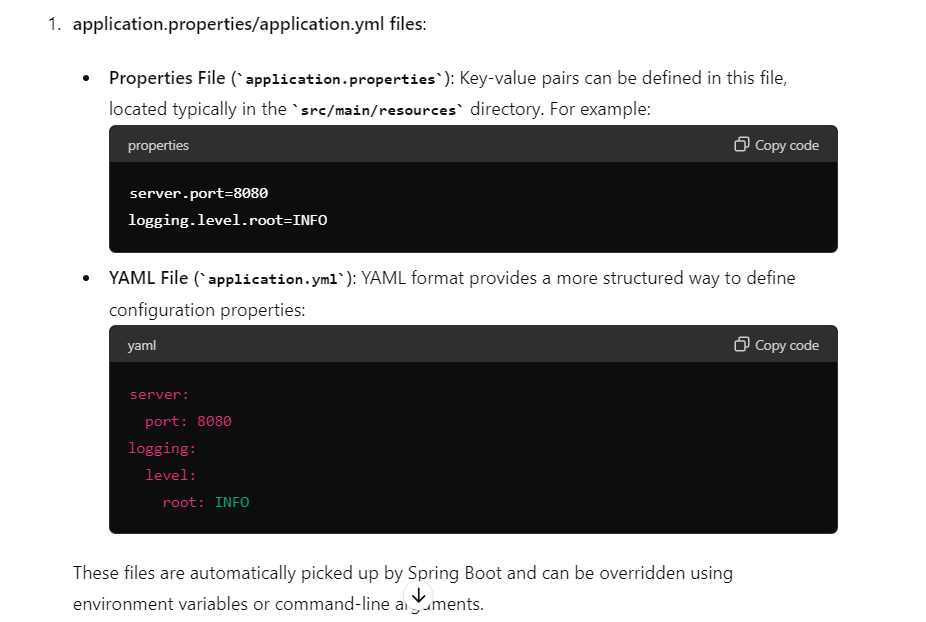
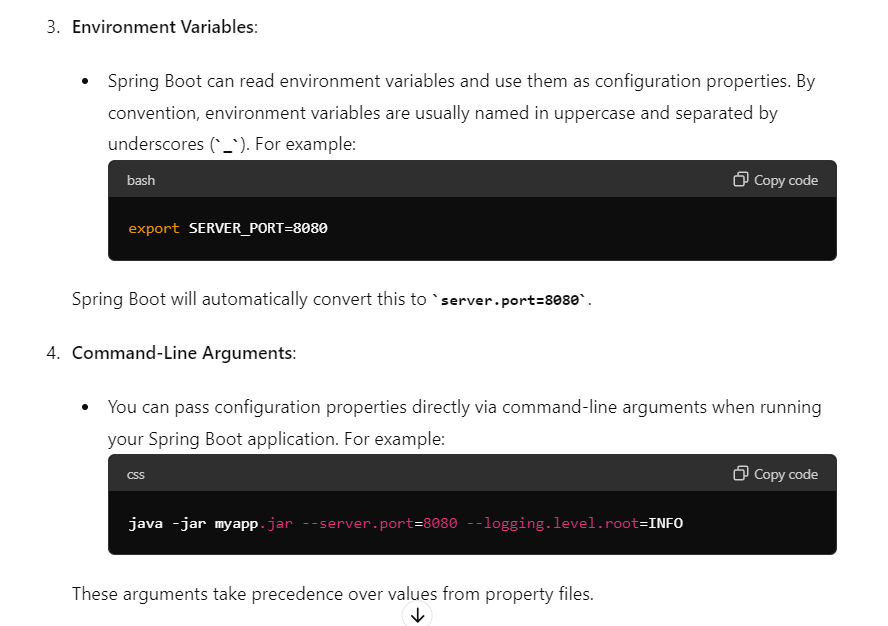
Question with solution

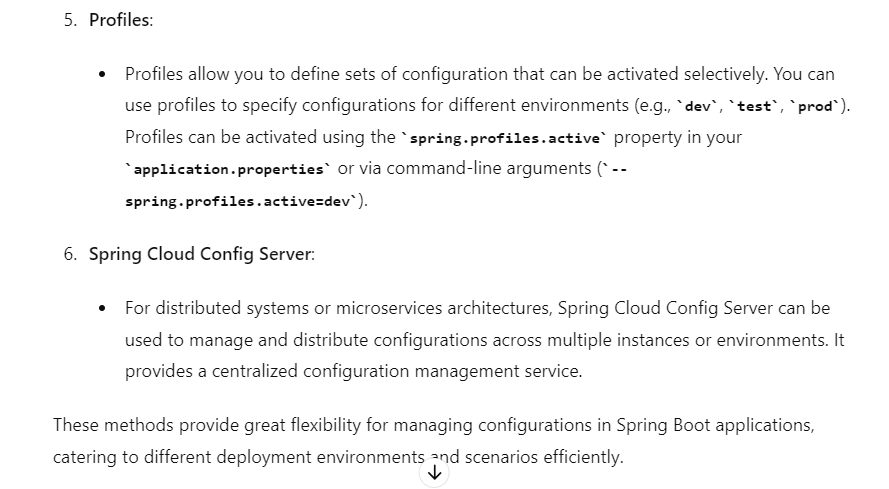
1---------- Springboot

Q1 : What are the different ways to externalize configuration in Spring Boot?

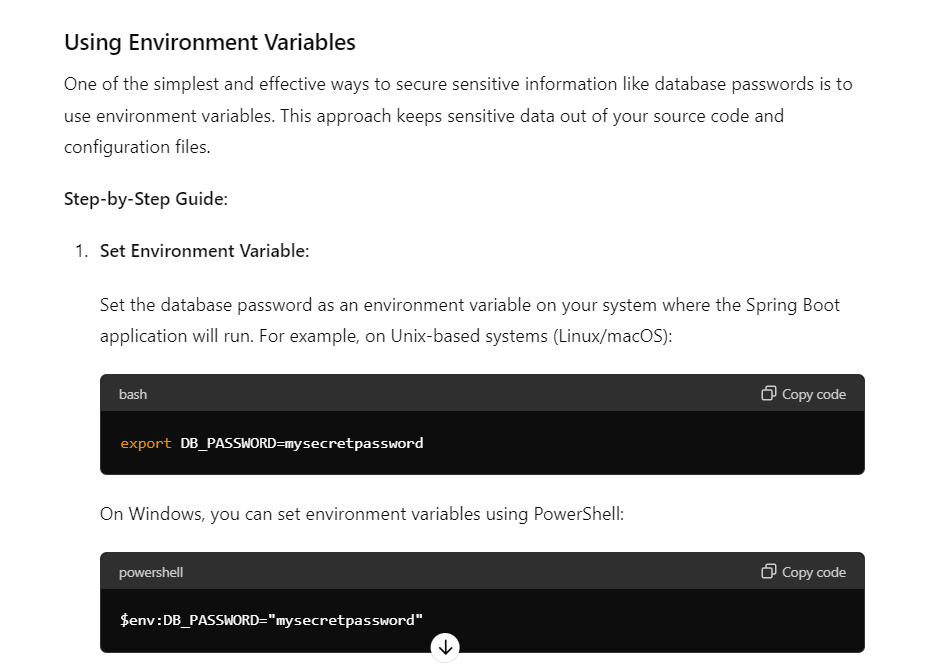
In Spring Boot, there are several ways to externalize configuration, allowing you to keep configuration details separate from your application code. This flexibility is essential for applications deployed in different environments or configurations without modifying the codebase itself. Here are the main methods to externalize configuration in Spring Boot:

