**Exercise 1: Constructor Injection**

**Problem:**

Create a Spring application for a simple messaging service. Define a **Message** class with attributes **content** and **sender**. Create a **MessageService** class that depends on the **Message** class using constructor injection. Display the message information using the **MessageService**.

Solution:

// Message.java

public class Message {

private String content;

private String sender;

public Message(String content, String sender) {

this.content = content;

this.sender = sender;

}

public String getContent() {

return content;

}

public String getSender() {

return sender;

}

@Override

public String toString() {

return "Message [content=" + content + ", sender=" + sender + "]";

}

}

// MessageService.java

public class MessageService {

private Message message;

public MessageService(Message message) {

this.message = message;

}

public void displayMessage() {

System.out.println("Received Message:");

System.out.println(message);

}

}

// AppConfig.java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class AppConfig {

@Bean

public Message message() {

return new Message("Hello, Spring!", "John Doe");

}

@Bean

public MessageService messageService(Message message) {

return new MessageService(message);

}

}

// MainApp.java

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {

public static void main(String[] args) {

try (AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class)) {

// Retrieve the MessageService bean from the context

MessageService messageService = context.getBean(MessageService.class);

// Use the MessageService object to display the message

messageService.displayMessage();

}

}

}

**Exercise 2: Setter Injection**

Problem:

Extend the previous exercise by using setter injection instead of constructor injection. Update the **MessageService** class to have a setter method for injecting the **Message** dependency.

Solution:

// Message.java

public class Message {

private String content;

private String sender;

public Message(String content, String sender) {

this.content = content;

this.sender = sender;

}

public String getContent() {

return content;

}

public String getSender() {

return sender;

}

@Override

public String toString() {

return "Message [content=" + content + ", sender=" + sender + "]";

}

}

// MessageService.java

public class MessageService {

private Message message;

// Setter for Message dependency (DI)

public void setMessage(Message message) {

this.message = message;

}

public void displayMessage() {

System.out.println("Received Message:");

System.out.println(message);

}

}

// AppConfig.java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class AppConfig {

@Bean

public Message message() {

return new Message("Hello, Spring!", "John Doe");

}

@Bean

public MessageService messageService() {

return new MessageService();

}

}

// MainApp.java

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {

public static void main(String[] args) {

try (AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class)) {

// Retrieve the MessageService and Message beans from the context

MessageService messageService = context.getBean(MessageService.class);

Message message = context.getBean(Message.class);

// Use the MessageService object to display the message

// Note: This time we set the Message dependency using the setter

messageService.setMessage(message);

messageService.displayMessage();

}

}

}

**Exercise 3: Constructor and Setter Injection with Multiple Dependencies**

**Problem:**

Create a Spring application for managing a library. Define classes for **Book**, **Author**, and **Library**. The **Library** class should depend on multiple **Book** and **Author** instances. Use both constructor and setter injection to manage these dependencies.

**Solution:**

// Author.java

public class Author {

private String name;

public Author(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

// Book.java

public class Book {

private String title;

private Author author;

public Book(String title, Author author) {

this.title = title;

this.author = author;

}

public String getTitle() {

return title;

}

public Author getAuthor() {

return author;

}

@Override

public String toString() {

return "Book [title=" + title + ", author=" + author.getName() + "]";

}

}

// Library.java

import java.util.List;

public class Library {

private List<Book> books;

private List<Author> authors;

// Constructor with Book and Author dependencies (DI)

public Library(List<Book> books, List<Author> authors) {

this.books = books;

this.authors = authors;

}

// Setter for Book dependencies (DI)

public void setBooks(List<Book> books) {

this.books = books;

}

// Setter for Author dependencies (DI)

public void setAuthors(List<Author> authors) {

this.authors = authors;

}

public void displayLibraryInfo() {

System.out.println("Library Information:");

System.out.println("Books:");

for (Book book : books) {

System.out.println(book);

