

Spring Dependency Injection

524730-1
2022년 봄학기¹
5/25/2022
박경신

Dependency (의존성)

- Spring 생산/유지보수 용이성의 핵심요소
- Single class > Dependency per code > Dependency Injection

Single class 의존성

- ▣ 하나의 클래스에 모든 기능을 다 집어넣는 경우
 - 4만 줄짜리 클래스 본 적 없죠?
- ▣ 사소한 변경 하나에 의해서도 전체 코드를 수정하는 결과
 - 편집의 어려움
 - 비효율적 재사용
 - 이 클래스에 의존하고 있는 다른 클래스들이 변경에 취약해짐

코드 레벨 의존성의 문제

- 일반적인 자바 개발에서 이뤄지는 코드 구성의 결과

```
private Encryptor enc = new Encryptor();
```

- 이 코드가 정상적으로 작동하기 위해서는

- Encryptor 타입의 클래스 Encryptor가 존재해야 할 것

- 만약 Encryptor 대신 그 서브타입인 **FastEncryptor**를 사용해야 한다면?

- 내 코드를 수정해야 함

```
private Encryptor enc = new FastEncryptor();
```

- Encryptor를 사용하던 클래스가 많으면 많을수록 더 많은 수정
 - 더 많은 테스트
 - 더 많은 오류
 - 더 많은 수정 전파

- Encryptor가 완성될 때 까지 내 코드를 테스트 불가능

조금 더 나은 모델

- 외부 코드에 의한 탑재

```
public class FileEncryptor{  
    private Encryptor enc;  
  
    public FileEncryptor(Encryptor enc) {  
        this.enc = enc;  
    }  
}
```

- 생성자 호출시 Encryptor 탑재의 아무 클래스나 존재하면 가능
- 전통적인 code with interface 모델

Dependency Injection (의존성 주입)

- 위 모델을 사용하기 위한 코드

```
Encryptor enc = new Encryptor();  
FileEncryptor fileEnc = new FileEncryptor(enc);
```

- FileEncryptor가 의존하는 객체는 자신의 코드가 아닌 외부에서 생성되어 넣어 줌
 - 이러한 방식을 **의존성 주입 (Dependency Injection)**이라 부름

의존성 주입 방식

□ Factory 패턴

- 한 타입(Encryptor)을 만족시키는 다양한 서브타입(FastEncryptor, PlainEncryptor, ...)을 생성해서 돌려주는 코드

```
Enc = EncFactory.getEncryptor(EncType.Fast);
```



- 변경이 일어나면 이 조립기의 코드에만 영향이 미침
 - ▣ 팩토리에 Fast 대신 SuperFast Encryptor를 쓰는 코드만 넣으면 다른 코드에는 영향이 없음

의존성 주입 방식

▣ 또 다른 장점

- Encryptor 클래스가 완성되지 않았다면
 - ▣ 팩토리에 MockEncryptor 임시 클래스를 장착
 - ▣ 해당 클래스는 가짜 객체로, 무조건 일정한 결과를 돌려주는 단순한 코드
 - ▣ 이러한 클래스를 MockObject 혹은 test stub이라 부름
- Encryptor 서브클래스들과 FileEncryptor 클래스를 작업하는 사람이 동시에 작업 분담 가능

생성자 방식 DI

- 속성(Property)로 의존 객체를 전달받음

```
public class FileEncryptor{  
    private Encryptor enc;  
  
    public FileEncryptor(Encryptor enc) {  
        this.enc = enc;  
    }  
}
```

- 생성자 실행 후 언제나 객체 사용이 가능

Setter 방식 DI

- 속성(Property)로 의존 객체를 전달받음
- 자바빈즈 영향으로 setPropertyName()과 같은 메소드로 속성 설정 가능

```
public class FileEncryptor{  
    private Encryptor encryptor;  
  
    public void setEncryptor(Encryptor enc) {  
        this.encryptor = enc;  
    }  
}
```