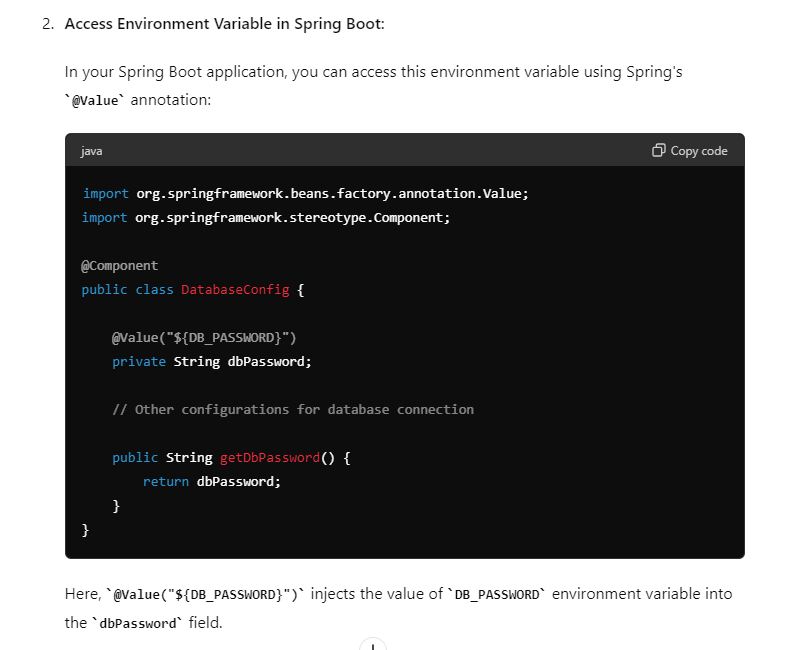


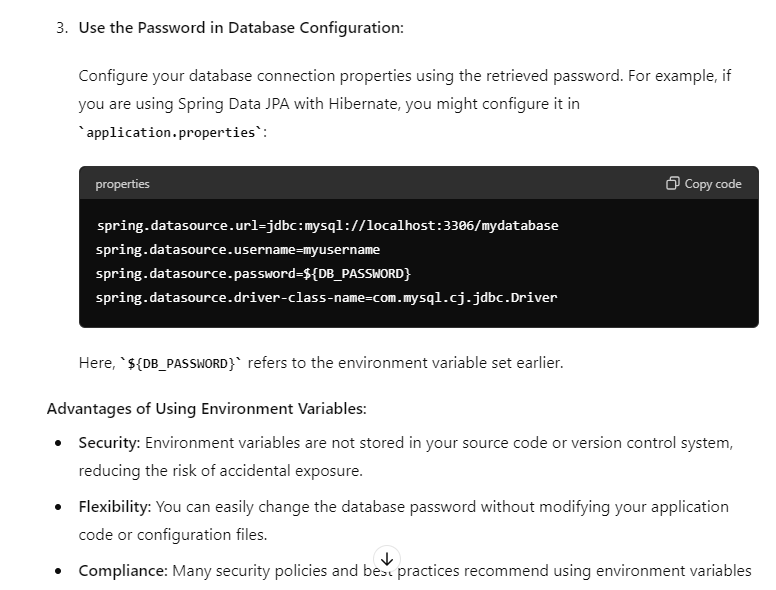


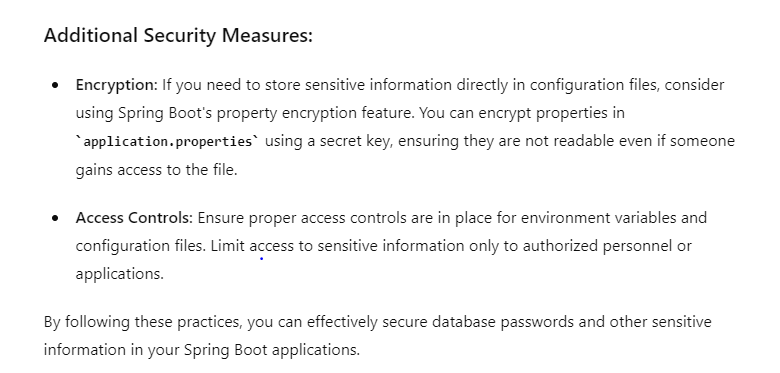


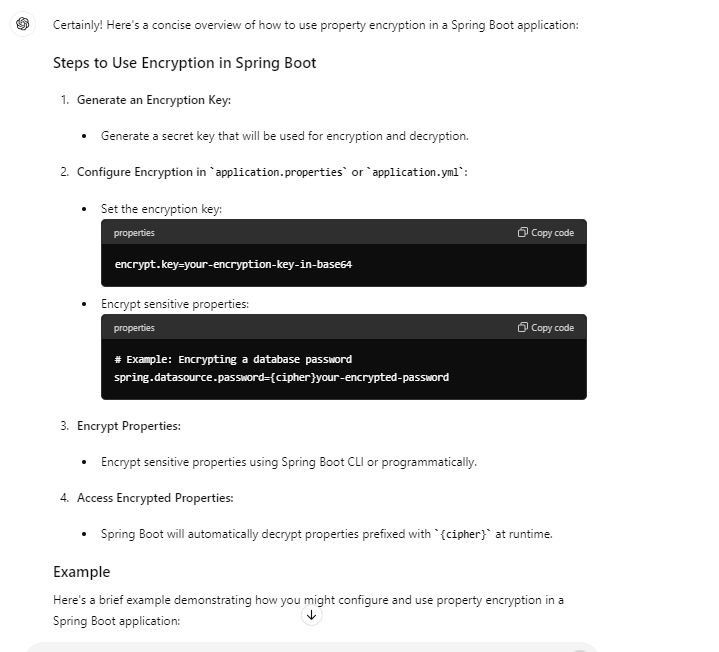
Q2: How can you secure sensitive configuration properties in Spring Boot?

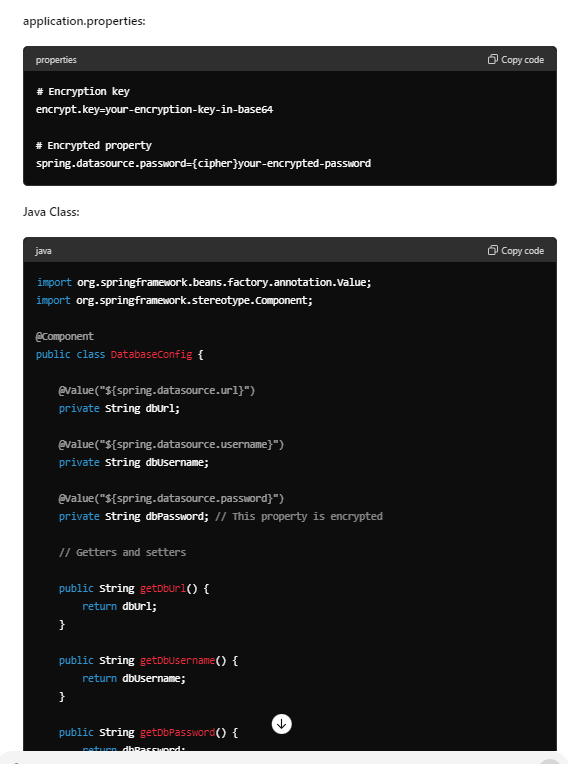


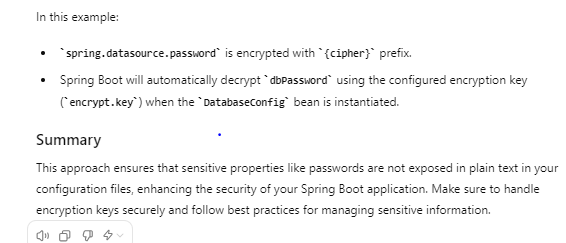












Q3: Explain the difference between application.properties and application.yml.

* **application.properties:** Simple and widely adopted format suitable for straightforward key-value configurations.
* **application.yml:** Offers more flexibility and readability, especially for complex configurations with nested structures.

Choose application.properties for simplicity and compatibility, while application.yml is ideal for more structured and hierarchical configurations in modern applications.

Q4: How can you switch between different configurations for different environments in Spring Boot?

In Spring Boot, switching between different configurations for different environments (such as development, testing, production) is typically achieved using profiles. Profiles allow you to define sets of configuration properties that are used based on the environment or specific conditions. Here’s how you can switch between configurations for different environments:

**Using Spring Boot Profiles**

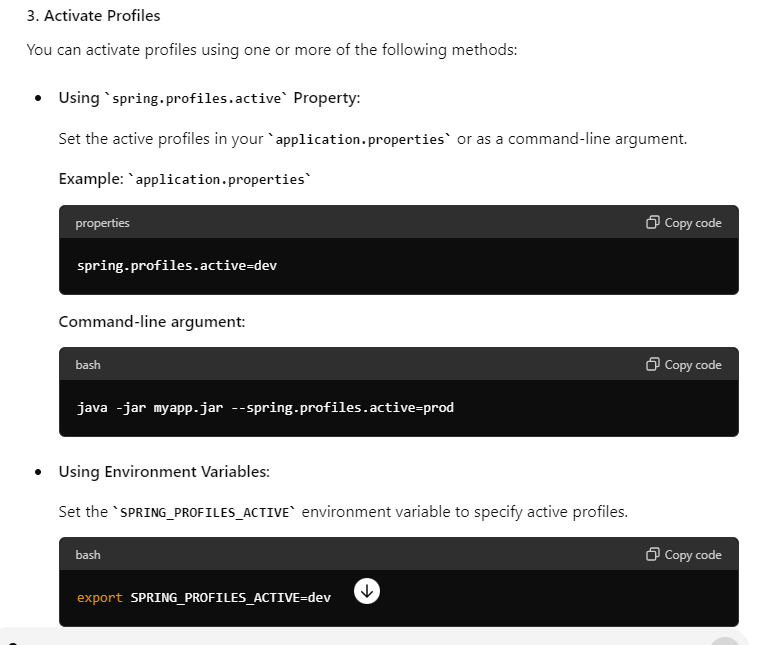
**1. Define Configuration Files**

Create separate configuration files for each environment or profile you want to support. By default, Spring Boot loads application.properties (or application.yml). For profiles, you can use specific naming conventions to load profile-specific configurations:

* **application-{profile}.properties** or **application-{profile}.yml**

For example:

* + application-dev.properties for development environment
  + application-prod.properties for production environment



Q5 : What is the purpose of the @ConfigurationProperties annotation?

