

# Spring DI, IoC

---

558280-1  
2025년 봄학기  
4/9/2025  
박경신

# Overview

---

- ❑ IoC (Inversion of Control)
- ❑ Spring Container - BeanFactory, ApplicationContext
- ❑ Singleton Pattern - Bean
- ❑ DI (Dependency Injection)
- ❑ @ComponentScan
- ❑ @Component - @Controller, @Service, @Repository
- ❑ AppConfig - @Configuration
- ❑ @Bean
- ❑ @Autowired

# IoC (Inversion of Control)

---

- 제어의 역전 IoC (Inversion of Control)는 프로그램의 제어를 다른 대상에게 맡기는 것임.
- 스프링에서는 스프링 컨테이너가 Bean 생성, 관리, 의존관계 주입과 같은 작업을 담당함.
  - Bean 생성부터 소멸까지의 생명주기 관리를 개발자가 아닌 컨테이너가 대신 해줌.
- 스프링 컨테이너는 **ApplicationContext**이며, IoC (Inversion of Control) 컨테이너 혹은 DI (Dependency Injection) 컨테이너라고도 부름.

# Spring Container

## □ 스프링 컨테이너

### ■ BeanFactory

- 스프링 컨테이너의 최상위 인터페이스이며, 스프링 빈을 관리하고 조회하는 순수한 DI 역할을 담당함.
- BeanFactory 계열의 인터페이스만 구현한 클래스는 단순히 컨테이너에서 객체를 생성하고 DI를 처리하는 기능만 제공함.
- Factory Design Pattern을 구현한 것으로 BeanFactory는 빈을 생성하고 분배하는 책임을 지는 클래스.
- Bean을 조회할 수 있는 **getBean() 메소드**가 정의되어 있음.

### ■ ApplicationContext

- BeanFactory + 앱 개발에 필요한 편리한 부가기능 추가

# Singleton Pattern

---

- 스프링 컨테이너는 객체 인스턴스를 **Singleton**으로 관리함
  - 객체 (Bean)을 스프링 컨테이너를 등록하고, 빈 조회 요청 시 새로 생성하지 않고 스프링 컨테이너에서 빈을 찾아서 반환함
  - 스프링은 **@Conguration**이 붙은 클래스를 설정 정보로 사용함
- Singleton pattern 은 객체를 하나만 생성하게 해서, 하나의 객체에서 처리하게 함.

# Singleton Pattern

---

```
public class Singleton {
    // static field containing its only instance
    private static Singleton uniqueInstance;

    // private default constructor
    private Singleton() { }

    // static factory method for obtaining the
instance
    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
} // classical implementation
Singleton singleton = Singleton.getInstance();
```

# 스프링 컨테이너 생성, 빈 등록, 의존관계 설정

- 스프링은 스프링 컨테이너를 생성하고,
- 스프링 컨테이너는 Key=빈 이름, Value=빈 객체 형태로 빈을 저장,
  - Key (빈 이름)은 메서드 이름으로 사용, 실제 반환하는 객체를 Value (빈 객체)에 저장함
- 스프링 컨테이너는 설정 정보 (Configuration)을 참고해서 의존 관계 주입 (Dependency Injection) 함.

# DI (Dependency Injection)

---

- 의존 주입 DI (Dependency Injection)는 객체 간의 의존을 외부에서 주입하는 것을 말함.
  - 예를 들어, 회원 가입을 위한 MemberService 클래스가 MemberRepository의 save()를 필요로 한다면 MemberService 클래스는 MemberRepository 클래스에 의존한다고 볼 수 있음.
- DI (Dependency Injection) 는 IoC의 대표적인 기능으로 의존하는 객체를 직접 생성하는 대신, 의존 객체를 전달받는 방식을 사용함.



