Spring 5

1. Spring Framework
2. Spring Boot
3. Spring MVC
4. Spring Data - JPA

1

Spring Framework

1. Spring 개요
2. 객체지향과 Spring
3. Spring IoC/DI

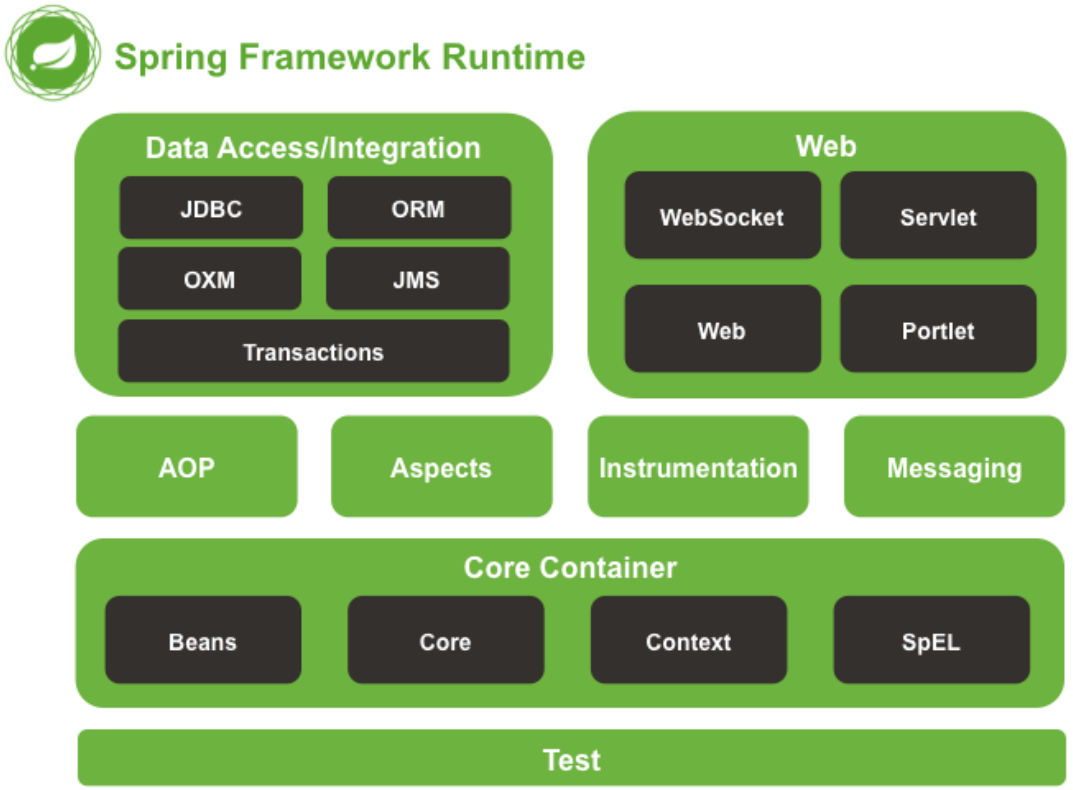
1.1. Spring 개요[1/2]

- 스프링(Spring)은 자바 언어로 개발된 엔터프라이즈 애플리케이션 개발을 단순화하고 확장성을 높이기 위한 강력한 도구로 폭넓게 사용되며, 자바 기반의 애플리케이션 개발에 필수적인 프레임워크 중 하나이다
- 스프링은 강력한 기능과 모듈화된 아키텍처를 제공하여 개발자가 애플리케이션을 보다 쉽게 구축하고 관리할 수 있도록 도와준다

• Spring의 특징

- IoC(Inversion of Control; 제어 역행)
  - 컨트롤의 제어권이 사용자가 아닌 스프링 프레임워크에 있어 필요에 따라 스프링이 상용자의 코드를 호출한다.
- DI(Dependency Injection; 의존성 주입)
  - 각각의 계층이나 서비스들 간 의존성이 존재할 경우 스프링 프레임워크가 서로 연결을 해준다.
- AOP(Aspect-Oriented Programming; 관점 지향)
  - 스프링은 AOP를 지원하여 애플리케이션의 핵심 비지니스 로직과 부가적인 기능(로깅, 트랜잭션 관리 등)을 분리하여 모듈화할 수 있다
- MVC 패턴 웹개발 지원 :
  - 웹 프로그래밍 개발 시 MVC(Model-View-Controller) 패턴을 사용한다

1.1. Spring 개요[2/2]



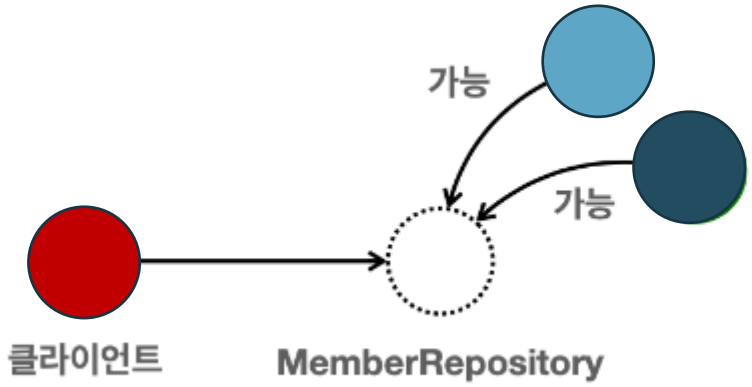
- 핵심 기술
  - ✓ 스프링 DI 컨테이너, AOP, 이벤트, 기타
- 웹 기술
  - ✓ 스프링 MVC, 스프링 WebFlux
- 데이터 접근 기술
  - ✓ 트랜잭션, JDBC, ORM 지원, XML 지원
- 기술 통합
  - ✓ 캐시, 이메일, 원격접근, 스케줄링
- 테스트
  - ✓ 스프링 기반 테스트 지원

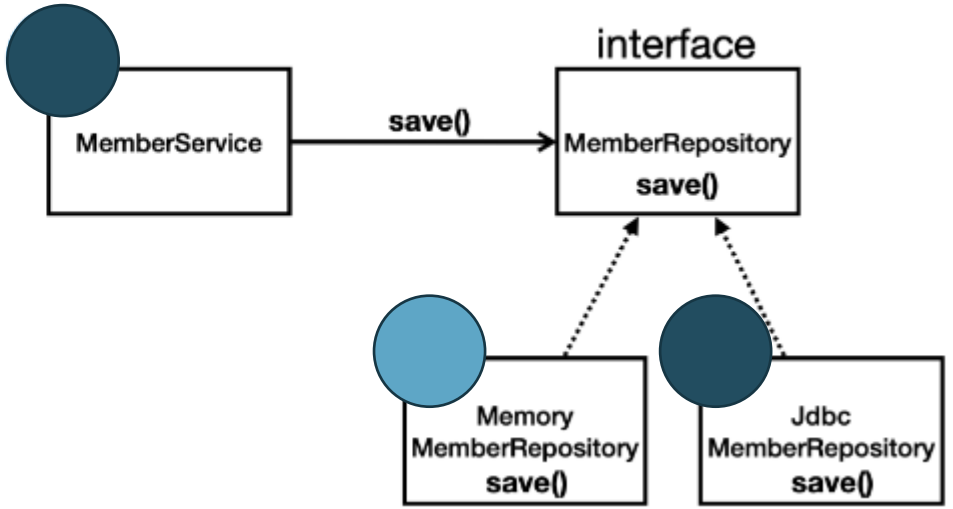
1.2. 객체지향과 Spring[1/2] - 객체지향 설계의 원칙

- SRP: 단일 책임 원칙 (Single Responsibility Principle)
  - ✓ 하나의 클래스는 하나의 책임만 가진다.
- OCP: 개방-폐쇄 원칙 (Open/Closed Principle)
  - ✓ 소프트웨어 요소는 확장에는 열려있지만 변경에는 닫혀있어야 한다.
- LSP: 리스코프 치환 원칙 (Liskov Substitution Principle)
  - ✓ 객체는 프로그램의 정확성을 깨지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야 한다.
- ISP: 인터페이스 분리 원칙 (Interface Segregation Principle)
  - ✓ 특정 클라이언트를 위한 인터페이스 여러 개가 범용 인터페이스 하나보다 낫다.
- DIP: 의존관계 역전 원칙 (Dependency Inversion Principle)
  - ✓ 구체적인 것이 아니라 추상적인 것에 의존해야 한다.
  - ✓ 구현 클래스에 의존하지 않고 인터페이스에 의존하라는 의미

객체지향과 Spring [2/2]

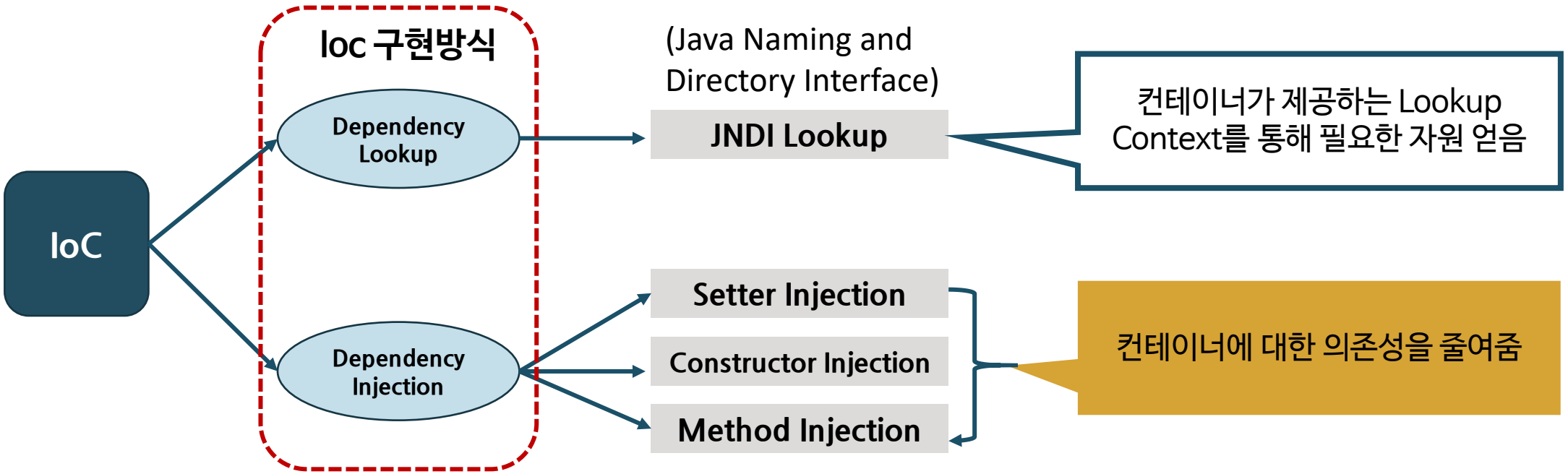
- 스프링의 IoC, DI는 다형성을 활용해서 역할과 구현을 편리하게 다룰 수 있도록 지원한다.
- 스프링을 사용하면 마치 레고를 조립하듯 구현을 편리하게 변경할 수 있다.
- 스프링은 DI와 DI 컨테이너로 다형성, OCP, DIP를 가능하게 지원해준다.