The @ConfigurationProperties annotation in Spring Boot serves the purpose of binding external configuration properties directly into a Java bean or a class. It simplifies the process of injecting configuration properties from various sources such as application.properties, application.yml, environment variables, or command-line arguments into your Spring components.

**Key Purposes of @ConfigurationProperties**

1. **Property Binding:**
   * **Binding to Beans:** It binds configuration properties to fields of a Java bean or class.
   * **Type Conversion:** Automatically converts the properties from string values to the appropriate data types of the bean fields (e.g., integers, booleans, lists).
   * **Validation:** Supports validation of properties using JSR-303/JSR-349 annotations.
2. **Type Safety:**
   * Ensures type safety by directly mapping properties to Java bean fields. This reduces the risk of runtime errors due to type mismatches.
3. **Simplification:**
   * Simplifies the reading and management of configuration properties compared to manually reading properties using Environment or @Value annotations.
4. **Centralized Configuration:**
   * Provides a centralized way to manage and access configuration properties across your application.
   * Promotes better organization and maintainability of configuration code.

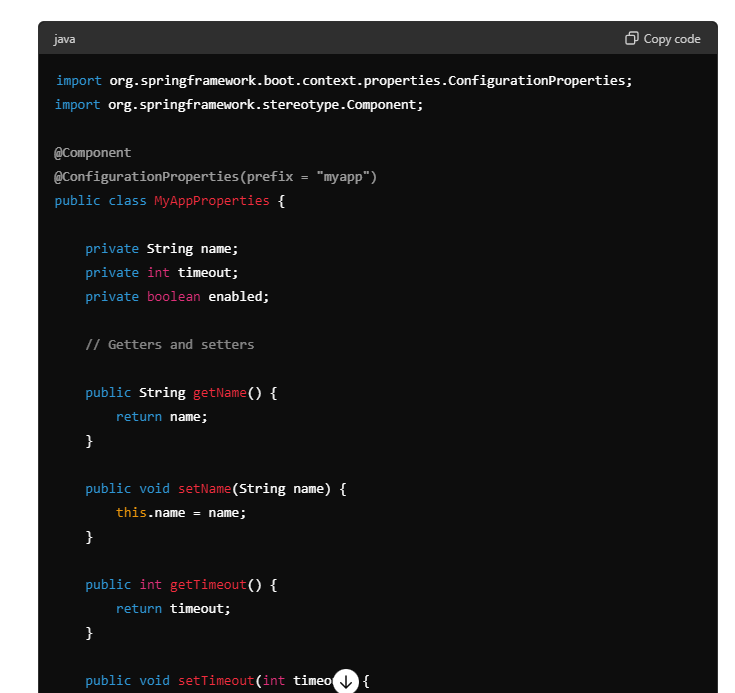
**How @ConfigurationProperties Works**

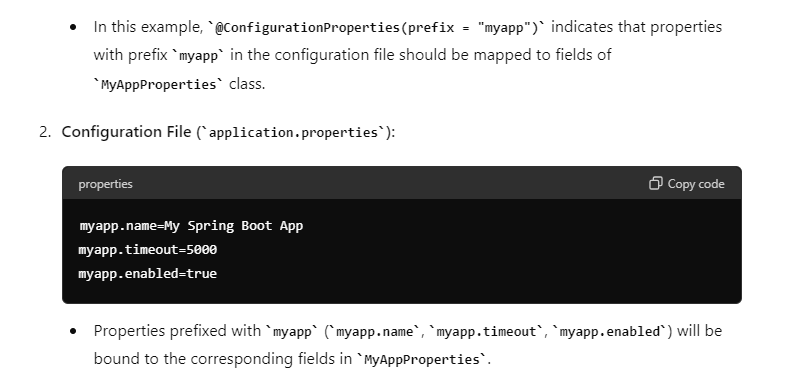
* **Annotation Usage:** Apply @ConfigurationProperties to a class or a bean to specify the prefix of the properties it should bind to.
* **Configuration File:** Define the properties in a configuration file (application.properties or application.yml) with the specified prefix.
* **Spring Boot Auto-Configuration:** Spring Boot automatically scans for classes annotated with @ConfigurationProperties and binds the properties from the configuration file to the corresponding fields of the annotated class or bean.

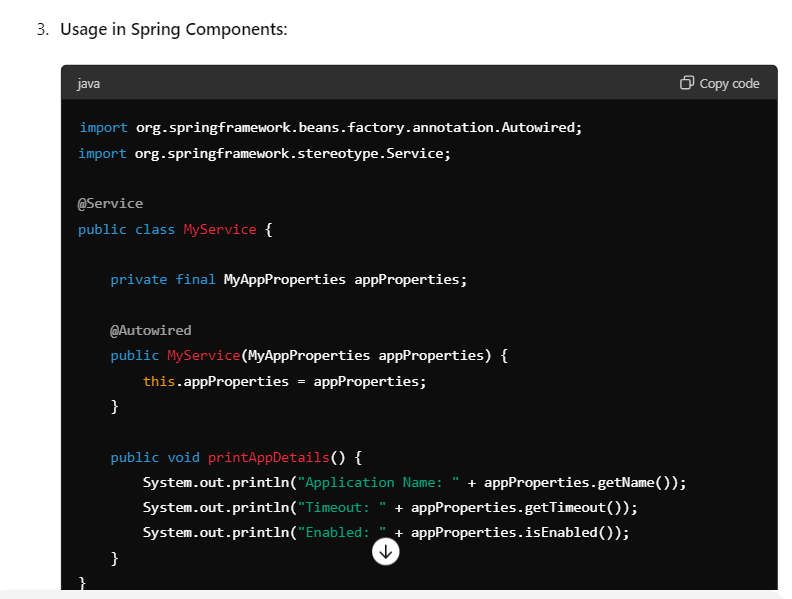
**Example**

Here's a quick example to illustrate how @ConfigurationProperties is used in a Spring Boot application:

1. **Define Configuration Class:**







* + The MyAppProperties bean is automatically injected into MyService using constructor injection.
  + appProperties.getName(), appProperties.getTimeout(), appProperties.isEnabled() will fetch the values from the configuration file (application.properties).

**Summary**

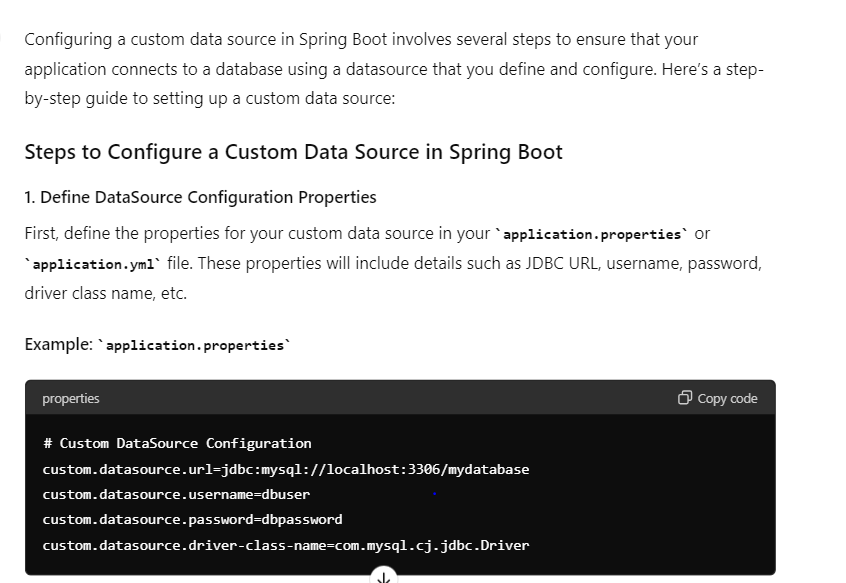
The @ConfigurationProperties annotation in Spring Boot simplifies the process of binding external configuration properties to Java beans or classes, ensuring type safety, automatic type conversion, and validation. It promotes cleaner and more maintainable code by centralizing configuration management and reducing boilerplate code for accessing properties across your application. Use it whenever you need to map configuration properties to a strongly-typed Java bean in your Spring Boot application.

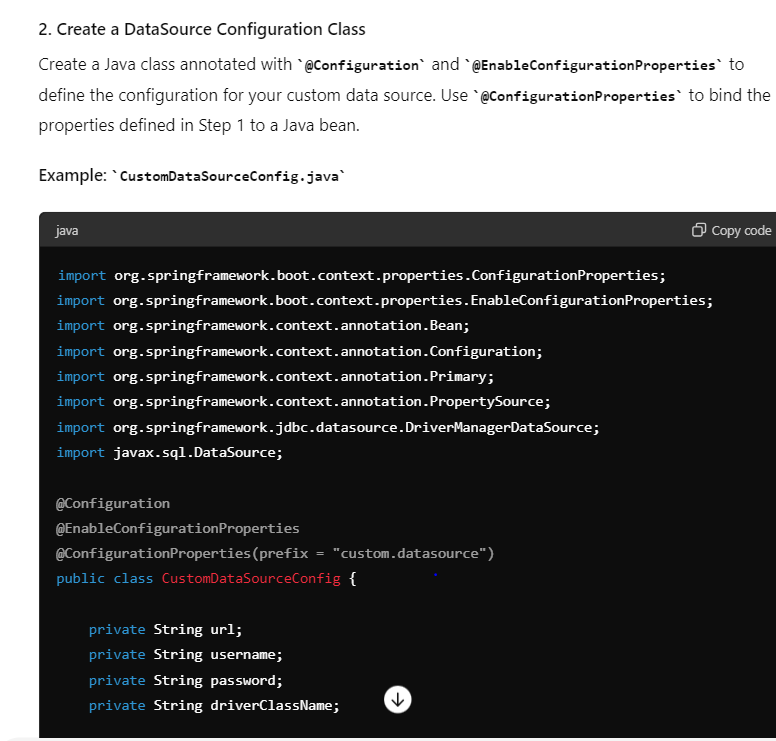
Q6 : why we need custom data source in springboot ?