}

System.out.println("Authors:");

for (Author author : authors) {

System.out.println(author.getName());

}

}

}

// AppConfig.java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import java.util.Arrays;

import java.util.List;

@Configuration

public class AppConfig {

@Bean

public Author author1() {

return new Author("John Doe");

}

@Bean

public Author author2() {

return new Author("Jane Smith");

}

@Bean

public Book book1(Author author1) {

return new Book("Spring in Action", author1);

}

@Bean

public Book book2(Author author2) {

return new Book("Hibernate Essentials", author2);

}

@Bean

public Library library(List<Book> books, List<Author> authors) {

Library library = new Library(books, authors);

// Set additional books using setter injection

library.setBooks(Arrays.asList(book1(author1()), book2(author2())));

return library;

}

}

// MainApp.java

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import java.util.List;

public class MainApp {

public static void main(String[] args) {

try (AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class)) {

// Retrieve the Library bean from the context

Library library = context.getBean(Library.class);

// Use the Library object to display library information

library.displayLibraryInfo();

}

}

}

**Exercise 4: Autowiring Dependencies**

**Problem:**

**Let's create an exercise that involves using the @Autowired annotation for dependency injection.**

Create a Spring application for managing a university. Define classes for **Student**, **Course**, and **University**. The **University** class should depend on multiple **Student** and **Course** instances. Use the **@Autowired** annotation for autowiring dependencies.

Solution:

// Student.java

public class Student {

private String name;

public Student(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

// Course.java

public class Course {

private String name;

public Course(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

// University.java

import org.springframework.beans.factory.annotation.Autowired;

import java.util.List;

public class University {

private List<Student> students;

private List<Course> courses;

// Autowired constructor for Student and Course dependencies

@Autowired

public University(List<Student> students, List<Course> courses) {

this.students = students;

this.courses = courses;

}

// Autowired setter for Student dependencies

@Autowired

public void setStudents(List<Student> students) {

this.students = students;

}

// Autowired setter for Course dependencies

@Autowired

public void setCourses(List<Course> courses) {

this.courses = courses;

}

public void displayUniversityInfo() {

System.out.println("University Information:");

System.out.println("Students:");

for (Student student : students) {

System.out.println(student.getName());

}

System.out.println("Courses:");

for (Course course : courses) {

System.out.println(course.getName());

}

}

}

// AppConfig.java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import java.util.Arrays;

import java.util.List;

@Configuration

public class AppConfig {

@Bean

public Student student1() {

return new Student("Alice");

}

@Bean

public Student student2() {

return new Student("Bob");

}

@Bean

public Course course1() {

return new Course("Introduction to Computer Science");

}

@Bean

public Course course2() {

return new Course("Advanced Java Programming");

}

@Bean

public University university() {

return new University();

}

}

// MainApp.java

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {

public static void main(String[] args) {

try (AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class)) {

// Retrieve the University bean from the context

University university = context.getBean(University.class);

// Use the University object to display university information

university.displayUniversityInfo();

}

}

}

In this exercise, we use the **@Autowired** annotation to perform autowiring of dependencies in the **University** class. The **University** class has both autowired constructor and autowired setter methods for injecting lists of **Student** and **Course** instances. The **AppConfig** class defines beans for students, courses, and the university, and Spring automatically resolves and injects these dependencies into the **University** bean during the context initialization.

**Exercise 5: Qualifier Annotation for Autowiring**

**Problem:**

Let's create an exercise that involves using the **@Qualifier** annotation for more fine-grained control over autowiring.

Extend the university management application from the previous exercise. Define additional courses and students. Modify the **University** class to autowire only specific courses and students using the **@Qualifier** annotation.