# 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 줄 짜리 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 타입 전달

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

- 생성자 호출시 Encryptor 타입의 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 setName()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

# Spring 빈 등록

## □ @ComponentScan, @Component 를 통한 자동 빈 등록

- @ComponentScan은 @Component가 붙은 객체를 찾아 자동으로 빈 등록하는 방법이 있음.
- **@Controller, @Service, @Repository**는 모두 @Component를 포함하고 있으며, 해당 어노테이션으로 등록된 클래스들은 스프링 컨테이너에 의해 자동으로 생성되어 스프링 빈으로 등록됨.
- 일반적으로 Bean을 생성하기 위해서는 @Component가 더 많이 사용됨

# Spring Bean

## □ Spring Bean

- Spring Bean Container 가 관리하는 순수 자바 객체(POJO)
- Spring Bean Container 는 설정을 읽은 후 Spring Bean 객체를 생성하고, 서로 의존성 있는 Bean 객체들을 주입(DI)함
- Spring Bean Container 는 Bean 생성부터 주입, 소멸하는 전체 과정 (즉, Spring Bean Life Cycle) 을 관리함

## □ Spring Bean 설정

- @ComponentScan으로 스캔되어 Spring Bean으로 생성
  - @Configuration, @Component, @Controller, @Service, @Repository
- Auto-configuration
  - @EnableAutoConfiguration은 프로젝트의 외부 라이브러리들의 빈들을 등록함. @Configuration 또는 @Conditional로 시작하는 클래스를 대상으로 함.
- User-defined Configuration
  - Spring Bean Container 가 로딩하는 설정(@Configuration) 파일에 @Bean 을 사용하여 정의

# Spring 빈 등록

## □ @Configuration, @Bean을 통한 직접 빈 등록

- 스프링이 뜰 때에 스프링은 자동으로 @Configuration 이 붙은 클래스(예를 들어, AppConfig)를 찾아서 구성 정보로 사용함
  - 클래스에 @Configuration 주석을 달면 해당 클래스가 **JavaConfig**에서 Bean 정의의 소스로 사용된다는 것을 나타냄
  - @Configuration 주석이 달린 클래스를 하나만 사용할 수도 있고 **여러 개 사용할 수도 있음**
  - @Configuration은 XML의 <beans/> 요소와 동등함
- @Configuration 구성 정보에 @Bean을 통해 스프링 컨테이너에 직접 빈을 등록하고 의존 관계 주입을 처리할 수 있음
- 빈 객체로 등록하고 싶은 **메서드 위에 @Bean 추가**

# @Bean

---

## □ @Bean [method]

- @Configuration 설정된 클래스의 메소드에서 사용 가능
- 메소드의 리턴 객체가 스프링 빈 객체임을 선언함
- **빈의 이름**은 기본적으로 **메소드의 이름**
- @Bean(name="name")으로 이름 변경 가능
- @Scope를 통해 객체 생성을 조정할 수 있음

# @Configuration, @Bean

- ▣ @Configuration은 @SpringBootApplication 내에서도 제공됨

```
@SpringBootApplication
//@ComponentScan
//@EnableAutoConfiguration
//@SpringBootConfiguration // ==> wraps @Configuration
public class SpringBootHelloControllerApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBootHelloControllerApplication.class, args);
    }
    @Bean
    public Hello hello() {
        return new Hello("Application @Bean says Hello");
    }
}
```

# @Configuration, @Bean

- ❑ @Configuration 해당 클래스가 스프링 설정임을 나타냄
- ❑ @Bean 해당 메소드가 빈(Bean) 객체를 만들어 냄을 의미

## @Configuration

```
public class CoffeeConfig {  
    // Configuring origin for Coffee  
    @Bean // 메소드 이름이 빈 객체 이름임 origin  
    public Origin origin() {  
        return new Origin("Costa Rica");  
    }  
    // Configuring roast for Coffee  
    @Bean // 메소드 이름이 빈 객체 이름임 roast  
    public Roast roast() {  
        return new Roast(2);  
    }  
}
```

# Bean Visibility

## □ AppConfig.java

@Configuration

public abstract class VisibilityConfiguration {

    @Bean

    public Bean publicBean() {

        Bean bean = new Bean();

        bean.setDependency(hiddenBean());

        return bean;

    }

    @Bean

    protected HiddenBean hiddenBean() {

        return new Bean("protected bean");

    }

    @Bean

    HiddenBean secretBean() {

        Bean bean = new Bean("package-private bean");

        // hidden beans can access beans defined in the 'owning' context

        bean.setDependency(outsideBean());

    }

    @ExternalBean

    public abstract Bean outsideBean()

}

<https://docs.spring.io/spring-javaconfig/docs/1.0.0.m3/reference/html/creating-bean-definitions.html>