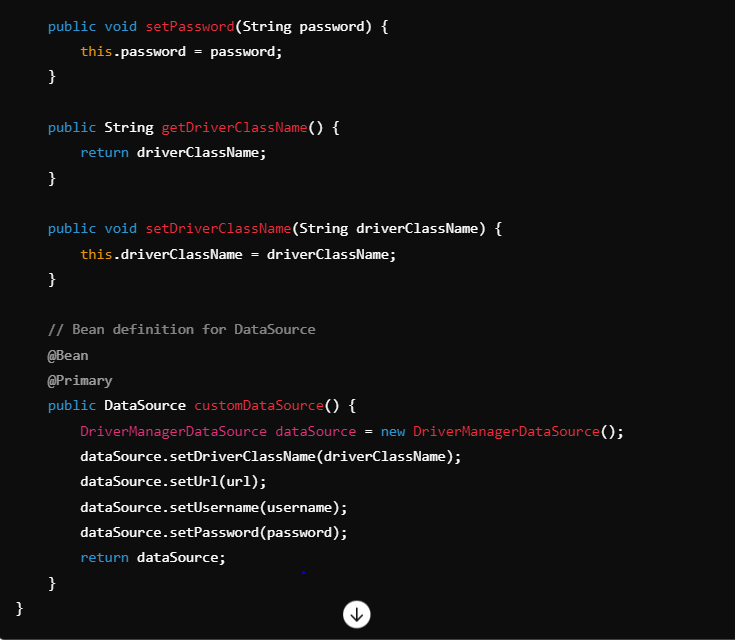
In Spring Boot, the need for a custom data source typically arises from specific requirements or constraints that go beyond what the default data source configurations provided by Spring Boot can support. Here are some common scenarios where a custom data source becomes necessary:

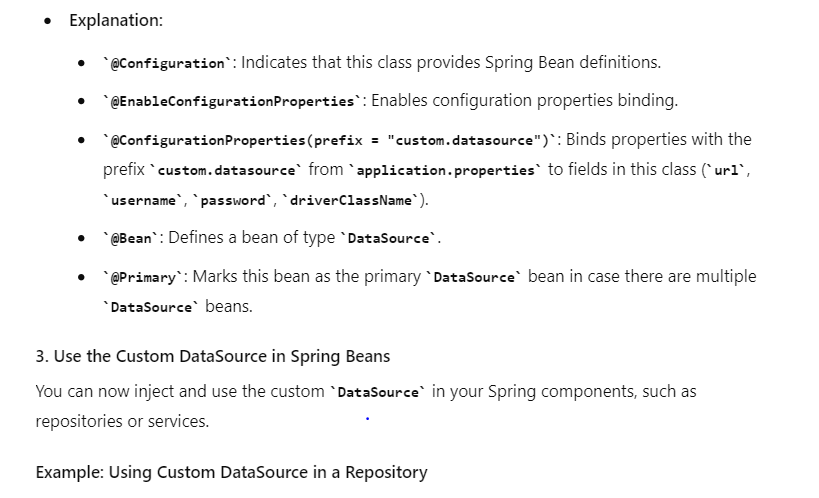
1. **Integration with Non-Standard or Legacy Databases:**
   * **Legacy Systems:** Sometimes applications need to integrate with older or legacy databases that don't conform to modern JDBC standards or require custom connection settings.
   * **Non-Standard Databases:** Connecting to databases like MongoDB, Cassandra, or proprietary systems may require custom drivers or configurations that aren't part of the default Spring Boot setup.
2. **Advanced Configuration Requirements:**
   * **Custom Connection Pooling:** You may need specific configurations for connection pooling (e.g., max connections, timeout settings) that differ from the defaults provided by Spring Boot's embedded connection pool (e.g., HikariCP).
   * **SSL or Encryption:** Applications dealing with secure data may require custom configurations for SSL certificates, encryption protocols, or custom authentication mechanisms.
3. **Multiple Data Sources:**
   * Applications often require connections to multiple databases or data sources. Spring Boot's auto-configuration supports primary and secondary data sources, but custom configurations are needed for more complex setups, such as sharding or distributed databases.
4. **Performance Optimization:**
   * In some cases, applications may have specific performance requirements that necessitate fine-tuning of database connection settings or using specialized features of a particular database engine.
5. **Compliance and Security:**
   * Applications may need to comply with specific security standards or regulations that require custom database configurations for encryption, auditing, or data masking.

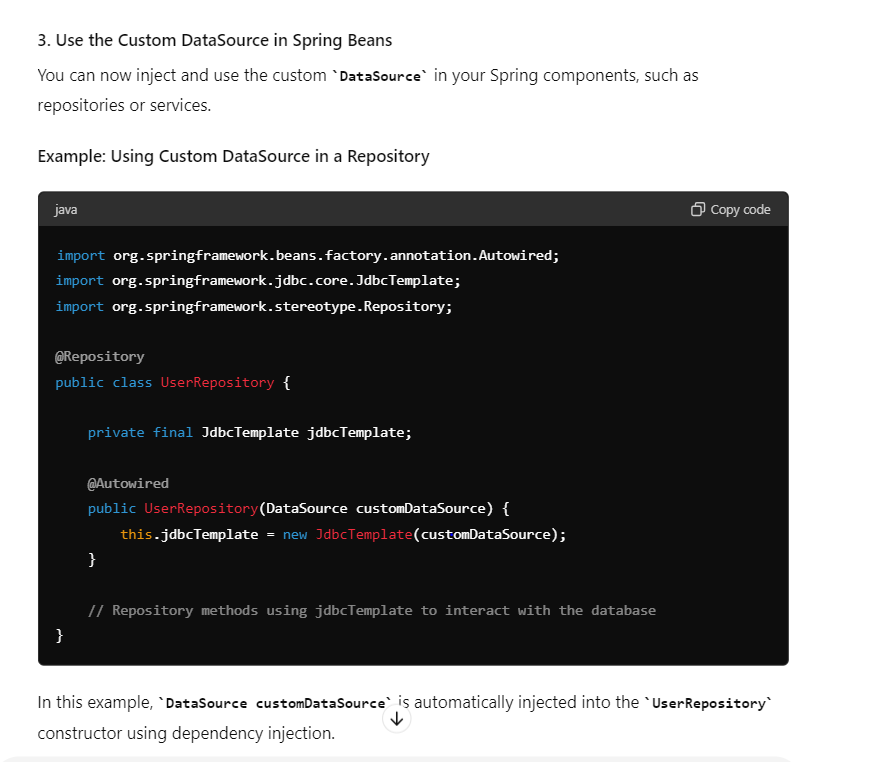
Q7 : How do you configure a custom data source in Spring Boot?

