Core Annotation

In Spring Boot, annotations such as `@Component`, `@Service`, `@Repository`, `@Controller`, and `@RestController` are used to define Spring-managed beans, which are automatically detected and registered by Spring's component scanning mechanism. These annotations play a key role in the Spring framework's dependency injection and IoC (Inversion of Control) container. Here's an explanation and example for each annotation, demonstrating their usage and the lifecycle within the Spring context:

1. `@Component`

- **Purpose**: Marks a class as a Spring-managed component. It is a generic stereotype for any Spring-managed component.
- **Example**:

```
import org.springframework.stereotype.Component;

@Component
public class MyComponent {
    public String greet() {
        return "Hello from MyComponent!";
    }
}
```

2. `@Service`

- **Purpose**: Specialization of `@Component` indicating that a class is a service. It is used in the service layer to encapsulate business logic.
- **Example**:

```
import org.springframework.stereotype.Service;

@Service
public class MyService {
    public String process() {
        return "Processing in MyService";
    }
}
```

3. `@Repository`

- **Purpose**: Specialization of `@Component` for data access components. It is used to indicate that a class interacts with the database.
- **Example**:

```
import org.springframework.stereotype.Repository;

@Repository
public class MyRepository {
    public String fetchData() {
        return "Data from MyRepository";
```

```
    }
}
```

4. `@Controller`

- **Purpose**: Marks a class as a Spring MVC controller. It is used to define request handling methods.
- **Example**:

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class MyController {
    @GetMapping("/home")
    public String home(Model model) {
        model.addAttribute("message", "Welcome to Home Page");
        return "home";
    }
}
```

5. `@RestController`

- **Purpose**: Combination of `@Controller` and `@ResponseBody`. It simplifies the creation of RESTful web services by returning data directly as JSON or XML.
- **Example**:

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MyRestController {
    @GetMapping("/api/greet")
    public String greet() {
        return "Hello from MyRestController!";
    }
}
```

Bean Lifecycle and ApplicationContext Example

To observe the lifecycle and how these beans are managed in the application context, you can create a simple Spring Boot application:

1. **Application Class**:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        var context = SpringApplication.run(MyApplication.class, args);

        // Fetch beans from context
```

```
        MyComponent myComponent = context.getBean(MyComponent.class);
        MyService myService = context.getBean(MyService.class);
        MyRepository myRepository = context.getBean(MyRepository.class);

        // Output bean methods
        System.out.println(myComponent.greet());
        System.out.println(myService.process());
        System.out.println(myRepository.fetchData());
    }
}
```

2. **Bean Lifecycle Methods**: Implementing `@PostConstruct` and `@PreDestroy` can help observe the initialization and destruction of beans.

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

@Component
public class LifecycleBean {
    @PostConstruct
    public void init() {
        System.out.println("Bean is initialized");
    }

    @PreDestroy
    public void destroy() {
        System.out.println("Bean is destroyed");
    }
}
```

Explanation

- **Component Scanning**: Spring Boot automatically scans the classpath for classes annotated with `@Component`, `@Service`, `@Repository`, `@Controller`, and `@RestController` within the `@SpringBootApplication` annotated class's package and its sub-packages.
- **Bean Factory and Application Context**: The `ApplicationContext` is the central interface for accessing the Spring container. It manages the lifecycle of beans, their dependencies, and ensures they are properly initialized and destroyed according to the Spring lifecycle callbacks.

Real-life analog

Let's use a real-life analogy to explain the scenario with `@Component`, `@Service`, `@Repository`, `@Controller`, and `@RestController`, along with how they fit into the Spring lifecycle, bean factory, and application context.