**Solution:**

// University.java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

import java.util.List;

public class University {

private List<Student> computerScienceStudents;

private List<Student> javaProgrammingStudents;

private List<Course> computerScienceCourses;

private List<Course> javaProgrammingCourses;

// Autowired with Qualifier for Computer Science Students

@Autowired

@Qualifier("computerScienceStudents")

public void setComputerScienceStudents(List<Student> computerScienceStudents) {

this.computerScienceStudents = computerScienceStudents;

}

// Autowired with Qualifier for Java Programming Students

@Autowired

@Qualifier("javaProgrammingStudents")

public void setJavaProgrammingStudents(List<Student> javaProgrammingStudents) {

this.javaProgrammingStudents = javaProgrammingStudents;

}

// Autowired with Qualifier for Computer Science Courses

@Autowired

@Qualifier("computerScienceCourses")

public void setComputerScienceCourses(List<Course> computerScienceCourses) {

this.computerScienceCourses = computerScienceCourses;

}

// Autowired with Qualifier for Java Programming Courses

@Autowired

@Qualifier("javaProgrammingCourses")

public void setJavaProgrammingCourses(List<Course> javaProgrammingCourses) {

this.javaProgrammingCourses = javaProgrammingCourses;

}

public void displayUniversityInfo() {

System.out.println("University Information:");

System.out.println("Computer Science Students:");

for (Student student : computerScienceStudents) {

System.out.println(student.getName());

}

System.out.println("Java Programming Students:");

for (Student student : javaProgrammingStudents) {

System.out.println(student.getName());

}

System.out.println("Computer Science Courses:");

for (Course course : computerScienceCourses) {

System.out.println(course.getName());

}

System.out.println("Java Programming Courses:");

for (Course course : javaProgrammingCourses) {

System.out.println(course.getName());

}

}

}

// AppConfig.java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Primary;

import java.util.Arrays;

import java.util.List;

@Configuration

public class AppConfig {

@Bean

@Primary

public Student student1() {

return new Student("Alice");

}

@Bean

public Student student2() {

return new Student("Bob");

}

@Bean

@Primary

public Course course1() {

return new Course("Introduction to Computer Science");

}

@Bean

public Course course2() {

return new Course("Advanced Java Programming");

}

@Bean("computerScienceStudents")

public List<Student> computerScienceStudents() {

return Arrays.asList(student1());

}

@Bean("javaProgrammingStudents")

public List<Student> javaProgrammingStudents() {

return Arrays.asList(student2());

}

@Bean("computerScienceCourses")

public List<Course> computerScienceCourses() {

return Arrays.asList(course1());

}

@Bean("javaProgrammingCourses")

public List<Course> javaProgrammingCourses() {

return Arrays.asList(course2());

}

@Bean

public University university() {

return new University();

}

}

// MainApp.java

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {

public static void main(String[] args) {

try (AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class)) {

// Retrieve the University bean from the context

University university = context.getBean(University.class);

// Use the University object to display university information

university.displayUniversityInfo();

}

}

}

In this exercise, we use the **@Qualifier** annotation to selectively autowire specific students and courses in the **University** class. The **@Primary** annotation is also used to indicate the primary bean when multiple candidates of the same type exist. The **AppConfig** class defines additional students and courses and specifies the qualifiers for each autowired dependency in the **University** bean. This allows for more fine-grained control over the autowiring process.

**Exercise 6: Injecting Values with @Value**

**Problem:**

Let's create an exercise that involves using the **@Value** annotation for injecting values from properties files.

Create a Spring application for managing employee information.

Define a **Employee** class with attributes **id**, **name**, and **designation**.

Create an **EmployeeService** class that depends on the **Employee** class and injects values from a properties file using the **@Value** annotation.