# Bean Visibility

## □ 다음 appConfig.xml과 일치함

```
<beans>
  <!-- the configuration above -->
  <bean class="my.java.config.VisibilityConfiguration"/>

  <!-- Java Configuration post processor -->
  <bean
class="org.springframework.config.java.process.ConfigurationPostProcessor"/>

  <bean id="mainBean" class="my.company.Bean">
    <!-- this will work -->
    <property name="dependency" ref="publicBean"/>
    <!-- this will *not* work -->
    <property name="anotherDependency" ref="hiddenBean"/>
  </bean>
</beans>
```

# Annotation 기반 의존 관계 주입

- @Autowired, @Qualifier를 통한 DI(의존성 주입)
  - Annotation 기반 Dependency Injection (의존성 주입)은 의존성 대상 객체를 주입받을 클래스 내부의 객체를 저장할 변수에 @Autowired 와 @Qualifier 애터네이션을 조합하여 정의함
  - Spring Bean Container 는 두 annotation의 속성 값을 파악하여 적절한 의존성 대상 객체를 찾아 주입함
    - 의존성 주입 시 의존성을 주입 받을 객체와 의존성 대상 객체 모두 Spring Bean 객체이어야 함

# @Autowired 의존관계 주입

- @Autowired를 통한 DI(의존성 주입)
  - 스프링 컨테이너에 빈들을 모두 등록한 후에 DI(의존성 주입)
  - @Autowired는 기존에 XML에 <property>, <constructor-arg>를 통해 **DI** 해오던 방식을 **자동**으로 해주는 Annotation
  - @Autowired 의존성 주입 방법은 다음 3가지 방식으로 정의
    - 1. **Field 방식 – 가장 기본적인 방식**
    - 2. **Setter 방식**
    - 3. **Constructor 방식 – 추천하는 방식**

# @Autowired Field 방식

- 필드(field) 기반 DI는 가장 기본적인 형식이며 @Autowired를 이용하여 간단하게 의존주입(객체생성)
  - 필드에 final 를 사용할 수 없음 – 불변성(immutable) 허용하지 않음
  - 사용이 간단하여 의존성 주입 대상이 많아질 수 있음
  - 의존관계가 가려짐

@Component

@Data

@NoArgsConstructor

@AllArgsConstructor

public class Car {

// field-based dependency injection (cannot use final)

@Autowired

private Engine engine; // 스프링이 engine 객체를 생성해서 주입

// field-based dependency injection (cannot use final)

@Autowired

private Transmission transmission; // 스프링이 transmission 객체를 생성해서

주입

}

# @Autowired Field 방식

---

```
@Configuration
public class CarConfig {
    // Configuring Engine for Car
    @Bean
    public Engine engine() {
        return new Engine("VR6", 6);
    }
    // Configuring Transmission for Car
    @Bean
    public Transmission transmission() {
        return new Transmission("Dual Clutch");
    }
}
```

# @Autowired Setter 방식

- ❑ @Autowired를 setter 메서드에 등록해서 사용함
  - 필드에 final 를 사용할 수 없음
  - 선택적이고 변화 가능한 의존관계에 사용함

@Component

@Data

@NoArgsConstructor

```
public class Car {  
    private Engine engine;  
    private Transmission transmission;  
    // setter-based dependency injection  
    @Autowired // setter 매개변수에 스프링이 자동으로 객체를 생성해서 주입  
    public void setEngine(Engine engine) {  
        this.engine = engine;  
    }  
    // setter-based dependency injection  
    @Autowired // setter 매개변수에 스프링이 자동으로 객체를 생성해서 주입  
    public void setTransmission(Transmission transmission) {  
        this.transmission = transmission;  
    }  
}
```

# @Autowired Setter 방식

## □ 속성 입력 방식

### ■ 자바 빈즈의 setter 직접 호출

@Configuration

```
public class CarConfig {
```

```
    @Bean
```

```
    public Engine engine() {
```

```
        return new Engine("v5", 3);
```

```
    }
```

```
    @Bean
```

```
    public Transmission transmission() {
```

```
        return new Transmission("automatic");
```

```
    }
```

```
    @Bean
```

```
    public Car car() {
```

```
        Car car = new Car();
```

```
        car.setEngine(engine()); // setter DI
```

```
        car.setTransmission(transmission()); // setter DI
```

```
        return car;
```

```
    }
```

```
}
```

