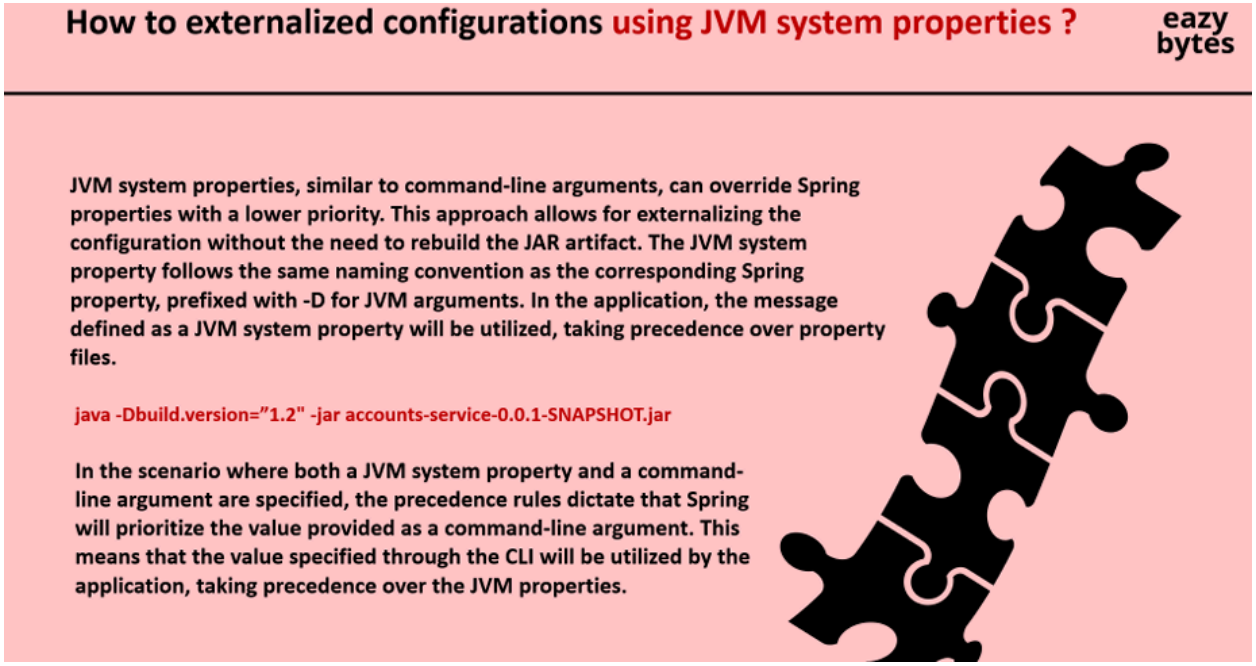
java -Dkey=value -jar app.jar

Notes

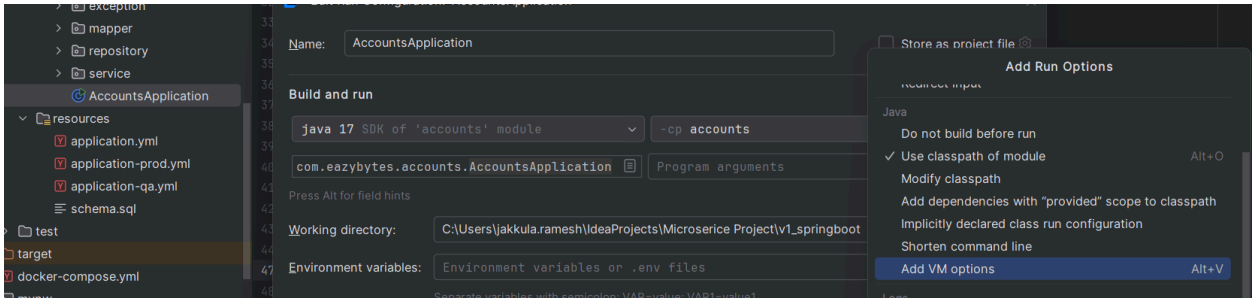
- System properties use the -D prefix.
- Follow the same naming as Spring keys.
- If both a system property **and** a command-line argument are provided, the command-line argument wins.

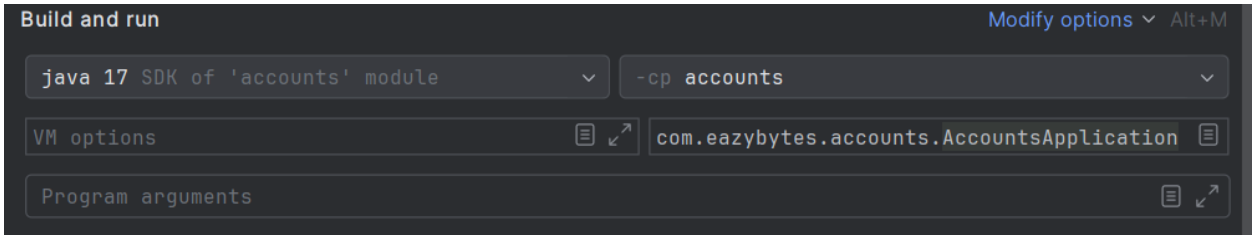
Note: Each property starts with -D, and they are separated by **spaces**

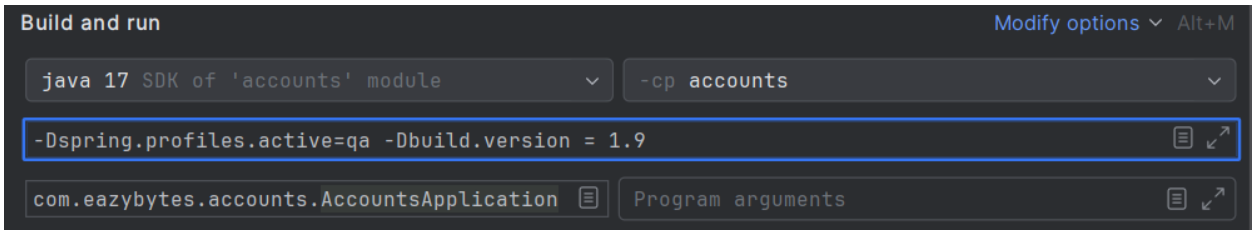
java -Dspring.profiles.active=prod -Dserver.port=9090 -jar app.jar

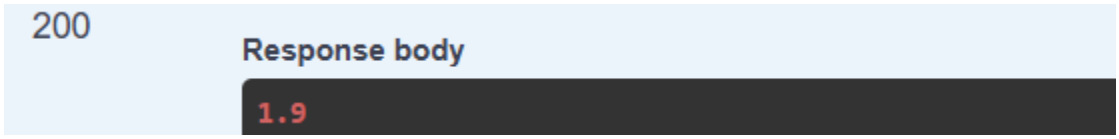


We can **also provide JVM System variables directly in the IDE**'s run configuration. Click on modify option , will get VM option.