Solution:

// Employee.java

public class Employee {

private int id;

private String name;

private String designation;

public Employee(int id, String name, String designation) {

this.id = id;

this.name = name;

this.designation = designation;

}

public int getId() {

return id;

}

public String getName() {

return name;

}

public String getDesignation() {

return designation;

}

@Override

public String toString() {

return "Employee [id=" + id + ", name=" + name + ", designation=" + designation + "]";

}

}

// EmployeeService.java

import org.springframework.beans.factory.annotation.Value;

public class EmployeeService {

private Employee employee;

// Inject values from properties file using @Value

@Value("${employee.id}")

private int employeeId;

@Value("${employee.name}")

private String employeeName;

@Value("${employee.designation}")

private String employeeDesignation;

public void initializeEmployee() {

// Create Employee object using injected values

employee = new Employee(employeeId, employeeName, employeeDesignation);

}

public void displayEmployeeInfo() {

System.out.println("Employee Information:");

System.out.println(employee);

}

}

// AppConfig.java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.PropertySource;

@Configuration

@PropertySource("classpath:employee.properties")

public class AppConfig {

@Bean

public EmployeeService employeeService() {

return new EmployeeService();

}

}

Create a **employee.properties** file in the **src/main/resources** directory with the following content:

employee.id=101

employee.name=John Doe

employee.designation=Software Engineer

// MainApp.java

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {

public static void main(String[] args) {

try (AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class)) {

// Retrieve the EmployeeService bean from the context

EmployeeService employeeService = context.getBean(EmployeeService.class);

// Initialize and display employee information

employeeService.initializeEmployee();

employeeService.displayEmployeeInfo();

}

}

}

In this exercise, we use the **@Value** annotation to inject values from a properties file into the **EmployeeService** class. The **employee.properties** file contains properties for **employee.id**, **employee.name**, and **employee.designation**. The **EmployeeService** class uses these properties to initialize an **Employee** object. The **AppConfig** class is annotated with **@PropertySource** to specify the location of the properties file. The **MainApp** class retrieves the **EmployeeService** bean from the Spring context and demonstrates how to initialize and display employee information.

**Exercise 7: Component Scanning and Auto-Discovery**

Let's create an exercise that involves using the **@ComponentScan** annotation for automatic component scanning and bean creation.

**Problem:**

Create a Spring application for managing products. Define a **Product** class and a **ProductService** class. Use the **@Component** annotation on the **ProductService** class and the **@Autowired** annotation for dependency injection.

Solution:

// Product.java

public class Product {

private String name;

private double price;

public Product(String name, double price) {

this.name = name;

this.price = price;

}

public String getName() {

return name;

}

public double getPrice() {

return price;

}

@Override

public String toString() {

return "Product [name=" + name + ", price=" + price + "]";

}

}

// ProductService.java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

@Component

public class ProductService {

private Product product;

// Autowired constructor for Product dependency

@Autowired

public ProductService(Product product) {

this.product = product;

}

public void displayProductInfo() {

System.out.println("Product Information:");

System.out.println(product);

}

}

// AppConfig.java

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan(basePackages = "com.example") // Specify the base package for component scanning

public class AppConfig {

// No specific bean definitions are needed in this case

}

// MainApp.java

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {

public static void main(String[] args) {

try (AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class)) {

// Retrieve the ProductService bean from the context

ProductService productService = context.getBean(ProductService.class);

// Use the ProductService object to display product information

productService.displayProductInfo();

}

}

}

In this exercise, we use the **@Component** annotation on the **ProductService** class, indicating that it should be automatically discovered and registered as a Spring bean during component scanning. The **@Autowired** annotation is used for constructor injection. The **AppConfig** class includes the **@ComponentScan** annotation, specifying the base package to scan for components. The **MainApp** class retrieves the **ProductService** bean from the Spring context and demonstrates how to display product information.

Make sure to adjust the **basePackages** attribute in **@ComponentScan** to match the package structure of your project.

Top of Form

**Exercise 8: Bean Scopes with @Scope**

Problem:

Let's create an exercise that involves using the **@Scope** annotation for defining different scopes of beans.

