

# 17강. 좋은 코드(Clean Code)는 왜 중요한가?!

## Section 3. 역할의 분리와 스프링 컨테이너

1. 좋은 코드가 왜 중요한지 이해하고, 원래 있던 Controller 코드를 보다 좋은 코드로 리팩토링한다.
2. 스프링 컨테이너와 스프링 빈이 무엇인지 이해한다.
3. 스프링 컨테이너가 왜 필요한지, 좋은 코드와 어떻게 연관이 있는지 이해한다.
4. 스프링 빈을 다루는 여러 방법을 이해한다.

# Clean Code는 왜 중요한가?!

Code

# Clean Code는 왜 중요한가?!

코드는 요구사항을 표현하는 언어이다.

# Clean Code는 왜 중요한가?!

개발자는 요구사항을 구현하기 위해 코드를 읽고 작성한다.

# Clean Code는 왜 중요한가?!

개발자는 요구사항을 구현하기 위해 코드를 읽고 작성한다.

# Clean Code는 왜 중요한가?!

코드를 읽는 것은 필수적이고 피할 수 없다!!!

# Clean Code는 왜 중요한가?!

```
public void good(U u){if (u.a <= 60 && u.a > 10 && u.b > 130) { c();}}
```

```
public class U {int a; int b;}
```



# Clean Code는 왜 중요한가?!

```
public void good(U u){if (u.a <= 60 && u.a > 10 && u.b > 130) { c();}}
```



```
public void getOnTheRideIfPossible(User user) {  
    if (user.canGetOnTheRide()) {  
        ride();  
    }  
}
```

# Clean Code는 왜 중요한가?!

```
public class U {int a; int b;}
```



```
public class User {  
    2 usages  
    private int age;  
    1 usage  
    private int height;  
  
    1 usage  
    public boolean canGetOnTheRide() {  
        return this.age > 10 && this.age <= 60 && this.height > 130;  
    }  
}
```

# Clean Code는 왜 중요한가?!

```
public void good(U u){if (u.a <= 60 && u.a > 10 && u.b > 130) { c();}}
```

```
public class U {int a; int b;}
```

코드만 봐서는 무슨 의미인지 알 수가 없다.

# Clean Code는 왜 중요한가?!

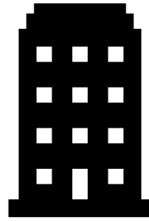
```
public class User {  
    2 usages  
    private int age;  
    1 usage  
    private int height;
```

```
public void getOnTheRideIfPossible(User user) {  
    if (user.canGetOnTheRide()) {  
        ride();  
    }  
}
```

```
    1 usage  
    public boolean canGetOnTheRide() {  
        return this.age > 10 && this.age <= 60 && this.height > 130;  
    }  
}
```

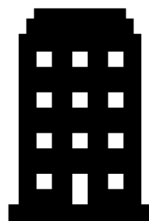
코드만 보고도 의미를 파악할 수 있다.

# 〈Clean Code〉의 사례



매우 인기있는 앱을 개발한 회사

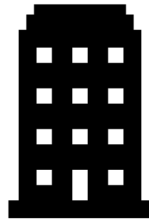
# 〈Clean Code〉의 사례



출시 주기가 점점 늘어지기 시작..!

버그가 여전히 남아 있고, 프로그램도 느려지고,  
아예 동작하지 않기도 함.

# 〈Clean Code〉의 사례



그렇게 사용자들이 점차 앱을 떠나가고..

# 〈Clean Code〉의 사례

(후일담) 다음 버전 출시가 바빠 코드를 마구 작성하였으며 기능을 추가할 수록 코드는 엉망이 되었고, 감당이 불가능해졌다.



# 안 좋은 코드가 쌓이면, 시간이 지날 수록 생산성이 낮아진다!

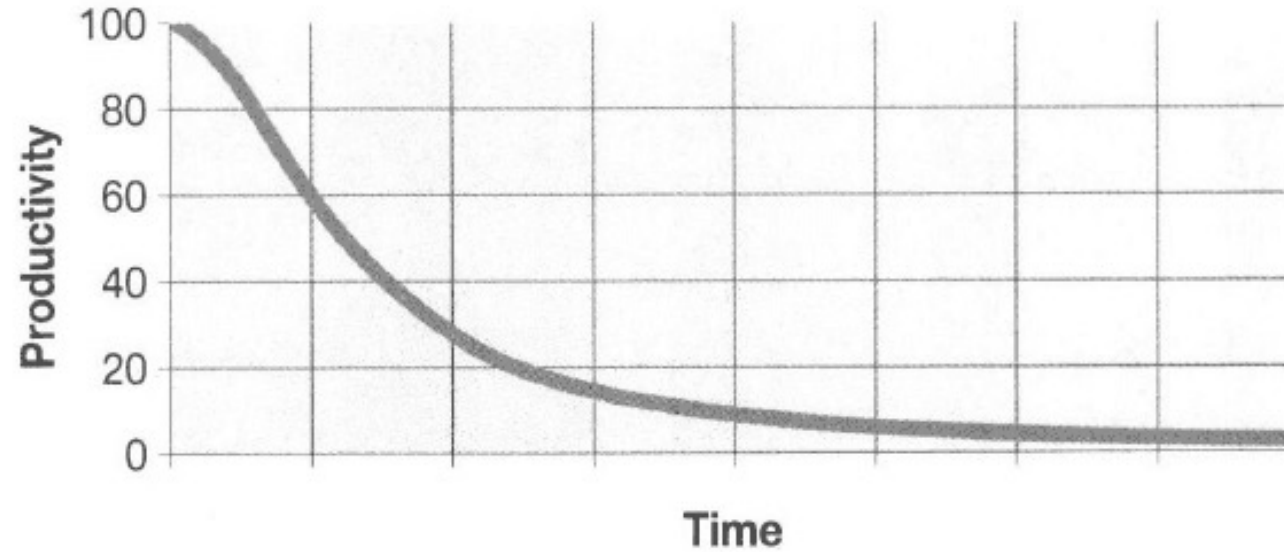


그림 1.1 생산성 대 시간

**그렇다면 Controller에서 모든 기능을 구현하면 왜 안될까?!**

<Clean Code>

함수는 최대한 작게 만들고 한 가지 일만 수행하는 것이 좋다.

**그렇다면 Controller에서 모든 기능을 구현하면 왜 안될까?!**

<Clean Code>

클래스는 작아야 하며 하나의 책임만을 가져야 한다.

# 우리가 작성한 Controller 함수 1개가 3000줄을 넘으면?!

1. 그 함수를 동시에 여러 명이 수정할 수 없다.
2. 그 함수를 읽고, 이해하는 것이 너무 어렵다.
3. 그 함수의 어느 부분을 수정하더라도 함수 전체에 영향을 미칠 수 있기 때문에 함부로 건들 수 없게 된다.