1.3. Spring IoC[1/10] - 개요

- IoC(Inversion Of Control)는 통제 방향의 변경을 의미한다
- 기존의 개발방식은 객체 결정 및 생성 -> 의존성 객체 생성 -> 객채 내의 메소드 호출 하는 작업을 사용자가 제어하는 구조였으나 IoC에서는 이 흐름의 구조를 바꾸어 제어의 흐름을 프레임워크가 통제하며 개발자는 내 프로그램에서 이벤트핸들러만 구현하는 방식이다
- 객체간 결합도가 높으면 해당 클래스가 변경될 때 결합된 다른 클래스도 수정해야할 가능성이 높아 향후 프로젝트를 수정하거나 확장시 발생할 문제점을 차단할 수 있도록 IoC는 객체의 생성을 컨테이너에게 위임하여 객체간의 결합도를 낮춘다



1.3. Spring IoC[2/10] - Container(1)

- Container는 DI를 사용하여 응용 프로그램을 구성하는 bean 객체를 관리한다
  - ✓ 객체(bean)를 생성하고
  - ✓ 객체들을 함께 묶고
  - ✓ 객체들을 구성하고
  - ✓ 객체들의 전체 수명주기(lifecycle)를 관리한다

- **컨테이너 메타데이터 설정 방법**

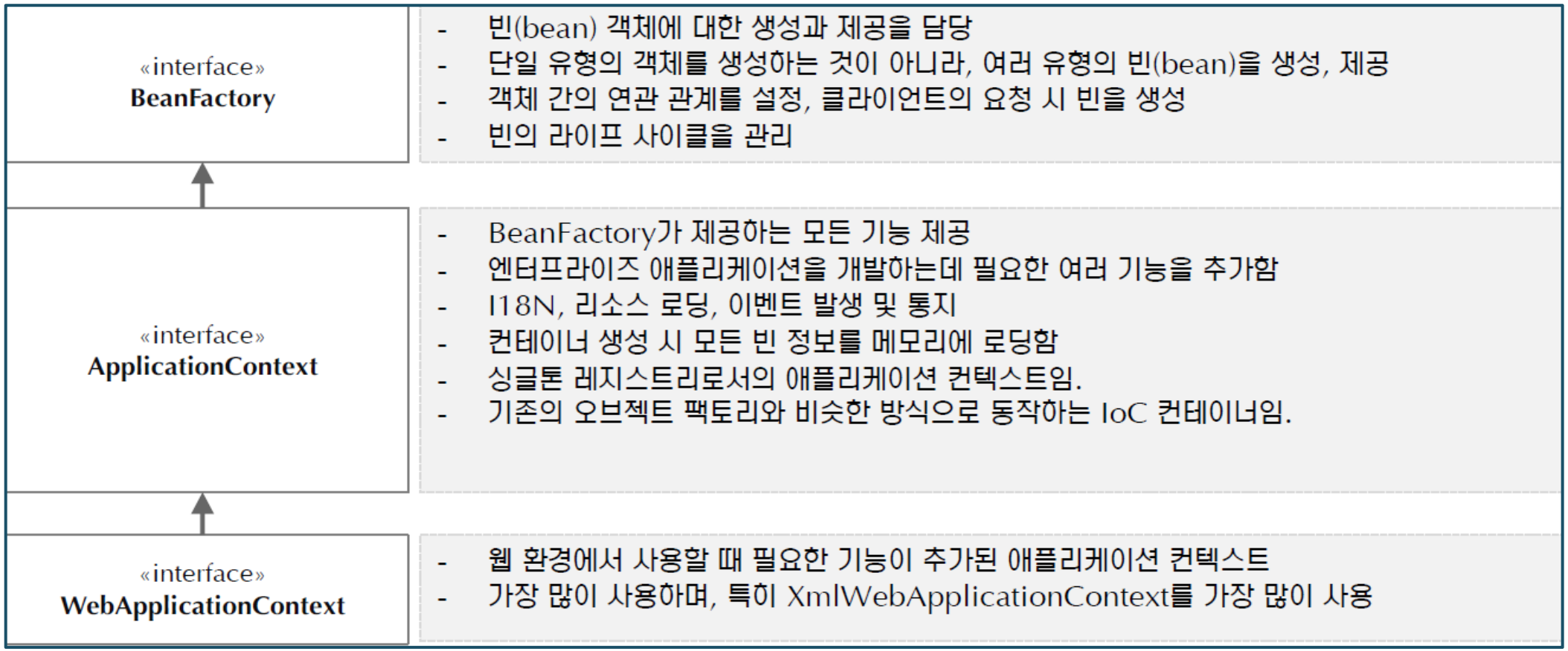
1. XML
   - ✓ 빈 객체 정의 (Bean Definition)
   - ✓ 의존성 주입 (Dependency Injection)
2. Java Annotations
3. Java Code

- **컨테이너 유형**

1. BeanFactory
   - ✓ 빈 객체 정의 (Bean Definition)
   - ✓ 의존성 주입 (Dependency Injection)
2. ApplicationContext

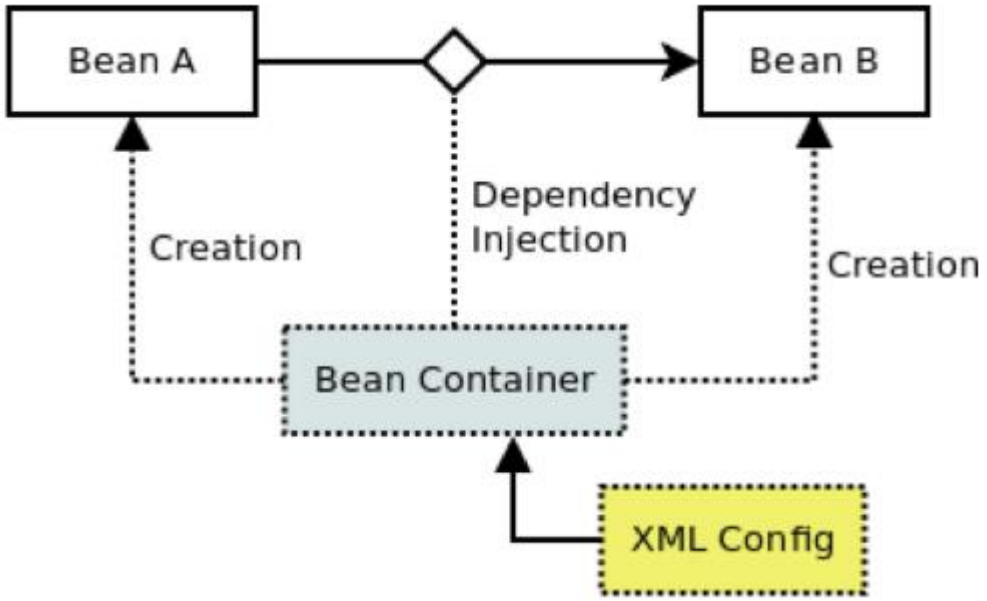
1.3. Spring IoC[3/10] - Container(2)

- 객체의 생성과 관계설정, 사용, 제거등의 작업을 독립된 컨테이너가 담당한다
- 프로그래머는 Bean으로 등록된 객체를 Spring IoC Container로부터 DI를 통해 가져와 사용할 수 있다.
- 스프링에서 IoC를 담당하는 컨테이너에는 빈 팩토리와 애플리케이션 컨텍스트가 있다
- 스프링은 별도의 설정이 없으면 컨테이너가 관리하는 빈 객체를 싱글톤(Singleton)으로 생성한다
  - 객체가 딱 하나만 생성되고 공유되도록 설계하는 것이 싱글톤 패턴이다



1.3. Spring IoC[4/10] - DI(Dependency Injection)

- 의존성 주입 (DI, Dependency Injection)
- 어떤 객체(B)를 사용하는 주체(A)가 객체(B)를 직접 생성하는게 아니라 객체를 외부(Spring)에서 생성해서 사용하려는 주체 객체(A)에 주입시켜주는 방식
- DI를 통해 외부에서 객체를 생성하는 시점에 참조하는 객체에게 의존관계를 제공한다
- 객체가 인터페이스만 알고 있으므로 loose coupling이 가능하다



1.3. Spring IoC[5/10] - DI 적용 절차

1. 빈(Bean) 클래스 생성

✓ 스프링 부트에서 의존성 주입을 사용하려면 먼저 빈 클래스를 생성해야 한다

✓ @~~어노테이션을 사용하여 해당 클래스를 스프링 빈으로 등록

```
import org.springframework.stereotype.Component;

@Component
public class MyService {
    public String sayHello() {
        return "Hello, Spring Boot!";
    }
}
```

2. 의존성 주입(@Autowired)

- 객체를 주입하기 위해 사용하는 스프링의 어노테이션이다.
- 객체를 주입하는 방식에는 @Autowired 외에 Setter 또는 생성자를 사용하는 방식이 있다.
- 순환참조 문제와 같은 이유로 @Autowired 보다는 생성자를 통한 객체 주입방식이 권장된다
- 단, 테스트 코드의 경우에는 생성자를 통한 객체의 주입이 불가능하므로 테스트 코드 작성시에만 @Autowired를 사용