Create a Spring application for managing customer information. Define a **Customer** class with attributes **customerId** and **customerName**. Create a **CustomerService** class that depends on the **Customer** class. Configure the **CustomerService** bean with different scopes using the **@Scope** annotation.

Solution:

// Customer.java

public class Customer {

private static int nextId = 1;

private int customerId;

private String customerName;

public Customer(String customerName) {

this.customerId = nextId++;

this.customerName = customerName;

}

public int getCustomerId() {

return customerId;

}

public String getCustomerName() {

return customerName;

}

@Override

public String toString() {

return "Customer [customerId=" + customerId + ", customerName=" + customerName + "]";

}

}

// CustomerService.java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

@Component

@Scope("singleton") // Default scope

public class CustomerService {

private Customer customer;

// Autowired constructor for Customer dependency

@Autowired

public CustomerService(Customer customer) {

this.customer = customer;

}

public void displayCustomerInfo() {

System.out.println("Customer Information:");

System.out.println(customer);

}

}

// AppConfig.java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Scope;

@Configuration

public class AppConfig {

@Bean

@Scope("prototype") // Different scope for the Customer bean

public Customer customer() {

return new Customer("John Doe");

}

}

// MainApp.java

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {

public static void main(String[] args) {

try (AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class)) {

// Retrieve the CustomerService bean from the context

CustomerService customerService1 = context.getBean(CustomerService.class);

CustomerService customerService2 = context.getBean(CustomerService.class);

// Display customer information from singleton-scoped CustomerService

customerService1.displayCustomerInfo();

// Display customer information from prototype-scoped CustomerService

customerService2.displayCustomerInfo();

}

}

}

In this exercise, we use the **@Scope** annotation to define different scopes for the **CustomerService** and **Customer** beans. The **CustomerService** bean is configured with the default **singleton** scope, while the **Customer** bean is configured with the **prototype** scope. The **MainApp** class retrieves two instances of the **CustomerService** bean from the Spring context and demonstrates the impact of different bean scopes on the customer information display.

Feel free to experiment with other scope values like **request**, **session**, or custom scopes based on your application's requirements.

**Exercise 9: Initializing and Destroying Beans with @PostConstruct and @PreDestroy**

Problem:

Let's create an exercise that involves using the **@PostConstruct** and **@PreDestroy** annotations for initializing and destroying beans.

Create a Spring application for managing resources. Define a **Resource** class with a method to initialize and release resources. Create a **ResourceService** class that depends on the **Resource** class. Use the **@PostConstruct** and **@PreDestroy** annotations to initialize and release resources.

// Resource.java

import javax.annotation.PostConstruct;

import javax.annotation.PreDestroy;

public class Resource {

private String resourceName;

public Resource(String resourceName) {

this.resourceName = resourceName;

}

public String getResourceName() {

return resourceName;

}

@PostConstruct

public void initializeResource() {

System.out.println("Initializing Resource: " + resourceName);

// Add resource initialization logic here

}

@PreDestroy

public void releaseResource() {

System.out.println("Releasing Resource: " + resourceName);

// Add resource release logic here

}

}

// ResourceService.java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

@Component

public class ResourceService {

private Resource resource;

// Autowired constructor for Resource dependency

@Autowired

public ResourceService(Resource resource) {

this.resource = resource;

}

public void displayResourceInfo() {

System.out.println("Resource Information:");

System.out.println(resource.getResourceName());

}

}

// AppConfig.java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class AppConfig {

@Bean

public Resource resource() {

return new Resource("Database Connection");

}

}

// MainApp.java

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {

public static void main(String[] args) {

try (AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class)) {

// Retrieve the ResourceService bean from the context

ResourceService resourceService = context.getBean(ResourceService.class);

// Display resource information

resourceService.displayResourceInfo();

}

}

}

In this exercise, we use the **@PostConstruct** and **@PreDestroy** annotations on methods within the **Resource** class to define logic that should be executed after bean creation and before bean destruction, respectively. The **ResourceService** class depends on the **Resource** class, and Spring ensures that the **initializeResource** method is called after the bean is created, and the **releaseResource** method is called before the bean is destroyed.