# 우리가 작성한 Controller 함수 1개가 3000줄을 넘으면?!

- 4. 너무 큰 기능이기 때문에 테스트도 힘들다.
- 5. 종합적으로 유지보수성이 매우 떨어진다.

# 우리가 작성한 Controller는?!

```
@PostMapping("/user")
public void updateUser(@RequestBody UserUpdateRequest request) {
    String readSql = "SELECT * FROM user WHERE id = ?";
    boolean isUserNotExist = jdbcTemplate.query(readSql, (rs, rowNum) -> 0, request.getId()).isEmpty();
    if (isUserNotExist) {
        throw new IllegalArgumentException();
    }

    String updateSql = "UPDATE user SET name = ? WHERE id = ?";
    jdbcTemplate.update(updateSql, request.getName(), request.getId());
}
```

[1] API의 진입 지점으로써 HTTP Body를 객체로 변환한다.

# 우리가 작성한 Controller는?!

```
@PostMapping("/user")
public void updateUser(@RequestBody UserUpdateRequest request) {
    String readSql = "SELECT * FROM user WHERE id = ?";
    boolean isUserNotExist = jdbcTemplate.query(readSql, (rs, rowNum) -> 0, request.getId()).isEmpty();
    if (isUserNotExist) {
        throw new IllegalArgumentException();
    }

    String updateSql = "UPDATE user SET name = ? WHERE id = ?";
    jdbcTemplate.update(updateSql, request.getName(), request.getId());
}
```

[2] 현재 유저가 있는지 없는지 확인하고 예외 처리를 한다.

# 우리가 작성한 Controller는?!

```
@PostMapping("/user")
public void updateUser(@RequestBody UserUpdateRequest request) {
    String readSql = "SELECT * FROM user WHERE id = ?";
    boolean isUserNotExist = jdbcTemplate.query(readSql, (rs, rowNum) -> 0, request.getId()).isEmpty();
    if (isUserNotExist) {
        throw new IllegalArgumentException();
    }

    String updateSql = "UPDATE user SET name = ? WHERE id = ?";
    jdbcTemplate.update(updateSql, request.getName(), request.getId());
}
```

[3] SQL을 사용해 실제 Database와의 통신을 담당한다.



# 우리가 작성한 Controller는?!

이렇게 무려 3가지 역할을 하고 있었다!!!

# 다음 시간에는

이 함수를 3가지 역할에 맞게 3단 분리를 하도록 하자!!

# 18강. Controller를 3단 분리하기 - Service와 Repository

# Controller의 함수 17개가 하고 있던 역할

1. API의 진입 지점으로써 HTTP Body를 객체로 변환하고 있다.
2. 현재 유저가 있는지, 없는지 등을 확인하고 예외 처리를 해준다.
3. SQL을 사용해 실제 DB와의 통신을 담당한다.

# Controller의 함수 1개가 하고 있던 역할

1. API의 진입 지점으로써 HTTP Body를 객체로 변환하고 있다.
2. 현재 유저가 있는지, 없는지 등을 확인하고 예외 처리를 해준다.
3. SQL을 사용해 실제 DB와의 통신을 담당한다.

Controller의 역할

# Controller의 함수 1개가 하고 있던 역할

1. API의 진입 지점으로써 HTTP Body를 객체로 변환하고 있다.
2. 현재 유저가 있는지, 없는지 등을 확인하고 예외 처리를 해준다.
3. SQL을 사용해 실제 DB와의 통신을 담당한다.

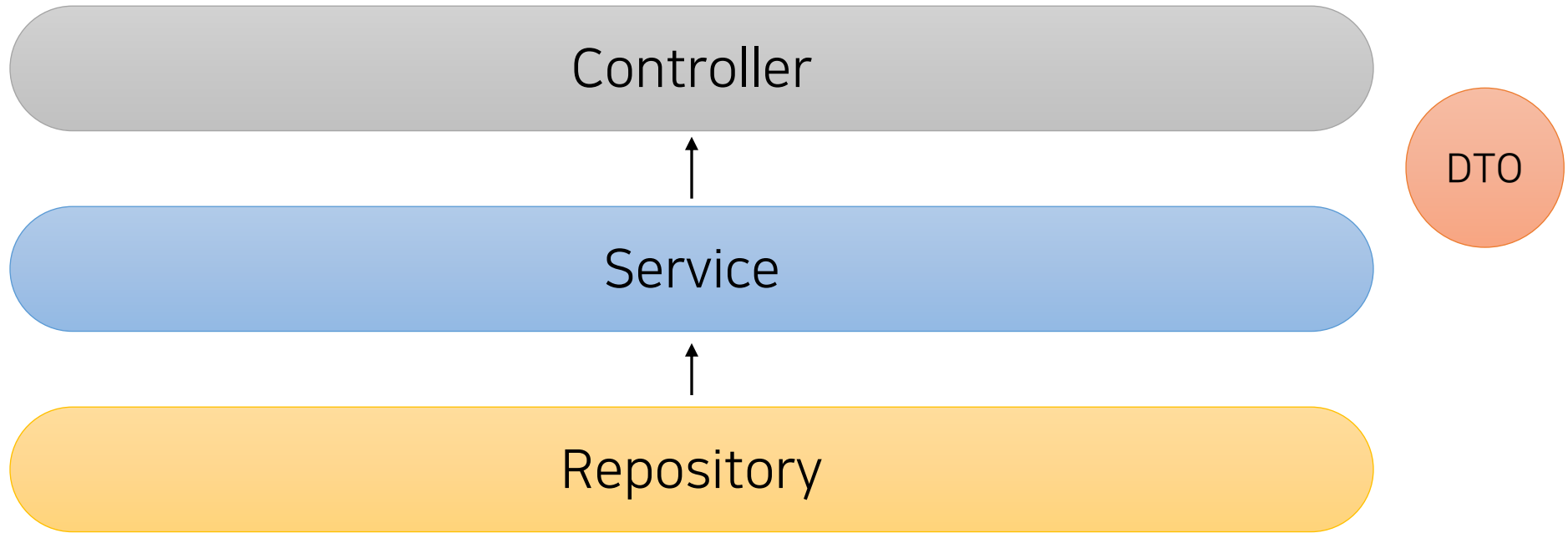
## Service의 역할

# Controller의 함수 1개가 하고 있던 역할

1. API의 진입 지점으로써 HTTP Body를 객체로 변환하고 있다.
2. 현재 유저가 있는지, 없는지 등을 확인하고 예외 처리를 해준다.
3. SQL을 사용해 실제 DB와의 통신을 담당한다.

Repository의 역할

# 3가지 역할로 구분된 구조



어려운 용어로 **Layered Architecture**라고 한다!