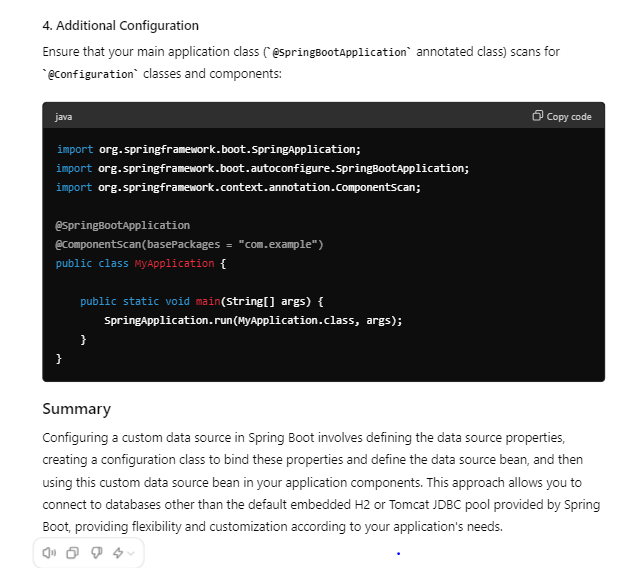




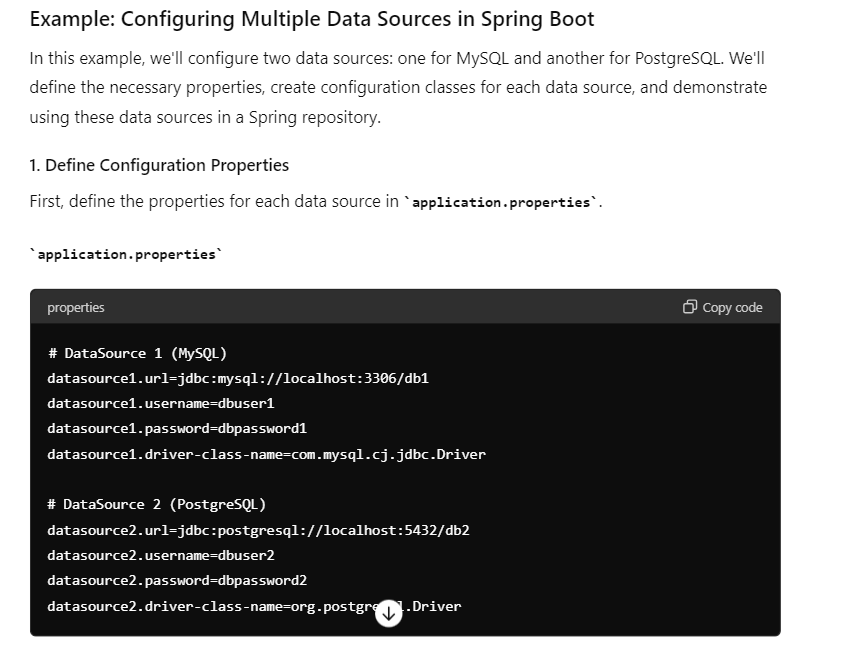


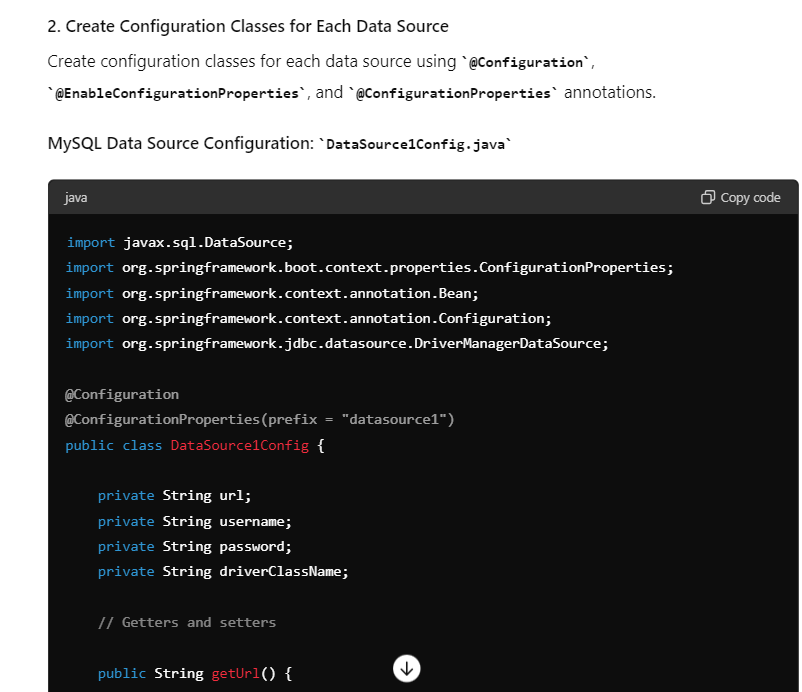


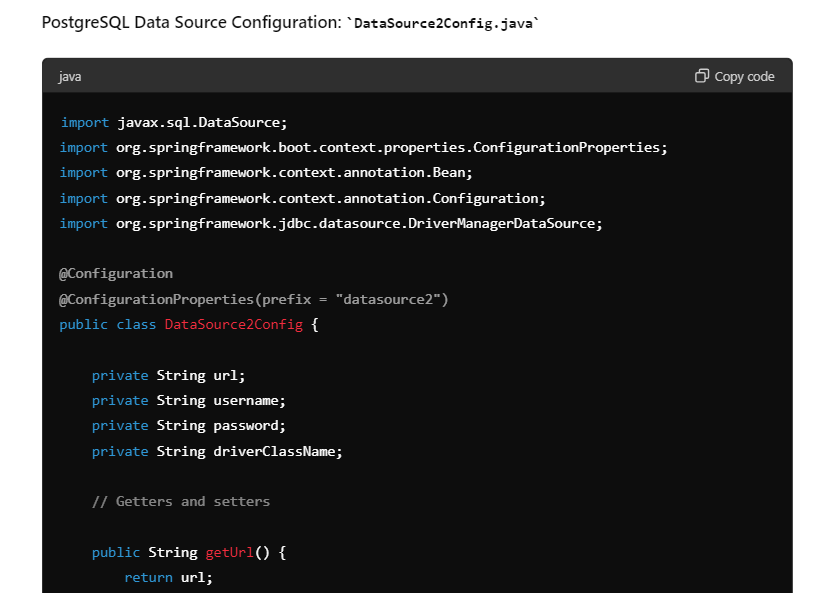


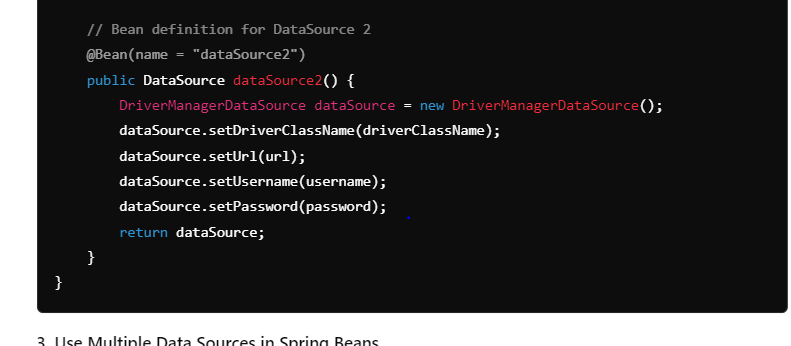


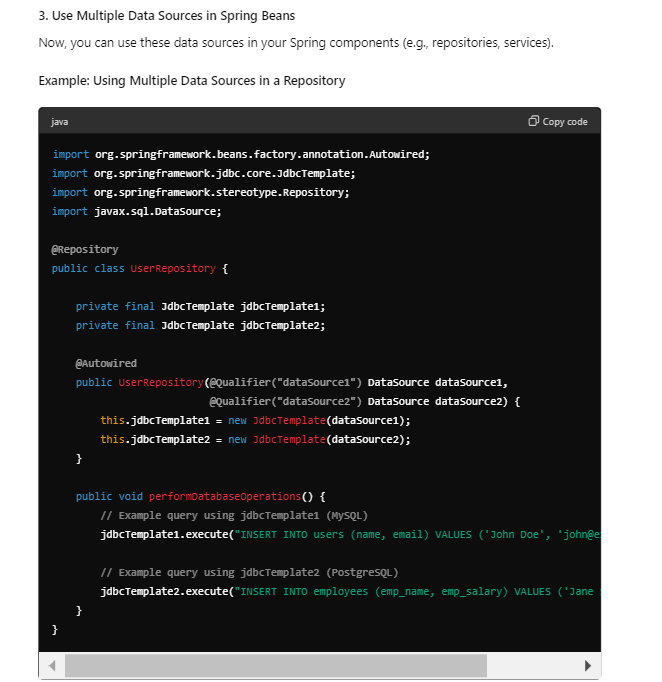
Q 8: how do we configure multiple data source in springboot ?