This mechanism is useful for managing resources such as opening and closing connections, acquiring and releasing locks, or any other resource-related initialization and cleanup tasks.

**Exercise 10: Using @Value with Default Values**

Problem:

Let's create an exercise that involves using the **@Value** annotation with default values for properties.

Extend the employee management application from Exercise 6. Modify the **EmployeeService** class to use the **@Value** annotation to inject values from a properties file. Provide default values for properties in case they are not specified in the properties file.

Solution:

// EmployeeService.java

import org.springframework.beans.factory.annotation.Value;

import org.springframework.stereotype.Component;

@Component

public class EmployeeService {

private String employeeId;

private String employeeName;

private String employeeDesignation;

// Inject values from properties file using @Value with default values

@Value("${employee.id:1001}") // Default value: 1001

public void setEmployeeId(String employeeId) {

this.employeeId = employeeId;

}

@Value("${employee.name:John Doe}") // Default value: John Doe

public void setEmployeeName(String employeeName) {

this.employeeName = employeeName;

}

@Value("${employee.designation:Software Engineer}") // Default value: Software Engineer

public void setEmployeeDesignation(String employeeDesignation) {

this.employeeDesignation = employeeDesignation;

}

public void displayEmployeeInfo() {

System.out.println("Employee Information:");

System.out.println("ID: " + employeeId);

System.out.println("Name: " + employeeName);

System.out.println("Designation: " + employeeDesignation);

}

}

Update the **employee.properties** file in the **src/main/resources** directory with the following content:

employee.properties

# No properties specified, defaults will be used

// AppConfig.java

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.PropertySource;

@Configuration

@ComponentScan(basePackages = "com.example")

@PropertySource("classpath:employee.properties")

public class AppConfig {

// No specific bean definitions are needed in this case

}

// MainApp.java

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {

public static void main(String[] args) {

try (AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class)) {

// Retrieve the EmployeeService bean from the context

EmployeeService employeeService = context.getBean(EmployeeService.class);

// Display employee information with default values

employeeService.displayEmployeeInfo();

}

}

}

In this exercise, we use the **@Value** annotation with default values to inject properties from a properties file. If the specified properties are not found in the file, default values are used. This allows for a graceful fallback mechanism when certain properties are not explicitly provided. The **employee.properties** file is intentionally left empty to demonstrate the use of default values.

**Exercise 11: Autowiring Dependencies with @Autowired**

**Problem:**

Create a Spring application for managing employees. Define classes for **Employee**, **Department**, and **EmployeeService**. The **EmployeeService** class should depend on multiple **Employee** instances and a **Department** instance. Use the **@Autowired** annotation for autowiring dependencies.

**Solution:**

// Employee.java

public class Employee {

private String name;

private int employeeId;

public Employee(String name, int employeeId) {

this.name = name;

this.employeeId = employeeId;

}

public String getName() {

return name;

}

public int getEmployeeId() {

return employeeId;

}

}

// Department.java

public class Department {

private String departmentName;

public Department(String departmentName) {

this.departmentName = departmentName;

}

public String getDepartmentName() {

return departmentName;

}

}

// EmployeeService.java

import org.springframework.beans.factory.annotation.Autowired;

import java.util.List;

public class EmployeeService {

private List<Employee> employees;

private Department department;

// Autowired constructor for Employee and Department dependencies

@Autowired

public EmployeeService(List<Employee> employees, Department department) {

this.employees = employees;

this.department = department;

}

// Autowired setter for Employee dependencies

@Autowired

public void setEmployees(List<Employee> employees) {

this.employees = employees;

}

// Autowired setter for Department dependency

@Autowired

public void setDepartment(Department department) {

this.department = department;