# @Autowired Constructor 방식

- 생성자(constructor) 기반 DI는 생성자에 @Autowired를 사용하여 의존주입(객체생성)
  - **final 선언이 가능**하므로 객체를 재할당하는 것을 방지할 수 있음
  - 생성자가 호출될 때 딱 한 번 주입됨. 순환 참조가 방지됨
  - 생성자가 하나일 경우 @Autowired를 생략할 수 있음
  - 필수 의존성 주입에 유용함

@Component

@Data

```
public class Car {  
    private final Engine engine;  
    private final Transmission transmission;  
    // constructor-based dependency injection  
    @Autowired // 스프링이 생성자 매개변수에 객체를 생성해서 주입  
    public Car(Engine engine, Transmission transmission) {  
        this.engine = engine;  
        this.transmission = transmission;  
    }  
}
```



# @Autowired Constructor 방식

## □ 생성자 방식

- 직접 생성자 호출하면서 매개변수 입력

@Configuration

```
public class CarConfig {  
    // Configuring Engine for Car  
    @Bean  
    public Engine engine() {  
        return new Engine("v8", 5); // constructor DI  
    }  
  
    // Configuring Transmission for Car  
    @Bean  
    public Transmission transmission() {  
        return new Transmission("sliding"); // constructor DI  
    }  
}
```

# @RequiredArgsConstructor

- ❑ @RequiredArgsConstructor
  - Lombok을 사용하여 간단한 방법으로 생성자 기반 DI 가능
  - **@RequiredArgsConstructor**는 **final 필드**나, **@NonNull** 이 붙은 **필드**에 대해 생성자를 자동 생성
  - 새로운 필드를 추가할 때 다시 생성자를 만들어서 관리해야하는 번거로움을 없애줌

@Component

@Data

**@RequiredArgsConstructor**

```
public class Car {  
    private final Engine engine;  
    private final Transmission transmission;  
}
```

# list, map, set

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

@Repository

@Data

```
public class ChocolateRepository {  
    @Autowired  
    private List<String> nameList;  
    public void printNameList() {  
        System.out.println(Arrays.toString(nameList.toArray()));  
    }  
}
```

@Configuration

```
public class ChocolateConfig {  
    // Configuring nameList for ChocolateRepository  
    @Bean  
    public List<String> nameList() {  
        return Arrays.asList("Lindt", "Godiva", "Ghirardelli");  
    }  
}
```

# list, map, set

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

@Repository

@Data

```
public class ChocolateRepository {  
    @Autowired  
    private Set<String> nameSet;  
    public void printNameSet() {  
        System.out.println(Arrays.toString(nameSet.toArray()));  
    }  
}
```

@Configuration

```
public class ChocolateConfig {  
    // Configuring nameSet for ChocolateRepository  
    @Bean  
    public Set<String> nameSet() {  
        return new HashSet<>(Arrays.asList("Milk Chocolate", "Dark Chocolate",  
"Pave Chocolate"));    }  
}
```

# list, map, set

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

@Repository

@Data

public class ChocolateRepository {

@Autowired

private Map<Integer, String> nameMap;

public void printNameMap() {

nameMap.entrySet().stream().forEach(e-> System.out.println(e));

}

}

# list, map, set

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

@Configuration

```
public class ChocolateConfig {
```

```
    // Configuring nameMap for ChocolateRepository
```

@Bean

```
    public Map<Integer, String> nameMap() {
```

```
        Map<Integer, String> nameMap = new HashMap<>();
```

```
        nameMap.put(1, "M&M's");
```

```
        nameMap.put(2, "Reese's");
```

```
        nameMap.put(3, "Hershey");
```

```
        return nameMap;
```

```
    }
```

```
}
```

# @Qualifier

---

- ❑ @Qualifier는 메서드, 파라미터, 필드, 애너테이션에 정의 가능함
  - value 속성값은 Spring Bean 의 이름이 됨
- ❑ @Qualifier는 사용할 의존 객체를 선택할 수 있게 해줌
  - @Qualifier는 클래스 타입은 같지만 이름이 다른 여러 Spring Bean 이 있을 경우 이 중 정의된 이름의 Spring Bean 을 주입 받기 위해 사용함
  - @Qualifier가 아닌 @Primary로도 처리 가능함