**이제 DELETE, POST, GET API도 변경해보자!**

**한 가지 궁금한 점이 남아 있다.....**

Controller에서 JdbcTemplate은 어떻게 가져온 것일까?!

**한 가지 궁금한 점이 남아 있다.....**

Repository에서 바로 JdbcTemplate을 가져올 수는 없을까?!

# 19강. UserController와 스프링 컨테이너

# UserController의 의아한 점

```
@RestController
public class UserController {

    private final UserService userService;

    public UserController(JdbcTemplate jdbcTemplate) {
        this.userService = new UserService(jdbcTemplate);
    }
}
```

[1] static이 아닌 코드를 사용하려면 **인스턴스화**가 필요하다.

# UserController의 의아한 점

```
@RestController
public class UserController {

    private final UserService userService;

    public UserController(JdbcTemplate jdbcTemplate) {
        this.userService = new UserService(jdbcTemplate);
    }
}
```

도대체 누가 UserController를 **인스턴스화** 하고 있는것인가?!

# UserController의 의아한 점

```
@RestController
public class UserController {

    private final UserService userService;

    public UserController(JdbcTemplate jdbcTemplate) {
        this.userService = new UserService(jdbcTemplate);
    }
}
```

[2] UserController는 JdbcTemplate이 필요하다.

# UserController의 의존한 점

```
@RestController
public class UserController {

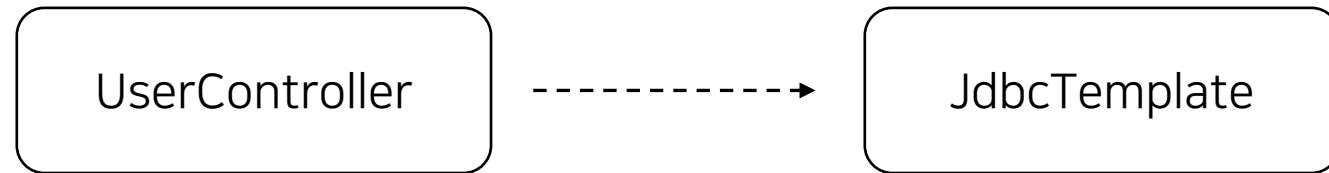
    private final UserService userService;

    public UserController(JdbcTemplate jdbcTemplate) {
        this.userService = new UserService(jdbcTemplate);
    }
}
```

즉, UserController는 JdbcTemplate에 의존하고 있다.



# UserController는 JdbcTemplate에 의존하고 있다.



UserController는 JdbcTemplate이 없으면 동작하지 않는다!

# UserController의 의아한 점

```
@RestController
public class UserController {

    private final UserService userService;

    public UserController(JdbcTemplate jdbcTemplate) {
        this.userService = new UserService(jdbcTemplate);
    }
}
```

그런데 우리는 JdbcTemplate이란 클래스를 설정해준 적이 없다.

# UserController의 의아한 점

```
@RestController
public class UserController {

    private final UserService userService;

    public UserController(JdbcTemplate jdbcTemplate) {
        this.userService = new UserService(jdbcTemplate);
    }
}
```

UserController는 어떻게 JdbcTemplate을 가져올 수 있었을까?

# 비밀은 바로..

```
@RestController  
public class UserController {  
  
    private final UserService userService;  
  
    public UserController(JdbcTemplate jdbcTemplate) {  
        this.userService = new UserService(jdbcTemplate);  
    }  
}
```

UserController 클래스를 API의 진입지점으로 만들 뿐 아니라,

# 비밀은 바로..

```
@RestController  
public class UserController {  
  
    private final UserService userService;  
  
    public UserController(JdbcTemplate jdbcTemplate) {  
        this.userService = new UserService(jdbcTemplate);  
    }  
}
```

UserController 클래스를 스프링 빈으로 등록시킨다!

# 스프링 빈이란?!!

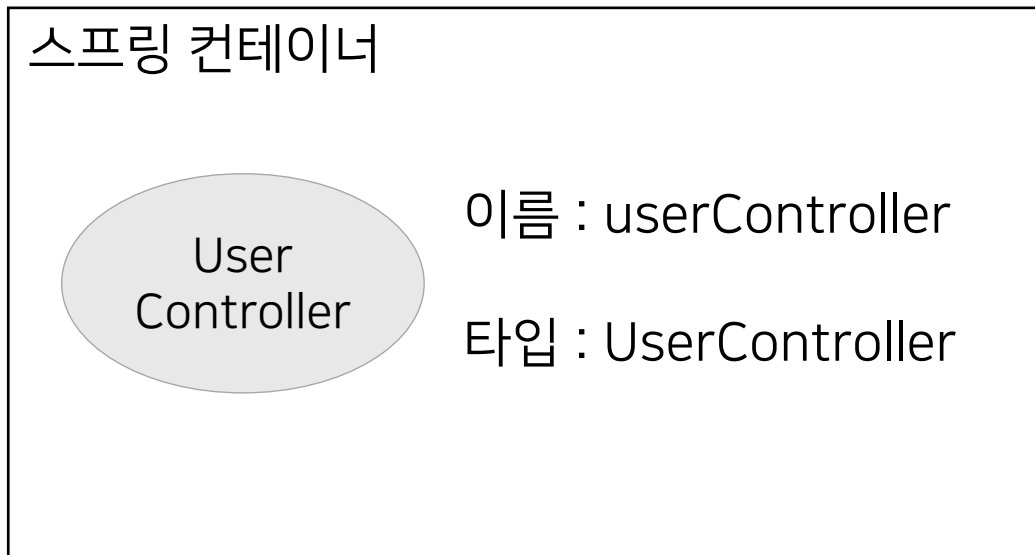
서버가 시작되면, 스프링 서버 내부에 거대한 컨테이너를 만들게 된다!



# 스프링 빈이란?!!

서버가 시작되면, 스프링 서버 내부에 거대한 컨테이너를 만들게 된다!

컨테이너 안에는 클래스가 들어가게 된다!!

