1. jdbc
2. WAR vs embedded containers
3. CDC
4. **@RequestParam and @PathVariable:** Used to extract values from URL queries and path parameters.
5. **Application Events:** Using ApplicationListener or @EventListener for handling custom events.
6. **Spring Cloud Gateway:** A powerful API gateway for routing requests.
7. **Spring Cloud Config:** Centralized configuration management for microservices.
8. **Circuit Breaker (Resilience4j/Hystrix):** To handle failures gracefully in microservices.
9. **Load Balancing (Spring Cloud LoadBalancer):** Client-side load balancing between microservices.
10. **Saga Pattern:** Managing long-running transactions across multiple services.
11. **Event-Driven Microservices:** Using messaging queues like **RabbitMQ** or **Kafka**.
12. **Distributed Caching (Redis):** Using Redis for caching in a distributed system.
13. **Rate Limiting:** Controlling the rate of requests to a service.
14. **Consul/Zookeeper:** Alternative tools for service discovery and configuration management.
15. Reactive Programming:

 **WebFlux:** Building non-blocking reactive applications using Spring WebFlux.

 **Mono and Flux:** Understanding reactive types in Spring.

 **Reactive Repositories:** Using ReactiveCrudRepository with MongoDB.

1. **Caching with Spring Cache:** Using @Cacheable and @CacheEvict.
2. **Validation:** Using @Valid and @NotNull for input validation.
3. **AOP (Aspect-Oriented Programming):** Using @Aspect and @Before, @After for cross-cutting concberns.
4. **@Scheduled:** Scheduling tasks using @Scheduled.
5. **@Async with ExecutorService:** For controlling thread pools in async methods.
6. **Swagger/OpenAPI:** Documenting APIs using **Springdoc OpenAPI** or **Swagger**.
7. **Liquibase/Flyway:** Managing database versioning with **Liquibase** or **Flyway**.
8. Spring Boot Application Events and Listeners:
9. Persistence unit
10. Listeners
11. Singleton
12. @Builder
13. Spring boot starter validation
14. @Pattern
15. Enumb
16. Feign Client
17. @RestCOntrollerAdvice
18. @Exceptionandeler
19. Fall tolerance
20. @CircuitBreaker
21. RateLimerter
22. @Retry

**Java**  
string builder

String buffer

1) Design patterns / Solid principles

4) JAVA 8 (Optional , Date and time , class reference)

5) Improvements done in Java 8 in collections and garbage collectors

6) Multithreading with Executers and synchronization, life cycle and inter thread communication . atomicity, locking and , volatile

7) Exception and its hierarchy, Exception propagation, throw and throws, Custome exception

8) Fundamental concepts (Loops and Patterns coding , String ,array, Oops, JVM , JDK, Memories mamgement, static and final , finalize)

9) Serialization, cloneable

**Web development**

1. **APIs (Application Programming Interfaces)**

An API is a set of rules and tools that allow different software applications to communicate with each other.

**Example:** When an app fetches weather data from a weather service, it uses the service’s API.

**Groovy:-**

Groovy is a dynamic, object-oriented programming language for the Java platform. It’s known for being more concise and easier to read than Java, with features like simpler syntax, closures (like anonymous functions), and support for scripting. Groovy can be used wherever Java is used, and it integrates seamlessly with existing Java code, making it popular for tasks like scripting, automation, and testing in Java-based projects.

1. **HTTP (Hypertext Transfer Protocol)**

Is the basic communication protocol used by web browsers and servers to exchange information. When you type a website URL and hit enter, HTTP is the "language" that makes your browser ask the server for the web page, and the server responds by sending the page back. It's the foundation of data exchange on the web.

**Purpose:** HTTP is the language used for communication between web browsers and servers.

**Example:** When you visit a website, your browser uses HTTP to request the page from the server.

1. **RESTful Web Services**

REST (Representational State Transfer) is a design style for building APIs. RESTful web services are APIs that follow the REST principles and use standard HTTP methods (GET, POST, PUT, DELETE) to interact with resources.

**Example:** A RESTful API might allow you to get data from a server (GET request), add new data (POST request), update data (PUT request), or delete data (DELETE request).

Think of **APIs** as the overall concept of how software interacts,

**HTTP** as the language used for these interactions on the web, and

**RESTful web services** as a popular, simple way of designing APIs using that language.

**Spring Boot**

Spring Boot is a framework that simplifies building Spring-based applications. It offers features like auto-configuration, embedded servers (like Tomcat), and production-ready metrics. With Spring Boot, you can start a new project quickly without extensive configuration.

**Summary:** Spring Boot simplifies building Spring-based applications by providing auto-configuration, embedded servers, and minimal setup.

**Actuator:-**

Spring Boot Actuator is a tool that helps you keep an eye on your Spring Boot application. Think of it like a dashboard for a car; it shows you important information about how the application is running.

**Key Points:**

* **Health Checks:** It can tell you if your application is working properly or if something is wrong. For example, it can check if the database is connected.
* **Metrics:** It provides data about how your application is performing, like how many requests it’s handling or how much memory it's using.
* **Management Endpoints:** Actuator offers several "endpoints" (URLs) that you can access to get this information. For example, you might visit /actuator/health to check the health of your app.
* **Easy to Use:** You just add a dependency to your project, and you get these features without much extra work.

**AOP (Aspect-Oriented Programming):-**

 in **Spring Boot** is a way to add extra behavior to your code **without modifying the actual code itself**. Think of it like adding reusable "side tasks" that can run **before**, **after**, or **around** the main logic of your methods.

Used for Logging, Security (which u were writing in every method), **Measure performance** (how long a method takes), **Handle security checks** (who is allowed to call a method), etc.

Instead of writing logging, timing, or security code **inside every method**, you can write it **once** using AOP. AOP lets you:

* Write the "extra task" (like logging) in one place, called an **Aspect**.
* Automatically apply it to any method you choose.

@Aspect

**ApplicationContext:-**

The **ApplicationContext** in Spring is the central interface to the Spring framework, managing the lifecycle of beans (objects) and providing many powerful features to your application. It’s like a **container** that holds and manages all the objects (beans) you define in your application.

Think of the **ApplicationContext** as a **bean factory** with extra superpowers:

* It creates and manages all the objects (beans) in your application.
* It handles dependencies between those beans automatically (Dependency Injection).
* It provides utilities like event handling, message resolution, and environment settings.

**Key Responsibilities of ApplicationContext:**

1. **Bean Management**:
   * It reads the configuration (XML, annotations, or Java-based) and creates and manages the lifecycle of beans.
2. **Dependency Injection (DI)**:
   * Automatically wires beans together based on their dependencies.
3. **Event Handling**:
   * Supports application-wide event publishing and listening.
4. **Internationalization (i18n)**:
   * Provides a way to load localized messages for different languages.
5. **Access to Spring Features**:
   * Lets you access utilities like property files, resources, and AOP (Aspect-Oriented Programming).

**Types of ApplicationContext:**

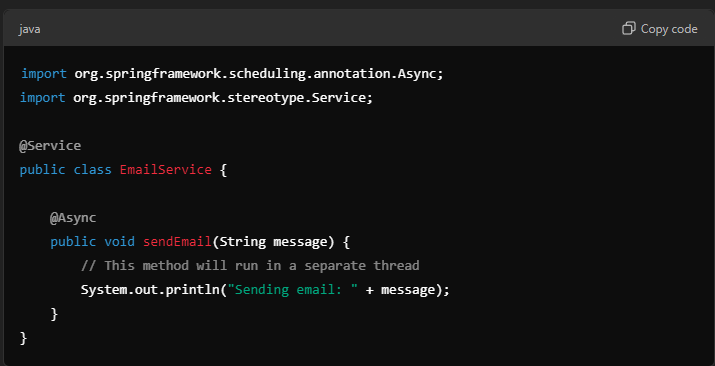
1. **ClassPathXmlApplicationContext**:
   * Loads configuration from an XML file on the classpath.
   * XML-based: ClassPathXmlApplicationContext();
   * We are telling the application that make this class a bean in a resource/xml file by using tags that the application can understand.
2. **AnnotationConfigApplicationContext**:
   * Loads configuration using Java annotations.
3. **GenericApplicationContext**:
   * A flexible context for various use cases.
4. **Web-specific Contexts**:
   * For web applications (e.g., WebApplicationContext).

**@Async:-**

**@Async** is an annotation in **Spring** that makes a method run **in the background**, without blocking the main thread.

**Key Points:**

* It allows a method to run **asynchronously**, meaning it can do its work without waiting for the result before moving on.
* Useful for tasks like sending emails, calling APIs, or processing data that don’t need an immediate response.
* To use it, you need to enable **async** support by adding @EnableAsync to your configuration class.



With @Async, sendEmail() runs in the background, allowing the main process to continue without waiting for the email to be sent.

**@Autowired:-**

is used for dependency injection. It tells Spring to automatically inject a bean into another bean where the annotation is placed. This helps in managing dependencies between different components.