Externalize the configuration using environment variables

Environment variables

KEY=value java -jar app.jar

- **Example (Windows CMD):**

```
cmd

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

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

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

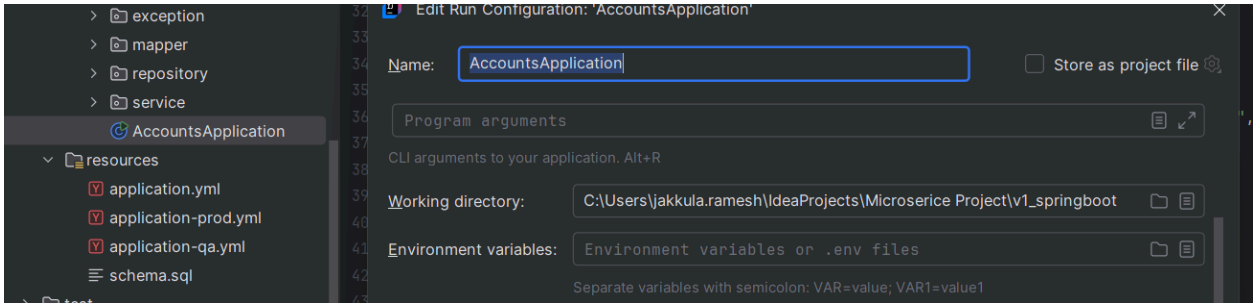
✘ Once you close CMD, those values disappear.

To map an environment variable to a Spring property:

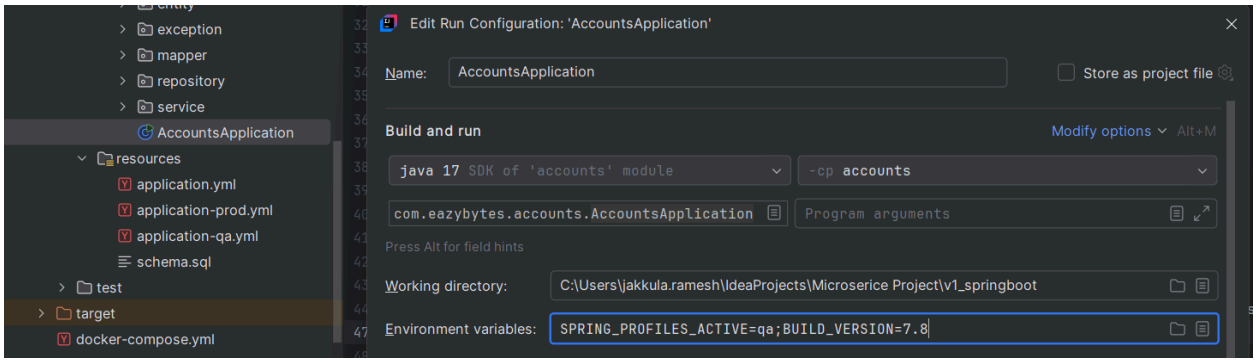
- Convert all letters to **uppercase**
- Replace dots (.) or dashes (-) with **underscores** (_)

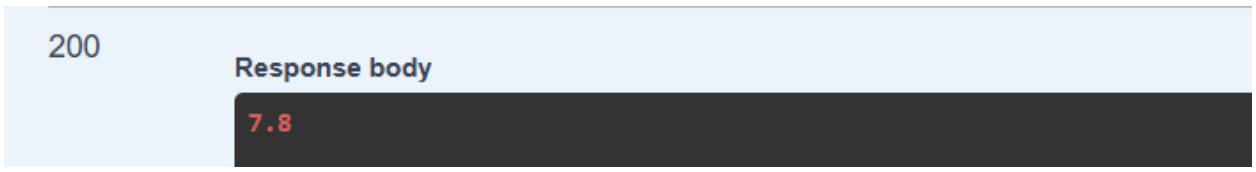
Example: BUILD_VERSION → recognized as build.version

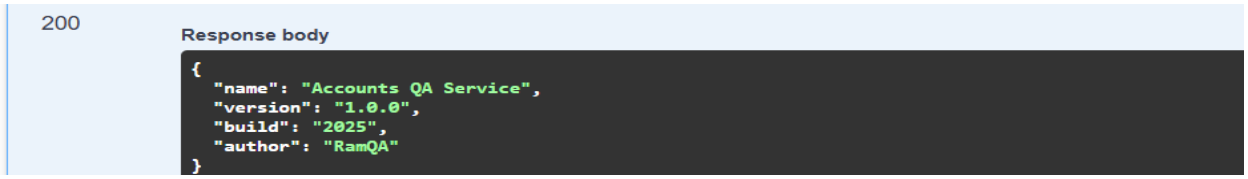
We can **also provide environment variables directly in the IDE**'s run configuration, **apart from just from the environment variables.**"

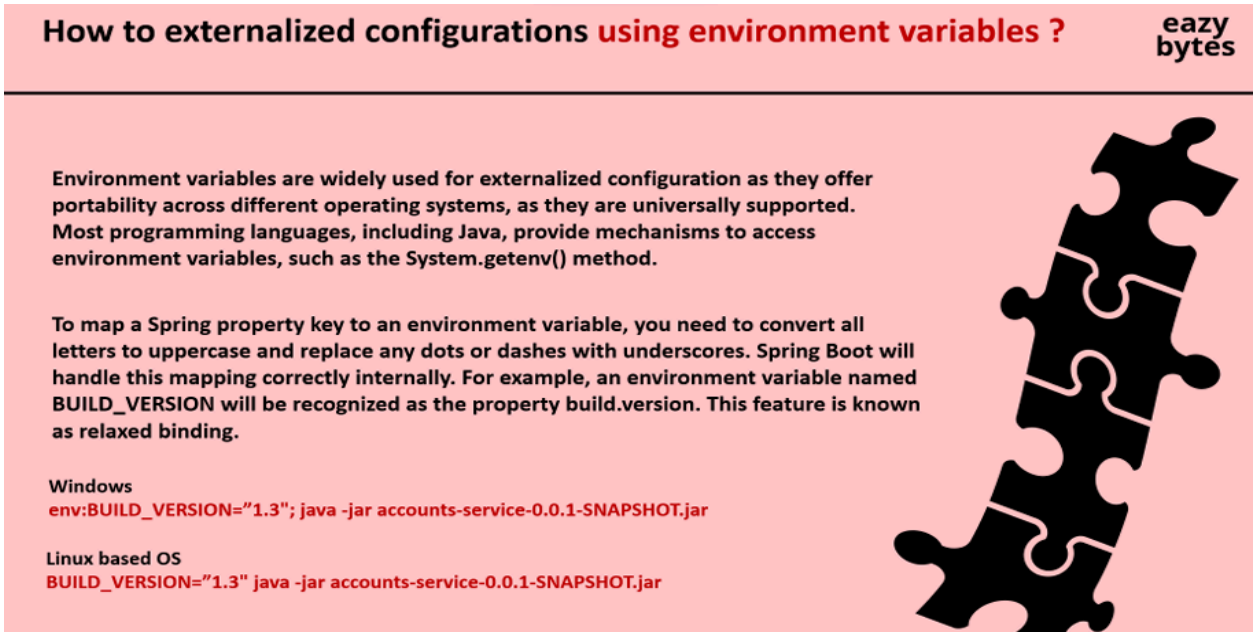


They are not separated by ; .







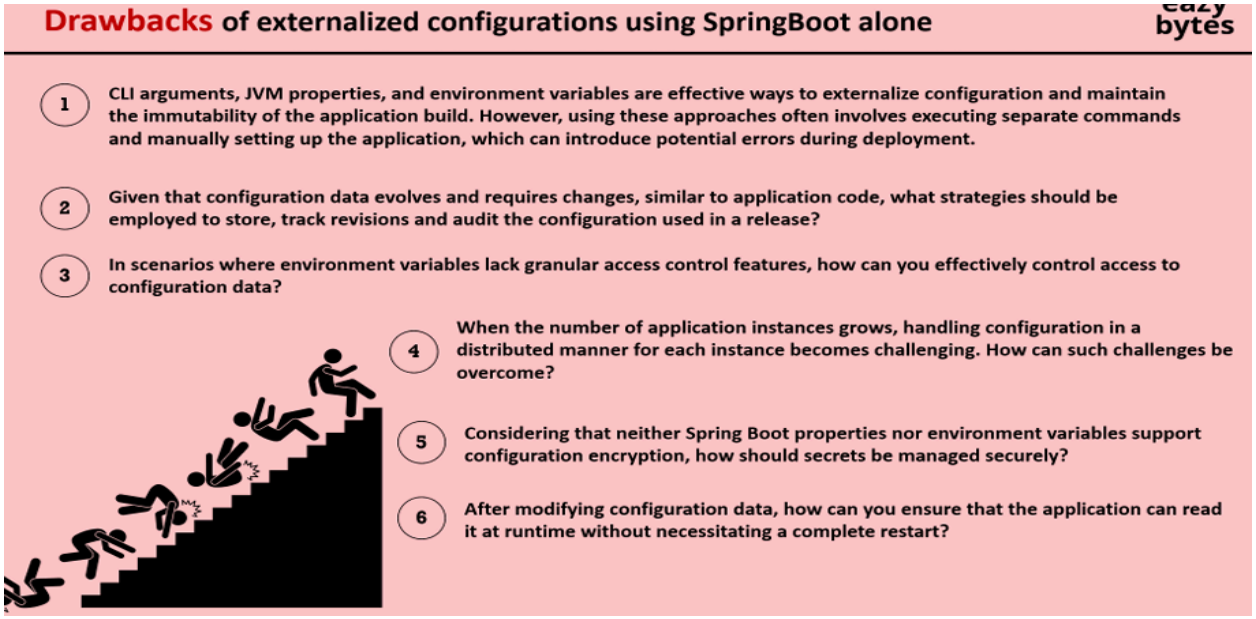


