Systems for Design and Implementation

Lecture 5

Outline

- IoC Containers

IoC Containers

- **DI, IoC**

Spring core tasks

- The IoC container - used to manage and configure POJOs
- Bean ⇔ POJO instance, Component ⇔ POJO class
- Goal:
    - Build applications with POJOs (minimal invasive apps)
    - Loose coupling (DI, interface orientation)
    - Declarative programming (aspects, conventions)
    - Eliminate boilerplate code (aspects, templates)

Steps for working with POJOs in Spring IoC

1. Design a POJO class
2. Create a Java Config class / setup components; @*Configuration*, *@Bean / @Component, @Repository, @Service, @Controller*
3. Instantiate the Spring IoC; scan for annotated classes
4. The POJO instances are now accessible

Bean names

- Implicitly method name
- @Bean(name=”newName”)

@Component, @Repository, @Service, @Controller

@Component(“componentName”) etc

```
public abstract class Product{
    private String name;
    private double price;
    public Product(){}
    public Product(String name, double price){
    //…
    }}
public Class Battery extends Product{
    private boolean rechargeable;
    public Batter(){}
    public Battery(String name, double price){
        super(name, price);
    }}
public class Disc extends Product{
    private int capacity;
    public Disc(){}
    public Disc(String name, double price){
        super(name, price);
    }}
@Configuration
public class ShopConfiguration{
    @Bean
    public Product battery1(){
        Battery b1=new Battery("battery1", 20);
        b1.setRechargeable(true);
        return b1;
    }
    @Bean
    public Product dvd1(){
        Disc d1=new Disc("dvd1", 10);
        d1.setCapacity(700);
        return d1;
    }}

//main
Product b1=context.getBean("battery1", Product.class);
Product d1=context.getBean("dvd1", Product.class);
```

POJO References and Autowiring

- The POJO instances often need to collaborate with each other
- For POJOs defined in Java Config classes - use standard Java code to create references between beans
- For autowiring - mark a field, setter, constructor or arbitrary method with *@Autowired*
- *@Autowired* - array, collection, map
- *@Autowired(required=false)*

spring-di

! bean names should be unique

Autowire ambiguity
@Primary; Qualifier("name") - may also be applied to a method argument

POJO references from multiple locations

1. *context.register({"Config1.class","Config2.class"});*
2. *@Import(Config1.class);*
    - *@Value("#{beanName}")*

POJO Scopes

- **singleton**
- prototype
- request
- session
- globalSession

```
@Component
public class ShoppingCart{
    private List<Product> items=new ArrayList<>();
    public void add(Product item){items.add(item);}
    public List<Product> get(){return items;}
}

@Configuration
public class ShopConfig{
    @Bean
    public Product battery(){
        return new Battery(“battery1”,10);}
    @Bean public Product cd(){
        return new Disc(“cd1”,20);}
    @Bean public Product dvd(){
        return new Disc(“dvd1”,20);}

//main
Product b1=context.getBean(“battery”);
Product c1=context.getBean(“cd”);
Product d1=context.getBean(“dvd”);

ShoppingCart cart1=context.getBean(“shoppingCart”);
cart1.add(b1);
cart1.add(c1);
System.out.println(cart1.get());

ShoppingCart cart2=context.getBean(“shoppingCart”);
cart2.add(d1);
System.out.println(cart2.get());
```

```
@Component
@Scope(“prototype”)
public class ShoppingCart{...}
```

Customize POJO initialization and destruction

- We want to perform certain tasks before the POJO is used (opening a file, network/db connection etc); and after it is used
- *@Bean(initMethod=”methodInitName”, destroyMethod=”methodDestroyName”)*
- *@PostConstruct*, *@PreDestroy*

Lazy POJO initialization --- *@Lazy*

POJO initialization order

- *@DependsOn(“beanName”)*
- *@DependsOn({“beanName1”, “beanName2”})*

Validate/modify POJOs using Post Processors

- We want to apply tasks to all bean instances during construction
- Allows bean processing before and after the bean initialization callback
- Processes all bean instances
- E.g: *@Required* is a bean built-in bean post processor --- may throw *BeanInitializationException*
- -> implement *BeanPostProcessor*

```
@Component
public class MyComponent implements BeanPostProcessor{
    public Object
postProcessBeforeInitialization(Object bean, String
beanName) throws BeansException{
        //process bean…
        return bean;
    }
    public Object
postProcessAfterInitialization(Object bean, String
beanName){
        //process bean…
        return bean;
    }
}
```

Bean creation lifecycle

1. Create bean instance
2. Set values and bean references to properties
3. Pass bean instance to `postProcessBeforeInitialization()`
4. Call the init callback
5. The bean is ready
6. Pass the bean instance to `postProcessAfterInitialization`
7. Call the destroy callback (when the container is shut down)

spring-jpa