- 메소드 이름으로 속성의 타입을 추측 가능

스프링 DI 컨테이너

- ▣ GenericXmlApplicationContext : 팩토리/조립기 클래스
 - configLocation의 정보를 토대로 객체를 생성하고 의존성을 확보한 뒤 보관 (container)
 - 이 객체들을 스프링 빈(Spring Bean)이라 부름

```
String configLocation = "classpath:applicationContext.xml";
AbstractApplicationContext ctx =
    new GenericXmlApplicationContext(configLocation);
Project project = ctx.getBean("sampleProject", Project.class);
project.build();
ctx.close();
```

스프링 DI 컨테이너

- Name을 기준으로 setPropertyName과 같은 DI 함수를 호출함
- 해당 객체에 이러한 함수를 구현해야 함

```
<bean id= "sampleProject" class= "kr.ac.dankook.ace.Project">
<property name= "srcDirs">
```



```
sampleProject.setSrcDirs(srcDirs);
```

스프링 컨테이너의 종류

- Spring **BeanFactory** Container
 - 구버전 스프링, 순수한 DI 생성기
- Spring **ApplicationContext** Container
 - BeanFactory + 부가기능

Spring DI

▣ Spring 의존성 주입

- Spring ApplicationContext 를 이용해 의존성 주입을 구현함
- ApplicationContext가 관리하는 객체들을 Bean이라고 함
- 빈과 빈사이의 의존관계는 다음 3가지 방식으로 정의
 1. **XML based configuration file**
 2. **Java based configuration**
 3. **Annotation based configuration**

Lombok Library

- ▣ 자바 컴파일 시점에서 특정 어노테이션으로 해당 코드를 추가할 수 있는 라이브러리
- ▣ <https://projectlombok.org/download>
- ▣ ~>java -jar lombok.jar

The screenshot shows the Project Lombok website on the left and its installer window on the right.

Project Lombok Website (Left):

- URL: <https://projectlombok.org/download>
- Download button: "Download 1.18.24"
- Links: "changelog", "older versions", "Feeling adventurous? Download the latest **snapshot** release", "credits", "Copyright © 2009-2022 The Project Lombok Authors, licensed under the MIT license".

Project Lombok v1.18.24 - Installer (Right):

- Java:** (and tools that invoke javac such as *ant* and *maven*)
Lombok works 'out of the box' with javac.
Just make sure the lombok.jar is in your classpath when you compile.
Example: `javac -cp lombok.jar MyCode.java`
- IDEs:**
Lombok can update your Eclipse or eclipse-based IDE to fully support all Lombok features.
Select IDE installations below and hit 'Install/Update'.
List of checked IDE installations:
 - C:\Users\park\eclipse\java-2021-12\eclipse\eclipse.exe
 - C:\Users\park\jee-2020-12\eclipse\eclipse.exe
 - C:\Users\park\jee-2021-12\eclipse\eclipse.exe
- Buttons: "Specify location...", "Show me what this installer will do to my IDE installation.", "Uninstall lombok from selected IDE installations.", "Install / Update", "Quit Installer".

SpringDI Test

□ Create a Maven Project

New Maven project
Select project name and location

Create a simple project (skip archetype selection)
 Use default Workspace location
Location:
 Add project(s) to working set
Working set:

New Maven project
Configure project

Artifact
Group Id: kr.ac.dankook
Artifact Id: SpringDI Test
Version: 0.0.1-SNAPSHOT
Packaging: jar
Name:
Description:
Parent Project
Group Id:
Artifact Id:
Version:
Advanced

?

< Back Cancel

?