**Summary:** @Autowired injects dependencies automatically between beans in Spring Boot.

**application.properties or application.yml:-**

These files are used for configuring application properties like database settings, server port, or custom application-specific settings. Spring Boot reads these files automatically to set up the environment for the application.

**Summary**: application.properties or application.yml configures settings like database info, server port, and environment variables in Spring Boot.

**@Bean:-**

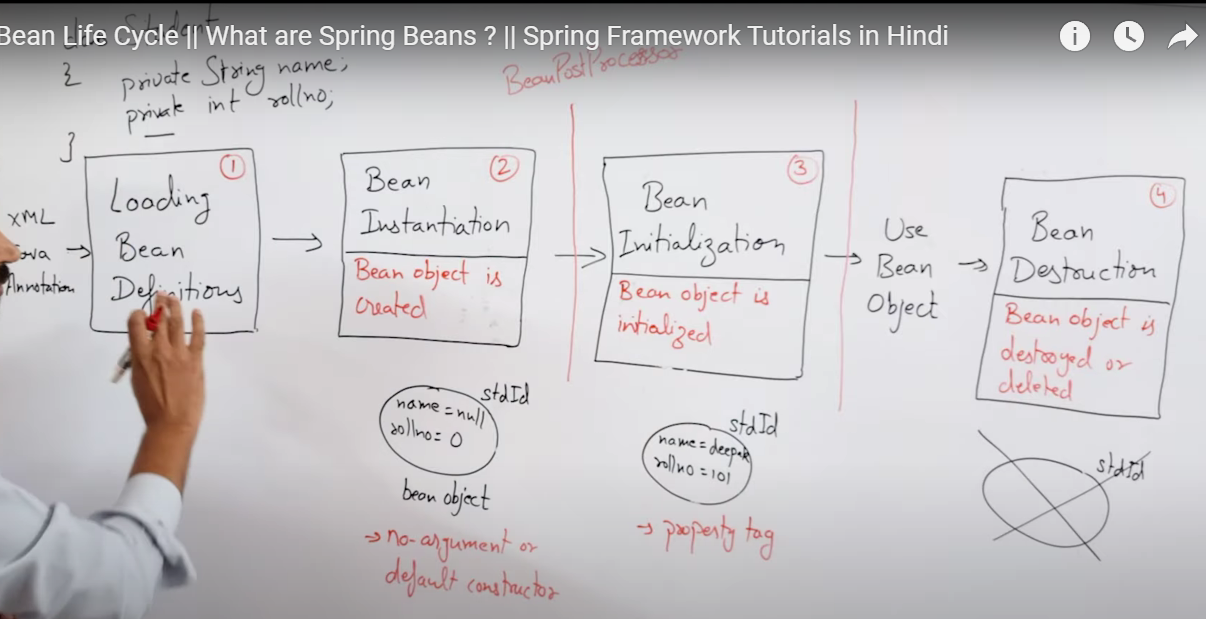
if we declare a method a bean in a configuration class we can inject and use it anywhere in the same Spring application context.

The primary use of declaring a method as a @Bean in a configuration class is to create and manage a Spring bean that can be injected throughout the application. Here are some additional points regarding its use:

1. **Centralized Configuration**: It allows for centralized control over the configuration of the bean.
2. **Custom Initialization**: You can customize the initialization logic of the bean within the @Bean method.
3. **Dependency Injection**: It enables the bean to be injected into other Spring components, facilitating dependency management.
4. **Testing**: The bean can be easily mocked or overridden in test configurations.

So while the main use is for dependency injection, it also supports configuration, customization, and testing.

**Bean Lifecycle:-**

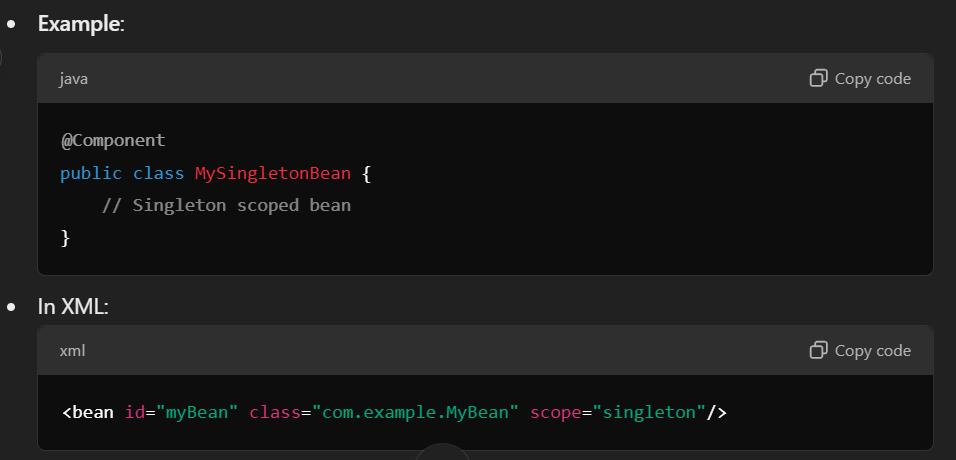
****

**Bean Scopes in Spring (@Scope):-**

A bean scope defines how Spring manages the lifecycle and visibility of a bean in the context of an application.

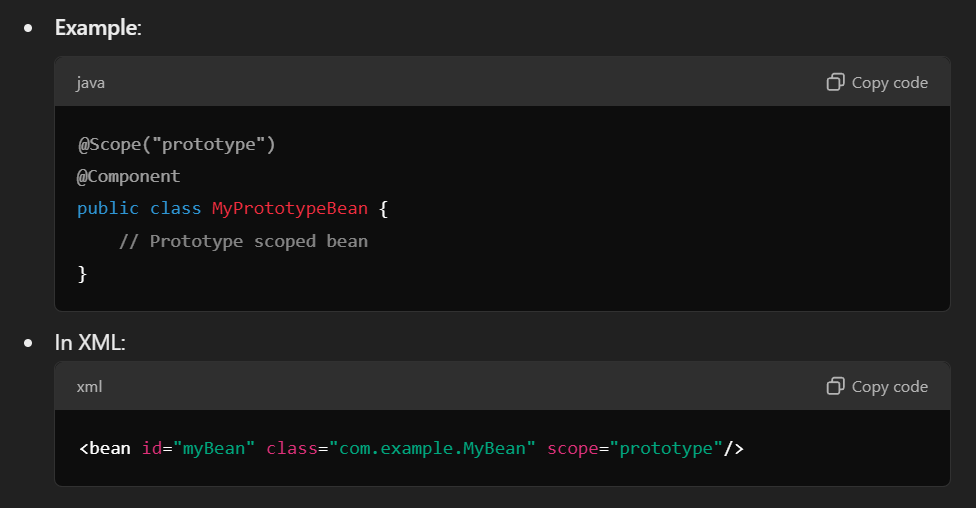
**Singleton Scope (Default Scope):-**

* What it means: A single instance of the bean is created and shared across the entire Spring container.
* Key Points:
  + The same instance is reused whenever the bean is requested.
  + Default scope in Spring.
* Use case: For stateless(does **not hold any data** that is specific to a user or a task) beans or components that are shared globally.



**Prototype Scope:-**

* What it means: A new instance of the bean is created every time it is requested.
* Key Points:
  + Not shared; each request gets a fresh instance.
  + The Spring container does not manage the full lifecycle of a prototype bean (e.g., destruction must be handled manually).
* Use case: For stateful(**stores data** (state) that changes depending on a user or a task) beans or beans that need to maintain unique data per request.



**Other Bean Scopes**

* **Request Scope**: Creates a new bean instance for each HTTP request (used in web applications).  
  Example: @Scope("request")
* **Session Scope**: Creates a new bean instance for each HTTP session.  
  Example: @Scope("session")
* **Application Scope**: A single instance is shared across the whole ServletContext.  
  Example: @Scope("application")
* **WebSocket Scope**: Scoped to a WebSocket session (Spring WebSocket support).

**@Component:-**

is an annotation in Spring Framework that tells the system that a class is a component. Here’s a simple breakdown:

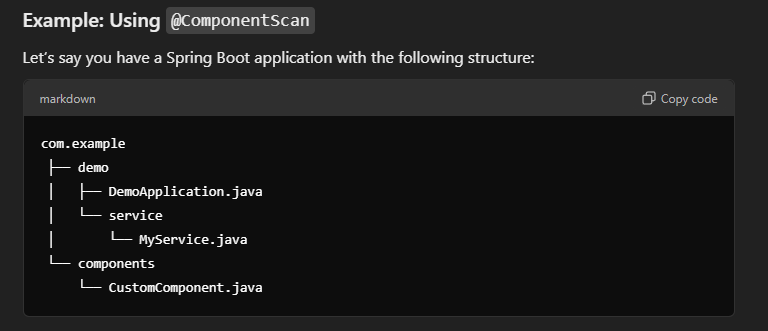
1. **Purpose**: Marks a class as a Spring-managed bean.
2. **Usage**: Spring can automatically detect and register this class when scanning for components.
3. **Dependency Injection**: Helps in managing dependencies, allowing Spring to inject other beans into this component.
4. **Example**: If you have a service class, you can annotate it with @Component to let Spring handle it.