Real-Life Analogy: A Coffee Shop

1. `@Component` - Barista (Generic Employee)

- **Analogy**: The barista is a general employee in a coffee shop who can perform various tasks, such as making coffee, cleaning, or taking orders.
- **Example**: In a Spring application, `@Component` is used for general-purpose classes that can be a part of any layer.

```
@Component
public class Barista {
    public String prepareCoffee() {
        return "Coffee is ready!";
    }
}
```

2. `@Service` - Coffee Maker (Service Provider)

- **Analogy**: The coffee maker in the shop is responsible for brewing coffee, which is a specific service provided to customers.
- **Example**: In a Spring application, `@Service` is used for business logic, similar to how a coffee maker is specifically focused on making coffee.

```
@Service
public class CoffeeMaker {
    public String brewCoffee() {
        return "Brewing a delicious cup of coffee!";
    }
}
```

3. `@Repository` - Inventory Manager (Data Access)

- **Analogy**: The inventory manager handles stock, like checking if coffee beans are available or ordering more when stock runs low.
- **Example**: In a Spring application, `@Repository` is used for data access, similar to how the inventory manager interacts with the database of stock items.

```
@Repository
public class InventoryManager {
    public String checkStock() {
        return "Stock is sufficient!";
    }
}
```

4. `@Controller` - Order Taker (Customer Interaction)

- **Analogy**: The order taker interacts with customers, taking their orders and passing them to the barista or coffee maker.
- **Example**: In a Spring MVC application, `@Controller` handles web requests, acting as the intermediary between the customer and the service providers in the coffee shop.

```
@Controller
```

```
public class OrderTaker {
    @GetMapping("/order")
    public String takeOrder(Model model) {
        model.addAttribute("message", "Order received!");
        return "orderPage";
    }
}
```

5. `@RestController` - Online Order System (Direct Service)

- **Analogy**: An online ordering system allows customers to place orders directly from their devices, receiving confirmations without needing to interact physically.
- **Example**: In a Spring Boot application, `@RestController` handles RESTful web services, returning data directly to the client.

```
@RestController
public class OnlineOrderSystem {
    @GetMapping("/api/order")
    public String placeOrder() {
        return "Order placed successfully!";
    }
}
```

Lifecycle and Application Context: The Coffee Shop's Management System

- **Bean Lifecycle**:
    - **Initialization (`@PostConstruct`)**: When the shop opens in the morning, all employees (beans) are prepared to start their tasks. The barista, coffee maker, inventory manager, order taker, and online system are all initialized.

```
@Component
public class Employee {
    @PostConstruct
    public void startShift() {
        System.out.println("Employee is ready for the shift.");
    }

    @PreDestroy
    public void endShift() {
        System.out.println("Employee's shift is over.");
    }
}
```

- **Application Context**: Think of the application context as the coffee shop’s management system. It oversees all the employees (beans), ensuring they have what they need (dependencies) and that they perform their roles properly. It manages when employees start (bean initialization) and finish (bean destruction) their tasks.
- **Bean Factory**: The bean factory is like the hiring manager who knows all the available employees and can provide any employee (bean) when needed. When a customer orders a coffee, the manager (bean factory) assigns the task to the barista or coffee maker, ensuring the process runs smoothly.

How It Works Together:

- **Customer Interaction**: A customer (user) places an order through the order taker (`@Controller`) or online system (`@RestController`).
- **Service Processing**: The order is processed by the coffee maker (`@Service`), which involves brewing coffee.
- **Data Access**: The inventory manager (`@Repository`) checks if the required ingredients are available.
- **Completion**: The barista (`@Component`) serves the coffee, and the customer receives their order.

Observe the lifecycle of Spring-managed beans

let’s observe the lifecycle of Spring-managed beans by using `System.out.println`. You can print messages at different points in the bean lifecycle to see how and when they are initialized, used, and destroyed. Here's how you can set this up in the coffee shop analogy example:

1. Adding Output to Lifecycle Events

Use the `@PostConstruct` and `@PreDestroy` annotations to print messages when beans are initialized and destroyed.