Most-Useful Priority

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

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

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

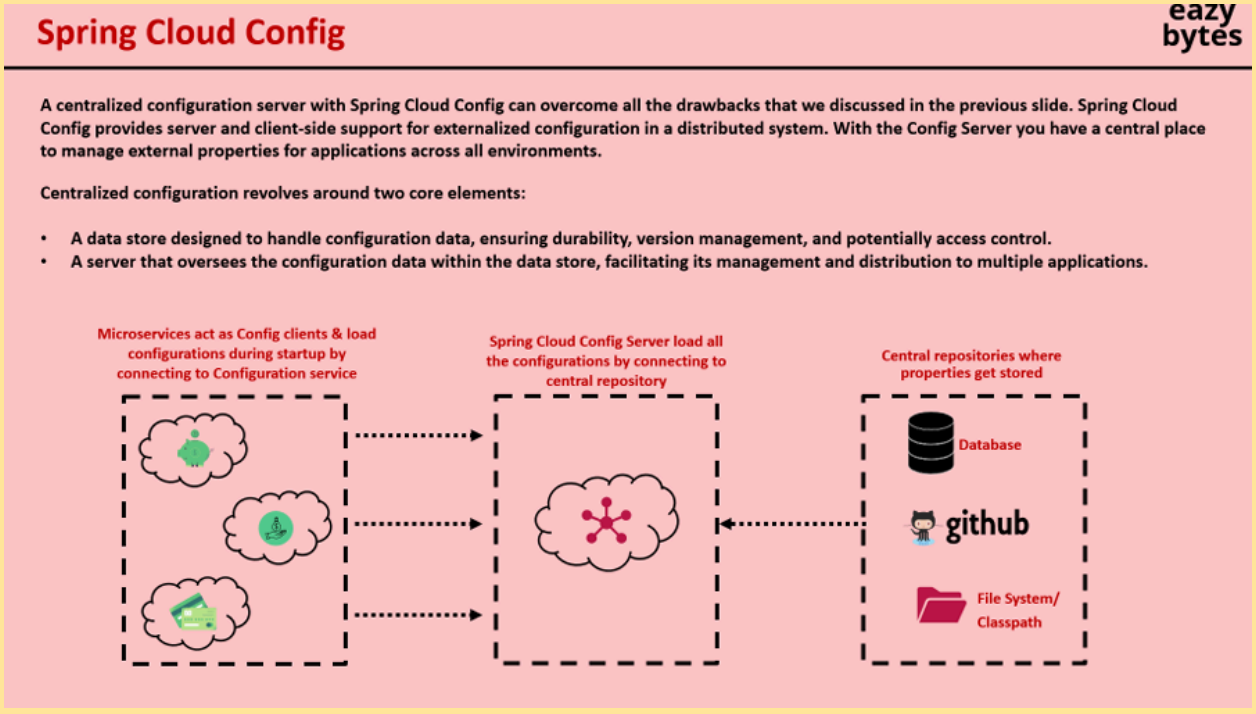


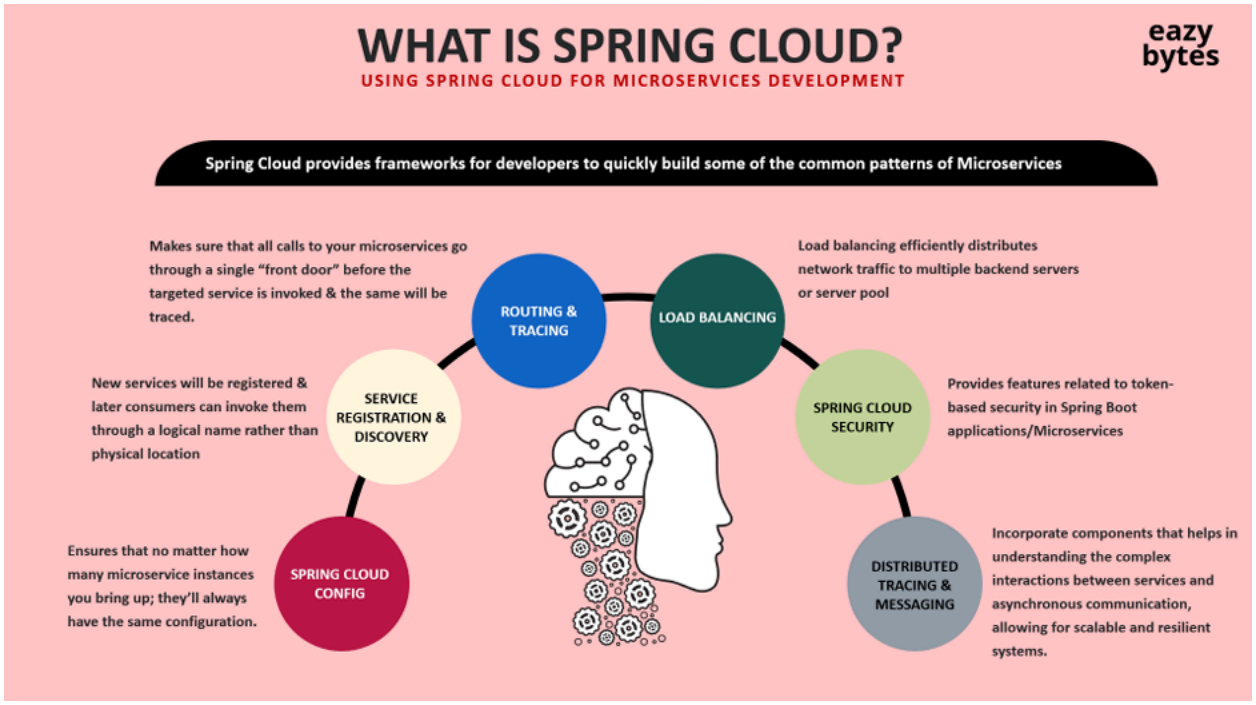
Spring Cloud Config

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

Key Points:

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





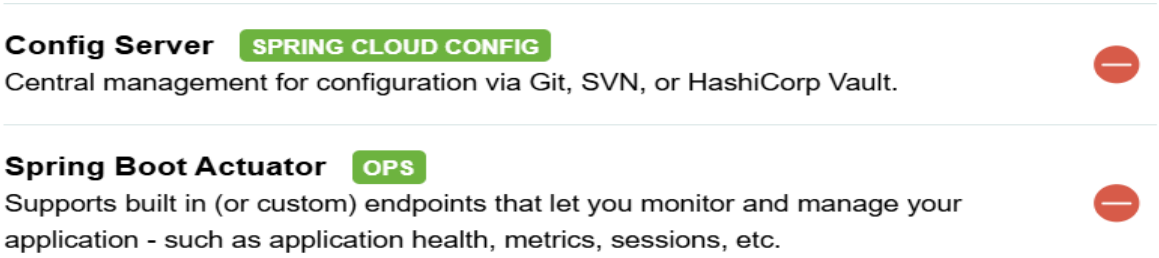
Spring Boot and Spring Cloud follow different release cycles, so their versions do **not** match automatically. Always use a **compatible combination** based on the official compatibility matrix, otherwise the application will show errors like:

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

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

Spring Cloud Config Server Setup

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



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