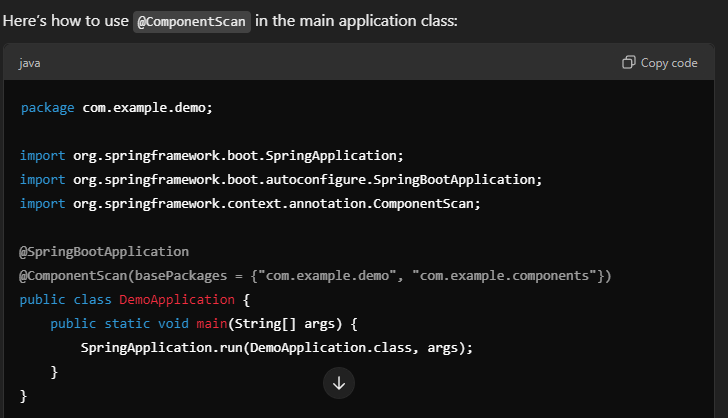
In short, it helps Spring know which classes to take care of!

**@ComponentScan:-**

@ComponentScan in Spring is used to specify which packages Spring should scan for finding components like @Component, @Service, @Repository, and other Spring-managed beans. When Spring scans these packages, it automatically detects and registers the classes annotated with these annotations, making them available for dependency injection.

**Custom Scanning**: You use @ComponentScan explicitly if you want Spring to scan additional packages outside the default package structure. This can be helpful if your components are located in packages that aren’t under the package of your @SpringBootApplication class.





**@SpringBootApplication**: Includes @ComponentScan by default, but only for com.example.demo and its sub-packages.

**@ComponentScan**: Here, we explicitly tell Spring to also scan com.example.components for any Spring-managed beans.

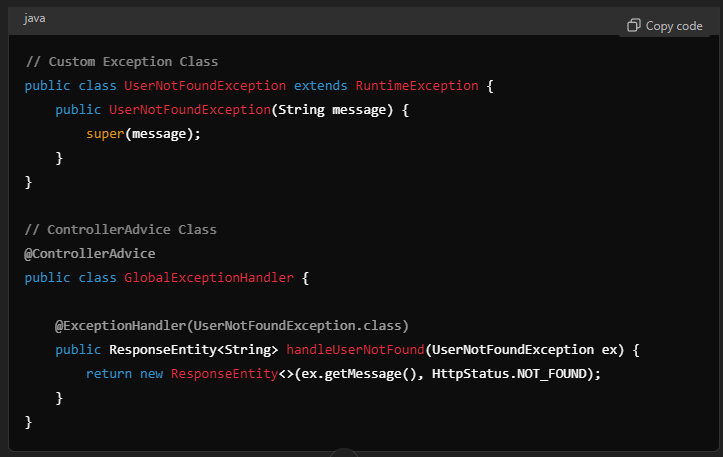
@ControllerAdvice:

* @ControllerAdvice is a Spring annotation used to handle exceptions across the whole application, not just in a single controller.
* It helps centralize error handling logic in one place.

a. @ExceptionHandler:

* @ExceptionHandler inside a @ControllerAdvice is used to specify which method should handle a specific exception.
* When an exception of the given type occurs, this method will be called.

Example:



Throw UserNotFoundException from a any controller method when a user is not found, and it will be handled by this method.

**@CrossOrigin:-**

Enables Cross-Origin Resource Sharing (CORS) for a controller or method to allow requests from different domains.

Use Case: Needed when frontend (e.g., Angular) and backend (Spring Boot) are hosted on different domains to avoid CORS errors.

**Custom exception in Spring Boot:-**

To create a custom exception, define a class extending RuntimeException, then use @ResponseStatus to specify the HTTP status code. You can throw this exception in your service or controller layer to handle specific errors.

**Summary:** Extend RuntimeException and use @ResponseStatus to create custom exceptions with specific HTTP status codes in Spring Boot.

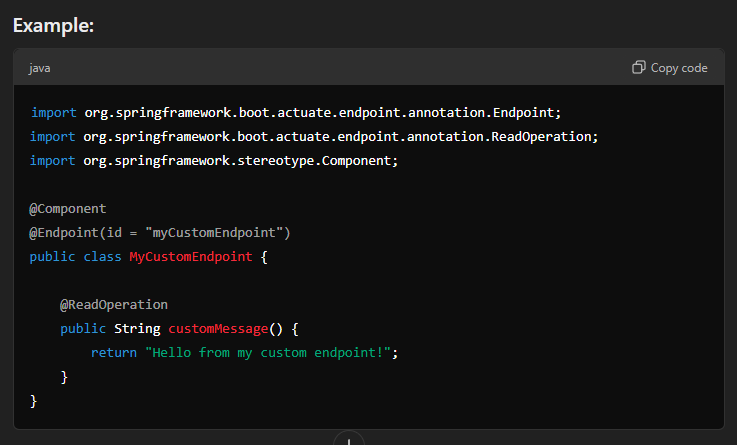
**@EnableAutoConfiguration:-**

Automatically sets up your application's configuration based on the dependencies you include. For example, if you add a web starter, Spring Boot automatically sets up things like a web server and REST controllers. It helps you avoid writing repetitive setup code, letting you focus more on your application's logic. You can still change default settings if needed using properties or custom configurations.

**@Endpoint:-**

@Endpoint lets you create your own tools to monitor or manage your application.

For example, if you want to create a special URL (/actuator/my-custom-endpoint) that gives specific information about your app, you use @Endpoint to set it up.

****

In this example:

* @Endpoint(id = "myCustomEndpoint") creates an endpoint that can be accessed at /actuator/myCustomEndpoint.
* @ReadOperation defines what happens when you make a GET request to this endpoint.

So, @Endpoint is like creating a custom control panel or information page for your application, tailored to your needs.

**Here are some examples of what such custom Actuator endpoints can expose or do:**

* Application Metrics or Status
* Configuration Details
* Debugging Information
* Database Insights
* Business Metrics
* Health Checks
* Actions or Tasks

**@Entity:-**

@Entity in Spring is used to mark a Java class as a database entity, which means it will be mapped to a table in the database.

In simple words, @Entity tells Spring, "This class represents a table in the database." Each instance (object) of this class corresponds to a row in that table, and each field in the class represents a column in the table.

**IoC (Inversion of Control) container:-**

Instead of we creating the object we are telling the spring boot to create the objects. Hence we are inversing the control.

The **IoC container** is a core part of the Spring Framework that **automatically creates and manages objects (beans)** for you. Instead of manually creating objects in your code, you tell Spring how to create them, and it takes care of the rest—like managing their lifecycle and dependencies.

In simple terms, **the IoC container takes control** of creating and managing the objects your application needs, rather than you doing it yourself.

**JPA:-**

Spring Data JPA is a part of the Spring Data project that simplifies data access and manipulation in Spring applications. It provides an abstraction layer over JPA (Java Persistence API) and allows you to work with databases using repositories, reducing boilerplate code for data access.

Summary: Spring Data JPA simplifies database operations in Spring applications by providing an easy-to-use repository pattern and reducing the need for boilerplate code.

<https://www.youtube.com/watch?v=GkkcZXF-mD8>

**Logging:-**

Logging is the process of recording information about an application's execution, typically to help track the flow of the application, identify issues, and understand the behavior of the code. It's like keeping a diary or a record of events that happen during the program's operation.

Why Is Logging Important?

Debugging

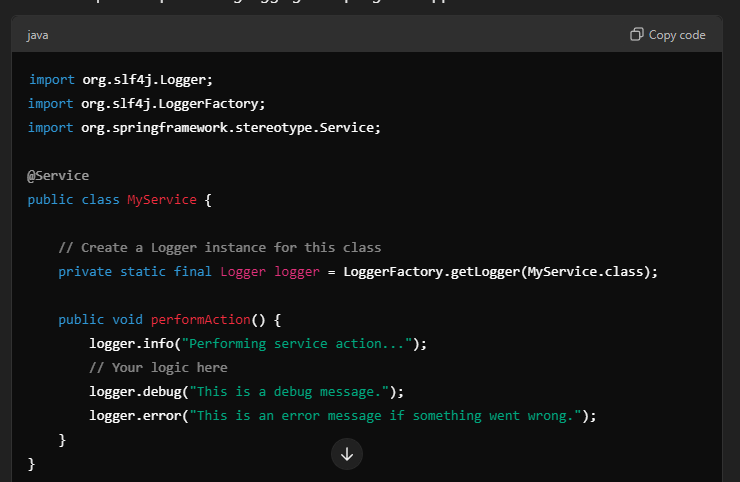
Monitoring

Audit Trails

Understanding Flow

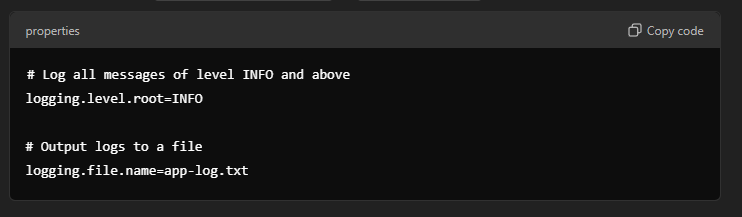
In Java, logging is often done using logging frameworks like SLF4J, Logback, or Log4j. Spring Boot, by default, uses SLF4J with Logback as its logging framework.