1. Field - DI

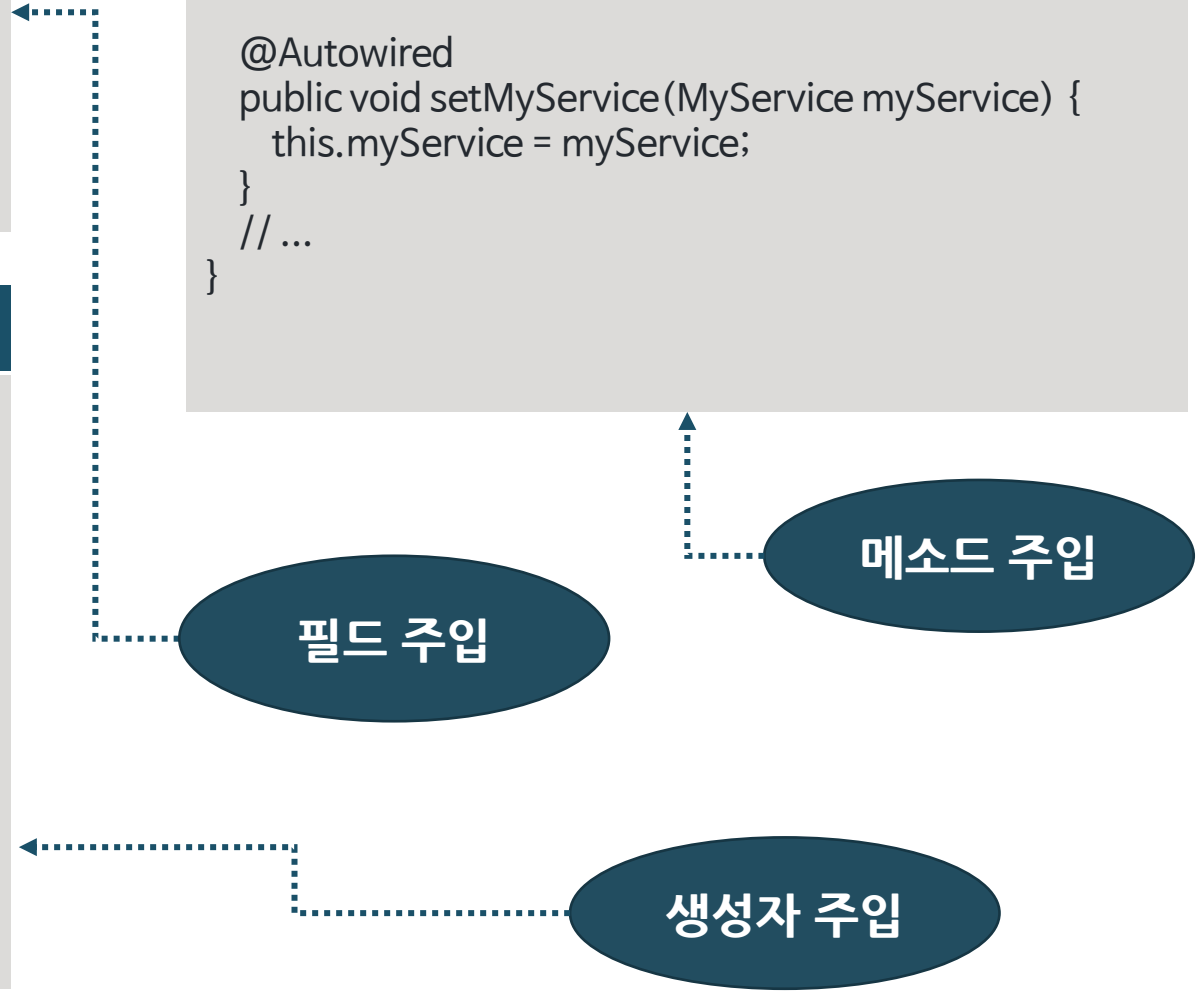
```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;

@Controller
public class MyController {
   @Autowired
   private MyService myService;
   // ...
}
```

2. Constructor - DI

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;

@Controller
public class MyController {
   private final MyService myService;

   @Autowired
   public MyController(MyService myService) {
      this.myService = myService;
   }
   // ...
}
```

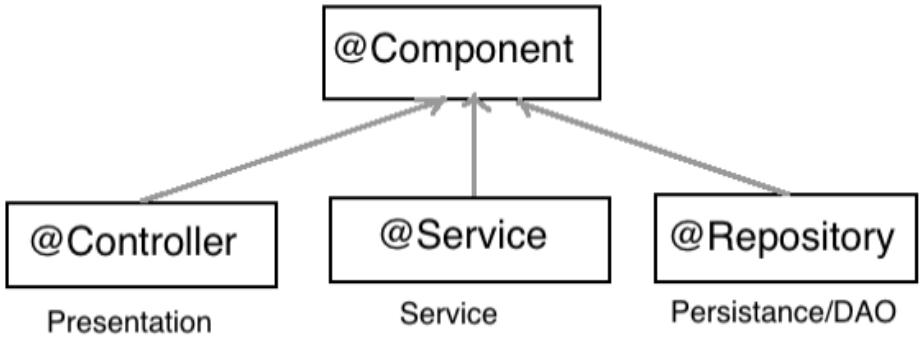
3. Method - DI

```
@Controller
public class MyController {
   private MyService myService;