```
@SpringBootApplication
@EnableConfigServer
public class SpringcloudconfigserverApplication {

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

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

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

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

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

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

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

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

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

  spring.application.name: accounts

  Ex: accounts-qa.yml

  accounts-prod.yml

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

- Enable native profile in **Config Server**

  spring.profiles.active=native

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

  Tell Config Server to read from classpath

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

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

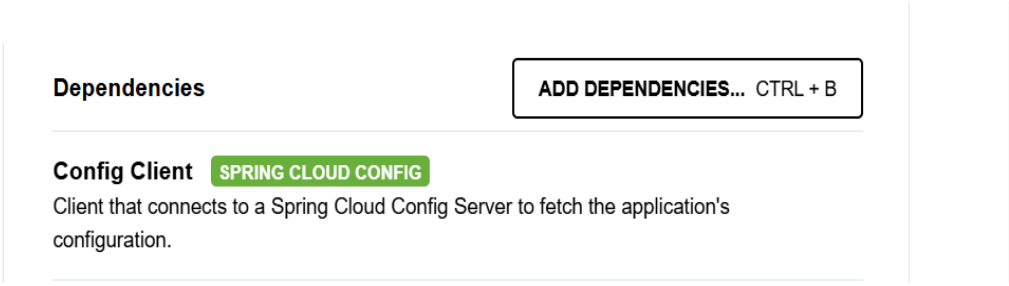
```yaml
server:
  port: "8071"
spring:
  application:
    name: "springcloudconfigserver"
  profiles:
    active: native
  cloud:
    config:
      server:
        native:
          search-locations: "classpath:/config"

```

- **Start Config Server**

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

- **Client microservice setup**

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




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

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

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

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

spring.profiles.active=prod

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

- **Start microservices and test**

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

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

200

Response body

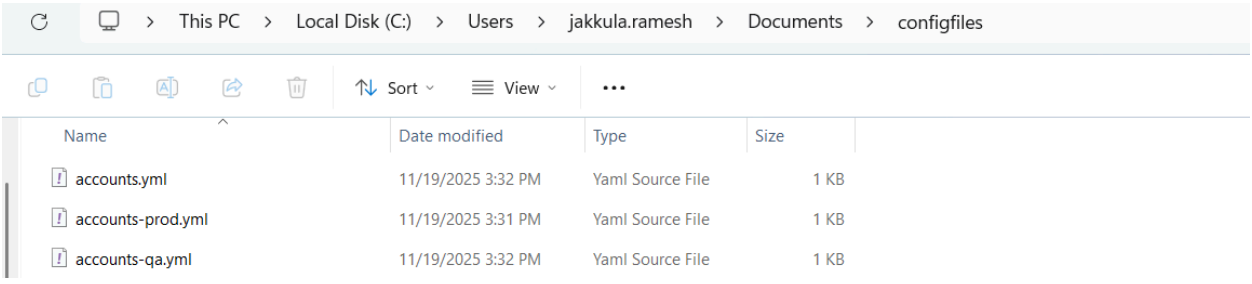
```
{
  "name": "Accounts Prod Service",
  "version": "1.0.0",
  "build": "2025",
  "author": "RamProd"
}
```

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

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

Ex 2: Reading Configurations from the System location

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

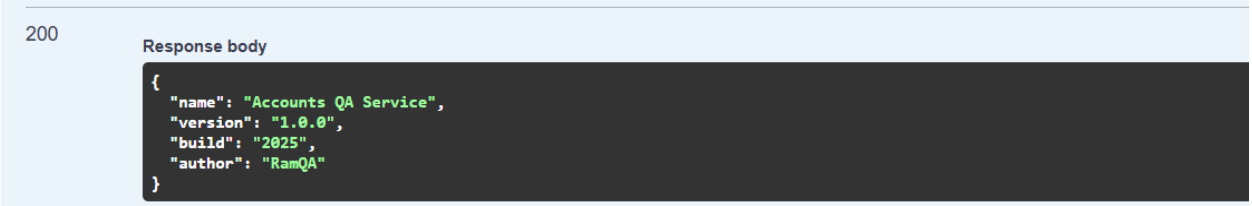


- Configure Spring Cloud Config Server to point to this system folder path using: spring.cloud.config.server.native.search-locations=file:///path/to/config-folder/

springcloudconfigserver\...\application.yml

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

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



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

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

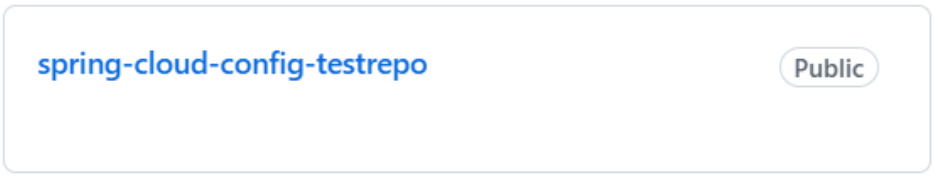
Ex 3: Reading Configurations from the GitHub Repository