Example with System Output:

1. **Barista Component (`@Component`)**:

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import org.springframework.stereotype.Component;

@Component
public class Barista {

    @PostConstruct
    public void init() {
        System.out.println("Barista initialized.");
    }

    public String prepareCoffee() {
        System.out.println("Barista is preparing coffee.");
        return "Coffee is ready!";
    }

    @PreDestroy
    public void destroy() {
        System.out.println("Barista is being destroyed.");
    }
}
```

2. **Coffee Maker Service (`@Service`)**:

```
import org.springframework.stereotype.Service;

@Service
public class CoffeeMaker {

    @PostConstruct
    public void init() {
        System.out.println("CoffeeMaker initialized.");
    }

    public String brewCoffee() {
        System.out.println("Brewing coffee.");
        return "Brewing a delicious cup of coffee!";
    }

    @PreDestroy
    public void destroy() {
        System.out.println("CoffeeMaker is being destroyed.");
    }
}
```

3. **Inventory Manager Repository (`@Repository`)**:

```
import org.springframework.stereotype.Repository;

@Repository
public class InventoryManager {

    @PostConstruct
    public void init() {
        System.out.println("InventoryManager initialized.");
    }

    public String checkStock() {
        System.out.println("Checking stock.");
        return "Stock is sufficient!";
    }

    @PreDestroy
    public void destroy() {
        System.out.println("InventoryManager is being destroyed.");
    }
}
```

4. **Application Class**:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        var context = SpringApplication.run(MyApplication.class, args);

        // Fetch beans from context
        Barista barista = context.getBean(Barista.class);
        CoffeeMaker coffeeMaker = context.getBean(CoffeeMaker.class);
```

```
        InventoryManager inventoryManager =
context.getBean(InventoryManager.class);

        // Use beans to trigger lifecycle output
        barista.prepareCoffee();
        coffeeMaker.brewCoffee();
        inventoryManager.checkStock();
    }
}
```

3. Running the Application

When you run this Spring Boot application, you will see the following output in the console:

```
Barista initialized.
CoffeeMaker initialized.
InventoryManager initialized.
Barista is preparing coffee.
Brewing coffee.
Checking stock.
Barista is being destroyed.
CoffeeMaker is being destroyed.
InventoryManager is being destroyed.
```

4. Understanding the Output

- **Initialization**: Messages from `@PostConstruct` show when each bean is initialized by the Spring context.
- **Usage**: Messages inside methods show when beans are used within the application.
- **Destruction**: Messages from `@PreDestroy` show when each bean is being destroyed as the application shuts down.

CHECK AVAILABLE BEANS

In Spring Boot, you can check the available beans in the **ApplicationContext** (which is the bean factory) by using the `ApplicationContext`'s `getBeanDefinitionNames()` method. This will return the names of all the beans that are currently defined and managed by the Spring context.

Here’s how you can do that:

Example: Checking Available Beans in the ApplicationContext

1. **Modify the Application Class** to List Beans: You can inject the `ApplicationContext` into your application and call `getBeanDefinitionNames()` to see all the beans that Spring has registered.

   ```
   import org.springframework.boot.CommandLineRunner;
   import org.springframework.boot.SpringApplication;
   import org.springframework.boot.autoconfigure.SpringBootApplication;
   ```

```
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class MyApplication implements CommandLineRunner {

    private final ApplicationContext context;

    // Inject the ApplicationContext into the constructor
    public MyApplication(ApplicationContext context) {
        this.context = context;
    }

    @Override
    public void run(String... args) throws Exception {
        // Get all available beans
        String[] beanNames = context.getBeanDefinitionNames();

        // Print out each bean name
        System.out.println("Available Beans in the ApplicationContext:");
        for (String beanName : beanNames) {
            System.out.println(beanName);
        }
    }

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

2. **Output**: When you run the application, you'll see a list of all the beans that Spring has registered in the context.

   Example output:

   ```
   Available Beans in the ApplicationContext:
   org.springframework.context.annotation.internalConfigurationAnnotationProcess
   or
   org.springframework.context.annotation.internalAutowiredAnnotationProcessor
   org.springframework.context.annotation.internalCommonAnnotationProcessor
   org.springframework.context.annotation.internalPersistenceAnnotationProcessor
   myApplication
   barista
   coffeeMaker
   inventoryManager
   ```

   - The bean names are listed, including both your application beans (`barista`, `coffeeMaker`, `inventoryManager`) and some internal beans managed by Spring, such as annotation processors.
   - If you want to filter and find only the beans you've defined, you can check for the ones that match the names of your classes or use more specific logic (like filtering by type).

3. **List All Beans of a Specific Type**: If you are interested in listing beans of a specific type (e.g., all beans of type `Barista`), you can do the following:

   ```
   @Override
   public void run(String... args) throws Exception {
       // Get all beans of a specific type (e.g., Barista)
   ```