   @Autowired
   public void setMyService(MyService myService) {
      this.myService = myService;
   }
   // ...
}
```



1.3. Spring IoC[7/10] - 어노테이션을 활용한 Bean 등록

- 빈을 만들기 위한 설정 메타정보는 파일이나 어노테이션 같은 리소스로부터 전용 리더를 통해 BeanDefinition 타입의 오브젝트로 변환된다.
- 스프링은 XML설정 파일과 어노테이션 설정 및 자바소스를 통해 메타정보를 설정할 수 있다
- 현재는 어노테이션을 이용하여 빈을 등록하고, DI 할 수 있도록 개선되었다



어노테이션을 붙이면
스프링 프레임워크가
auto scan과정을 거친 후
해당 클래스를
자동으로 Bean 등록한다

- @controller : 해당 클래스가 Presentation Layer 에서 Controller 임을 명시
- @Service : 해당 클래스가 Business Layer 에서 Service 임을 명시
- @Repository : 해당 클래스가 Persistence Layer 에서 DAO임을 명시
- @Component : 위 세가지 외로 Bean으로 등록하고 싶은 클래스에 명시

1.3. Spring IoC[8/10] - 어노테이션을 활용한 Bean 관리

- Spring은 ApplicationContext의 BeanDefinition을 통해 빈을 찾는다
- @Service , @Repository, @Component, @controller 등 스테레오타입 클래스들을 자동으로 인식한다
- 스프링이 관리하는 모든 컴포넌트를 계층별로 빈의 특성이나 종류에 맞는 스테레오타입 어노테이션을 사용한다

특정 계층에서 사용하는 스테레오타입 어노테이션

스테레오타입 어노테이션	설명
@Repository	데이터 액세스 계층의 DAO 또는 Repository 클래스에 사용 DataAccessException 자동변환과 같은 AOP의 적용 대상을 선정하기 위해 사용하기도 함
@Service	서비스 계층의 클래스에 사용
@Controller	프레젠테이션 계층의 MVC 컨트롤러에 사용. 스프링 웹 서블릿에 의해 웹 요청을 처리하는 컨트롤러 빈으로 선정
@Component	위의 계층 구분을 적용하기 어려운 일반적인 경우에 사용

1.3. Spring IoC[9/10] - Bean의 생명주기와 콜백

- Spring IoC 컨테이너는 인스턴스의 생성과 삭제 시점에 호출되는 메소드를 설정할 수 있다
- 빈의 초기화 콜백 메소드는 DI를 통해모든 프로퍼티가 셋팅된 후에만 가능한 초기화를 지원해 준다
- 빈 소멸전 콜백 메소드는 컨테이너가 종료될 때 호출되어 종료전에 처리해야하는 작업을 수행한다

1.3. Spring IoC[10/10] - Bean의 Scope

- 스프링 빈이 생성되고, 존재하며 적용되는 범위를 스코프라고 한다.
- 스프링 빈의 기본 스코프는 싱글톤이지만 경우에 따라 다른 스코프를 가질 수 있다.
  - ✓ 빈을 요청할 때 마다 새로운 오브젝트를 만드는 프로토타입 스코프,
  - ✓ HTTP 요청이 생길때마다 생성되는 Request 스코프,
  - ✓ 웹의 세션 스코프와 유사한 Session 스코프 등이 있다.

- Spring은 기본적으로 모든 빈을 싱글톤 빈으로 생성한다
  - ✓ 싱글톤이란 객체를 한번만 만들어서 사용한다는 것이다
  - ✓ 하나의 Bean 정의에 대해서 Container 내에 단 하나의 객체만 존재한다.
  - ✓ 애플리케이션에서 제한된 수, 대개 한 개의 오브젝트만 만들어 사용하는 것이 싱글톤 패턴의 원리이다
- 스프링이 싱글톤을 기본으로 하는 이유
  - ✓ 스프링이 주로 적용되는 대상이 자바 엔터프라이즈 기술을 사용하는 서버환경이기 때문에 클라이언트에서 요청이 올 때마다 각 로직을 담당하는 오브젝트를 새로 만들어서 사용한다면 부하가 걸려 서버가 감당하기 힘들 것이다. 따라서 서버환경에서는 서비스 싱글톤의 사용이 권장된다.

2

Spring Boot

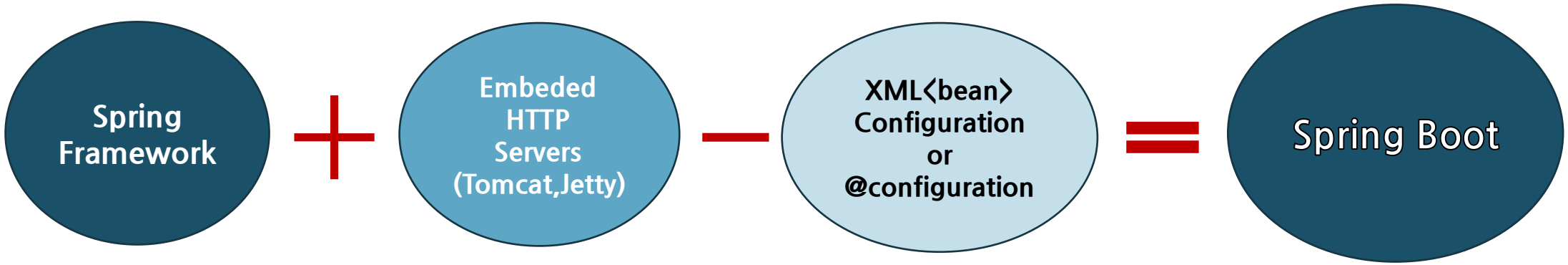
1. Spring Boot 탄생 배경
2. Spring Boot 개요
3. Auto Configuration
4. Starter와 의존성 관리

2.1. Spring Boot 탄생 배경(Spring의 단점 보완)

- 복잡한 설정 작업
  - ✓ 스프링은 강력한 기능을 제공하기 위해 많은 설정과 구성이 필요하다.
  - ✓ 이는 초기 설정의 복잡성을 증가시킬 수 있고, 초보자에게는 어렵게 느껴질 수 있다.
  - ✓ 개발자들이 애플리케이션 컨텍스트 설정, 빈 정의, 다양한 컴포넌트 구성 등을 위해 많은 설정 코드를 작성해야 한다.
- 높은 초기 학습 난이도
  - ✓ 스프링은 다른 프레임워크에 비해 학습이 필요한 부분이 많아서 다양한 개념과 기능을 이해하고 사용하기 위해 시간과 노력이 필요하다
- 의존성 관리 문제
  - ✓ 전통적인 스프링 프레임워크에서는 여러 의존성과 그들의 버전을 관리하는 것이 매우 복잡하다.
  - ✓ 스프링 레거시에서는 의존성 주입(Dependency Injection)을 구현하기 위해 XML 설정 파일에 많은 수의 빈(Bean)을 등록해야 하는데 이때문에 코드의 가독성이 떨어지고, 의존성 관리가 어려워질 수 있다
- 별도 WAS 서버 구성의 번거로움
  - ✓ 스프링을 웹상에서 사용하기 위해서는 별도의 Web Application Server(WAS)를 설치하고 설정해야 한다.
  - ✓ 또한, 애플리케이션을 서비스하기 위해서 별도의 서버에 수동으로 배포해야 하는 번거로움이 있다.

2.2. Spring Boot 개요

- 스프링 부트는 기본적인 설정과 보일러 플레이트 코드(여러 곳에서 재사용되는 코드) 작성을 최소화하고, 자동 설정과 컨벤션을 통해 개발자들이 빠르게 애플리케이션을 개발할 수 있도록 지원하는 스프링 프레임 워크이다
- 스프링 부트는 스프링(Spring Legacy)의 장점은 그대로 계승하면서, 기존의 문제가 되는 부분을 보완하여 기업용 애플리케이션의 개발 생산성 및 서비스 운영, 성능을 모두 해결해 줄 수 있다
- 개발자들의 개발 생산성을 높이고, 애플리케이션의 유연성, 확장성을 제공할 뿐만 아니라, 스프링 프레임 워크들과 강력하게 호환되고, 생태계와의 통합을 가능하게 해준다



2.3. Spring Boot - Auto Configuration

- 스프링 부트의 자동 구성은 많은 공통 작업과 설정을 대신 처리해주므로 개발자는 애플리케이션의 핵심 비지니스 로직에 집중할 수 있다.
- 자동 구성은 일관성 있고 안정적인 애플리케이션을 빠르게 개발하는 데 도움을 준다.
- 개발자가 필요한 경우에만 구성을 재정의하고 커스터마이징할 수 있기 때문에 높은 유연성도 제공한다

스프링부트 자동구성의 핵심 원칙

- ✓ Classpath Scanning: 애플리케이션의 클래스패스를 스캔하여 사용 가능한 라이브러리와 구성을 찾기 때문에 개발자는 필요한 의존성을 추가하면 자동으로 관련 설정 적용
- ✓ 조건부 자동 구성(Conditional Auto-Configuration): 스프링 부트는 자동 구성을 할 때, 특정 조건에 따라 자동으로 활성화 또는 비활성화할 수 있다. 예를 들어, 특정 빈이 클래스패스에 존재하거나 특정 프로퍼티가 설정되어 있는 경우에만 해당 자동 구성을 활성화시킬수 있다.
- ✓ 우선순위 지정(Priority Ordering): 여러 자동 구성 후보가 있을 경우, 우선순위를 지정하여 어떤 구성이 적용될지 결정할 수 있다. 개발자는 필요에 따라 구성 후보의 우선순위를 조절할 수 있다.
- ✓ 사용자 정의 자동 구성(Custom Auto-Configuration): 개발자는 자신만의 자동 구성을 작성할 수 있으며, 스프링 부트는 이러한 사용자 정의 구성을 자동으로 감지하여 애플리케이션에 적용한다.
- ✓ 자동 구성 보고서(Auto-Configuration Report): 스프링 부트는 자동 구성이 어떻게 적용되었는지에 대한 보고서를 제공하기 때문에 개발자는 애플리케이션의 구성을 이해하고 디버깅할 수 있다

2.4. Spring Boot - Starter와 의존성 관리

- Spring Boot Starter 및 의존성 관리는 Spring Boot 프레임워크에서 제공하는 핵심 기능 중 하나로, Spring 기반 애플리케이션을 빠르고 쉽게 구축하고 관리할 수 있도록 해준다

• Spring Boot Starter

- ✓ 특정 기능 또는 라이브러리에 대한 설정을 포함하는 스타터 모듈의 집합이다.
- ✓ 스타터 모듈은 메이븐이나 그레이들과 같은 빌드 도구를 사용하여 프로젝트에 추가된다.
- ✓ 스타터 모듈은 일반적으로 특정 용도에 맞게 구성된 의존성 세트를 제공하며, 프로젝트에 필요한 설정과 코드를 간단하게 가져올 수 있도록 도와준다.
  - spring-boot-starter-web 스타터 모듈을 사용 → Spring MVC, 내장 웹 서버(Tomcat)등의 의존성 자동 추가
- ✓ 개발자는 필요에 따라 자신만의 사용자 정의 스타터를 만들어 프로젝트에서 자주 사용하는 설정과 의존성을 패키지화하여 재사용할 수 있다

• 의존성 관리

- ✓ Spring Boot는 의존성 관리를 위한 자체적인 메커니즘을 제공하여 통해 버전 충돌을 방지하고 일관된 의존성을 유지할 수 있다.
- ✓ spring-boot-starter-parent라는 parent POM (Project Object Model)을 사용하여 프로젝트의 parent 설정을 관리하고, 의존성 버전을 관리.
- ✓ parent POM을 사용하면 Spring Boot 버전 및 일반적인 라이브러리의 버전을 쉽게 업데이트할 수 있다.
- ✓ Maven과 Gradle에서 사용할 수 있는 BOM (Bill Of Materials)을 제공하여 관련된 의존성 버전의 일관성 유지 지원

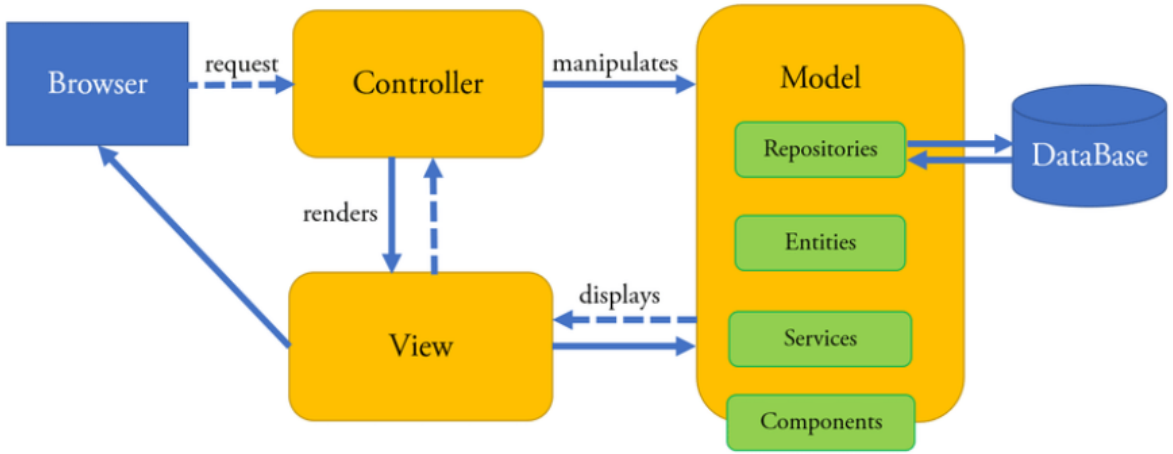
3

Spring MVC

1. Spring MVC 개요
2. MVC 각 요소의 역할
3. MVC 주요 인터페이스
4. Controller
5. View
6. Model

3.1. Spring MVC 개요

- 애플리케이션의 개발 영역을 MVC(Model, View, Controller)로 구분하여 각 역할에 맞게 코드를 작성하는 개발 방식이다.
- MVC 패턴은 UI 코드와 비지니스 코드를 분리함으로써 종속성을 줄이고, 재사용성을 높이고, 보다 쉬운 변경이 가능하도록 한다
- Spring MVC를 사용하면 Model, View, Controller 모두를 인터페이스를 사용해 규격화해서 유연하고 확장성 있게 웹 어플리케이션을 설계할 수 있다.



3.2. Spring MVC 각 요소의 역할

- MVC(Model-View-Controller) 패턴은 코드를 기능에 따라 Model, View, Controller 3가지 요소로 분리
  - ✓ Model : 어플리케이션의 데이터와 비지니스 로직을 담는 객체이다.
  - ✓ View : Model의 정보를 사용자에게 표시한다. 하나의 Model을 다양한 View에서 사용할 수 있다.
  - ✓ Controller : Model과 View의 중계역할을 한다. 사용자의 요청을 받아 Model에 변경된 상태를 반영하고, 응답을 위한 View를 선택한다.

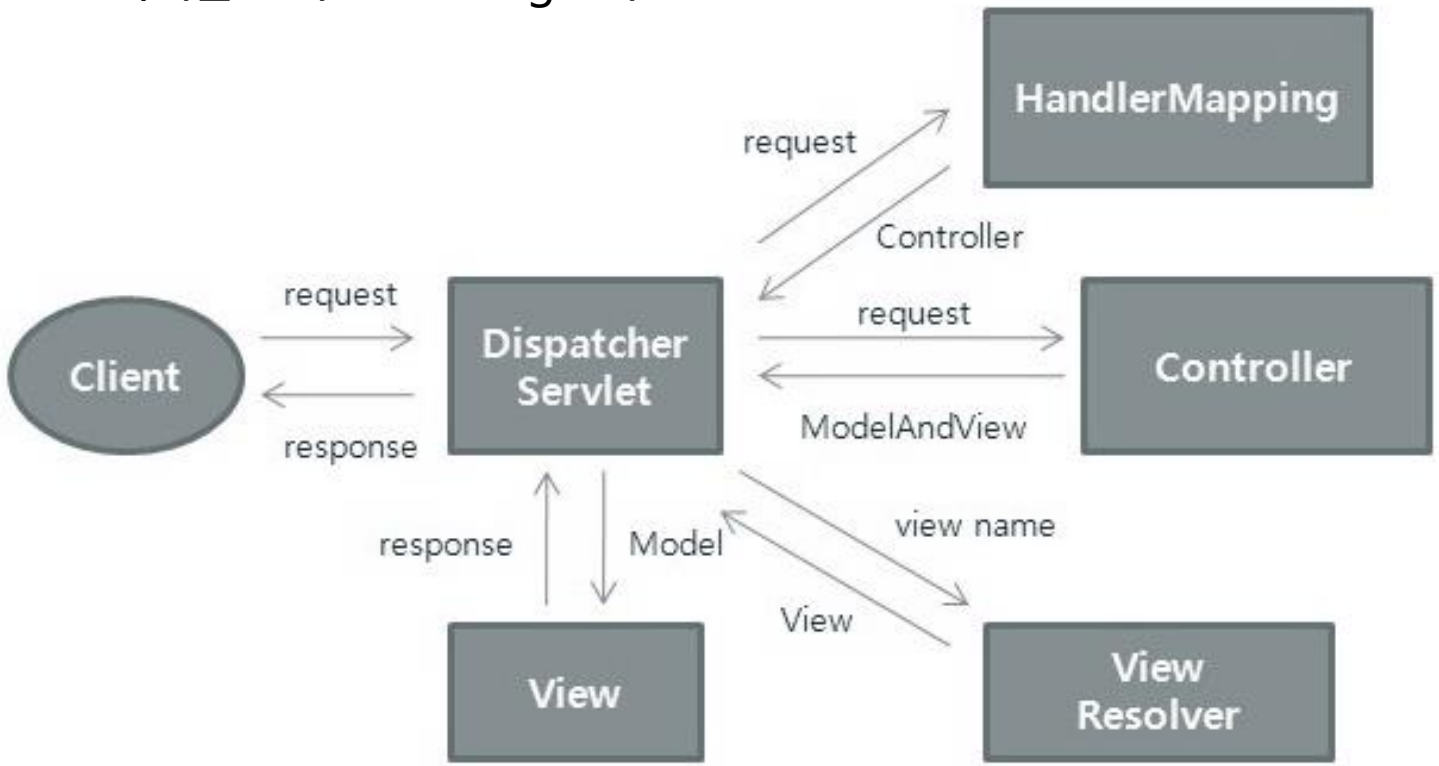
Model	• 애플리케이션 상태 캡슐화 • 상태 쿼리 응답 • 애플리케이션 기능 표현 • 변경사항 뷰에 알림
View	• 모델을 화면에 표현 • 모델에게 변경사항 요청 • 사용자입력을 컨트롤러에 전달 • 컨트롤러의뷰 선택 허용
Controller	• 애플리케이션 기능(행위) 정의 • 사용자 요청에 대한 모델 매핑 및 모델 업데이트 • 응답에 대한 뷰 선택

3.3. Spring MVC 주요 인터페이스 [1/2]

- DispatcherServlet
  - ✓ Spring MVC Framework의 Front Controller, 웹요청과 응답의 Life Cycle을 주관한다.
  - ✓ 단일 Front Controller로 모든 HTTP요청을 수신하여 그 밖의 오브젝트 사이의 흐름을 제어한다
- HandlerMapping
  - ✓ 웹요청시 해당 URL을 어떤 Controller가 처리할지 결정한다. (URL,Controller class명, Annotation 기준)
- Controller
  - ✓ 비지니스 로직을 수행하고 요청 처리 결과 데이터를 ModelAndView에 반영한다.
- ModelAndView
  - ✓ Controller가 수행 결과를 반영하는 Model 데이터 객체와 이동할 View객체로 이루어져 있다.
- ViewResolver
  - ✓ View 이름을 바탕으로 어떤 View를 선택할지 결정한다.
- View
  - ✓ 결과 데이터인 Model 객체를 display한다.

3.3. Spring MVC 인터페이스간의 관계와 흐름[2/2]

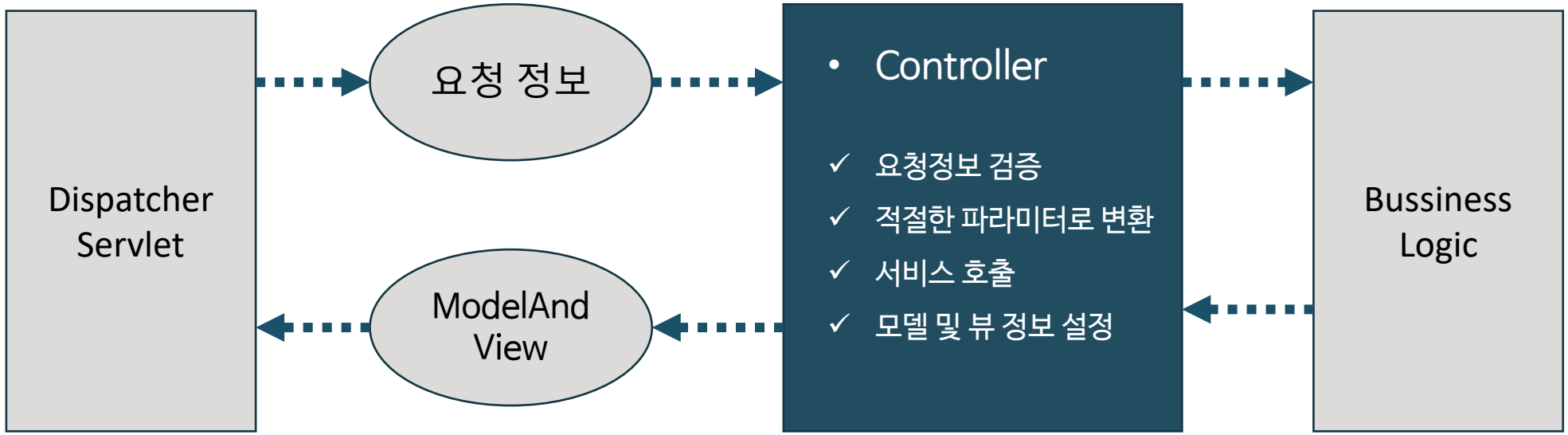
- Client의 요청이 들어오면 DispatchServlet이 가장 먼저 요청을 받는다.
- HandlerMapping이 요청에 해당하는 Controller를 return한다.
- Controller는 비지니스 로직을 수행(호출)하고 결과 데이터를 ModelAndView에 반영하여 return한다.
- ViewResolver는 view name을 받아 해당하는 View 객체를 return한다.
- View는 Model 객체를 받아 rendering한다.



3.4. Controller와 @Controller[1/5]

• Old Controller

- Controller는 클라이언트의 모든 요청을 받아서 처리한다
- 클라이언트의 요청을 처리할 서비스를 호출하여 비지니스 로직을 처리한다
  - ✓ 이를 위해 적절한 파라미터로 변환하여 서비스에게 전달
- 처리 결과를 View에 전달하여 결과를 화면에 렌더링하게 한다
  - ✓ 서비스로부터 처리 결과를 받으면 어떤 뷰를 보여줘야 할지 결정한 후 처리 결과를 뷰에 전달할 수 있는 형태로 생성
- 모델과 뷰를 생성하여 DispatcherServlet에게 전달한다
- Spring MVC 컨트롤러는 사용자의 요청을 처리하기 위한 Java 클래스로 구현된다



3.4. @Controller[2/5]

- @Controller 어노테이션을 클래스에 등록하여 XML의 빈등록 설정 작업 없이 컨트롤러를 빈으로 등록한다

- **@Controller**

- @MVC에서 Controller를 만들기 위해서는 작성한 클래스에 @Controller를 붙여주면 된다.
- 특정 클래스를 구현하거나 상속할 필요가 없다
- 어노테이션을 사용하기 위해서는 〈context:component-scan/〉 설정이 필요하다
  - ✓ @Controller만 스캔하려면 〈context:include-filter〉 조건을 추가한다
  - ✓ 다른 설정 정보와 충돌을 피하기 위해 excludeFilters로 스캔 대상에서 제외할 수도 있다