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

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

- Create a new GitHub repository

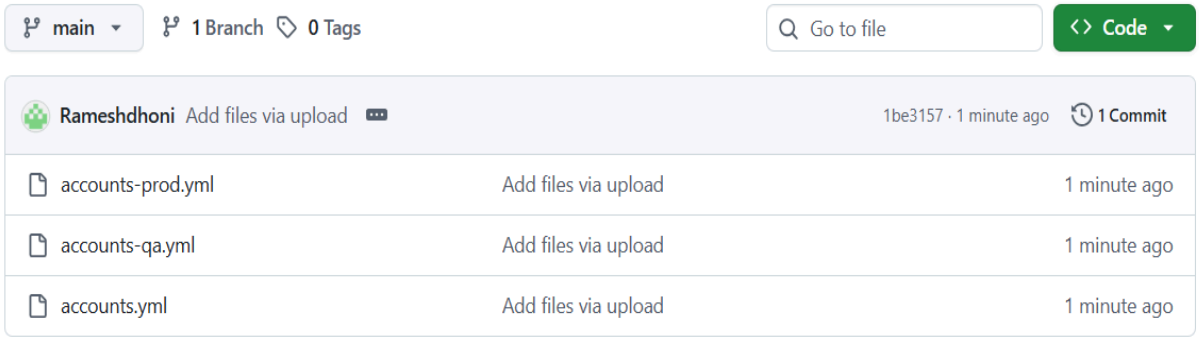
  Example: **spring-cloud-config-repo**



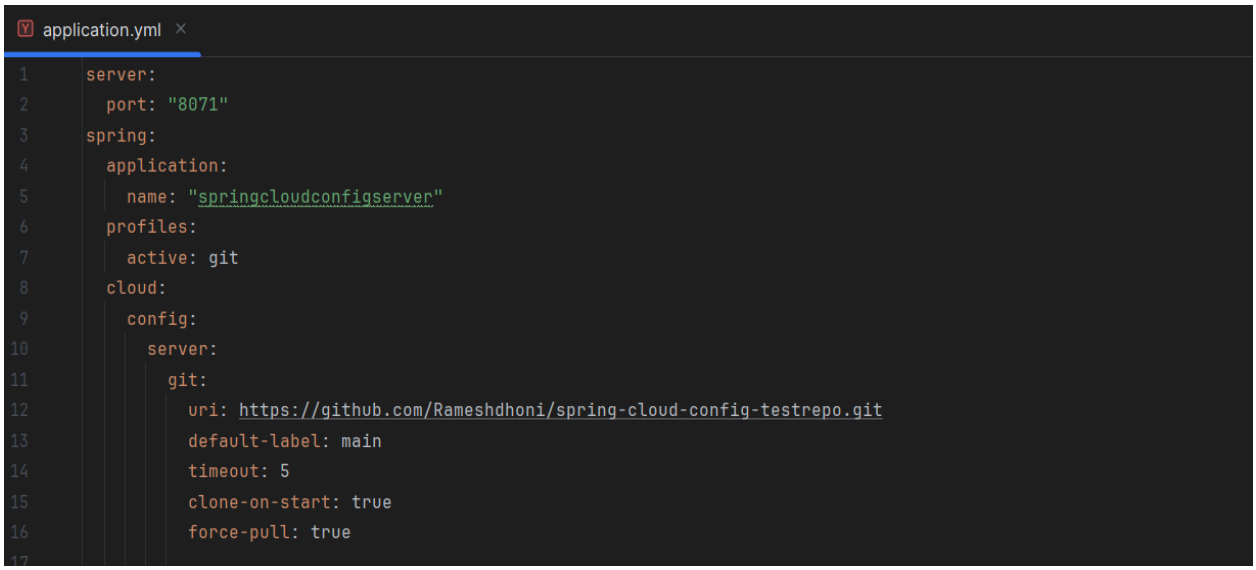
- Add your configuration files into the repo

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

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



- Configure Spring Cloud Config Server to use that GitHub repo



i) spring.profiles.active=git

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

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

Or

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

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

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

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

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

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

- Start the Config Server and microservices

200

Response body


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


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

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

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

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

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

You can use:

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

Example (symmetric key in Config Server):

```
encrypt:
  key: mysecretkey123
```

2. Encrypt values using Config Server endpoints

You can send plain values to:

POST http://localhost:8888/encrypt

It returns encrypted text like:

AQBsd908asda87asdasd...

3. Store encrypted values in GitHub repo

In your Git repo config file:

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

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

4. Microservices receive decrypted values automatically

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

So the microservice will see:

```
db.password = realpassword123
```

even though the Git repo stores:

```
{cipher}EncryptedStringHere
```

Refreshing the configuration at runtime

➤ Without any refresh mechanism

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

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

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

Way 1: Manual Refresh using /actuator/refresh

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

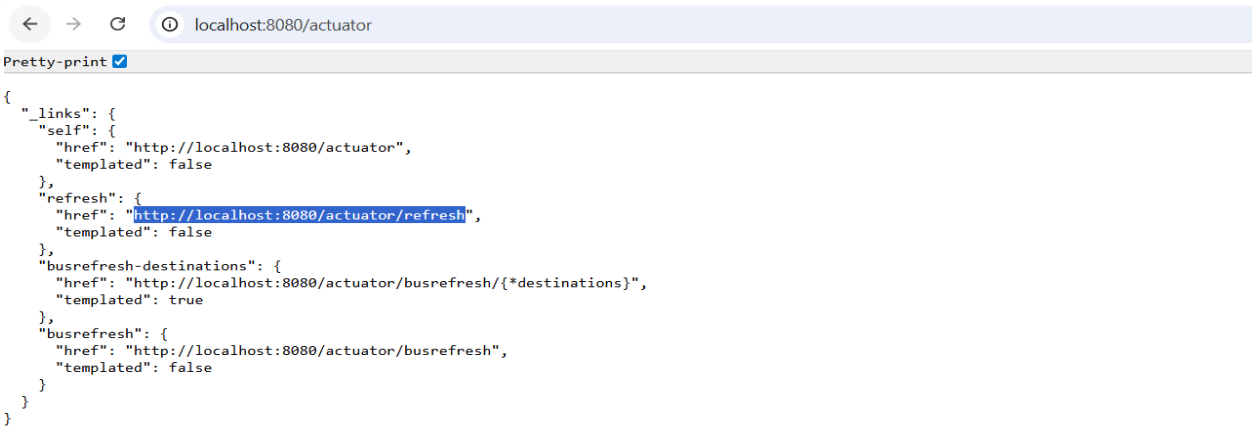
```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
```

- Expose the **/refresh** endpoint

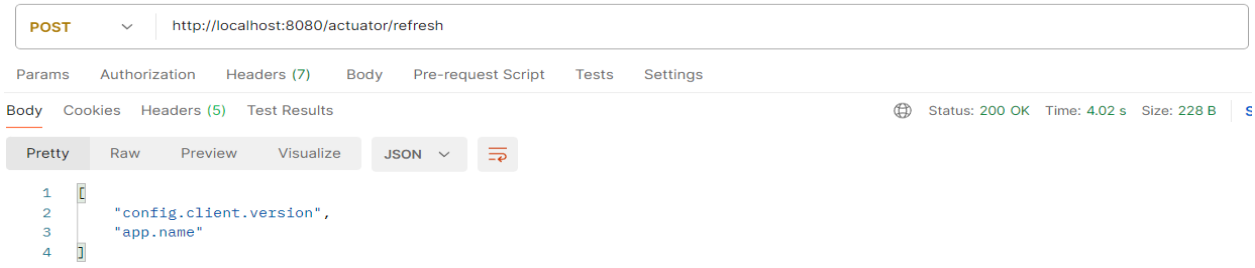
```yaml
management:
  endpoints:
    web:
      exposure:
        include: refresh
```