**Here's a simple example of using logging in a Spring Boot application:**

****

**Output**

Logs are usually output to the console or written to log files. In a Spring Boot application, you can configure this in the application.properties or application.yml:



Logging helps keep track of what is happening in the application, making it easier to maintain and troubleshoot issues. Let me know if you'd like a more specific example or explanation!

**Maven:-**

Maven is a tool that helps manage Java projects. It simplifies adding libraries (dependencies), building code, and packaging it into JAR or WAR files. Think of it as an assistant that automates repetitive tasks in Java development.

**OAuth:-**

OAuth (Open Authorization) is a way for apps and websites to share your data securely without needing your password.

**Summary:**

In a Spring Boot application, **OAuth** is used to authenticate users through trusted third-party providers and to control **who can access** which parts of the app. It’s like a gatekeeper that ensures only the right users get through, without them having to share their passwords directly with your app.

Example:- Login using google or Facebook or GitHub etc.

<https://www.youtube.com/watch?v=wfaKvQ0qY3E>

**Pact:-**

Pact is a contract testing tool used in microservices architecture. It focuses on ensuring that the communication between different microservices is reliable and that services integrate with each other as expected. It is particularly useful for testing APIs between a provider (a service that offers an API) and a consumer (a service that uses that API).

**Explanation in Simple Words:**

* Pact allows you to create contracts (agreements) between two services:
  + Consumer: The service that makes requests to another service.
  + Provider: The service that receives those requests and responds.
* These contracts define what kind of requests the consumer expects the provider to accept and what responses it expects in return.

**Why Is Pact Useful in Microservices?**

* In a microservices architecture, many services often communicate with each other through APIs.
* Pact ensures that the consumer and provider agree on the structure of requests and responses, so that when one service changes, it doesn’t unexpectedly break another service.
* It helps to test these interactions without needing to run all the services together, making testing faster and easier.

**How Does It Work?**

Pact works by allowing the consumer to define the interactions it expects from the provider. It does this in two phases:

1. **Consumer Contract Test:**
   * The consumer creates a Pact file that describes the expected requests and responses when interacting with the provider.
   * The consumer tests run against these expectations, generating a Pact file as the output.
2. **Provider Verification:**
   * The provider then runs tests to verify that it can fulfill the expectations described in the Pact file.
   * If the provider can successfully meet all the expectations, it means that the two services are compatible.

**Practical Example**

**1. Create a Pact for a Consumer:**

* Suppose you have two services:
  + A Consumer service that requests flight details from a Provider service.
  + The consumer creates a Pact that specifies:
    - If the consumer sends a request to GET /flights/1, it expects a response with the details of flight 1.

**2. Generate a Pact File:**

* The Pact file might look like this:

json

Copy code

{

"consumer": {

"name": "FlightConsumer"

},

"provider": {

"name": "FlightService"

},

"interactions": [

{

"request": {

"method": "GET",

"path": "/flights/1"

},

"response": {

"status": 200,

"body": {

"id": "1",

"destination": "New York",

"departure": "2024-11-01T10:00:00"

}

}

}

]

}

**3. Verify Pact with the Provider:**

* The provider (FlightService) tests its API to ensure it can respond as expected when the consumer sends a request for GET /flights/1.
* If the provider meets these expectations, the contract is valid, and the two services can communicate reliably.

**Key Benefits of Using Pact:**

* Faster Feedback: Allows testing of interactions without needing to start up both services.
* Reduced Integration Issues: Ensures that services agree on how they should .communicate before changes are deployed.
* Improves Collaboration: Facilitates communication between teams working on different microservices by defining clear expectations.

**Summary:**

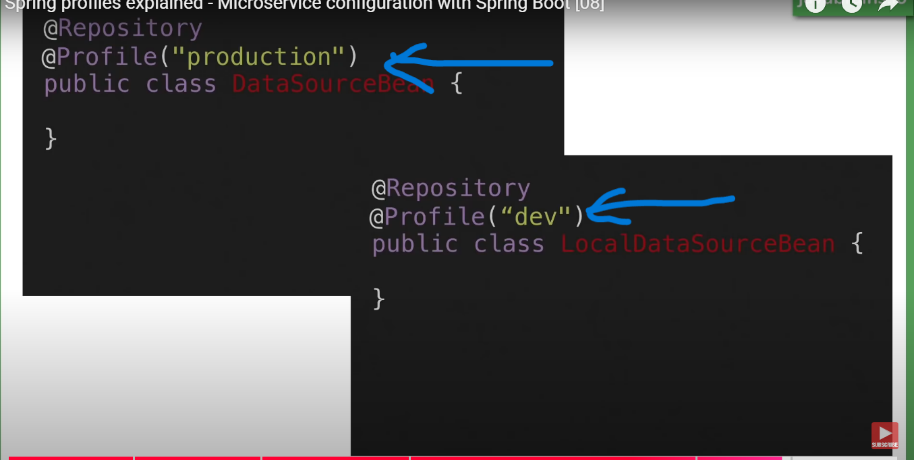
Pact is a powerful tool in microservices architecture for contract testing. It helps ensure that changes in one service don’t break its interactions with another service by creating and verifying contracts for API interactions. This makes it easier to maintain and scale microservices-based applications.

**Profiles:-**

Spring Profiles provide a way to group configuration settings and beans for different environments (e.g., development, testing, production). You can define different profiles in your application.properties or application.yml files, allowing you to activate specific configurations based on the environment.

Summary: Spring Profiles enable you to manage environment-specific configurations in your Spring applications, making it easy to switch between different setups.

Spring profiles for beans



When application is running dev profile spring boot initialize the local data source bean if application is running in production spring boot is going to initialize production data source bean.

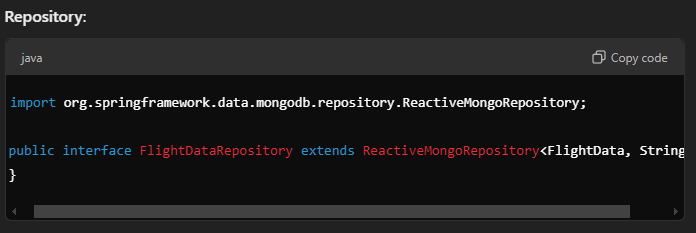
**Reactive extensions in microservices:-**

If u want to build real time applications and deploy it REACTIVE PROGRAMMING is the only way

* Threads don’t get stuck waiting when they don’t have to.
* For example, if a thread is waiting for data from a database, it can switch to handle another task instead of just sitting idle.
* Once the data is ready, the thread can come back and finish the job.

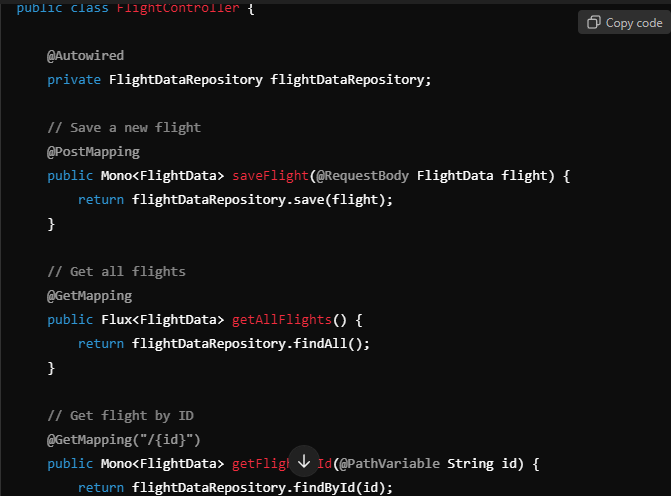
So, Reactive programming uses threads but tries to use fewer threads and make them more active and busy by doing multiple things without blocking. This is different from the traditional way, where each request might have its own dedicated thread, which can be wasteful when waiting for things like database responses.

In short, Reactive programming makes better use of threads to keep everything running smoother and faster when handling many users or tasks at once.



Use ReactiveMongoRepository for Reactive applications

Controller:



* **Mono<FlightData>** is used to return a **single flight** when saving or retrieving a flight by ID.
* **Flux<FlightData>** is used to return a **list of flights** when retrieving all flights.