```
<context:component-scan base-package="com.backend.spring.mvc">
<context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"/>
</context:component-scan>
```

3.4. @Controller[3/5]

- 스프링 부트는 핸들러 매핑과 핸들러 어댑터를 자동으로 등록해준다.
- 핸들러매핑은 HTTP 요청정보를 처리하는 컨트롤러(핸들러)를 찾아주는 역할을 한다
- Spring은 이미 필요한 핸들러 매핑과 핸들러 어댑터를 대부분 구현해두었기 때문에 개발자가 직접 핸들러 매핑과 핸들러 어댑터를 구현하는 일은 거의 없다.

- **HandlerMapping 순위**

1. RequestMappingHandlerMapping
   - ✓ 어노테이션 기반 컨트롤러인 @RequestMapping에서 사용한다
2. BeanNameUrlHandlerMapping
   - ✓ 스프링 빈 이름으로 핸들러를 찾는다.

- **HandlerAdapter 순위**

1. RequestMappingHandlerAdapter
   - ✓ 어노테이션 기반 컨트롤러인 @RequestMapping에서 사용한다
2. HttpRequestHandler
   - ✓ AdapterHttpRequestHandler를 처리한다
3. SimpleControllerHandlerAdapter
   - ✓ Controller 인터페이스를 처리한다.

3.4. @Controller[4/5] - @RequestMapping

- @RequestMapping

- 요청에 대해 어떤 Controller, 어떤 메소드가 처리할지를 맵핑하기 위한 어노테이션이다
- 상위(Parent) 컨트롤러 클래스에 @RequestMapping 을 추가하면 하위 컨트롤러 클래스에도 적용된다
  - ✓ 하위클래스에서 @RequestMapping을 재정의하면 상위클래스의 @RequestMapping은 무시하고 하위클래스의 @RequestMapping이 적용된다
  - ✓ @RequestMapping을 인터페이스에 작성한 경우 그 구현 클래스에도 동일하게 적용된다

```java
public class ParentController {
    @RequestMapping("find")
    public String find() {
        return "find";
    }
}
```

```java
@Controller
public class ChildController extends ParentController {
    @Override
    public String find() {
        System.out.println("child find");
        return "find";
    }
}
```

자식 클래스의 @RequestMapping 적용
→ “child find” 출력

3.4. @Controller[5/5] - @RequestMapping

이름	타입	설명
value	String []	URL 값으로 맵핑 조건을 부여한다. • @RequestMapping(value="/hello.do") • @RequestMapping(value={"/hello.do", "/world.do" }) • @RequestMapping("/hello.do")
method	RequestMethod []	HTTP Request 메소드값을 맵핑 조건으로 부여한다. • HTTP 요청 메소드값이 일치해야 한다. • @RequestMapping(method = RequestMethod.POST) 같은 형식으로 표기한다. • 사용 가능한 메소드는 GET, POST, HEAD, OPTIONS, PUT, DELETE, TRACE이다
params	String []	HTTP Request 파라미터를 맵핑 조건으로 파라미터 값이 지정한 값과 일치할 경우에만 요청을 매핑한다 • params="myParam=myValue" → myParam이라는 파라미터가 있어야 하고 값은 myValue일 때만 맵핑 • params="!myParam" → myParam이라는 파라미터가 없는 요청만 맵핑

3.5. View[1/4] - 개요

- MVC패턴에서 뷰는 모델의 데이터를 전달받아서 화면에 다양한 형식으로 표현하는 역할을 한다
- 뷰는 일반적으로 HTML로 생성되어 브라우저를 통해 결과를 표현한다
- 뷰를 직접 사용하는 대신 메시지 컨버터를 사용하면 XML, Json을 생성할 수도 있다

• View 지정 방법

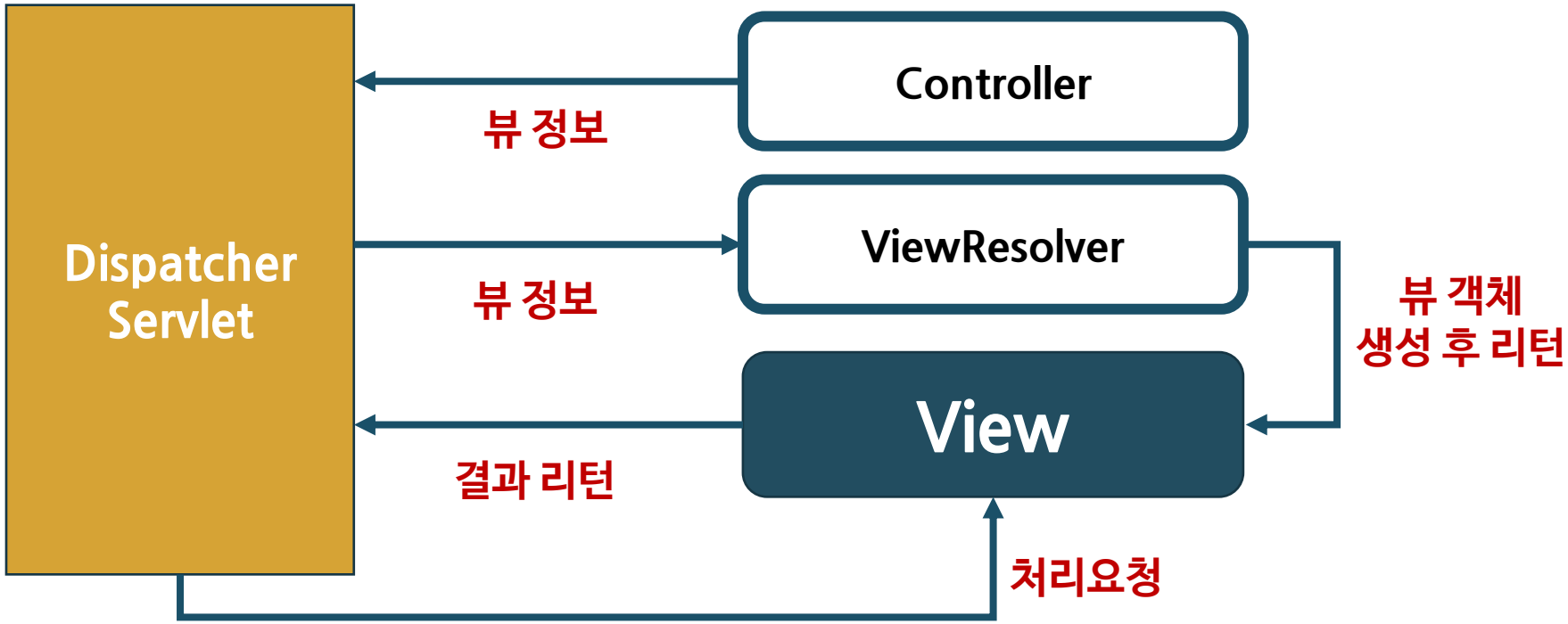
- 컨트롤러가 ModelAndView의 View객체에 뷰이름을 설정
- 스프링 부트에서 MVC 뷰를 지정하는 가장 일반적인 방법 중 하나는 `@Controller` 어노테이션을 사용하여 컨트롤러 클래스를 만들고, 해당 컨트롤러의 메소드에서 뷰 이름을 반환하는 것이다. 이 때, `@RequestMapping` 어노테이션을 사용하여 URL과 매핑한다