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

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

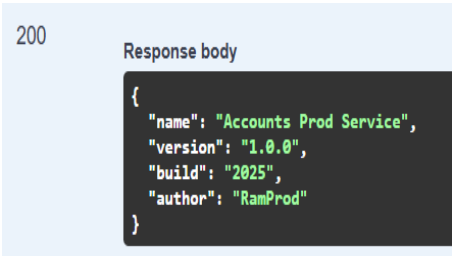


  Then invoke refresh endpoint mentioned in refresh
  POST http://localhost:8080/actuator/refresh



  Before invoking refresh

  After invoking refresh



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

Refresh configurations at runtime using /refresh path

eazy bytes

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

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

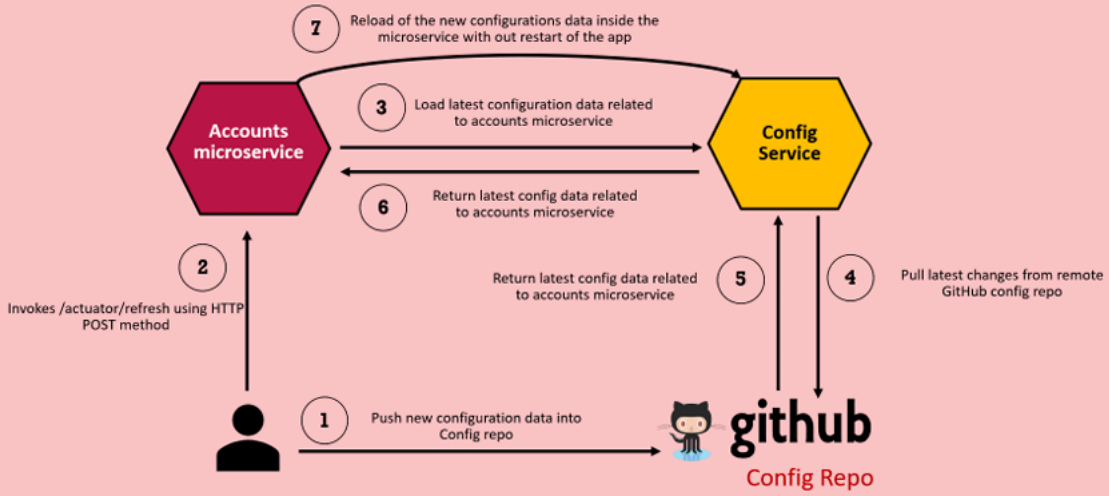
1. Add actuator dependency in the Config Client services: Add Spring Boot Actuator dependency inside pom.xml of the individual microservices like accounts, loans, cards to expose the /refresh endpoint

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

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

Refresh configurations at runtime using /refresh path

eazy bytes



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

❌ Drawbacks

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

- Not suitable for production systems with many services

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

Way 2: Automatic Refresh using Spring Cloud Bus

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

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

A single refresh triggers updates across all services.

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

Ex: Steps to Integrate Spring Cloud Bus With RabbitMQ

1. Install RabbitMQ

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

The rabbitmq:4-management image includes two parts:

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

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

Run 'docker run --help' for more information

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

2. Add Spring Cloud Bus Dependency in Each Microservice

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

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

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

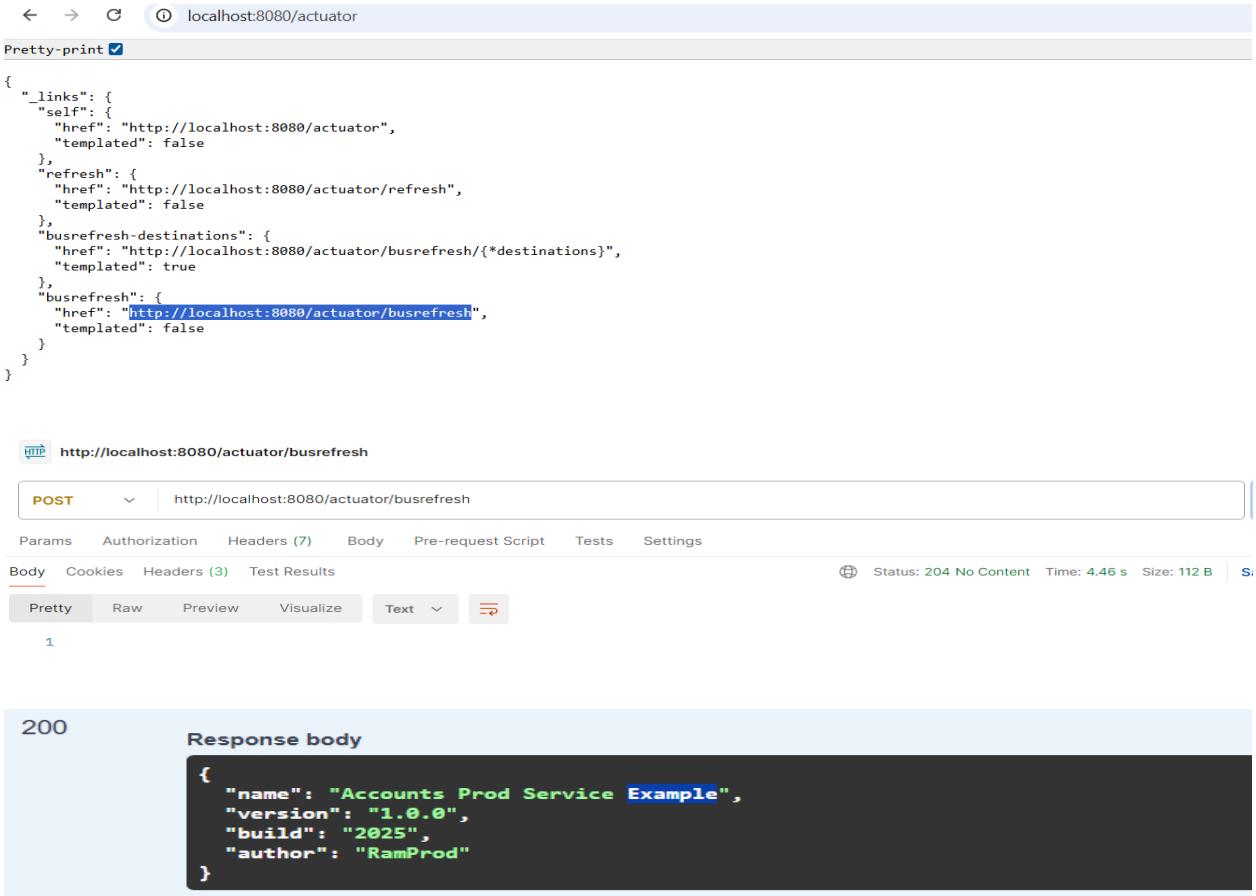
```yaml
rabbitmq:
  host: localhost
  port: 5672
  username: guest
  password: guest
management:
  endpoints:
    web:
      exposure:
        include: refresh, bus-refresh
```

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

4. Trigger Config Refresh

After updating configuration in Spring Cloud Config repo, call:

All microservices connected to RabbitMQ will reload their configuration automatically.