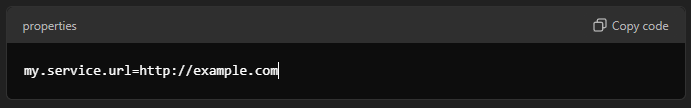
In Spring Boot, the concepts related to building **reactive applications** fall under the **Spring WebFlux** framework.

**Relaxed binding:-**

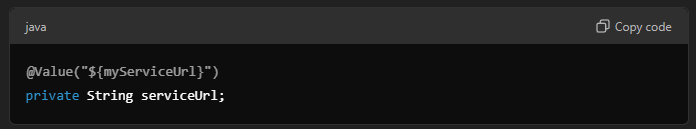
* It means you don't have to follow exact casing or syntax when defining configuration properties.
* Spring Boot is smart enough to match properties that look similar, even if they have slight differences in naming.

**How It Works:**

* Spring Boot can bind **property names** in different formats to **Java fields**.
* For example, suppose you have a property called my.service.url in your application.properties file:



In your Java class, you could bind this property to a field using the @Value annotation or a @ConfigurationProperties class, and it would still work with different styles, like:



**@ResponceBody:-**

When you call this /hello endpoint, it directly returns **"Hello, world!"** as plain text, instead of looking for a web page.

@RestController is a shortcut annotation in Spring Boot that combines @Controller and @ResponseBody.

**ResponseEntity<String>:-**

**ResponseEntity<String>** is a class in Spring that represents an HTTP response, including its status code, headers, and body content.

**Key Points:**

* **Generic Type**: The <String> part indicates that the response body will be of type String.
* **Status Code**: You can specify HTTP status codes (like 200 for success or 404 for not found).
* **Headers**: You can add custom headers to the response.



In this example:

* The method returns a greeting message as the body, with an HTTP status of **200 OK**.

**RestTemplate:-**

A Spring class for making HTTP requests to REST APIs.(calls external URL)

Use Case: Call external REST APIs and handle responses easily in a Spring Boot application.

<https://www.youtube.com/watch?v=P91tqdWUHE4>

**Semantic Monitoring:-**

The primary benefit of Semantic Monitoring is to find out the factors which are more profitable to your business. Semantic monitoring along with service layer monitoring approaches monitoring of microservices from a business point of view. It combines monitoring of the entire application along with automated tests.

**Example:**

* Suppose you have a **Payment Service** in your microservice architecture.
* Traditional monitoring will tell you if the Payment Service is **up** and using a certain amount of memory or CPU.
* **Semantic monitoring** would involve making **test payments** at regular intervals to see if they are processed successfully.
  + If the payment is processed correctly, everything is fine.
  + If the payment fails or doesn’t produce the expected outcome, it means there’s a problem that needs attention.

**Summary:**

Semantic monitoring in microservices ensures that your services are delivering the correct business outcomes. It’s about validating the real-world functionality of a service, beyond just checking if it's running, making it a critical part of maintaining a reliable and high-quality microservice-based application.

**@SpringBootApplication:-**

combines @Configuration, @EnableAutoConfiguration, and @ComponentScan to configure and start a Spring Boot application.

**Spring Boot starter:-**

A Spring Boot starter is a dependency that simplifies including common libraries in your project. For example, spring-boot-starter-web includes dependencies for building web applications like Spring MVC and an embedded Tomcat server.

spring-boot-starter, spring-boot-starter-jdbc, spring-boot-starter-jpa, spring-boot-starter-security, spring-boot-starter-app, spring-boot-starter-aop, spring-boot-starter-test

**Summary:** Spring Boot starters bundle common libraries, making it easier to include necessary dependencies, like spring-boot-starter-web for web apps.

**Spring Cloud:-**

Spring Cloud is like the infrastructure for a city iocof microservices, providing tools and patterns to make sure everything works together smoothly. It helps microservices with things like finding each other, sharing data, and staying reliable.

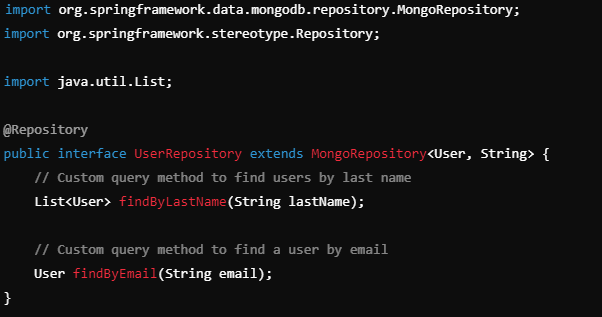
**Key Features of Spring Cloud:**

1. **Service Discovery** (like a city map):
   * Uses **Eureka** to help microservices **find each other** so they can communicate.
   * Example: A **User Service** can find where the **Order Service** is located.
2. **API Gateway** (like a central receptionist):
   * Uses **Spring Cloud Gateway** to **manage requests** to multiple services.
   * Example: All client requests first go through the **API Gateway** before reaching different services.
3. **Configuration Management** (like a central control room):
   * Uses **Spring Cloud Config** to manage **configuration settings** for all services from a **central location**.
   * Example: Instead of each service storing its own settings, they all **fetch settings** from a **central config server**.
4. **Load Balancing** (like traffic control):
   * Uses **Ribbon** or **Spring Cloud Load Balancer** to **distribute requests evenly** across multiple instances of a service.
   * Example: If there are 3 instances of **User Service**, it **spreads the requests** evenly so no instance gets overwhelmed.
5. **Fault Tolerance** (like backup systems):
   * Uses **Hystrix** or **Resilience4j** for **circuit breaking** and **fallbacks** when a service fails.
   * Example: If **Order Service** is down, the **User Service** can use a **default response** instead of crashing.

**Spring Data:-**

**Spring Data** is a part of the larger Spring ecosystem, designed to simplify working with databases and data access layers in Java applications. It provides an easy way to interact with different types of databases—both SQL and NoSQL—by reducing boilerplate code.

It simplifies database interactions, query creation, and repository implementation.

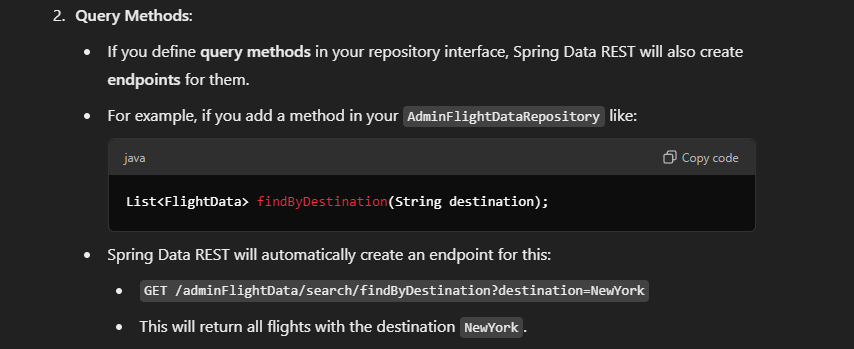


**Spring Data Rest:-**

if i use spring data rest i dont need to wright methods in controller class for getting or post or edit or delete or getbyid or getbyName

Spring Data REST automatically provides the following endpoints for **basic operations**:

* GET /entities – Retrieves a list of all entities.
* GET /entities/{id} – Retrieves a single entity by its ID.
* POST /entities – Creates a new entity.
* PUT /entities/{id} – Updates an existing entity.
* PATCH /entities/{id} – Partially updates an entity.
* DELETE /entities/{id} – Deletes an entity by its ID



**Summary:**

* **Spring Data REST** takes care of **automatically creating** RESTful **endpoints** for **CRUD operations** and simple **query methods** in your repository.
* This saves you from writing the **boilerplate code** for a controller if your API needs are basic.
* It’s great for **prototyping** or when you want to expose your data quickly with minimal setup.

**When You Might Still Need a Controller:**

* If you have **complex business logic** or need to **validate** data before saving it.
* If you want to customize the **structure** of the **response** or **request handling**.
* If you need to define **custom endpoints** that don’t directly map to repository operations.

**Thymeleaf:-**

Thymeleaf is a Java template engine used to create dynamic web pages. It allows you to mix HTML with Java code in a way that makes it easy to display data on a web page.

Key Points:

* It works well with Spring Boot.
* Allows you to bind data to HTML templates (like showing a list of users).
* Helps create views (web pages) that are both readable and editable as plain HTML.

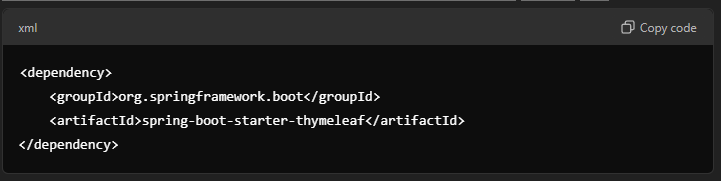
Example:

If you want to display a list of names on a web page, Thymeleaf helps you write HTML with placeholders that get replaced with actual data from your Java code.