< Back Cancel

POM 수정하기

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apa</pre>

<modelVersion>4.0.0</modelVersion>



<groupId>kr.ac.dankook</groupId>



<artifactId>SpringDITest</artifactId>



<version>0.0.1-SNAPSHOT</version>



<properties>



<java-version>11</java-version>



<org.springframework-version>5.2.8.RELEASE</org.springframework-version>



<org.aspectj-version>1.6.10</org.aspectj-version>



<org.slf4j-version>1.6.6</org.slf4j-version>



</properties>



<dependencies>



<!-- Spring --&gt;</p>

<dependency>



<groupId>org.springframework</groupId>



<artifactId>spring-context</artifactId>



<version>${org.springframework-version}</version>



<exclusions>



<!-- Exclude Commons Logging in favor of SLF4j -->



<exclusion>



<groupId>commons-logging</groupId>



<artifactId>commons-logging</artifactId>



</exclusion>



</exclusions>



</dependency>



<!-- lombok --&gt;</p>

<dependency>



<groupId>org.projectlombok</groupId>



<artifactId>lombok</artifactId>



<version>1.18.24</version>



<scope>provided</scope>



</dependency>



<!-- @Inject -->



<dependency>



<groupId>javax.inject</groupId>



<artifactId>javax.inject</artifactId>



<version>1</version>



</dependency>



</dependencies>



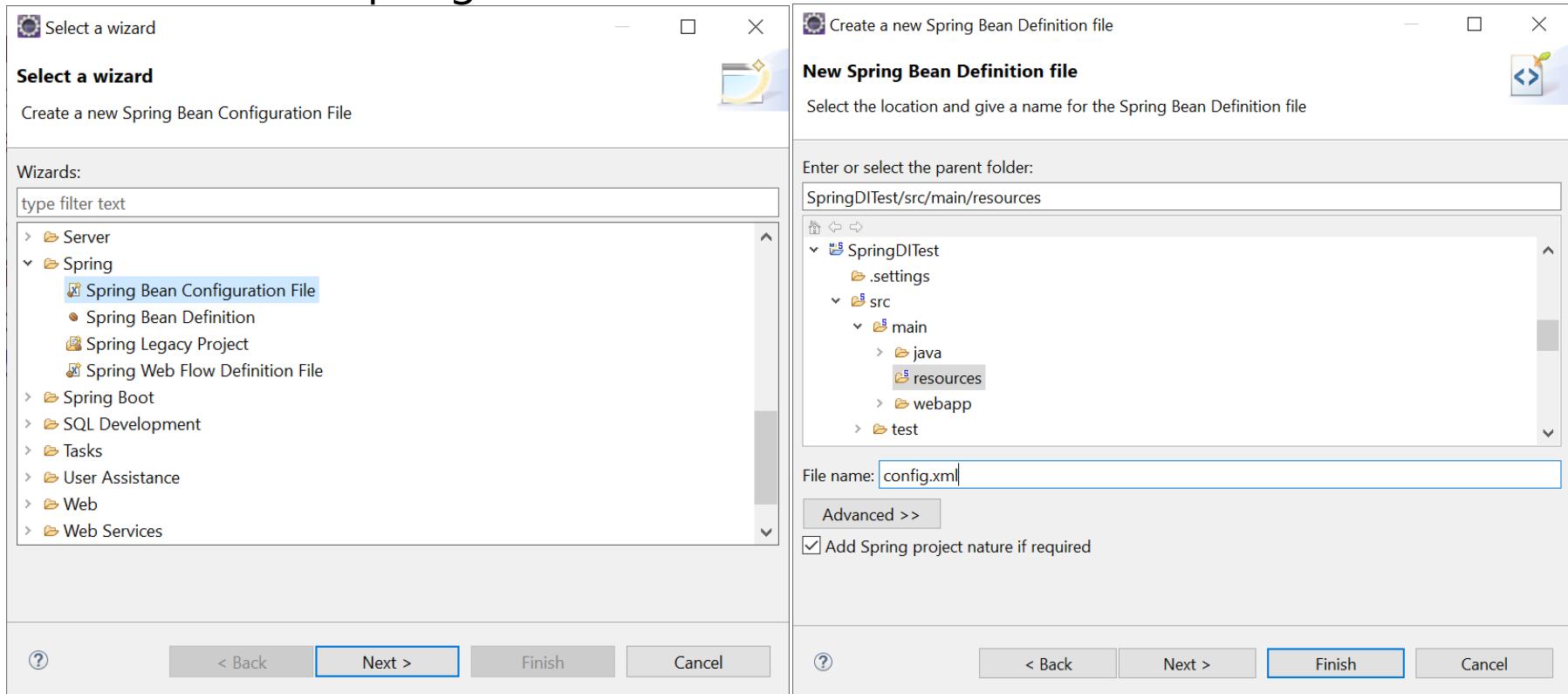
</project>


```

XML 설정파일 만들기

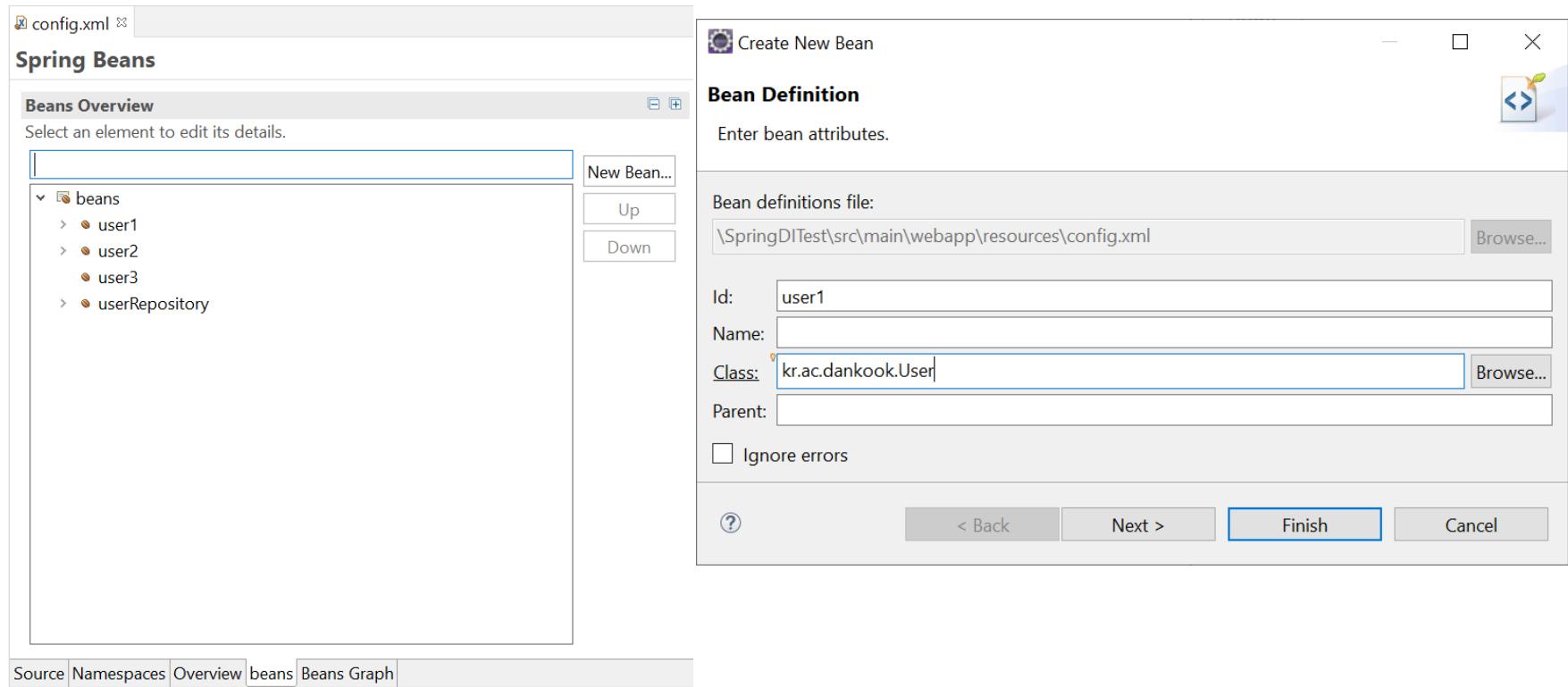
▣ XML 설정파일

- New->Other->Spring Bean Configuration File: **config.xml** 생성
- Folder: SpringDITest/src/main/resources



<bean>

- ▣ config.xml 파일을 열고, 생성할 bean 객체들 설정
 - id : 빈의 고유한 이름(user1, user2, ...)
 - class : 빈의 완전한 이름(fully qualified name)
(kr.ac.dankook.User)



<constructor-arg>

- ▣ 각각 <constructor-arg> 생성자 방식 또는 <property> 프로퍼티 설정 방식으로 DI 수행
- ▣ 생성자에 들어가야 하는 파라미터 수 만큼 <bean>의 자식 태그로 <constructor-arg> 생성
 - **public User(String name, int password)** 생성자라면