# @Qualifier

## □ 다음 두 단계를 설정해주어야 함

1. Configuration 에서 빈의 한정자(@Qualifier) 이름을 설정
2. @Autowired 주입 대상(field, setter, constructor)에 @Qualifier를 설정 (이때 Configuration 에서 설정한 한정자 이름을 사용)
  - 여기서 주의할 점은 한정자 이름이 일치하지 않을 시 객체가 존재하지 않아 Exception 발생

## @Repository

### @Data

```
public class ChocolateRepository {  
    @Autowired  
    @Qualifier("chocolateList") // 주입된 bean 이름을 chocolateList로 설정  
    private List<Chocolate> chocolateList;  
    public void printChocolateList() {  
        chocolateList.forEach(System.out::println);  
    }  
}
```



# @Qualifier

## @Configuration

```
public class ChocolateConfig {
```

```
    @Bean
```

```
    @Qualifier("chocolateList")
```

```
    public List<Chocolate> chocolateList() {
```

```
        List<Chocolate> chocolateList = new ArrayList<Chocolate>();
```

```
        chocolateList.add(new Chocolate(new Brand("Lindt", "Swiss"), new Type("Milk  
Chocolate")));
```

```
        chocolateList.add(new Chocolate(new Brand("Godiva", "Belgium"), new  
Type("Dark Chocolate")));
```

```
        chocolateList.add(new Chocolate(new Brand("Ghirardelli", "USA"), new  
Type("Dark Chocolate")));
```

```
        chocolateList.add(new Chocolate(new Brand("Ferrero Rocher", "Italy"), new  
Type("Nutella Chocolate")));
```

```
        chocolateList.add(new Chocolate(new Brand("Royce", "Japan"), new  
Type("Pave Chocolate")));
```

```
        return chocolateList;
```

```
    }
```

```
}
```

# LoginUser 자바빈

---

## ▣ LoginUser 클래스

@Component // Spring Component

@Data // Lombok @Getter/@Setter/@ToString

@NoArgsConstructor // Lombok Default Constructor

@AllArgsConstructor // Lombok Parameterized Constructor

```
public class LoginUser {  
    private String loginId;  
    private int password;  
}
```

# LoginUserRepository 자바빈

## ▣ LoginUserRepository 클래스

@Repository // Spring Component

@Data // Lombok @Getter/@Setter/@ToString

```
public class LoginUserRepository {
```

```
    List<LoginUser> userList;
```

```
    public void printList() {
```

```
        System.out.println("Repository printList userList=" + this.userList);
```

```
    }
```

```
}
```

# LoginUserService 자바빈

## ▣ LoginUserService 클래스

@Service // Spring Component

@NoArgsConstructor

@Data

```
public class LoginUserService {  
    private LoginUserRepository userRepository;  
    // setter-based dependency injection  
    @Autowired  
    public void setUserRepository(LoginUserRepository userRepository) {  
        this.userRepository = userRepository;  
    }  
    public List<LoginUser> findAll() {  
        return this.userRepository.getUserList();  
    }  
}
```

# LoginUserController 자바빈

## ▣ LoginUserController 클래스

@Controller // Spring Component

@RequestMapping("/user")

public class LoginUserController {

    private final LoginUserService userService;

    // constructor-based dependency injection

    @Autowired // 생성자가 1개만 있으면 생략 가능

    public LoginUserController(LoginUserService userService) {

        this.userService = userService;

    }

    @GetMapping("/list") // localhost:8080/di/user/list

    public String list(Model model) {

        model.addAttribute("userList",

        this.userService.getUserRepository().getUserList());

        return "userlist"; // templates/userlist.html

    }

}

# LoginUserConfig에서 @Bean 주입

## @Configuration

```
public class LoginUserConfig {
```

## @Bean

```
protected List<LoginUser> userList() {
```

```
    List<LoginUser> userList = new ArrayList<>();
```

```
    userList.add(new LoginUser("Kim", 12345));
```

```
    userList.add(new LoginUser("Lee", 6789));
```

```
    userList.add(new LoginUser("Park", 321));
```

```
    userList.add(new LoginUser("DIS", 98765));
```

```
    return userList;
```

```
}
```

## @Bean

```
public LoginUserRepository userRepository() {
```

```
    LoginUserRepository userRepository = new LoginUserRepository();
```

```
    userRepository.setUserList(userList());
```

```
    return userRepository;
```

```
}
```

```
}
```