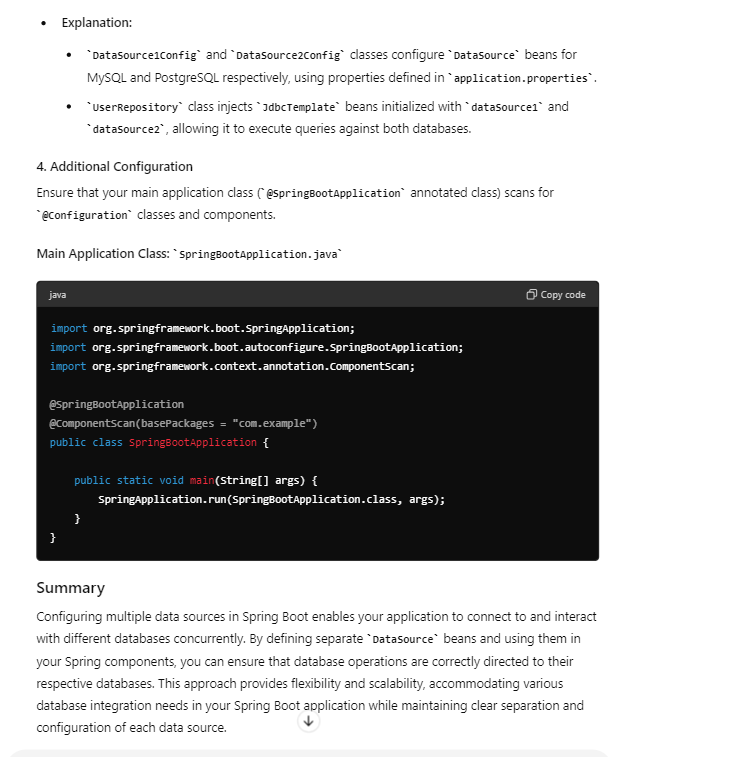




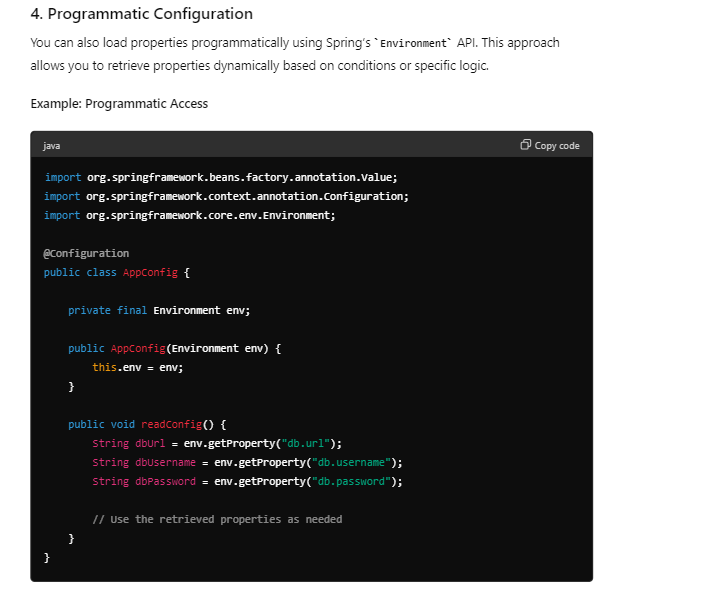








Q 9: how to read value from environment value in springboot ?



Q 10 : How can you configure logging in a Spring Boot application?

Using logging in a Spring Boot application is straightforward, and Spring Boot provides seamless integration with popular logging frameworks like Logback, Log4j2, and Java Util Logging (JUL). Here, I'll demonstrate how to use logging with an example using Logback, which is the default logging framework in Spring Boot.

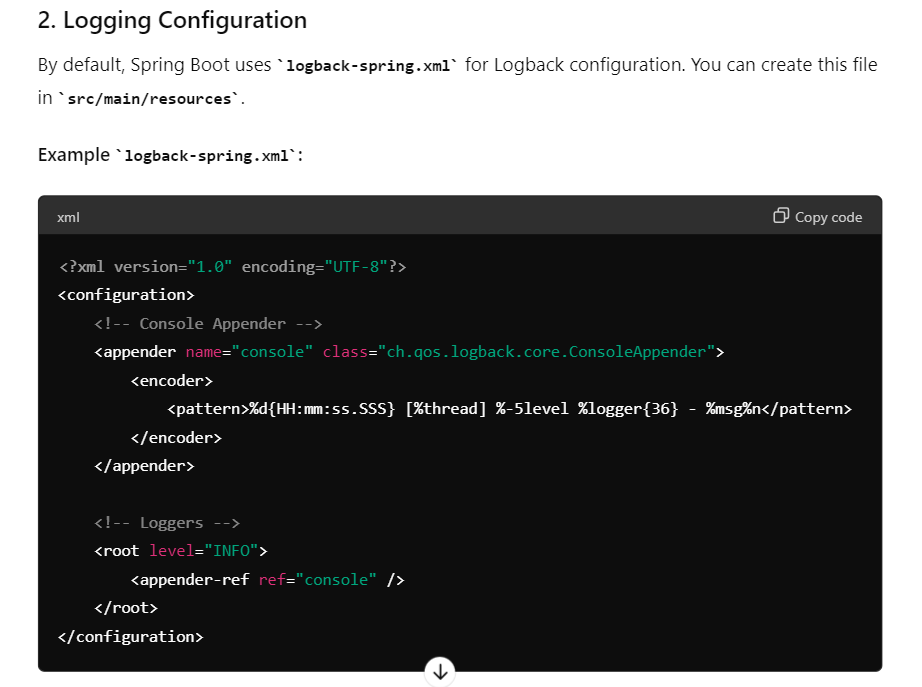
### 1. Dependency Setup

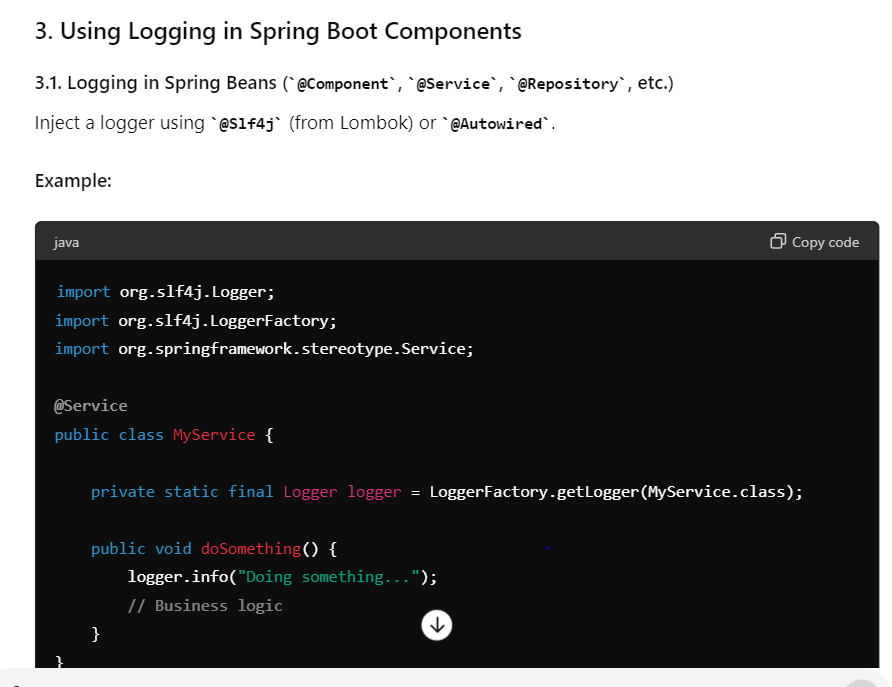
Ensure you have spring-boot-starter dependency in your pom.xml or build.gradle file. Spring Boot automatically includes the necessary logging dependencies.

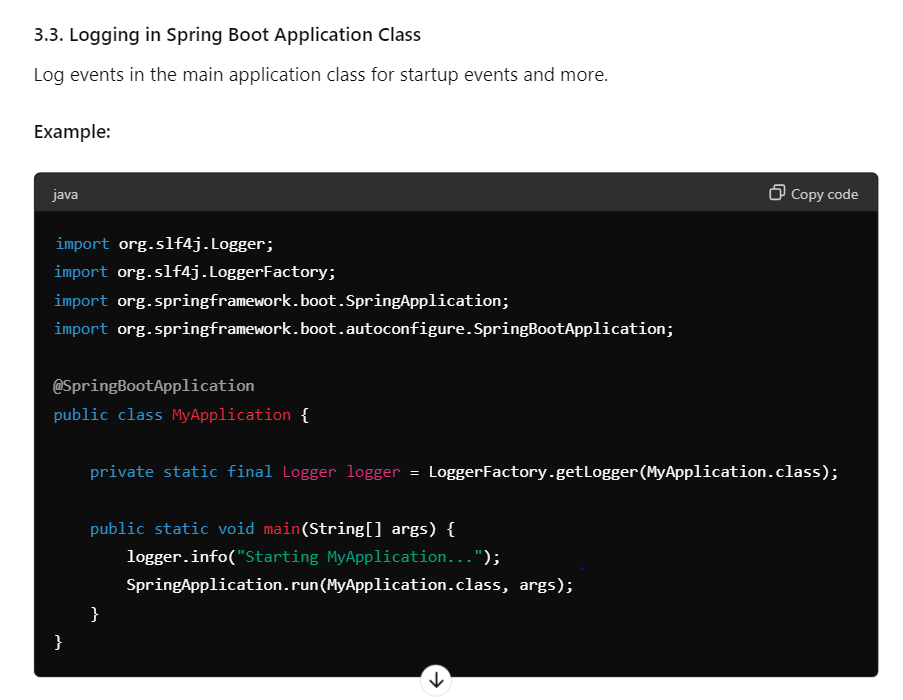
### 2. Logging Configuration

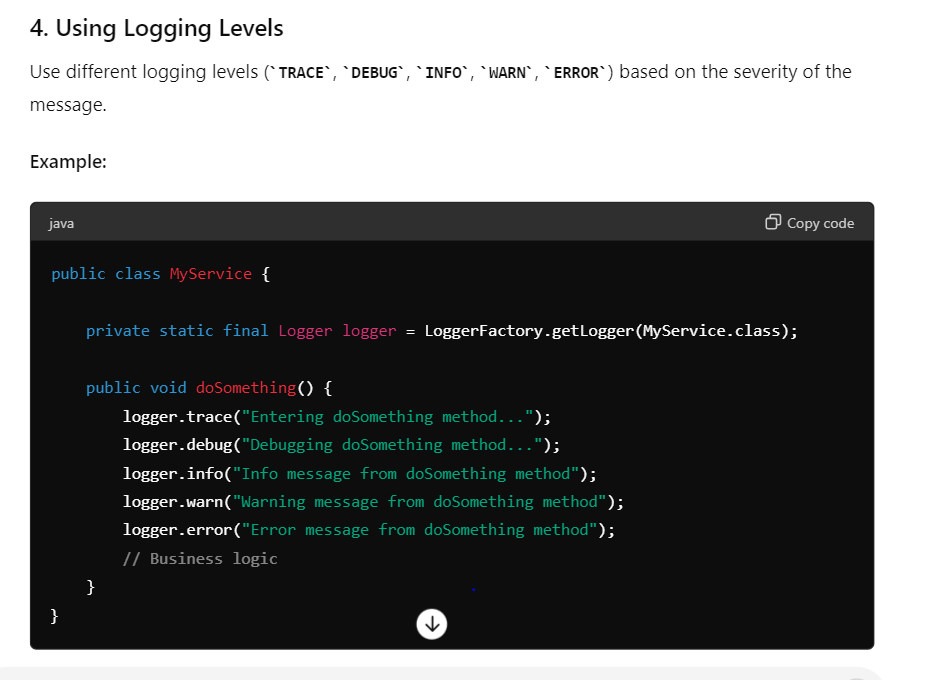
By default, Spring Boot uses logback-spring.xml for Logback configuration. You can create this file in src/main/resources.