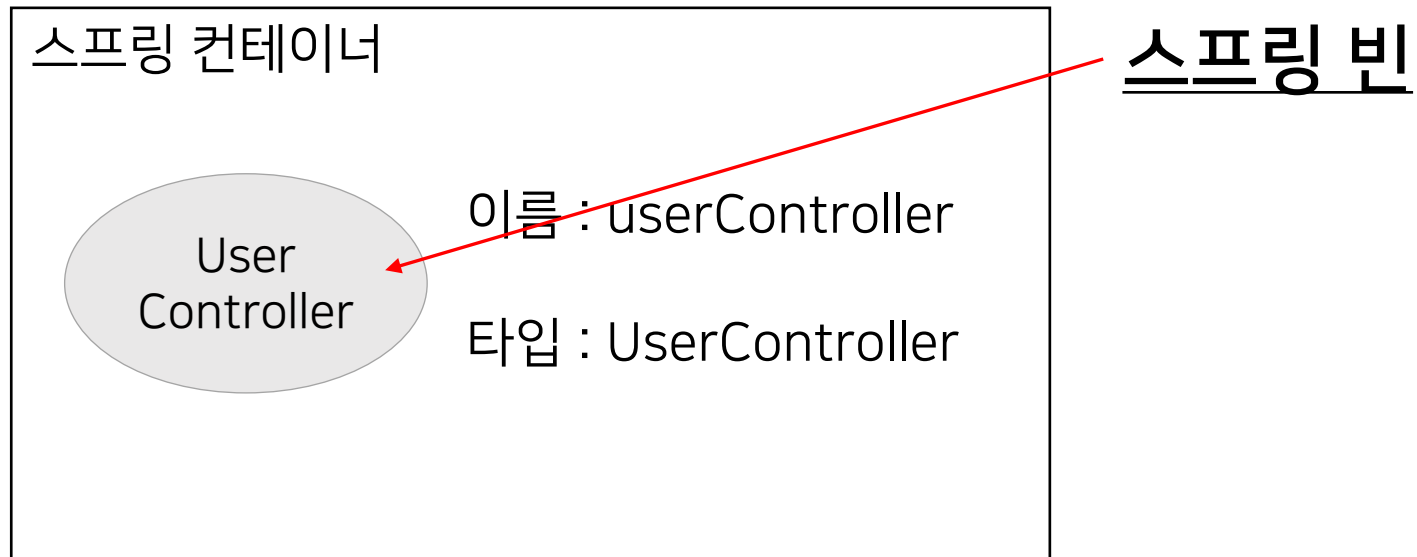


이때 다양한 정보도 함께 들어있고,  
인스턴스화도 이루어진다!

# 스프링 빈이란?!!

서버가 시작되면, 스프링 서버 내부에 거대한 컨테이너를 만들게 된다!

컨테이너 안에는 클래스가 들어가게 된다!!

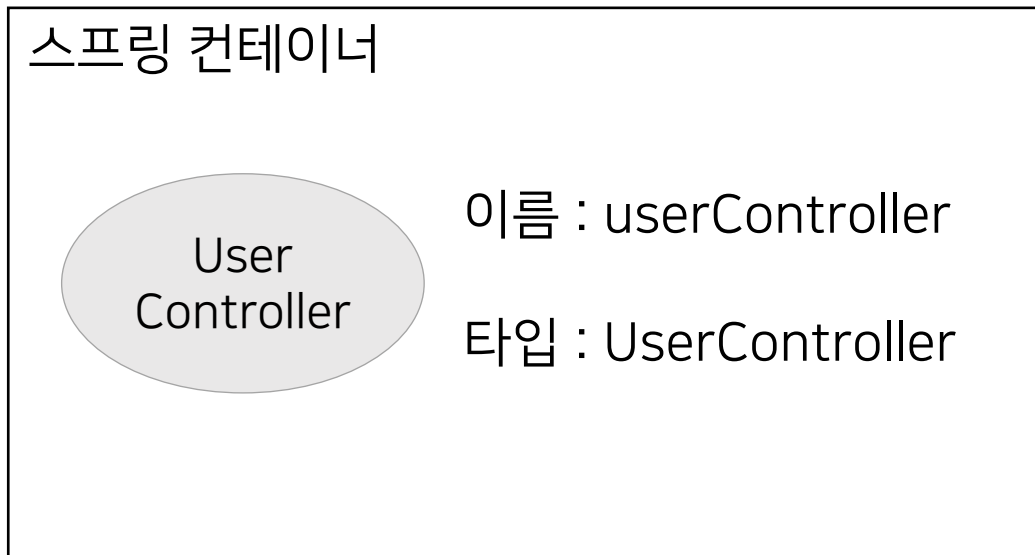




# 스프링 빈이란?!!

서버가 시작되면, 스프링 서버 내부에 거대한 컨테이너를 만들게 된다!

컨테이너 안에는 클래스가 들어가게 된다!!

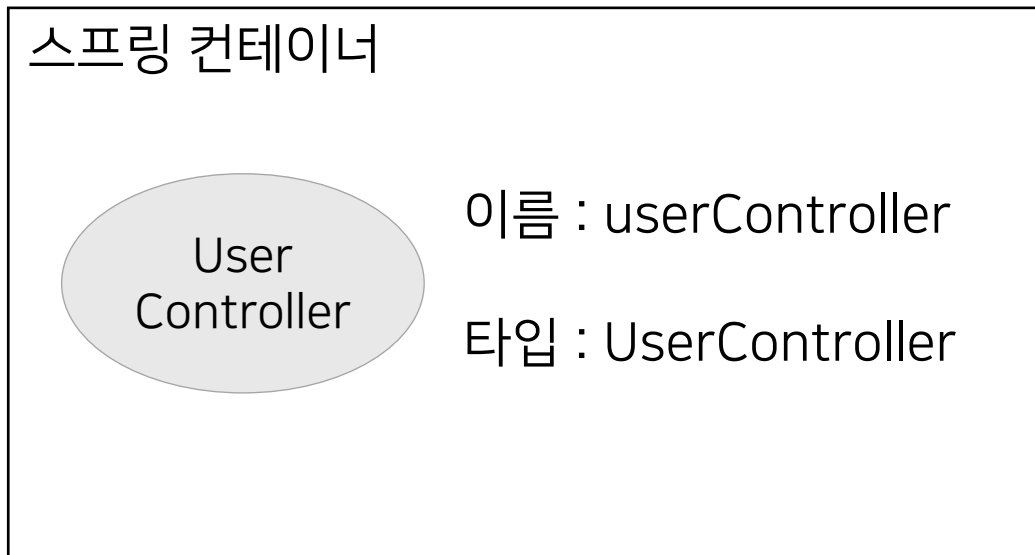


잠깐! 그런데 UserController를  
인스턴스화 하려면  
**JdbcTemplate**이 필요하다!

# 스프링 빈이란?!!

서버가 시작되면, 스프링 서버 내부에 거대한 컨테이너를 만들게 된다!

컨테이너 안에는 클래스가 들어가게 된다!!