```
<bean id="user1" class="kr.ac.dankook.User">
    <constructor-arg value="ace"/>
    <constructor-arg value="12345" type="int"/>
</bean>
```

config.xml

- ▣ 이는 **User user1 = new User("ace", 12345);** 에 해당

<property>

- ▣ 프로퍼티 설정 방식
- ▣ 각 객체의 속성 이름을 name으로 이용해 setter 함수 호출
 - User 객체의 속성이 name, password

```
<bean id="user2" class="kr.ac.dankook.User">
    <property name="name" value="Park" />
    <property name="password" value="20" />
</bean>
```

config.xml

- ▣ user2.setName("Park"); user2.setPassword(20); 과 같은 효과
- 객체의 속성이 threshold라면

```
<bean id="authFailLogger" class="kr.ac.dankook.AuthFailLogger">
    <property name="threshold" value="2" />
</bean>
```

config.xml

- ▣ authFailLogger.setThreshold(2); 와 같은 효과
- 여러 값을 담기 위해 <list>태그 포함 가능

<ref>

- ▣ <constructor-arg>, <property> 태그 내에서 다른 빈을 참조하기 위해 사용
 - <ref bean="다른 빈 id"> 와 같이 사용

```
<bean id="userRepository" class="kr.ac.dankook.ace.UserRepository">
    <property name="users">
        <list>
            <ref bean="user1"/>
            <ref bean="user2"/>
            <ref bean="user3"/>
        </list>
    </property>
</bean>
```

메인자바코드

▣ 메인자바 테스트 프로그램에서 할 일

- config.xml에서는 bean 객체들이 설정
 - ▣ 3 User - 각각 생성자로 아이디/패스워드 가짐
 - ▣ UserRepository - 위의 사용자들을 users라는 속성으로 가짐
- resources 폴더 안의 config.xml을 참조하라고 프로그램에 알려주기

SpringDITest.java

```
GenericXmlApplicationContext context =  
    new GenericXmlApplicationContext("classpath:config.xml");  
User user1 = (User) context.getBean("user1");  
System.out.println(user1);  
User user2 = (User) context.getBean("user2");  
System.out.println(user2);  
User user3 = (User) context.getBean("user3");  
System.out.println(user3);  
UserRepository users = (UserRepository) context.getBean("userRepository");  
System.out.println(users);
```

자바빈

>User 클래스

```
package kr.ac.dankook;
import org.springframework.stereotype.Component;
import lombok.Data;

@Component // Spring component
@Data // Lombok @Getter/@Setter/@ToString
public class User {
    private String name;
    private int password;
    public User() {
        this("", 0);
    }
    public User(String name, int password) {
        this.name = name;
        this.password = password;
    }
}
```

자바빈

>UserRepository 클래스

```
package kr.ac.dankook;

import java.util.List;
import org.springframework.stereotype.Component;
import lombok.Data;

@Component // Spring component
@Data // Lombok @Getter/@Setter/@ToString
public class UserRepository {
    List<User> users;
}
```

XML 설정파일 지정하기

- ❑ 실제로는 관리의 용이성을 위해 여러 파일에 나누어 사용
- ❑ 매개변수에 여러 파일 삽입 가능

```
new ClassPathXmlApplicationContext("classpath:config.xml", "classpath:config-board.xml");
```

- ❑ 매개변수에 상대경로 사용 가능

```
new ClassPathXmlApplicationContext("classpath:/conf/config.xml",  
"classpath:config-board.xml");
```

- ❑ 매개변수에 파일 시스템 사용 가능

```
new ClassPathApplicationContext("file:../local/config.xml", "classpath:config-board.xml");
```

list, map, set

- ▣ 각각 자바의 List, Map, Set 컬렉션 타입에 대응
- ▣ list