To use Thymeleaf with a Spring Boot application, follow these simple steps:

1. Add Thymeleaf Dependency:

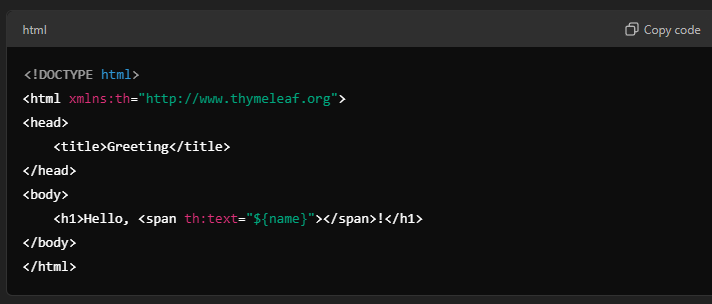
Make sure you have the Thymeleaf dependency in your Maven project’s pom.xml file:

****

This will include Thymeleaf and make it ready for use in your Spring Boot project.

2. Create an HTML Template:

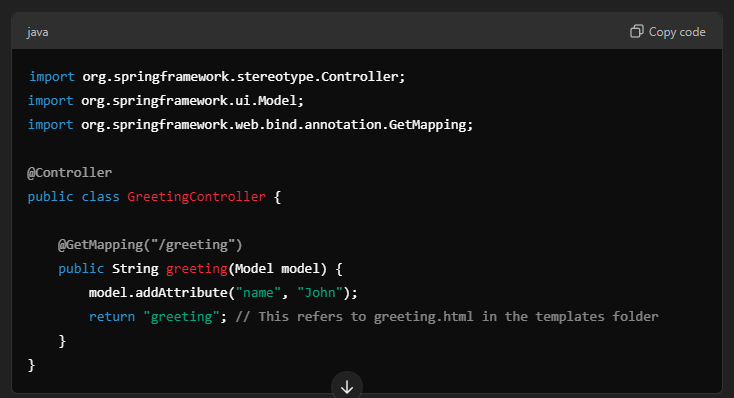
* Inside the src/main/resources/templates folder, create an HTML file (e.g., greeting.html):

****

* Here, th:text="${name}" is a **Thymeleaf** expression that will display the value of the name variable passed from your **Java** controller.

**3. Create a Spring Boot Controller:**

* Create a **controller** class to handle requests and return the **HTML template**:



In this example:

* When a user visits /greeting, the greeting method is called.
* It sets name to "John".
* The greeting.html template is rendered, and "John" will be displayed in the HTML where ${name} is used.

**4. Run Your Application:**

* Start your **Spring Boot** application.
* Visit http://localhost:8080/greeting in your web browser.
* You should see the message **"Hello, John!"**.

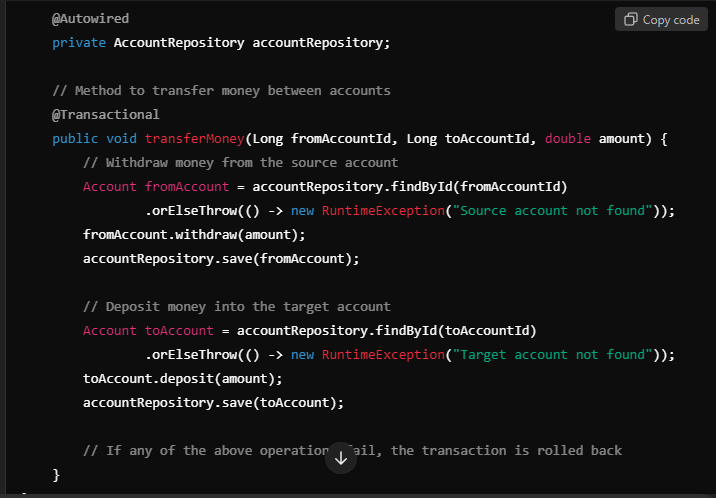
**Summary:**

* **Thymeleaf** templates are placed in the src/main/resources/templates folder.
* Use @Controller in Spring Boot to map requests to HTML templates.
* Use th:text and other Thymeleaf tags to bind data from your **Java model** to the **HTML view**.

**@Transactional:-**

@Transactional in Spring Boot is used to manage database transactions automatically. It ensures that a series of operations on the database either all succeed or all fail together.

So, @Transactional helps to maintain data integrity during database operations by automatically handling rollbacks (undoing changes) if something fails.



Annotate with @EnableTransactionManagement in the main class.

**Summary:**

By using @Transactional on the transferMoney() method, you make sure that either both operations (withdraw and deposit) complete successfully, or neither of them are applied if an error occurs. This helps maintain data integrity during database operations.

**@Value:-**

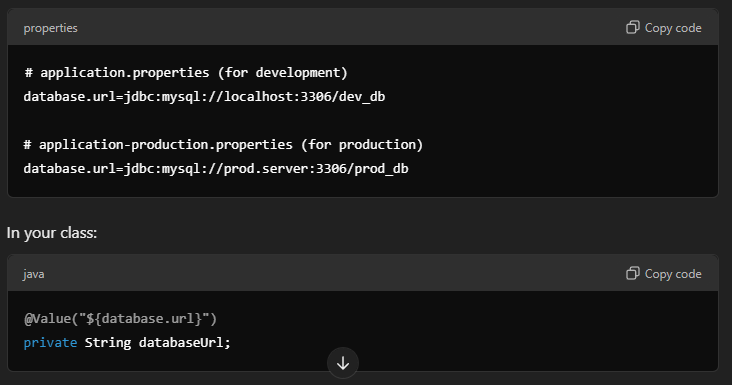
@Value in Spring Boot is helpful because it allows you to easily configure values outside of your code, making your application more flexible, maintainable, and environment-specific. Here’s how and why it's useful:

**1. External Configuration:**

* Instead of hardcoding values directly in your Java classes, you can store them in an external configuration file like application.properties or application.yml.
* This makes it easier to change values like database URLs, API keys, or environment-specific settings without modifying the code.

**2. Environment-Specific Settings:**

* You can have different values for different environments (e.g., development, testing, production) without changing the code.
* For example, you can use @Value to inject a different database connection URL for production compared to your local development environment.



The databaseUrl will automatically change based on the active environment.  
**3. Easier Maintenance:**

* If you need to update a setting, you only change it in one place (the application.properties file), rather than searching through the code.
* This reduces the risk of making mistakes when making updates and makes the application easier to maintain.

**WAR:-**

stands for **Web Application Archive**. It is a file format used to package a **web application** in Java.

**Key Points:**

* **File Extension**: A WAR file has a .war extension.
* **Packaging**: It contains all the components of a web application, including HTML, JSP, Servlets, classes, libraries, and configuration files (like web.xml).
* **Deployment**: WAR files are deployed to web servers or application servers (like Tomcat or Jetty) to run Java web applications.

**Example:**

When you build a Java web application, it is packaged into a WAR file, which can then be easily deployed to a server for users to access through a web browser.

**@webEndPoint:-**

in @Endpoint we can only read but in @webEndPoint we can read, write, edit ect.  
**In Simple Terms:**

* With @Endpoint, you’re mainly **displaying** or **sharing** information.
* With @WebEndpoint, you have the option to **interact** with the application by **updating**, **modifying**, or **deleting** data, making it more versatile for custom management needs.

So, use @WebEndpoint when you want your custom endpoint to handle different types of HTTP requests, beyond just reading data.



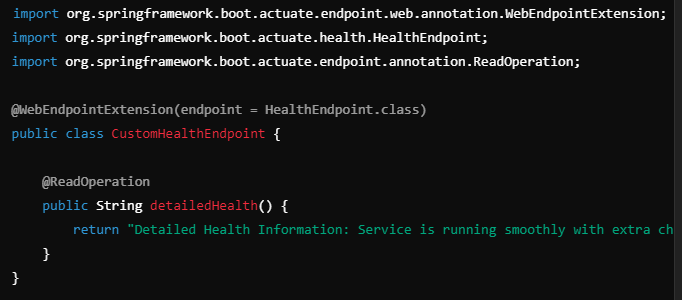
**@WebEndpointExtension:-**

@WebEndpointExtension in Spring Boot is used when you want to extend or enhance an existing Actuator endpoint with additional web-specific behaviours or functionality. It allows you to add new operations (like read, write, or delete) to an existing endpoint without changing the original endpoint’s code.

In Simple Words:

* @WebEndpointExtension is like adding extra features to an existing endpoint, specifically for web access.
* If you have an endpoint already (maybe provided by Spring Boot or your custom one), and you want to customize how it behaves over HTTP without altering the original code, you use @WebEndpointExtension.