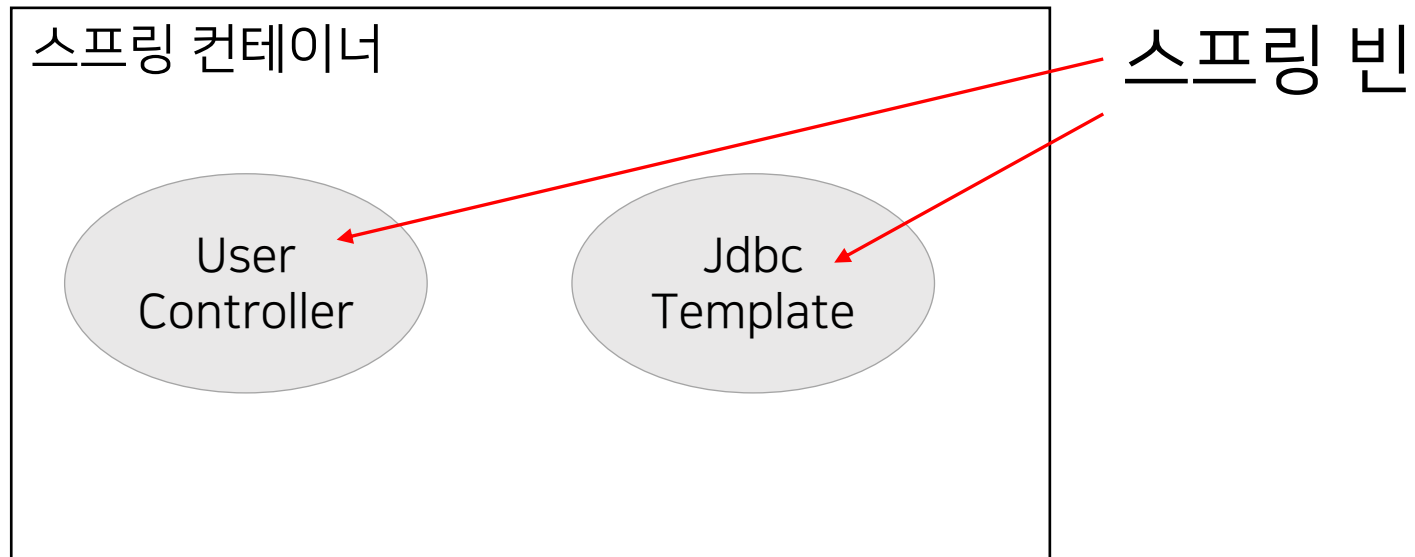


사실 이 JdbcTemplate도  
스프링 빈으로 등록되어 있다!

# 스프링 빈이란?!!

서버가 시작되면, 스프링 서버 내부에 거대한 컨테이너를 만들게 된다!


컨테이너 안에는 클래스가 들어가게 된다!!



# JdbcTemplate은 누가 스프링 빈으로 등록해주었지?!

```
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'  
}
```

# JdbcTemplate은 누가 스프링 빈으로 등록해주었지?!

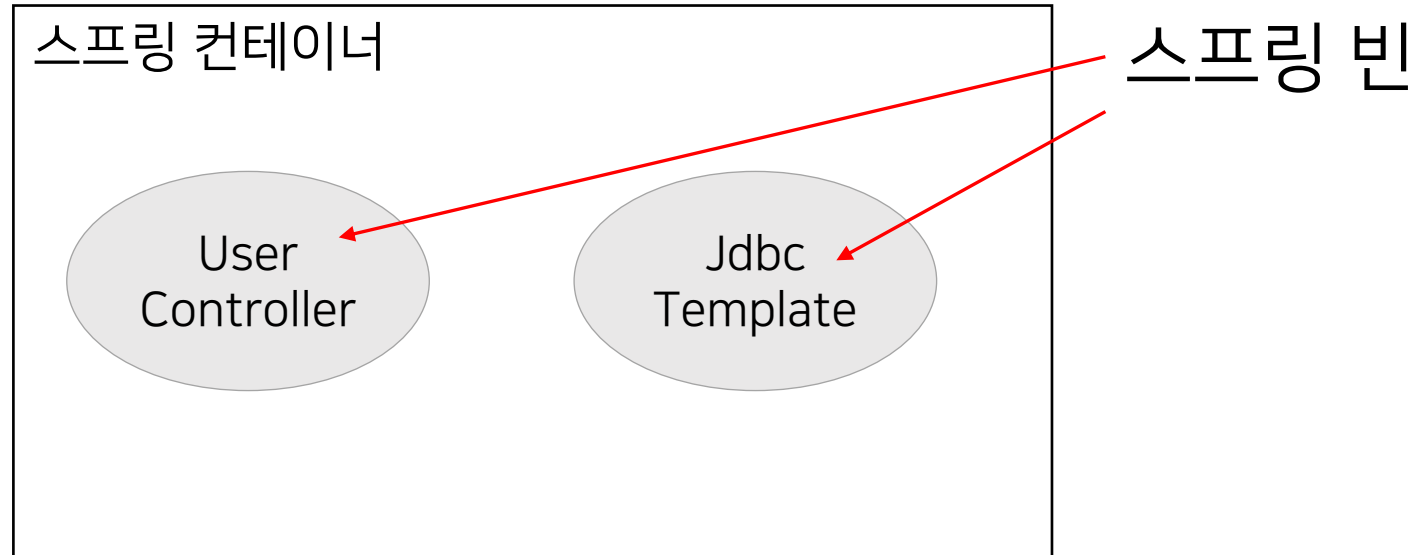


```
public UserController(JdbcTemplate jdbcTemplate) {  
    this.userService = new UserService(jdbcTemplate);  
}
```

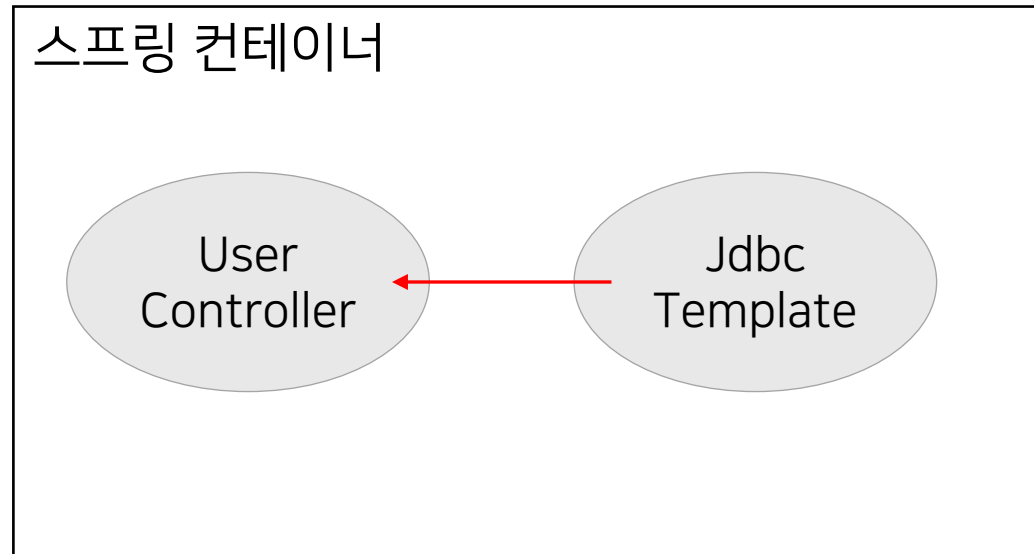
# JdbcTemplate은 누가 스프링 빈으로 등록해주었지?!