```
import org.springframework.stereotype.Controller;
import
org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

   @RequestMapping("/myPage")
   public String myPage() {
      return "myPage"; // 뷰 이름 반환
   }
}
```

1. "/myPage" URL이 `myPage()` 메소드와 매핑
2. myPage()는 " myPage " 라는 뷰 이름 반환
3. 스프링 부트는 “myPage” 이름을 가진 뷰 템플릿을 찾아 렌더링
4. 뷰 템플릿은 src/main/resources/templates폴더에 저장
5. 뷰 템플릿
   - ✓ src/main/resources/templates/myPage.html

3.5. View[2/4] - Controller와의 관계

- 컨트롤러가 요청 처리후 리다이렉트할 뷰를 지정한 경우 DispatcherServlet은 해당 뷰에 처리를 요청
- 명확한 뷰객체 지정없이 뷰 이름 정보만 있어도 ViewResolver를 통해서 뷰객체를 생성할 수 있다
- 아무런 뷰정보도 알려주지 않은 경우 DefaultRequestToViewNameTranslator를 통해 뷰이름을 설정할 수 있다



3.5. View[3/4] - ViewResolver

- 컨트롤러가 뷰 이름을 반환하면 ViewResolver를 통해 실제 사용할 뷰를 결정한다
- BeanNameViewResolver혹은 InternalResourceViewResolver를 통해서 랜더링을 진행한다.
- ModelAndView가 반환되는 Dispatcher 시점에서 View를 랜더링한다.
- Spring Boot는 container를 초기화할 때
  - ✓ InternalResoureceViewResolver와
  - ✓ BeanNameViewResolver를 bean으로 자동 등록한다
- 기본으로 제공되는 ViewResolver의 구현체를 등록하여 사용하거나 기능을 확장하여 구현할 수 있다
- 특정 ViewResolver를 빈으로 등록하지 않으면 InternalResoureceViewResolver가 자동등록된다

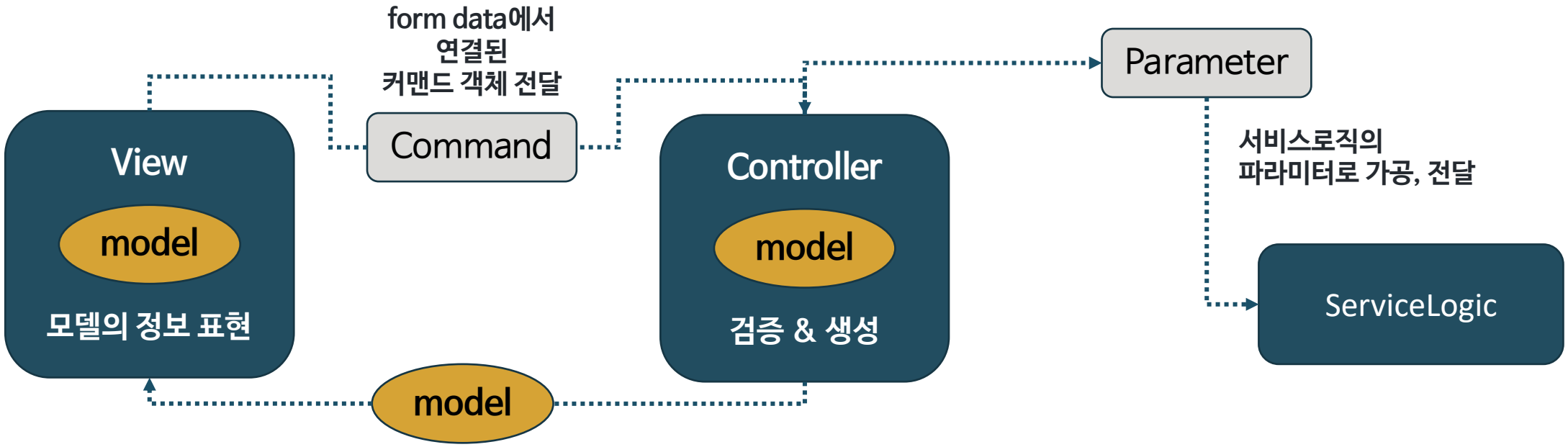
3.5. View[4/4] - RedirectView

- 리다이렉트 뷰는 실제로 뷰 생성은 하지 않고 URL을 생성하여 다른 페이지로 리다이렉트한다
- 리다이렉트 뷰는 RedirectView객체나 접두어 “redirect: “를 포함한 문자열을 리턴하여 지정한다
  - ✓ RedirectView 오브젝트를 직접 사용하지 않아도 문자열에 접두어“redirect: “를 포함시키면 ViewResolver에서 리다이렉트를 처리해준다

```
//회원을 등록하고 전체목록 조회화면으로 리다이렉트

@RequestMapping(value – “/user”, method = RequestMethod.POST)
public String addUser(User user)    {

            user.creatUser(user);
            return “redirect:/”;
}
```

3.6. Model[1/3]

- 컨트롤러는 뷰에서 전달된 데이터를 커맨드 객체로 전달받아 처리
- 컨트롤러에서 생성된 모델은 검증 후 서비스로직의 메소드에 전달하는 파라미터가 될수 있다
- 컨트롤러가 모델을 리턴하면 DispatcherServlet은 모델을 뷰에 전달
- 뷰는 전달받은 모델을 클라이언트(예 브라우저)에 표현한다



3.6. Model[2/3] - Command 객체

- Command 객체는 사용자가 웹 어플리케이션에서 제출하는 데이터를 수신하고 처리하는 데 사용되는 객체를 말한다
- 주로 HTML 폼으로부터 제출된 데이터를 수신하고 처리하는 데 사용된다
- 컨트롤러는 뷰의 HTML 폼으로부터 전달된 데이터를 파라미터로 받아 처리할 수 있다
- 커맨드 객체는 개발자가 모델에 추가하지 않아도 자동으로 추가되어 뷰에게 전달된다
- 뷰에서는 커맨드 객체의 클래스명(첫글자 : 소문자)을 사용하여 커맨드 객체에 접근한다
  - ✓ 클래스명이 아닌 다른 이름으로 정하고자 할 때는 @ModelAttribute("변경할 이름")을 파라미터 앞에 붙이면 된다

```
<form action="posting" method="post">
    <input type="text" name="postings[0].id" />
    <input type="text" name="postings[0].title" />
    <input type="text" name="postings[0].contents" />
    <br/>
    <input type="text" name="postings[1].id" />
    <input type="text" name="postings[1].title" />
    <input type="text" name="postings[1].contents" />
    <br/>
    <input type="submit" />
```

```
public class PostingCommand {

    private List<Posting> postings;

    public void setPostings(List<Posting> postings) {
        this.postings = postings;
    }

}
```

```
@Controller
@RequestMapping("/posting")
public class PostingController {

    @RequestMapping(method=RequestMethod.POST)
    public String regist(PostingCommand command) {
        ...
    }
```

3.6. Model[3/3] - @RequestParam, @ModelAttribute

- @RequestParam은 요청 파라미터를 1:1로 받을 때 사용
- @ModelAttribute는 요청 파라미터를 객체 형태로 받을때 사용
- @ModelAttribute가 붙은 객체는 자동으로 모델에 추가되므로 뷰에서 바로 사용할 수 있다