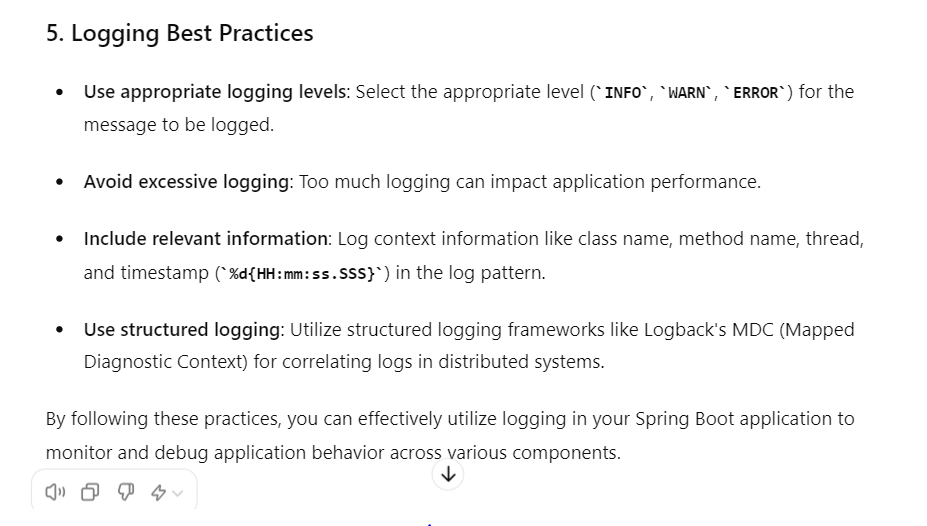
**Example logback-spring.xml:**











Q11 : Explain the use of the @Value annotation in Spring Boot.

In Spring Boot, the @Value annotation is used to inject values from properties files, environment variables, command-line arguments, or other Spring beans into your Spring components, such as @Component, @Service, @Controller, or @Configuration classes. This annotation provides a convenient way to externalize configuration and inject it directly into your Spring managed beans.

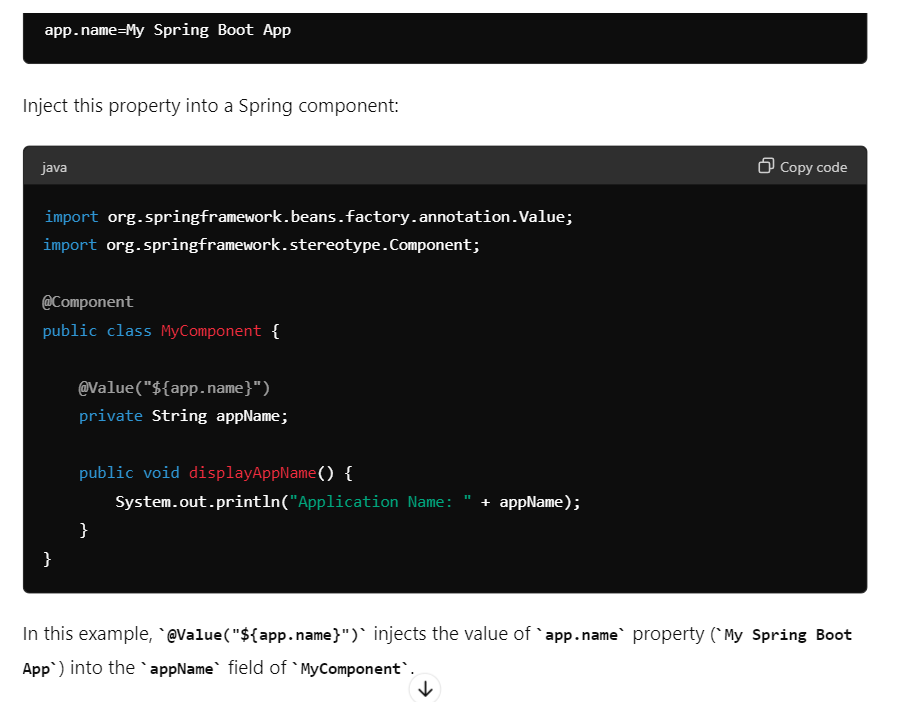
### Usage of @Value

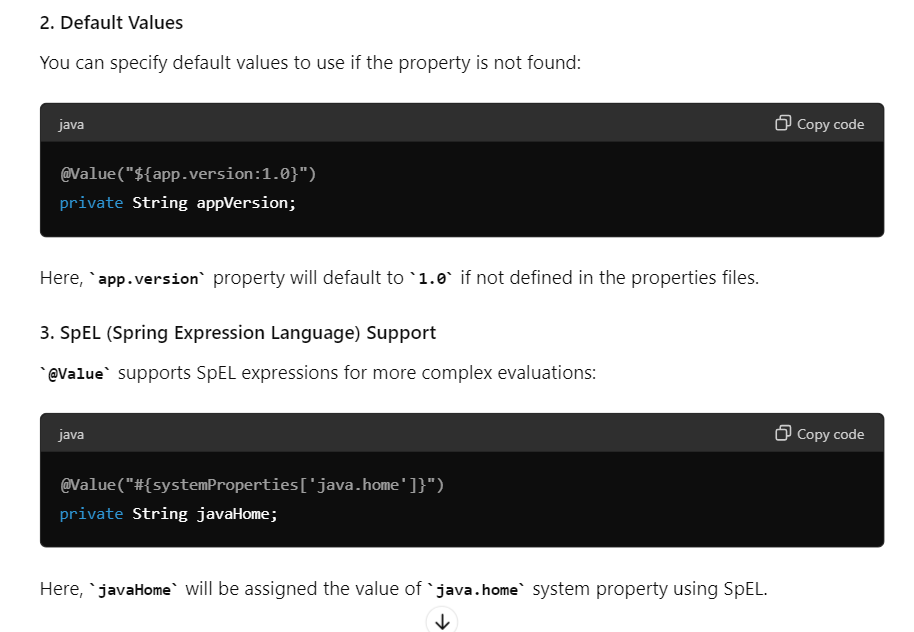
#### 1. Injecting Simple Properties

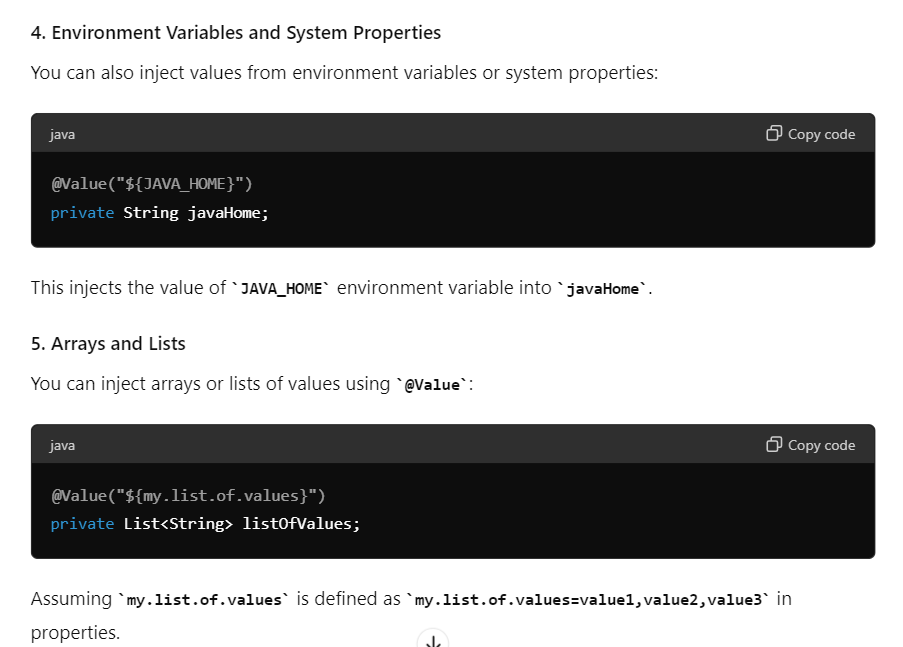
You can inject simple values from properties files (application.properties or application.yml) using @Value.