}

public void displayEmployeeServiceInfo() {

System.out.println("Employee Service Information:");

System.out.println("Employees:");

for (Employee employee : employees) {

System.out.println("ID: " + employee.getEmployeeId() + ", Name: " + employee.getName());

}

System.out.println("Department: " + department.getDepartmentName());

}

}

// AppConfig.java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import java.util.Arrays;

import java.util.List;

@Configuration

public class AppConfig {

@Bean

public Employee employee1() {

return new Employee("Alice", 101);

}

@Bean

public Employee employee2() {

return new Employee("Bob", 102);

}

@Bean

public Department department() {

return new Department("IT Department");

}

@Bean

public EmployeeService employeeService() {

return new EmployeeService();

}

}

// MainApp.java

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {

public static void main(String[] args) {

try (AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class)) {

// Retrieve the EmployeeService bean from the context

EmployeeService employeeService = context.getBean(EmployeeService.class);

// Use the EmployeeService object to display employee service information

employeeService.displayEmployeeServiceInfo();

}

}

}

In this exercise, we use the **@Autowired** annotation to perform autowiring of dependencies in the **EmployeeService** class. The **EmployeeService** class has both autowired constructor and autowired setter methods for injecting lists of **Employee** instances and a **Department** instance. The **AppConfig** class defines beans for employees, a department, and the employee service, and Spring automatically resolves and injects these dependencies into the **EmployeeService** bean during the context initialization.

If you want to achieve the same dependency injection without using the **@Autowired** annotation, you can use constructor injection and setter methods. Here's the modified solution without **@Autowired**:

// Employee.java

public class Employee {

private String name;

private int employeeId;

public Employee(String name, int employeeId) {

this.name = name;

this.employeeId = employeeId;

}

public String getName() {

return name;

}

public int getEmployeeId() {

return employeeId;

}

}

// Department.java

public class Department {

private String departmentName;

public Department(String departmentName) {

this.departmentName = departmentName;

}

public String getDepartmentName() {

return departmentName;

}

}

// EmployeeService.java

import java.util.List;

public class EmployeeService {

private List<Employee> employees;

private Department department;

// Constructor for Employee and Department dependencies

public EmployeeService(List<Employee> employees, Department department) {

this.employees = employees;

this.department = department;

}

// Setter for Employee dependencies

public void setEmployees(List<Employee> employees) {

this.employees = employees;

}

// Setter for Department dependency

public void setDepartment(Department department) {

this.department = department;

}

public void displayEmployeeServiceInfo() {

System.out.println("Employee Service Information:");

System.out.println("Employees:");

for (Employee employee : employees) {

System.out.println("ID: " + employee.getEmployeeId() + ", Name: " + employee.getName());

}

System.out.println("Department: " + department.getDepartmentName());

}

}

// AppConfig.java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import java.util.Arrays;

import java.util.List;

@Configuration

public class AppConfig {

@Bean

public Employee employee1() {

return new Employee("Alice", 101);

}

@Bean

public Employee employee2() {

return new Employee("Bob", 102);

}

@Bean

public Department department() {

return new Department("IT Department");

}

@Bean

public EmployeeService employeeService(List<Employee> employees, Department department) {

EmployeeService employeeService = new EmployeeService(employees, department);

// Set additional properties if needed using setter methods

return employeeService;

}

}

// MainApp.java

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import java.util.List;

public class MainApp {

public static void main(String[] args) {

try (AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class)) {

// Retrieve the EmployeeService bean from the context

EmployeeService employeeService = context.getBean(EmployeeService.class);

// Use the EmployeeService object to display employee service information

employeeService.displayEmployeeServiceInfo();

}

}

}

In this version, constructor injection and setter methods are used to achieve dependency injection without relying on the **@Autowired** annotation. The **EmployeeService** bean is created in the **AppConfig** class, and dependencies are provided through the constructor and setter methods.