**@WebEndpointExtension Example**: Suppose Spring Boot already has a built-in endpoint like health that gives basic health information, and you want to add extra data when someone accesses this endpoint over HTTP:



**Output**: This adds a custom feature to the HealthEndpoint, allowing it to return extra information when accessed via HTTP.

**Difference:-**

**@Controller and @RestController:-**

Use @RestController for REST APIs and @Controller for web pages.

@Controller: Used for handling web page requests. Returns a view (e.g., HTML, JSP). Requires @ResponseBody to return JSON.

@Controller

public class HomeController {

@GetMapping("/home")

public String homePage() {

return "home"; // Renders a view named 'home'

}

}

**@RestController**: Combines @Controller and @ResponseBody. Directly returns data as JSON for REST APIs.

@RestController

public class ApiController {

@GetMapping("/api/data")

public List<String> getData() {

return List.of("Data1", "Data2");

}

}

**Jar Vs War:-**

****

**JAVA**

Functional interfaces Stream pipeline

Jre and jdk

forEach

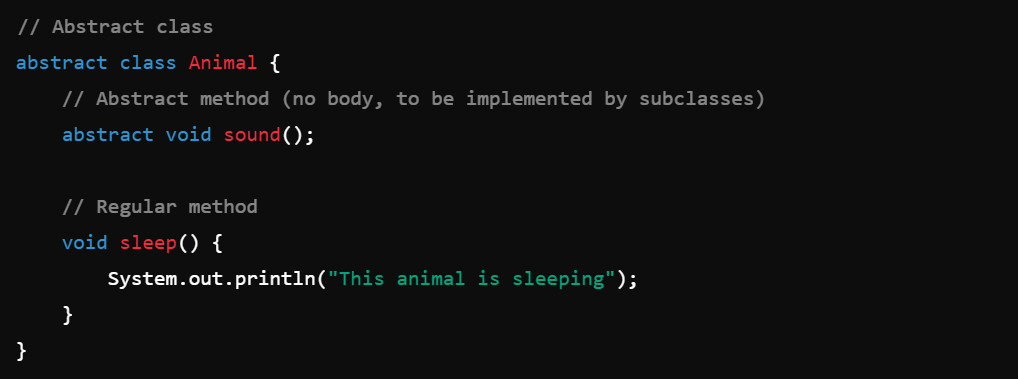
wrapper class

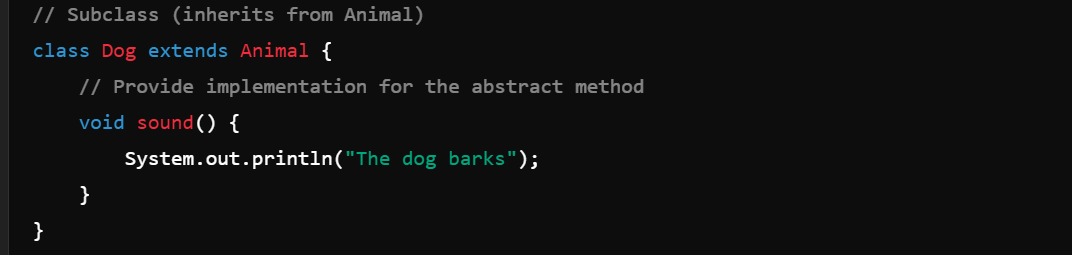
string buffer and string builder

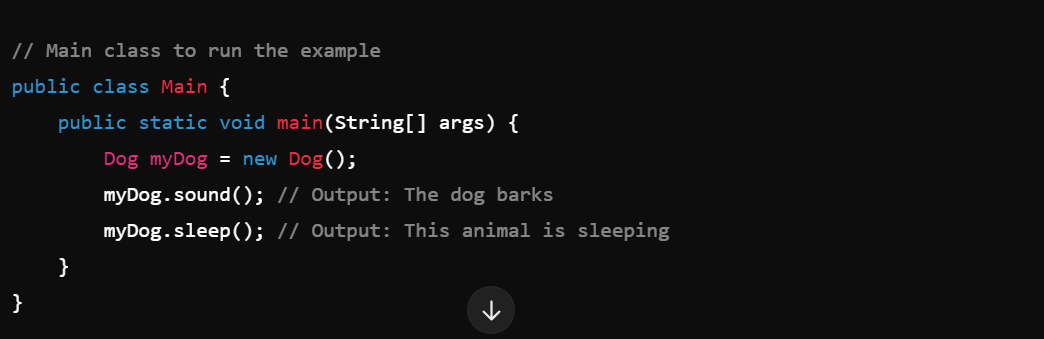
**Multi Threading:-**

**Abstract:-**

Abstract class is a class in which 1 or more abstract methods are present.  
**Abstract Method**: A method without a body, meant to be completed by subclasses.



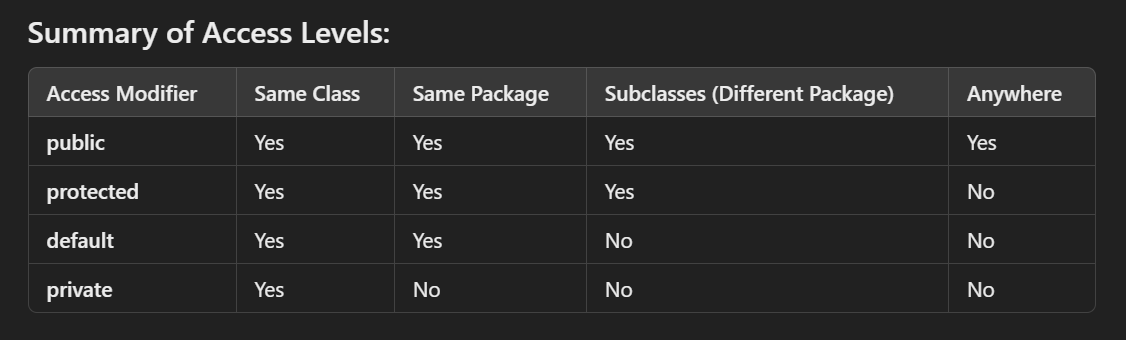




In this example:

* Animal is an abstract class with an abstract method sound() and a regular method sleep().
* Dog is a subclass of Animal and provides the implementation for the sound() method.

**Access modifiers:-**



**Collections:-**

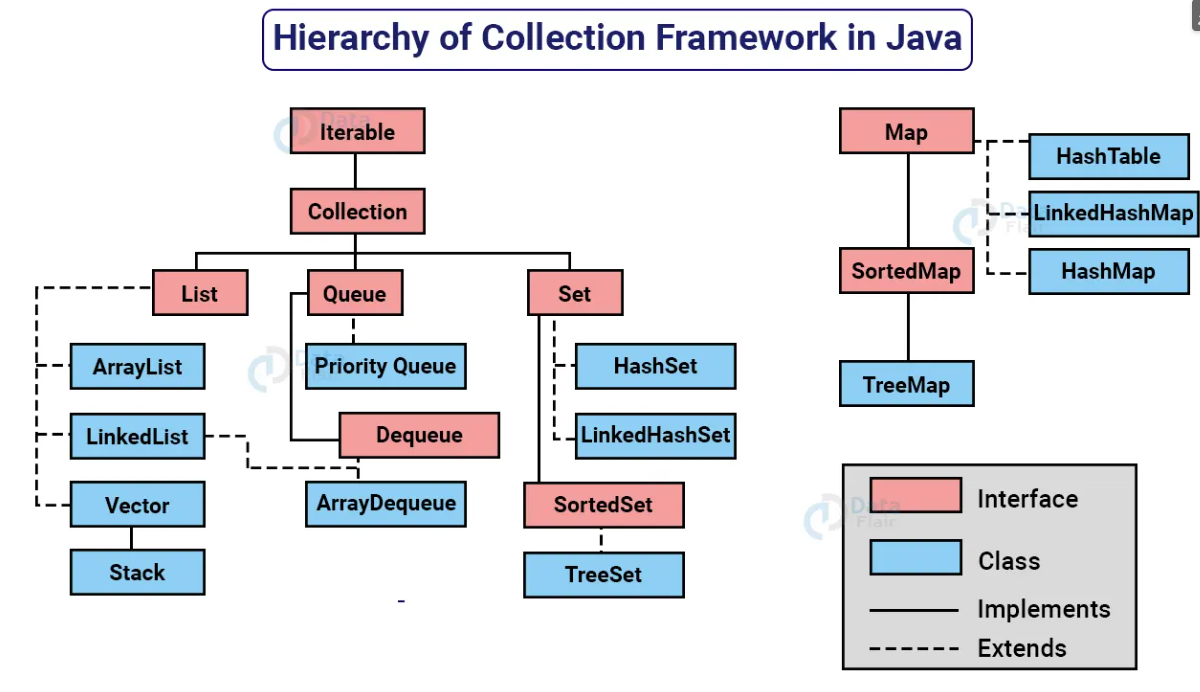
A **collection** in Java is simply a group of objects stored together. It can hold a single type of object (like numbers or strings) and provides methods to add, remove, and manipulate these objects.