```
        String[] baristaBeans =
    context.getBeansOfType(Barista.class).keySet().toArray(new String[0]);

        System.out.println("Barista Beans in the ApplicationContext:");
        for (String beanName : baristaBeans) {
            System.out.println(beanName);
        }
    }
```

Let’s Check Custom Created Beans

To display only the **custom-created beans** (i.e., your own `@Component`, `@Service`, `@Repository`, and `@Controller` beans), you can filter out the built-in Spring beans and only list the beans that you have defined in your application.

You can achieve this by using the `ApplicationContext` to fetch only beans that belong to your custom types, excluding internal Spring beans.

Steps to Display Only Custom Beans

1. **Filter Beans by Type**: You can use the `getBeansOfType` method to find only your custom beans by their class type or use the `getBeanDefinitionNames()` method to exclude beans that you don't define.

Example: Display Only Custom Beans

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class MyApplication implements CommandLineRunner {

    private final ApplicationContext context;

    // Injecting ApplicationContext
    public MyApplication(ApplicationContext context) {
        this.context = context;
    }

    @Override
    public void run(String... args) throws Exception {
        System.out.println("Custom Beans in the ApplicationContext:");

        // Get all beans of a specific type (excluding internal Spring beans)
        String[] customBeanNames = context.getBeanDefinitionNames();
```

```
        // Filter custom beans based on class name pattern (or your own custom
logic)
        for (String beanName : customBeanNames) {
            Object bean = context.getBean(beanName);
            if (bean.getClass().getPackageName().startsWith("com.example")) {  //
replace with your package name
                System.out.println(beanName);
            }
        }
    }

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

// Example Custom Beans
@Component
class Barista {
    // Bean definition
}

@Service
class CoffeeMaker {
    // Bean definition
}

@Component
class InventoryManager {
    // Bean definition
}
```

Explanation:

- **`getBeanDefinitionNames()`**: Returns all the bean names, including both custom and internal Spring beans.
- **Filter Logic**: In the loop, we're checking the package name of each bean's class using `bean.getClass().getPackageName()`. We can then filter out any Spring internal beans by comparing the package name to the package where you typically define your custom beans.
- Replace `"com.example"` with the actual base package name of your application, where you store your custom beans.

Example Output:

If your beans are in the package `com.example`, the output would look like this:

```
Custom Beans in the ApplicationContext:
barista
coffeeMaker
inventoryManager
```

This way, you'll only see the beans that you have defined (your custom beans), excluding any internal Spring beans.

Alternative: Using Bean Class Type

Another approach is to directly check for beans of your custom classes. For example, if you're interested in listing beans of type `Barista`, `CoffeeMaker`, and `InventoryManager` specifically:

```
@Override
public void run(String... args) throws Exception {
    // Get beans by type
    System.out.println("Custom Beans in the ApplicationContext:");

    // List of custom beans by specific type
    Object baristaBean = context.getBean(Barista.class);
    Object coffeeMakerBean = context.getBean(CoffeeMaker.class);
    Object inventoryManagerBean = context.getBean(InventoryManager.class);

    // Display the names of the custom beans
    System.out.println("Barista Bean: " + baristaBean.getClass().getName());
    System.out.println("CoffeeMaker Bean: " +
coffeeMakerBean.getClass().getName());
    System.out.println("InventoryManager Bean: " +
inventoryManagerBean.getClass().getName());
}
```