Refresh configurations at runtime using Spring Cloud Bus

eazy bytes

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

Below are the steps to follow,

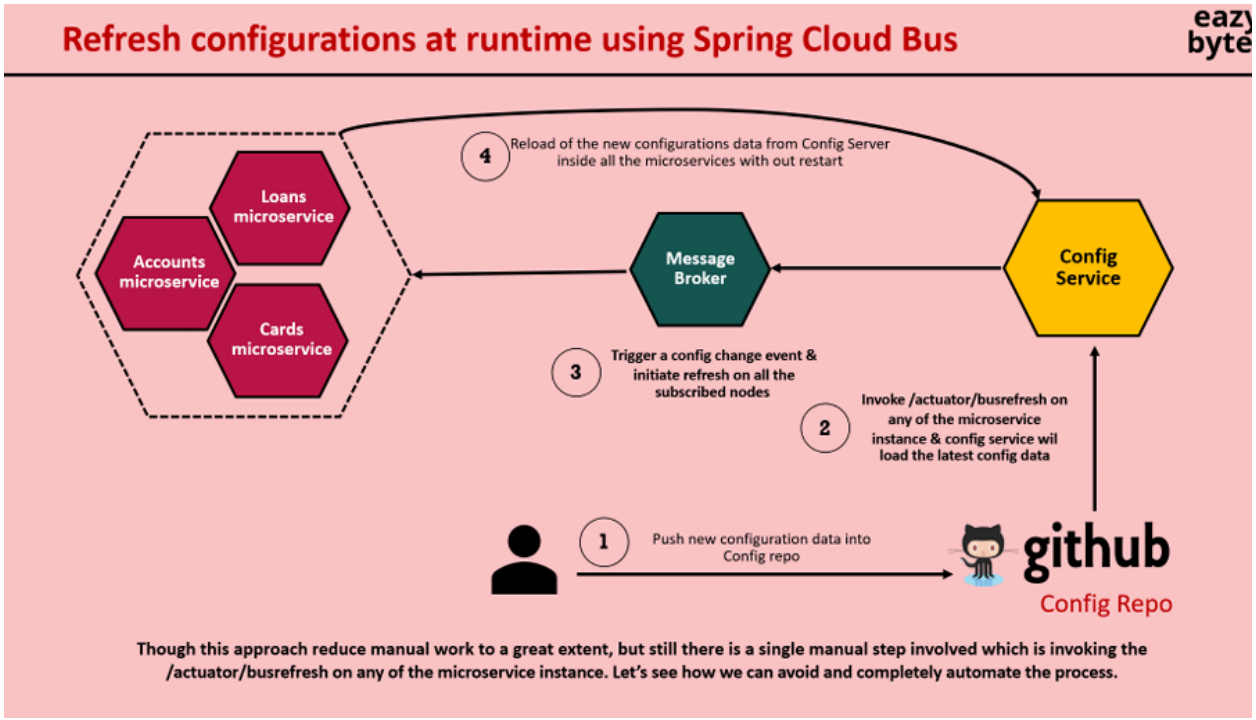
1. **Add actuator dependency in the Config Server & Client services**: Add Spring Boot Actuator dependency inside pom.xml of the individual microservices like accounts, loans and cards to expose the /busrefresh endpoint

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

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

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

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



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

Drawback

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

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

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

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

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

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

- Expose webhook endpoint
-

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

Config Server needs RabbitMQ

It must publish Bus Refresh events to all microservices.

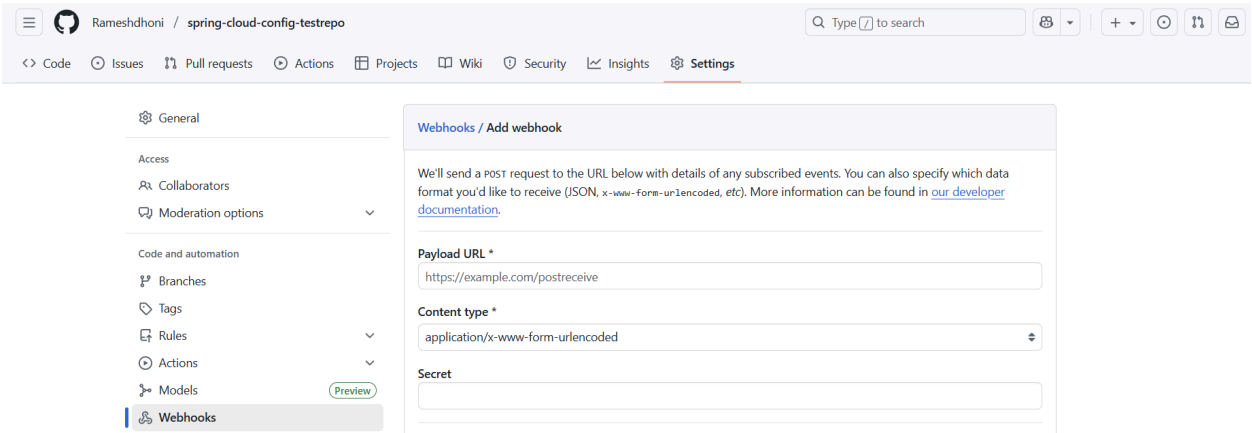
Config Server needs all Actuator endpoints exposed

Especially:

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

Including "*" makes all actuator endpoints available.

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



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

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

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

eazy bytes

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

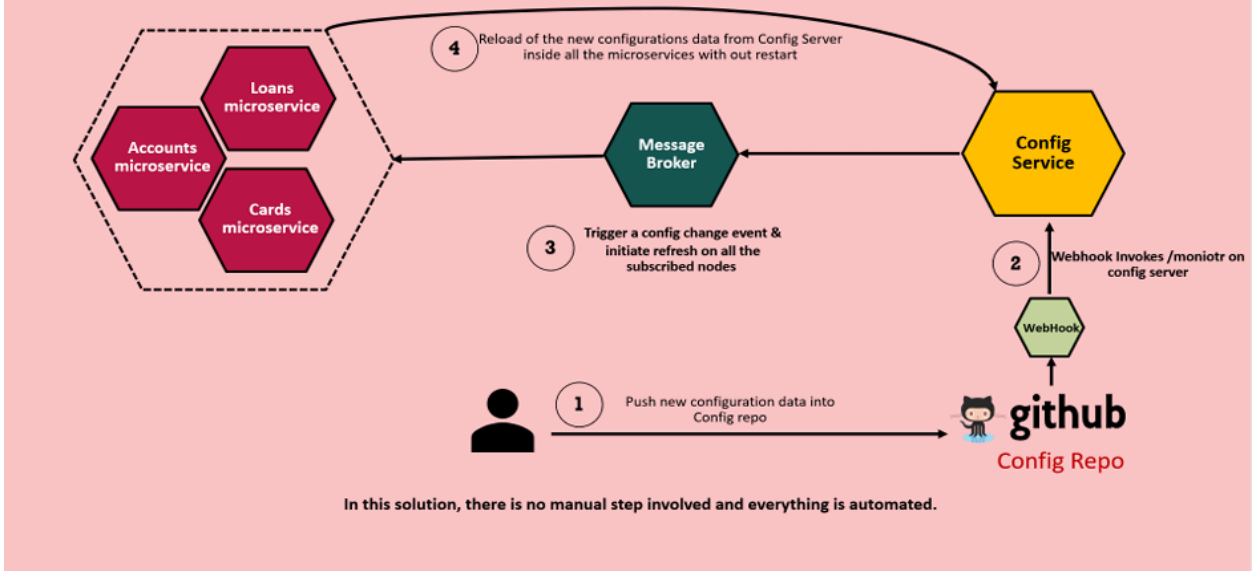
1. Add actuator dependency in the Config Server & Client services: Add Spring Boot Actuator dependency inside pom.xml of the individual microservices like accounts, loans, cards and Config server to expose the /busrefresh endpoint
2. Enable / busrefresh API : The Spring Boot Actuator library provides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