For example:

* A List is a collection that stores items in order.
* A Set is a collection that stores unique items without duplicates.

In short, collections help organize and manage groups of data.

**hierarchy of collection framework**

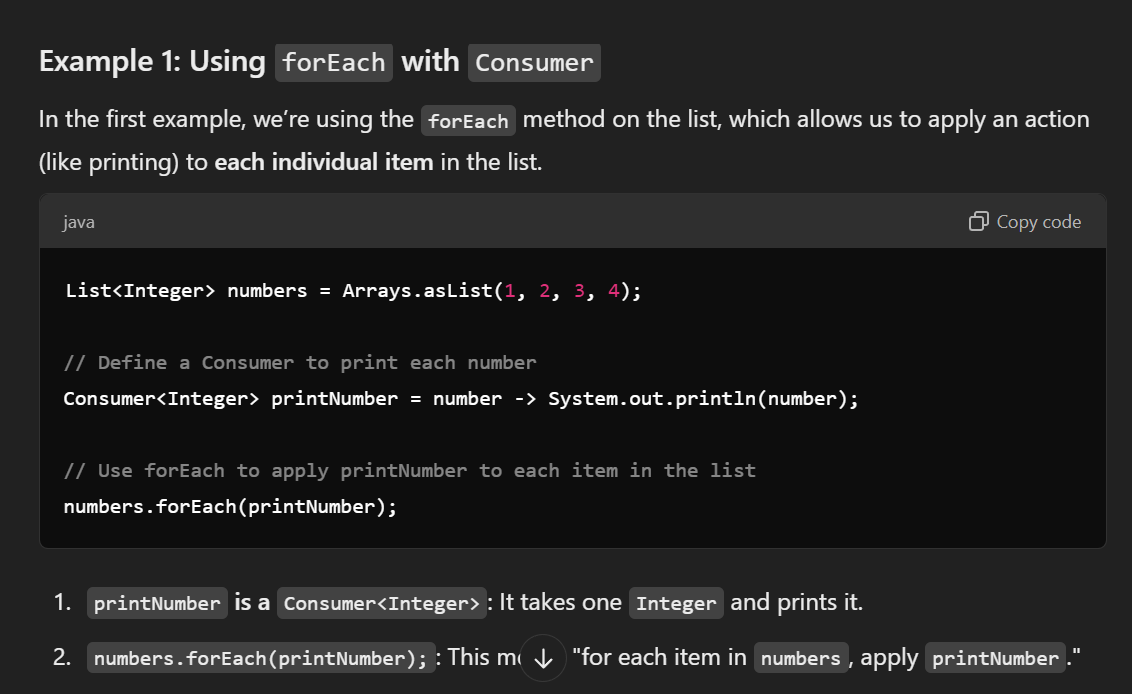


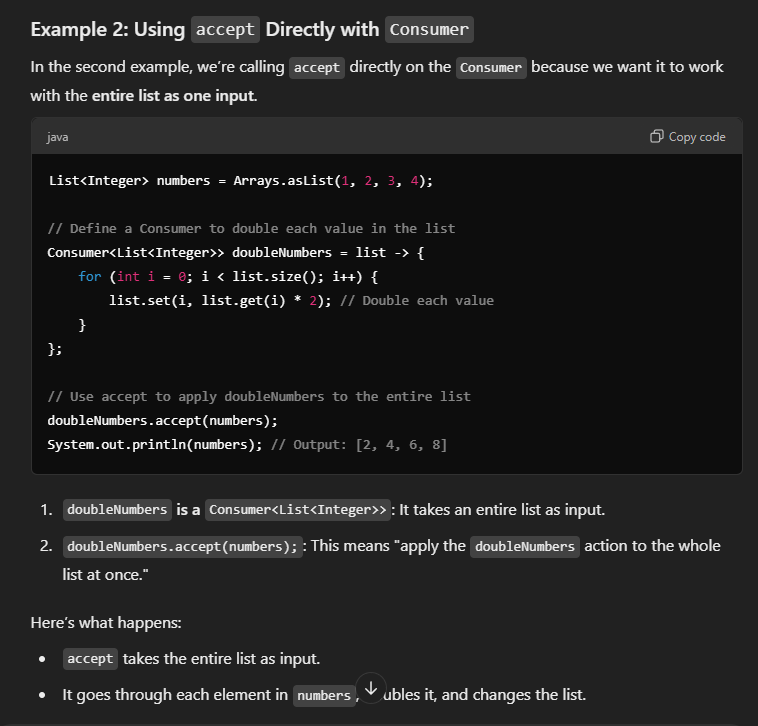
**Consumer:-**

**Definition**:  
A Consumer in Java represents an operation that **accepts a single input and performs an action on it without returning any result**. It’s commonly used for operations like printing or modifying data without needing to return anything.

**Key Methods**

1. **accept**: Executes the action defined in the Consumer.
2. **andThen**: Chains two Consumers together, allowing both actions to run in sequence on the same input.





**Exception Handling:-**

Built in methods in Exception class:

System.out.println(e.getMessage);

System.out.println(e);

e.peintStackTrace();

**Final keyword:-**

If we use final keyword to a class or a method we cannot extend this class.  
cannot use final and static keyword in the abstract class.

**Generics:-**

Generics allow you to define classes, methods, or interfaces that can work with any data type (like Integer, String, etc.) in a safe way without losing type information.

**Generics** are about making that data storage flexible so you can use different types of data without writing multiple versions of the same code.

**Interface:-**

An **interface** is like a contract in programming. It defines a list of methods (functions) that a class *must* have, but it doesn’t give the details of how those methods work.

**Purpose**: Interfaces let you set rules for what a class should do without saying how it should do it.

**Static:-**

* **For variables**:

There is only *one* copy of that variable for the whole class, no matter how many objects (instances) of that class you create.

* **For methods**:

You can call the method without creating an object of the class

class MathUtils {

// Static method for adding two numbers

static int add(int a, int b) {

return a + b;

}

}

public class Main {

public static void main(String[] args) {

// Calling the static method using the class name

int result = MathUtils.add(5, 3);

System.out.println("Result of static method: " + result);

}

}

**Predicate:-**

Is a functional interface used by lambda function which is used in filter method.  
it returns true or false.

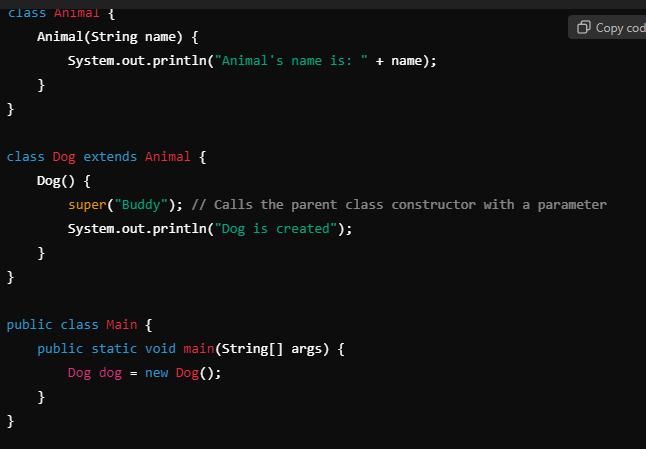


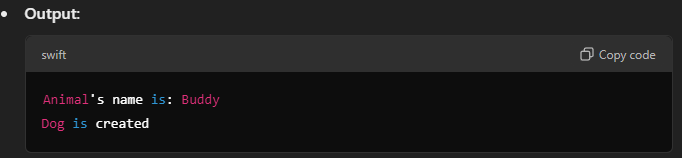
**Super:-**

In Java, the super keyword is used to refer to **the parent class** (or superclass) of the current class. It allows a child class to access the **methods** or **constructors** of its **superclass**.

**In Simple Words:**

* Think of super as a way to reach out to the **parent version** of a method or a constructor.
* It’s useful when you want to **call a method** or **constructor** from the **parent class** while you are inside the **child class**.





**Oops:-**

**Abstraction:-**

**Encapsulation:=**

encapsulation is like a protective wrapper around an object's data, allowing access only through defined methods. This helps maintain data integrity and promotes modular design.  
class Car {

// Private attribute

private String model;

// Constructor to initialize the model

public Car(String model) {

this.model = model;

}

// Public method to get the model

public String getModel() {

return model;

}

}

public class Main {

public static void main(String[] args) {

// Creating a Car object

Car myCar = new Car("Toyota");

// Accessing the model through the public method

System.out.println("Car Model: " + myCar.getModel()); // Outputs: Car Model: Toyota

}

}

**Inheritance:-**

It allows a class (called a **child class** or **subclass**) to inherit properties (fields) and behaviours (methods) from another class (called a **parent class** or **superclass**). This promotes code reuse and establishes a natural hierarchy between classes.

A class can extend only one class but can implement many interfaces.