# 메인에서 Bean 등록 확인 테스트

## ▣ Main코드에서 ApplicationContext의 getBean 사용

```
public static void main(String[] args) {  
    ConfigurableApplicationContext appContext =  
        SpringApplication.run(SpringBootDiApplication.class, args);  
    Car car = appContext.getBean(Car.class);  
    System.out.println("car=" + car);  
    Coffee coffee = appContext.getBean(Coffee.class);  
    coffee.print();  
    ChocolateRepository chocolateRepository =  
        appContext.getBean(ChocolateRepository.class);  
    chocolateRepository.printChocolate();  
    chocolateRepository.printNameList();  
    chocolateRepository.printNameSet();  
    chocolateRepository.printNameMap();  
    chocolateRepository.printChocolateList();  
}
```

# 같은 인터페이스를 구현한 Bean을 구분하여 의존성 주입

```
public interface Client {  
    void doSomething();  
}
```

```
@Component("client1") // 빈 이름 client1는 ClientA 객체  
public class ClientA implements Client {  
    @Autowired  
    private Service service1; // client1 <- service1  
    @Override  
    public void doSomething() {  
        System.out.println("ClientA: " + service1.getInfo());  
    }  
}
```



# 같은 인터페이스를 구현한 Bean을 구분하여 의존성 주입

```
public class ClientB implements Client {
    Service service;
    public ClientB(Service service) {
        this.service = service;
    }
    @Override
    public void doSomething() {
        System.out.println("ClientB: " + service.getInfo() + " " + service.getInfo());
    }
}

public class ClientC implements Client {
    Service service;
    public ClientC(Service service) {
        this.service = service;
    }
    @Override
    public void doSomething() {
        System.out.println("ClientC: " + service.getInfo() + " " + service.getInfo() + " " +
service.getInfo());
    }
}
```

# 같은 인터페이스를 구현한 Bean을 구분하여 의존성 주입

```
public interface Service {  
    String getInfo();  
}
```

```
@Component("service1") // 빈 이름 service1은 ServiceE 객체  
public class ServiceE implements Service {
```

```
    @Override  
    public String getInfo() {  
        return "ServiceE's Info";  
    }  
}
```

# 같은 인터페이스를 구현한 Bean을 구분하여 의존성 주입

`@Component("service2")` // 빈 이름 service2는 ServiceF 객체  
public class **ServiceF** implements **Service** {

```
    @Override
    public String getInfo() {
        return "ServiceF's ServiceF's Info";
    }
}
```

```
}
public class ServiceG implements Service {
```

```
    @Override
    public String getInfo() {
        return "ServiceG's ServiceG's ServiceG's Info";
    }
}
```

```
}
```

# 같은 인터페이스를 구현한 Bean을 구분하여 의존성 주입

@Configuration

```
public class AppConfig {
```

```
    @Bean("service3") // 빈 이름 service3는 ServiceG 객체
```

```
    public Service getService3() {  
        return new ServiceG();  
    }
```

```
    @Bean("client2") // client2 (ClientB) <- service2 (ServiceF)
```

```
    public Client getClient3(@Qualifier("service2") Service service2) {  
        return new ClientB(service2);  
    }
```

```
    @Bean("client3") // client3 (ClientC) <- service3 (ServiceG)
```

```
    public Client getClient4(@Qualifier("service3") Service service3) {  
        return new ClientC(service3);  
    }
```

```
}
```

```
}
```

# 같은 인터페이스를 구현한 Bean을 구분하여 의존성 주입

## ▣ 메인자바 테스트

```
public static void main(String[] args) {  
    ConfigurableApplicationContext appContext =  
        SpringApplication.run(SpringBootDiApplication.class, args);  
    Client client1 = (Client)appContext.getBean("client1");  
    client1.doSomething(); // client1 <- service1  
    Client client2 = (Client)appContext.getBean("client2");  
    client2.doSomething(); // client2 <- service2  
    Client client3 = (Client)appContext.getBean("client3");  
    client3.doSomething(); // client3 <- service3  
}
```

# 같은 인터페이스를 구현한 Bean을 구분하여 의존성 주입

## □ 실행 결과

ClientA: ServiceE's Info

ClientB: ServiceF's ServiceF's Info ServiceF's ServiceF's Info

ClientC: ServiceG's ServiceG's ServiceG's Info ServiceG's

ServiceG's ServiceG's Info ServiceG's ServiceG's ServiceG's Info