### Spring Udemy Notes

**why spring:**

popular framework

simper and lightweight alternative to J2EE

large number of helper classes. make things easier

get rid of EJB, former bad reputation.

**New for Spring 5:**

java 8 or higher

Upgrade Spring MVC to use new versions of Servlet API 4.0

Add new reactive programming framework: spring webFlux

**Core Container:** Beans:Core:SpEL:Context:

**Infrastructure:** AOP: Aspect Oriented Programming

**Data Access Layer:** JDBC ORM Transactions OXM JMS:Java message service

**Web layer**: Servlet, websocket, web ,porlet

**Test layer:** unit, integration, mock

**Inversion of Control(IoC)**

The approach of outsourcing the construction and management of objects.

Software Engineering Best Practice: code to an interface

**Spring Container:**

Primary functions:

1. Create and manager objects(Inversion of Control)

2. Inject object's dependencies(Dependency Injection)

**Configuring Spring Container:**

1. XML configuration file(legacy)

2. Java Annotations(modern)

3. Java Source Code(modern)

**Spring Development Process**

1. Configure ur spring Beans: treat the beans as java objects

2. Create a Spring Container: generically known as ApplicationContext

3. Retrieve Beans from Spring Container.

**Dependency Injection:**

Injection types two common: 1. Constructor Injection. 2. Setter Injection !!// treat dependency as a helper

**Setter Injection:**

1. Create setter method(s) in class for injections. 2. Configure the dependency injection in Srping config file

**Injecting Literal Values:**

1. Create setter methods in class for injections.: create private fields, create setter methods.

2. Configure the injection in Spring config file.



**Singleton:**

1.Spring Container creates only one instance of the bean, by default.

2.It's cached in memory.

3.All requests for bean, will return a SHARED reference to the SAME bean.

**Bean Prototype Scope: @Scope(“prototype”)**

new bean object for each request: different address different object

IF not mentioned,would be considered by default: singleton scope: only one instance, different requests only share the bean.

**Default scope is singleton: @Scope(“singleton”)**

Spring Container creates only one instance of the bean, by default.

It is cached in memory.

All requests for the bean: will return a SHARED reference to the SAME bean.

**Bean lifecycle:**

container started--> Bean Instantiated--> Dependencies Injected --> Internal Spring Processing --> your custom Init Method --> Bean is ready for use --

-- container is shutdown --> your custom destroy method --> stop

**Bean lifecycle methods/ hooks:**

1. you can add custom code during bean initializaion, calling custom business logic methods, setting up handles to resources(db, sockets, file etc).

2. you can add custom code during bean destruction, calling custom business logic method, clean up handles to resources(db, sockets, files ect).

Init: method configuration:set up bean initialization(init-method="any method name").

Sestroy:method configuration: set up bean destroy method(destroy-method="any method name").

**Development process:**

1. Define your methods for init and destroy.

2. Configure the method names on Spring config file

**Java Annotations:**

1. Special labels/markers added to Java classes.
2. Provide meta-data about the class.
3. Processed at compile time or run-time for special processing.

**Why Spring configuration with annotations:**

1. XML configuration can be verbose.
2. Configure Spring beans with annotations.
3. Annotations minimizes the XML configuration.

**Scan for component classes:**

1. Spring will scan Java classes for special annotations.
2. Automatically register beans in Spring container.

**Development Process:**

1. Enable component scanning in Spring config file.
2. Add the @Component Annotation to Java classes.
3. Retrieve bean from Spring container.

**Default Bean IDs**: the class name, make first letter lower-case

**Spring Dependency Injection with Annotations and Autowiring:**

**Spring Auto-wiring:**

1. For dependency injection, Spring can use auto wiring.
2. Spring will look for a class that matches the property: matches by type: class or interface
3. Spring will inject it automatically ... hence it is autowired

**Autowiring example:**

1. Injecting FortuneService into a Coach implementation.
2. Spring will scan @Components.
3. Any one implements FortuneService interface.
4. If so, let’s inject them. FOr example: HappyFortuneService.

**Autowiring Injection Types:**

1. Constructor Injection
2. Setter Injection
3. Field Injections

**Development Process - Constructor Injection**

1. Define the dependency interface and class.
2. Create a constructor in class for injections. :constructor always have the same name as class.
3. Configure the dependency injection with @Autowired Annotation

**Setter Injection:** Inject dependencies by calling setter methods on class.

**Development Process - Setter Injection:**

1. Create setter methods in class for injections.
2. Configure the dependency injection with @Autowired Annotation

**Field Injection:** Inject dependencies by setting field values on class directly(even private fields)

Accomplished by using Java Reflection

**Development Process - Field Injection:**

1. Configure the dependency injection with Autowired Annotation: applied directly to the field, no need for setter methods.

**Annotation Autowiring and Qualifiers:**

**Can apply @Qualifier annotation to:** 1. Constructor injection. 2. Setter injection methods. 3. Field injection