```
<bean id= "userRepository" class= "kr.ac.dankook.UserRepository">
</property>
<list>
    <ref bean= "user1"/>
    <ref bean= "user2"/>
    <ref bean= "user3"/>
</list>
</property>
</bean>
```

list, map, set

□ map – generic 사용

```
<property name="sensorMap">
  <map>
    <entry>
      <key>
        <value>frontDoor</value>
      </key>
      <ref bean="sensor1"/>
    </entry>
    <entry key="backDoor" value-ref="sensor2" />
  </map>
</property>
```

□ set

```
<property name="agentCodes">
  <set>
    <value>200</value>
    <value>300</value>
  </set>
</property>
```

<props>, <prop>

- K-V 쌍으로 이루어진 설정 값의 목록을 나타낼 때는 java.util.Properties 사용 추천
- 예 : 서버 관련 설정 파일 servers.conf 존재시

```
<property name="additionalInfo">
<props>
<prop key="threshold">1500</prop>
<prop key="retry">3</prop>
</props>
</property>
```



```
Properties prop = new Properties();
prop.setProperty("threshold", "1500");
prop.setProperty("retry", "3");
```

<import>

- 설정 xml 파일 안에서 다른 xml의 내용을 가져오고자 할 때

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <import resource="classpath:/config-service.xml" />
    <import resource="classpath:/config.xml" />

</beans>
```

자바 코드를 이용한 DI 설정

▣ 자바 코드를 이용한 DI 설정

- 많은 개발자들이 xml보다 자바 문법에 익숙함
- **Annotation** 메커니즘을 이용해 자바 코드로 설정을 생성

자바 코드를 이용한 DI 설정

- ▣ `@Configuration`

- 해당 클래스가 스프링 설정임을 나타냄

- ▣ `@Bean`

- 해당 메소드가 빈 객체를 만들어 냄을 의미

```
package kr.ac.dankook;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;
```

UserConfig.java

```
@Configuration  
public class UserConfig {  
    @Bean  
    public User user1() { 메소드 이름 = id  
        return new User("ace", 12345);  
    }  
    @Bean(name = "user2") name 값 = id  
    public User user() {  
        return new User("park", 20);  
    }  
}
```

자바 코드를 이용한 DI 설정

▣ 생성자

- 직접 return문에 생성자 호출하면서 매개변수 입력

▣ 속성 입력 방식

- 자바빈즈의 setPropertyName 직접 호출

```
@Bean  
public PasswordChangeService pwChangeService() {  
    return new PasswordChangeService(userRepository);  
}
```

```
@Bean  
public AuthFailLogger authFailLogger() {  
    AuthFailLogger logger = new AuthFailLogger();  
    logger.setThreshold(2);  
    return logger;  
}
```

자바 코드를 이용한 DI 설정

- ▣ 다른 객체 참조하기 (<ref>)
 - 해당 빈의 어노테이션(생성 메소드) 호출

```
@Configuration  
public class UserConfig {  
    @Bean // userRepository <- user1, user2  
    public UserRepository userRepository() {  
        UserRepository users = new UserRepository();  
        users.setUsers(Arrays.asList(user1(), user0()));  
        return users;  
    }  
}
```

자바 코드를 이용한 DI 설정

- ▣ <import>태그처럼 다른 설정 파일 불러들이기

```
@Configuration  
@Import(UserConfig.class)  
public class AppConfig {
```

자바 코드를 이용한 DI 설정 사용하기

- Main코드의 ClassPathXmlApplicationContext 대신 AnnotationConfigApplicationContext 사용

```
ApplicationContext contex = new ApplicationContext(AppConfig.class);
```

@Configuration
클래스

```
ApplicationContext contex = new ApplicationContext("di.conf");
```

@Configuration
클래스들이 묶인
패키지

xml 설정파일에 자바 설정 추가하기

- context:annotation-config 태그 추가 후
- @configuration 클래스를 빈으로 등록

```
<context:annotation-config />  
  
<bean class="di.UserConfig" />
```

자바 설정에 xml 설정파일 불러오기

- ▣ @ImportResource 어노테이션 사용

```
@Configuration  
@ImportResource("classpath:config-sensor.xml")  
public class ConfigWithXmlImport {
```