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

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

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

eazy bytes

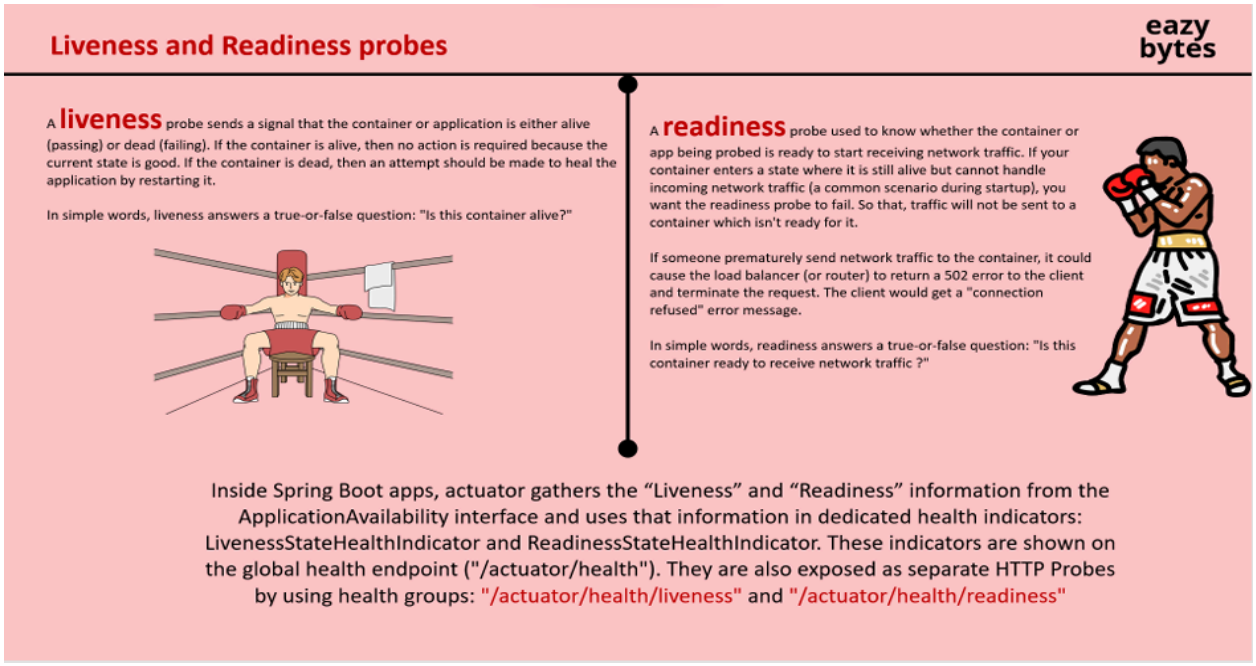


In this solution, there is no manual step involved and everything is automated.

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

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

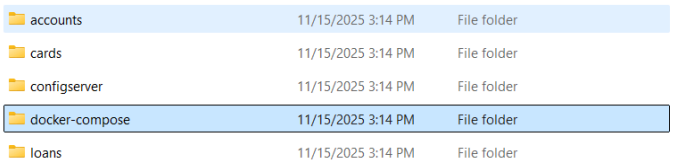
- All services run together
- Config Server reads from Git + Webhook + RabbitMQ
- Microservices auto-refresh on config changes
- Liveness & Readiness probes determine container health
- Multiple environments (dev, qa, prod) run using separate compose files
- Docker images are built once and reused everywhere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



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

This folder will contain:

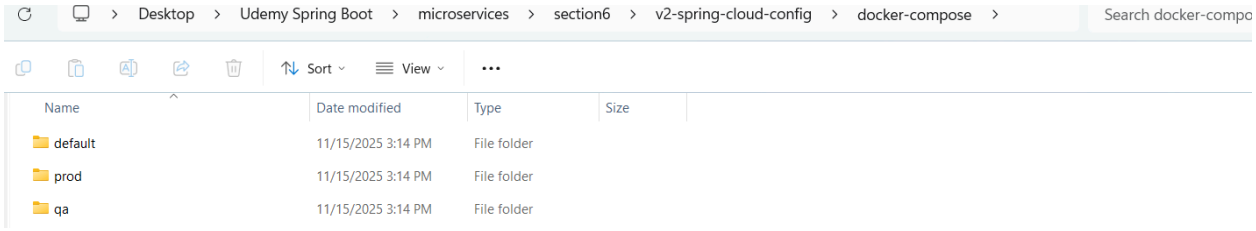
```
docker-compose/
  |-- dev/
  |     |-- docker-compose-dev.yml
  |
  |-- qa/
  |     |-- docker-compose-qa.yml
  |
  |-- prod/
        |-- docker-compose-prod.yml
```

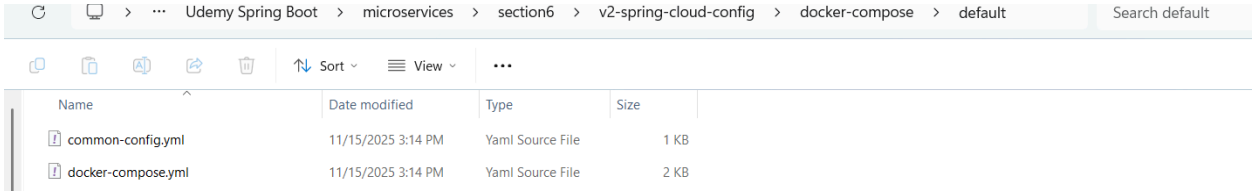


Each environment folder has its own compose file.

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

This ensures clean separation of ENV-based configs.





2. Add Services to Compose File

(A) docker-compose.yml → main file

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

docker-compose.yml

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

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


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

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

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

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

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

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

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

Common-config.yml

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

Detailed Breakdown of Each Service

1️⃣RabbitMQ Service

Docker Compose:

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

Explanation:

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

Health Check:

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

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

Extends:

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

Meaning:

- shared network config

2️⃣ Config Server

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

Explanation:

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

Health Check:

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

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

Extends:

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

Meaning:

- memory limit
- RabbitMQ host
- network config

3️⃣ Microservices (accounts, loans, cards)

Example: **accounts**

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

Explanation:

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

Extends → configserver config

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

This injects:

```
SPRING_PROFILES_ACTIVE=default
```

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

SPRING_RABBITMQ_HOST=rabbit

So:

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

Same applies for:

- loans (8090)
- cards (9000)

4. Build Docker Images for All Microservices

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

5. Push Docker All Images to Docker Hub

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

6. Run Environment Using Docker Compose

Example: run dev

cd docker-compose/dev

docker compose up -d

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

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

7. Switching Environments

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

Step 1 — Bring down existing containers

docker compose down

- Stops all running containers

Step 2 — Start containers with a new environment

To work in QA:

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

To work in PROD:

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

This is the whole point of immutable deployment.

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

Once containers are running:

Whenever you push config changes to Git:

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

This works even inside Docker.