```
@RequestMapping(method=RequestMethod.POST)
public String regist(@ModelAttribute("posting") Posting posting) {
    ...
    return "posting";
}
```

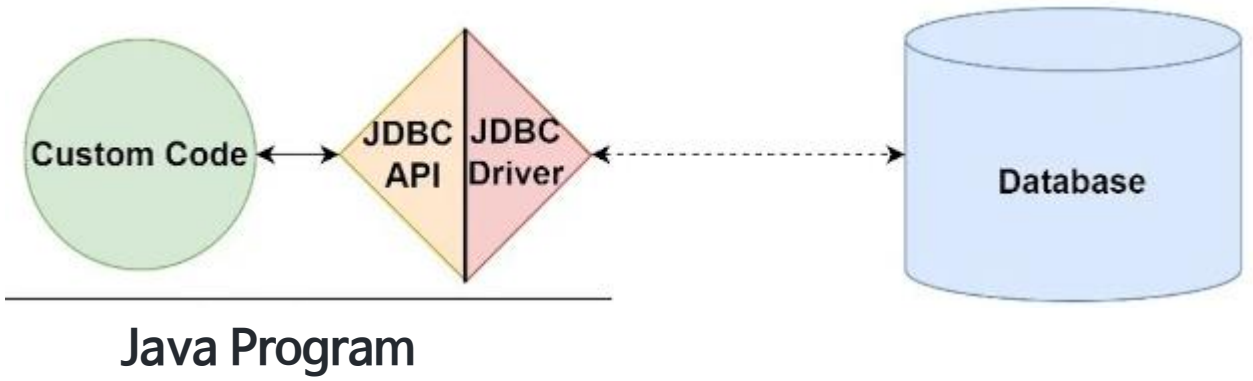
4

Spring Data

1. JDBC
2. SQL Mapping
3. OR Mapping
4. Spring Data JPA

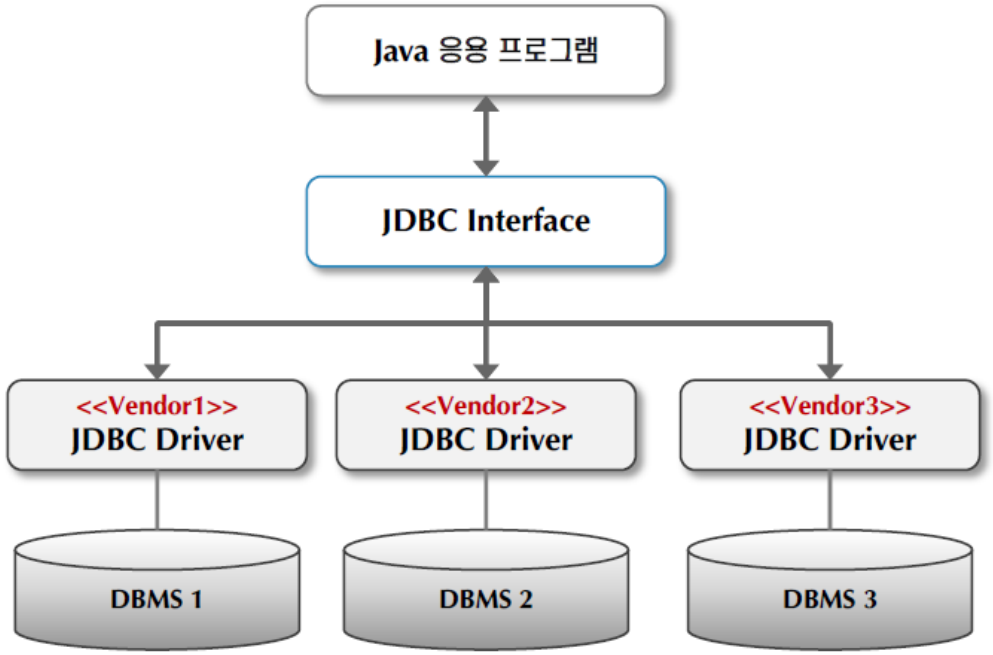
4.1. JDBC[1/2]

- Java Database Connectivity의 약자로, Java 언어를 사용하여 데이터베이스에 연결하고 상호 작용하기 위한 자바 API이다
- 웹 애플리케이션은 정보 저장이 필요할 때 주로 DB를 이용한다.
- 또한 웹 애플리케이션 서버(web application server – WAS)는 애플리케이션에 Connection Pooling 등 DB 의존적인 서비스를 제공하기 위하여 DB와 통신할 필요가 있다. 이러한 필요를 보다 체계적이고 효율적으로 충족시키기 위해 DB 클라이언트들과 DB 간 인터페이스를 정의한 것이 바로 Java Database Connectivity (JDBC) 표준이다.
- JDBC 표준은 애플리케이션의 DB Connection 사용 방법 및 SQL 작업 수행 방법에 대해 기술하고 관련 API를 제공한다



4.1. JDBC API [2/2]

- 객체중심의 자바 애플리케이션과 테이블 중심의 관계형데이터베이스는 서로의 목표가 달라 패러다임의 불일치로 개발과정에서 많은 비용과 노력이 소요된다
- JDBC는 데이터의 영속적인 저장을 위해 위 둘을 연결해주는 인터페이스이다
- 자바에서는 객체를 RDBMS에 영속화하는 방법으로 OR Mapping이나 SQL Mapping을 사용한다



객체지향	RDBMS
객체, 클래스	테이블, 로우
속성(attribute, property)	컬럼
Identity	Primary Key
Relationship/다른 엔티티 참조	Foreign Key
상속/다형성	없음
메소드	SQL 로직, SP, 트리거
코드의 이식성	벤더 종속적임

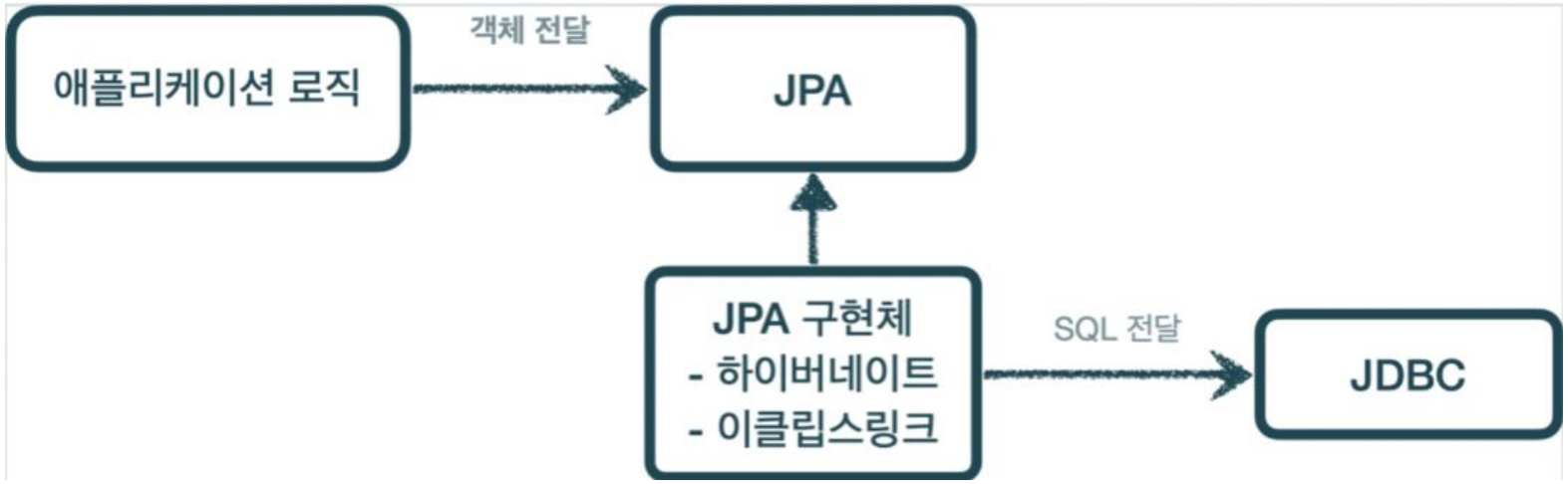
4.2. SQL Mapping

- SQL Mapper는 자바코드와 SQL을 분리하며 개발자가 JDBC를 편리하게 사용할 수 있는 툴을 제공한다.
  - ✓ SQL 응답 결과를 객체로 변환해준다.
  - ✓ JDBC의 반복 코드를 제거해준다.

- 개발자가 SQL을 직접 작성해야 한다.
- 대표 기술 : 스프링 Jdbc Template, MyBatis

4.3. OR Mapping[1/2]

- OR Mapping은 객체를 RDB 테이블과 매핑해주는 역할을 하며 SQL을 생성하여 패러다임의 불일치를 해결해준다
- 개발자가 직접 SQL을 작성하지 않아도 동적으로 대신 만들어 실행한다.
- DB마다 다른 SQL을 사용하는 문제도 중간에 해결해준다.
- JPA : Java진영의 ORM 표준인터페이스
  - ✓ 하이버네이트
  - ✓ 이클립스 링크

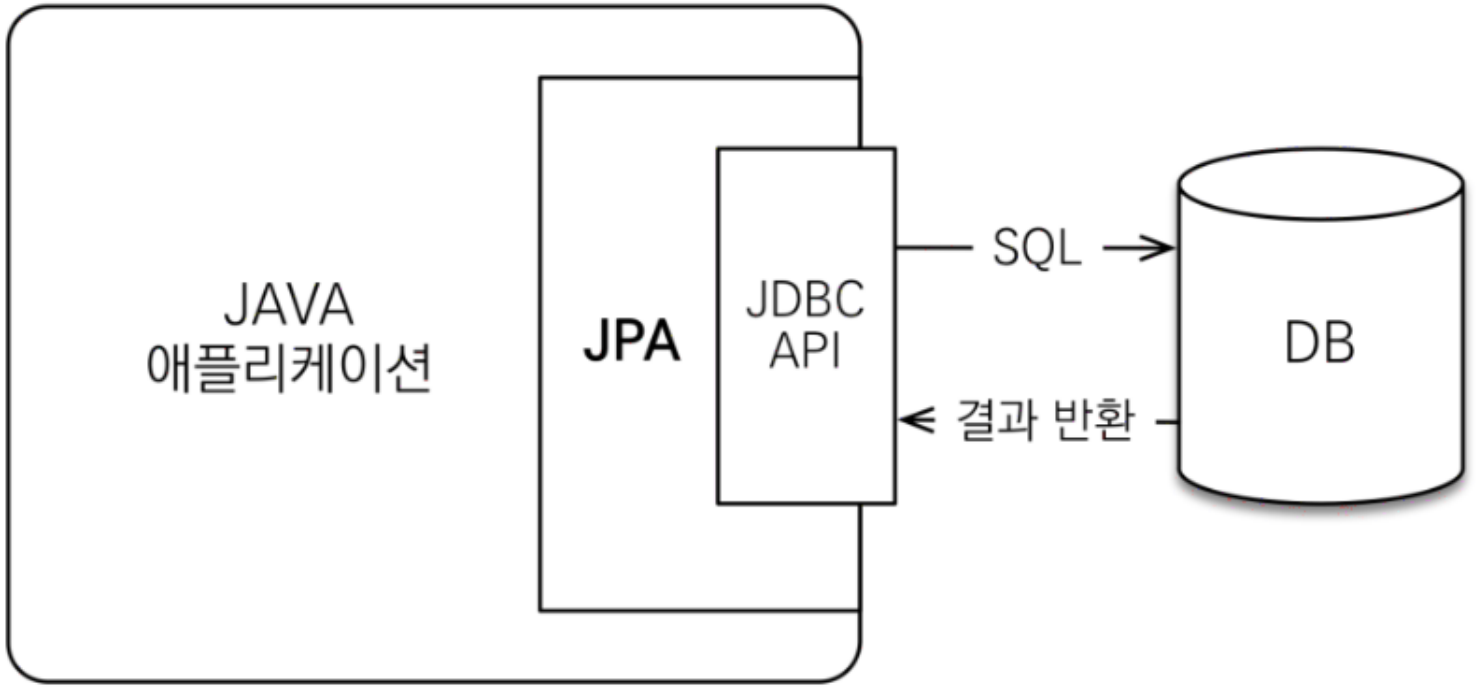


4.3. OR Mapping[2/2] - 특징

- 객체 모델과 관계형 데이터베이스 간의 매핑 기능인 ORM(Object-Relational Mapping) 기능을 제공함으로써, SQL이 아닌 객체을 이용한 업무 로직의 작성이 가능하도록 지원함
- 객체와 관계형 데이터베이스 테이블 간의 매핑
  - ✓ 프레임워크 설정 정보에 저장된 ORM 매핑 정보를 이용하여 객체와 관계형 데이터베이스 테이블간의 매핑 지원
- 객체 로딩
  - ✓ 객체와 매핑되는 관계형 데이터베이스의 값을 읽어와 객체의 속성 값으로 설정함
- 객체 저장
  - ✓ 저장하고자 하는 객체의 속성 값을 객체와 매핑되는 관계형 데이터베이스에 저장
- 다양한 연관 관계 지원
  - ✓ 객체와 객체 간의 1:1, 1:n, n:n 등의 다양한 연관 관계를 지원
  - ✓ 객체의 로딩 및 저장 시, 연관 관계를 맺고 있는 객체도 로딩 및 저장 지원
- Caching
  - ✓ 객체에 대한 Cache 기능을 지원하여 성능을 향상시킴

4.4. JPA(Java Persistence API)[1/7]

- JPA는 자바프로그램에서 RDBMS에 접근하는 방식을 명세화한 인터페이스이다
- 자바진영의 ORM 기술표준으로 자바애플리케이션과 JDBC사이에서 동작하며 구현체의 사용은 일반적으로 Hibernate라이브러리를 사용한다
- JPA를 적용할 경우 도메인 객체는 기술에 비의존적이면서 재사용성을 높일 수 있다
- JPA를 사용하기 위해서는 객체에 Annotation을 추가하거나 별도의 메타데이터를 구성해야한다



4.4. Spring Data JPA[2/7] - @Entity

- 엔티티는 데이터베이스 테이블과 매핑되는 자바 클래스를 말한다
- 도메인 객체를 Entity로 매핑할 때에는 별도의 매핑 클래스(Jpo)를 둔다
- 매핑클래스를 RDBMS의 테이블로 매핑할 때 @Entity 어노테이션을 사용한다
- @Entity 어노테이션만 선언했을 때 테이블명은 클래스 이름으로 지정되고 대소문자 치환은 일어나지 않는다
  - 대부분의 RDBMS에서는 대소문자를 가리지 않아서 JPA의 작명규칙이 문제가 되진 않는다

@Id

- 고유 번호 id 속성에 적용한 @Id 어노테이션은 id 속성을 기본 키로 지정한다.
- 기본 키로 지정하면 이제 id 속성의 값은 데이터베이스에 저장할 때 고유(Unique)한 값으로 저장된다
- 고유 번호를 기본 키로 한 이유는 고유 번호는 엔티티에서 각 데이터를 구분하는 유효한 값으로 중복되면 안 되기 때문이다.
  ✓ RDBMS에서는 id와 같은 특징을 가진 속성을 기본 키(primary key)라고 한다.

4.4. Spring Data JPA[3/7] - @GeneratedValue

- @GeneratedValue 어노테이션을 적용하면 데이터를 저장할 때 해당 속성에 값을 따로 세팅하지 않아도 1씩 자동으로 증가하여 저장된다.
  - ✓ strategy는 고유번호를 생성하는 옵션으로 GenerationType.IDENTITY는 해당 컬럼만의 독립적인 시퀀스를 생성하여 번호를 증가시킬 때 사용한다.
  - ✓ strategy 옵션을 생략할 경우에 @GeneratedValue 어노테이션이 지정된 컬럼들이 모두 동일한 시퀀스로 번호를 생성하기 때문에 일정한 순서의 고유번호를 가질수 없게 되므로 보통 GenerationType.IDENTITY를 많이 사용한다.

@Column

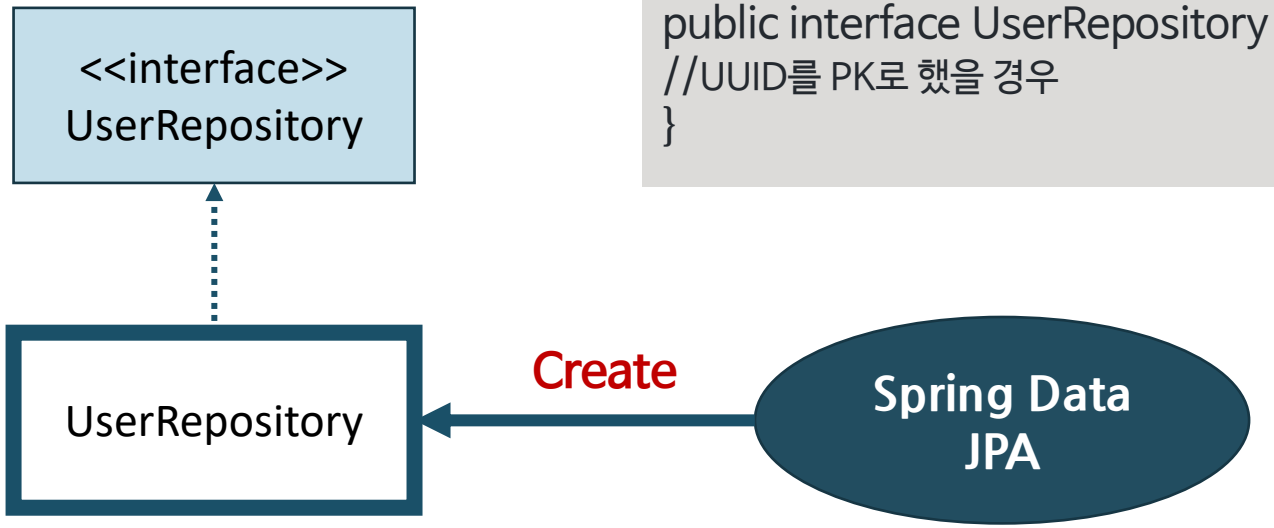
- 엔티티의 속성은 테이블의 컬럼명과 일치하는데 컬럼의 세부 설정을 위해 @Column 어노테이션을 사용한다.
  - length는 컬럼의 길이를 설정할때 사용하고
    - columnDefinition은 컬럼의 속성을 정의할 때 사용한다.
    - columnDefinition = "TEXT"은 글자 수를 제한할 수 없는 경우에 사용한다.

4.4. Spring Data JPA[4/7] - @Table

- RDBMS의 테이블 설계시 관습적인 부분으로 테이블명에 “_”를 사용한다
- @Table 어노테이션은 스키마를 지정하거나 테이블 명을 구체적으로 명시할 수 있다
  - “SA.USER”를 사용하면 스키마 지정이 가능하다
  - @Table(name= “USER”, category=“CATEGORY”) → 카테고리 지정

4.4. Spring Data JPA[5/7] - JpaRepository

- 리포지터리는 엔티티에 의해 생성된 데이터베이스 테이블에 접근하는 메소드들(예: findAll, save 등)을 사용하기 위한 인터페이스이다.
- 데이터 처리를 위해서는 테이블에 어떤 값을 넣거나 값을 조회하는 등의 CRUD(Create, Read, Update, Delete)가 필요한데 이때 CRUD를 어떻게 처리할지 정의하는 계층이 바로 리포지터리이다
- JpaRepository 인터페이스를 상속하는 것만으로도 기본적인 CRUD기능을 사용할 수 있다
- JpaRepository를 상속할 때는 제네릭 타입으로 리포지터리의 대상이 되는 엔티티의 타입과 해당 엔티티의 PK의 속성 타입(식별자 타입)을 지정해야 한다.