우리가 가져온 Dependency가 등록해주고 있었다!

# 스프링 컨테이너는 필요한 클래스를 연결해준다!



# 스프링 컨테이너는 필요한 클래스를 연결해준다!





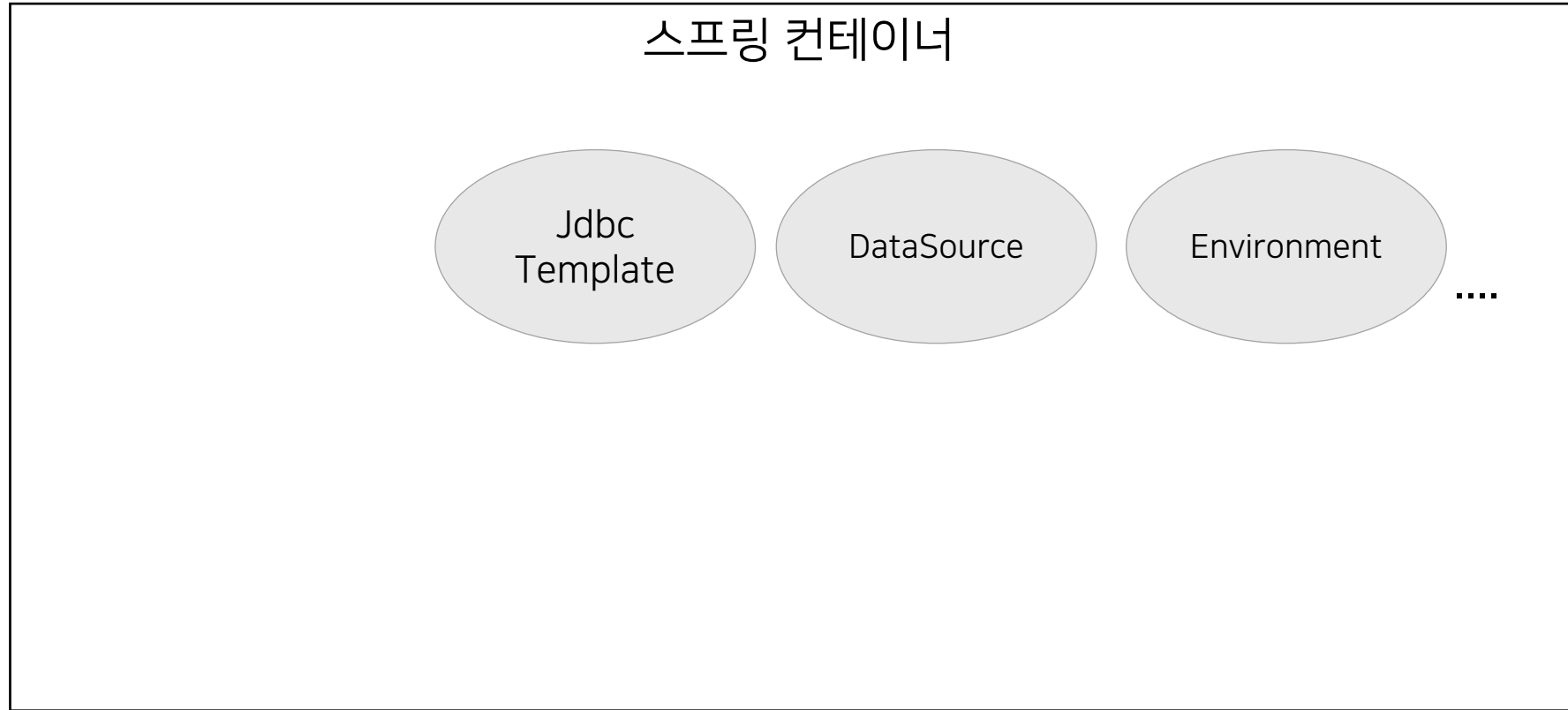
**처음부터 정리해보자!**

# 서버가 시작되면 다음과 같은 일이 일어난다.

스프링 컨테이너

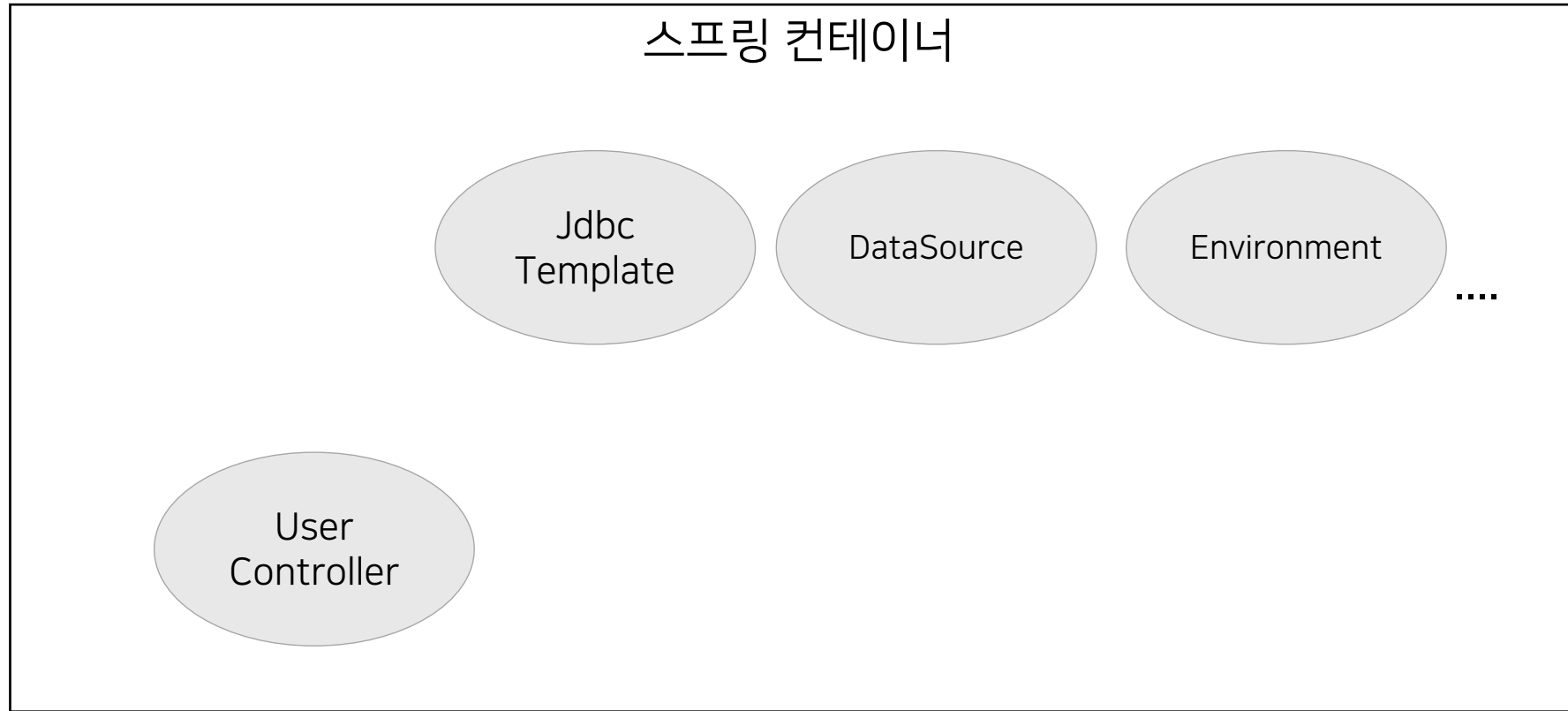
[1] 스프링 컨테이너(클래스 저장소)가 시작된다!