Annotation으로 자동주입

□ Annotation으로 DI 설정

- @Component & @Autowired 사용

```
@Component("userTest")
```

```
public class UserTest {  
    @Autowired  
    private User user; // userTest <- user  
    public void set(String name, int password) {  
        user.setName(name);  
        user.setPassword(password);  
    }  
    public void test() {  
        System.out.println(user);  
    }  
}
```

UserTest.java

userTest에 user을
넣어줌

Annotation으로 자동주입

- Main코드에서 AnnotationConfigApplicationContext 사용

```
ApplicationContext context = new ApplicationContext();
context.scan("kr.ac.dankook");
context.refresh();
UserTest userTest = (UserTest)context.getBean("userTest");
userTest.set("Cho", 123);
userTest.test();
userTest.set("Choi", 1234);
userTest.test();
```

DI using XML Configuration

- ▣ 프로젝트 src/main/resources/**applicationContext.xml**
다음 코드 추가

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xsi:schemaLocation="  
           http://www.springframework.org/schema/beans  
           http://www.springframework.org/schema/beans/spring-beans.xsd">  
  
<bean id="service1" class="kr.ac.dankook.ServiceE" />  
<bean id="client1" class="kr.ac.dankook.ClientA">  
    <constructor-arg ref="service1" />  
</bean>  
  
<import resource="classpath:/config.xml" />  
</beans>
```

applicationContext.xml

ClientA client1에 ServiceE
객체 service1을 넣어줌

config.xml 포함

DI using XML Configuration

▣ 메인자바 테스트

```
// Dependency Injection using XML (client1 <- service1)
ApplicationContext appContext =
new ClassPathXmlApplicationContext("classpath:applicationContext.xml");
Client client1 = (Client) appContext.getBean("client1");
client1.doSomething();
```

DI using Java Config (AppConfig.java)

- src/main/java/kr/ac/dankook/AppConfig.java 다음 추가

```
@Configuration  
@Import(UserConfig.class)  
public class AppConfig {  
    @Bean("client3") // client3 <- service3  
    public Client getClient3(Service service3) {  
        return new ClientC(service3);  
    }  
    @Bean("client4") // client4 <- service3  
    public Client getClient4(Service service3) {  
        return new ClientD(service3);  
    }  
    @Bean("service3")  
    public Service getService3() {  
        return new ServiceG();  
    }  
}
```

UserConfig.java 포함

AppConfig.java

ClientC client3에
ServiceG 객체 service3을
넣어줌

ClientD client4에
ServiceG 객체 service3을
넣어줌

DI using Java Config

▣ 메인자바 테스트

```
// Dependency Injection using Java Config (AppConfig.java) (client3 <- service3)
ApplicationContext appContext =
    new AnnotationConfigApplicationContext(AppConfig.class);
Client client3 = (Client) appContext.getBean("client3");
client3.doSomething();

Client client4 = (Client) appContext.getBean("client4");
client4.doSomething();

Service service = (Service) appContext.getBean("service3");
System.out.println(service.getInfo());
```

DI using Annotation

- ▣ @Component & @Autowired 추가

```
@Component("client2")
```

```
public class ClientB implements Client {  
    @Autowired  
    private Service service2; // client2 <- service2  
    public void doSomething() {  
        String info = service2.getInfo();  
        System.out.println("ClientB: " + info + ", " + info);  
    }  
}
```

ClientB.java

ClientB client2에
ServiceF service2을
넣어줌

```
@Component("service2")
```

```
public class ServiceF implements Service {  
    public String getInfo() {  
        return "ServiceF's Info";  
    }  
}
```

ServiceF.java

DI using Annotation

▣ 메인자바 테스트

```
// Dependency Injection using Annotation (client2 <- service2)
AnnotationConfigApplicationContext appContext2 =
    new AnnotationConfigApplicationContext();
appContext2.scan("kr.ac.dankook");
appContext2.refresh();
Client client2 = (Client) appContext2.getBean("client2");
client2.doSomething();
```