**Example:**

Assume you have a property defined in application.properties:







### Conclusion

The @Value annotation in Spring Boot provides a flexible mechanism for injecting configuration properties into Spring managed beans. It simplifies accessing externalized configuration, allowing you to configure your application's behavior without hard-coding values in your codebase. This makes your application more configurable, adaptable to different environments, and easier to maintain.

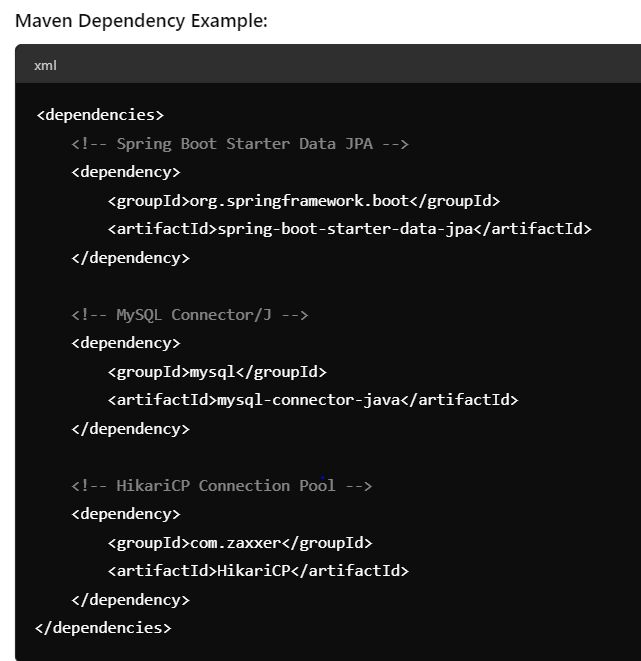
Q 12 : How do you configure a database connection pool in Spring Boot?

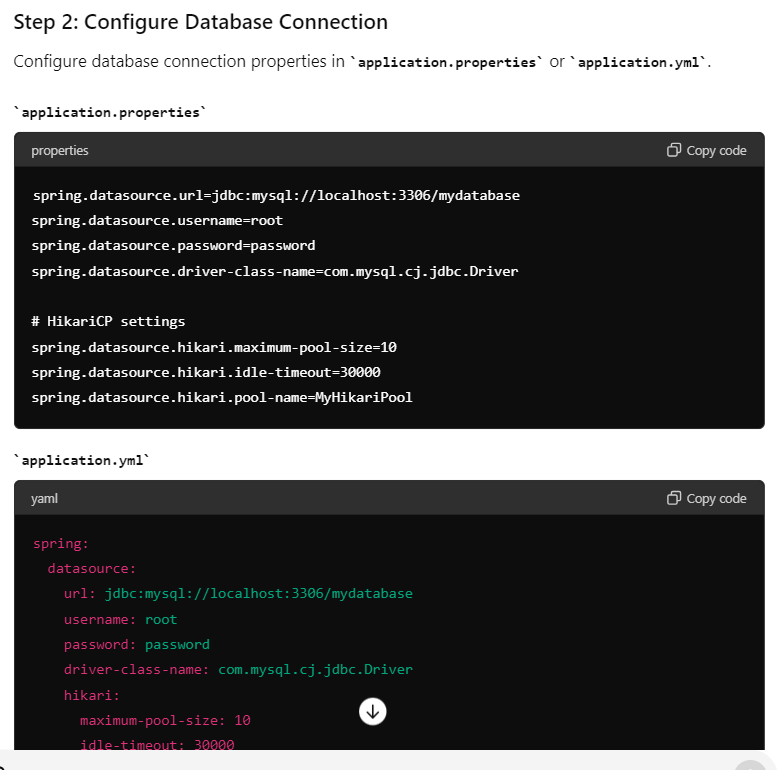
To create a complete example of setting up database connection pooling in a Spring Boot application, we'll go through the steps from configuring application.properties, setting up the necessary dependencies, defining a database configuration class, and using the connection in a service class.

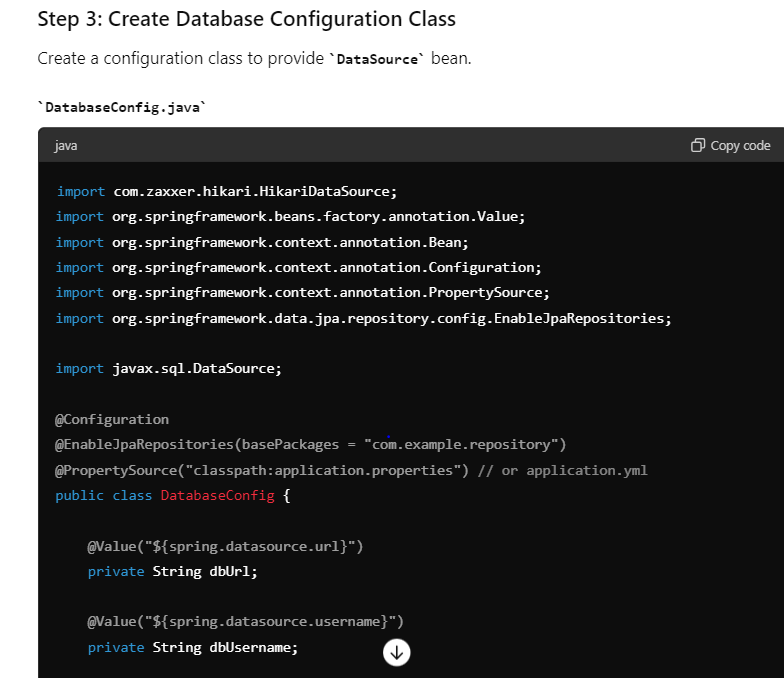
**Step 1: Dependencies**

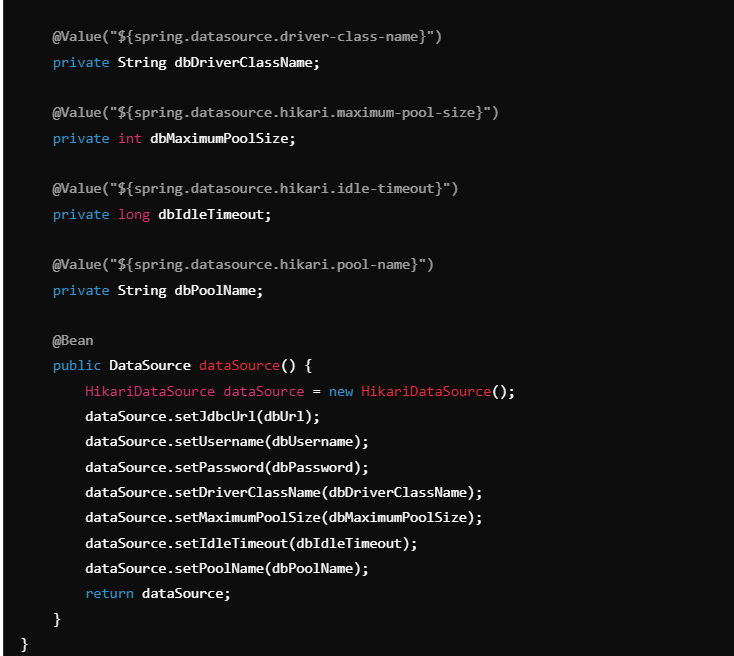
Ensure you have the necessary dependencies in your pom.xml (for Maven) or build.gradle (for Gradle).

**Maven Dependency Example:**









**Step 4: Use the DataSource in a Service**

Inject the DataSource into a service class and use it to obtain a database connection.

**Example Service Class**



**Step 5: Ensure Proper Shutdown**

Spring Boot manages the lifecycle of beans, including DataSource and connection pools. However, if you need custom shutdown behavior (e.g., releasing resources), implement DisposableBean or use @PreDestroy annotated methods.

**Conclusion**

This setup demonstrates how to configure and use database connection pooling in a Spring Boot application using HikariCP. By following these steps, you can efficiently manage database connections, optimize resource usage, and enhance the performance and scalability of your Java application. Adjust configurations and settings based on your specific application requirements and database setup.

Q 13 : what is hikari CP ?

HikariCP (formerly known as HikariCP) is a high-performance JDBC connection pool for Java applications. It is widely regarded as one of the fastest and most reliable connection pooling libraries available for Java



Q 14 : How do you handle transactions in Spring Boot?

Handling transactions in Spring Boot is crucial for ensuring data integrity and consistency when interacting with databases. Spring Boot simplifies transaction management through its integration with the Spring Framework's transaction management capabilities. Here’s how you can handle transactions in a Spring Boot application:

**1. Configure DataSource and Transaction Manager**

First, ensure you have configured your DataSource and TransactionManager appropriately in your Spring Boot application. Spring Boot auto-configures these components if you are using spring-boot-starter-data-jpa or similar starters.

Example application.properties or application.yml for configuring a datasource (MySQL in this case):