# 서버가 시작되면 다음과 같은 일이 일어난다.



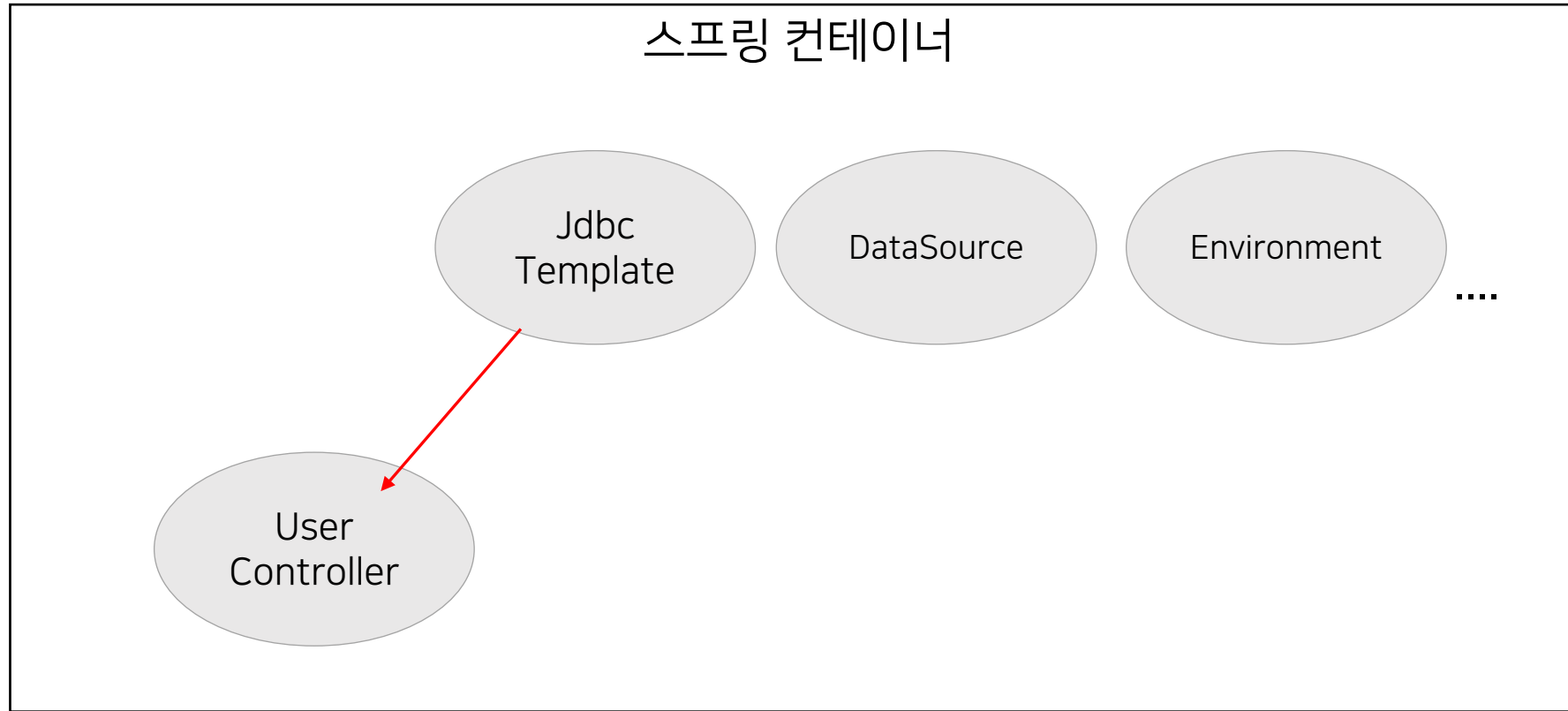
[2] 기본적으로 많은 스프링 빈들이 등록된다!

# 서버가 시작되면 다음과 같은 일이 일어난다.



[3] 우리가 설정해준 스프링 빈이 등록된다!

# 서버가 시작되면 다음과 같은 일이 일어난다.



[4] 이때 필요한 의존성이 자동으로 설정된다!