```
public interface UserRepository extends JpaRepository<User, String> {
//UUID를 PK로 했을 경우
}
```

4.4. Spring Data JPA[6/7] - JpaRepository 인터페이스 생성 & CRUD

엔티티 클래스 생성

```
@Entity
public class Product {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String name;
   private double price;
 .....
}
```

JpaRepository 인터페이스 생성

```
@Repository
public interface ProductRepository extends
                              JpaRepository<Product, Long> {
}
```

서비스 클래스 생성 – CRUD 수행

```
@Service
public class ProductService {
   private final ProductRepository productRepository;

   @Autowired
   public ProductService(ProductRepository
productRepository) {
      this.productRepository = productRepository;
   }

   public List<Product> getAllProducts() {
      return productRepository.findAll();
   }

   public Product getProductById(Long id) {
      return productRepository.findById(id).orElse(null);
   }

   public Product saveProduct(Product product) {
      return productRepository.save(product);
   }

   public void deleteProduct(Long id) {
      productRepository.deleteById(id);
   }
}
```

4.4. Spring Data JPA[7/7] - JPA Query Methods

메소드 이름 키워드	샘플	설명
And	findByEmailAndUserId(String email, String userId)	여러필드를 and 로 검색
Or	findByEmailOrUserId(String email, String userId)	여러필드를 or 로 검색
Between	findByCreatedAtBetween(Date fromDate, Date toDate)	필드의 두 값 사이에 있는 항목 검색
LessThan	findByAgeGraterThanEqual(int age)	작은 항목 검색
GreaterThanEqual	findByAgeGraterThanEqual(int age)	크거나 같은 항목 검색
Like	findByNameLike(String name)	like 검색
IsNull	findByJobIsNull()	null 인 항목 검색
In	findByJob(String … jobs)	여러 값중에 하나인 항목 검색
OrderBy	findByEmailOrderByNameAsc(String email)	검색 결과를 정렬하여 전달

참고자료 - JSON

- JSON(JavaScript Object Notation)은 데이터를 구조화하고 표현하기 위한 간단하고 가독성이 좋은 형식을 제공하며, 다양한 프로그래밍 언어에서 지원되는 표준 형식이다
- JSON 데이터는
  - ✓ 키-값 쌍 (Key-Value Pairs): JSON 데이터는 이름(키)과 해당 값의 쌍으로 구성.
    (키는 문자열로 표현되며, 값은 숫자, 문자열, 불리언, 배열, 객체, null 등 다양한 데이터 유형이다.)
  - ✓ 객체 (Object): JSON 데이터는 중괄호 {} 내에 키-값 쌍들을 포함하는 객체로 시작할 수 있다. 이러한 객체는 여러 키-값 쌍을 포함할 수 있으며, 각 쌍은 쉼표로 구분된다
  - ✓ 배열 (Array): JSON 데이터는 대괄호 [] 내에 값들의 목록으로 시작할 수도 있다. 이러한 배열은 여러 값을 순서대로 저장할 수 있으며, 값들은 쉼표로 구분된다.

```
{
 "name": "John",
 "age": 30,
 "city": "New York",
 "isStudent": false,
 "hobbies": ["reading", "swimming", "coding"],
 "address": {
  "street": "123 Main St",
  "zipCode": "10001"
 }
}
```

hobbies" 키의 값은 배열로 여러 가지 취미를 저장

"address" 키의 값은 또 다른 객체를 포함하며,
이 객체에는 "street" 및 "zipCode"라는 키-값 쌍이 포함