# 그렇다면 왜 UserRepository는 JdbcTemplate을 가져오지 못할까?!

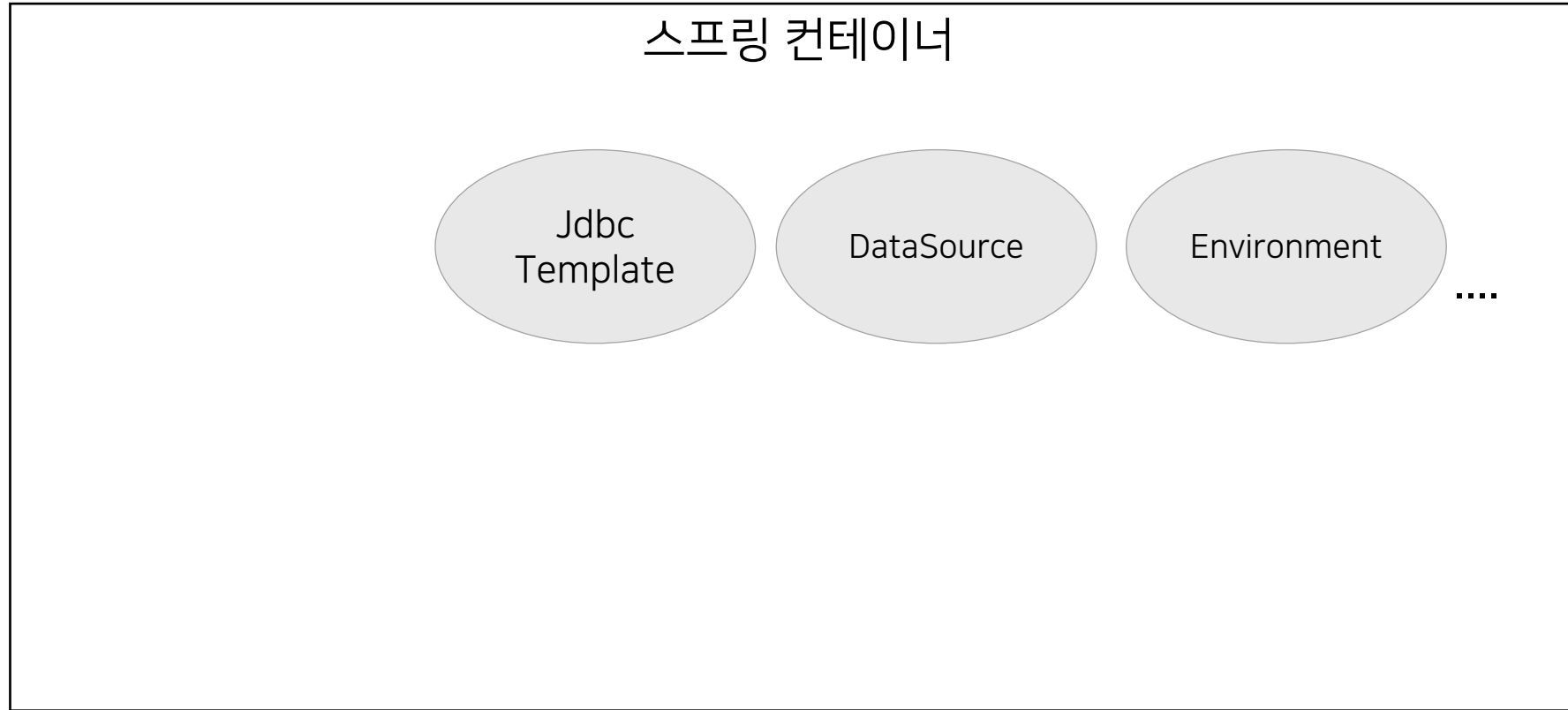
JdbcTemplate을 가져오려면 UserRepository가 스프링 빈이어야 하는데  
UserRepository는 스프링 빈이 아니다!!!

**UserRepository를 스프링 빈으로 등록하자!**

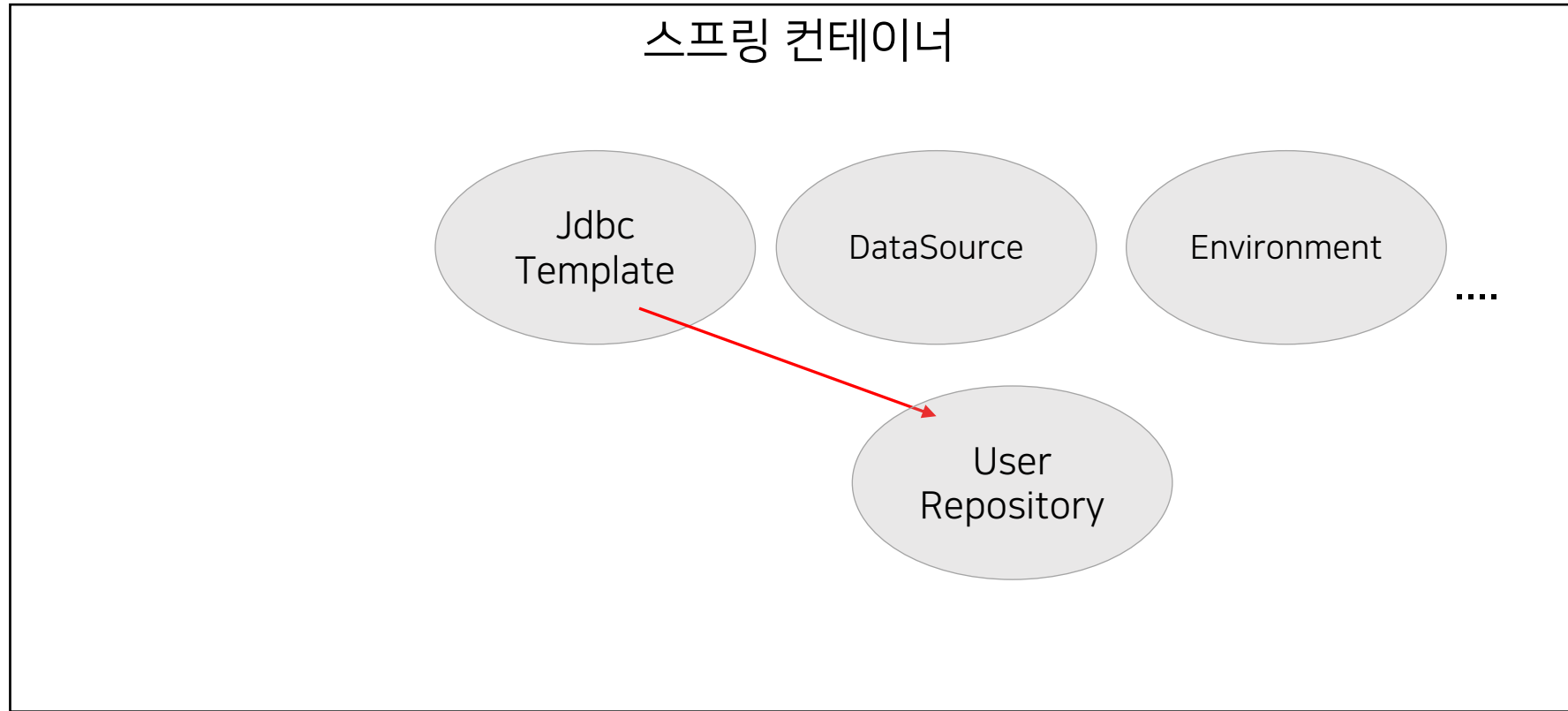
**UserService도 스프링 빈으로 등록하자!**



# 이제 3개의 클래스는 서버가 시작할 때 다음 순서를 따른다!

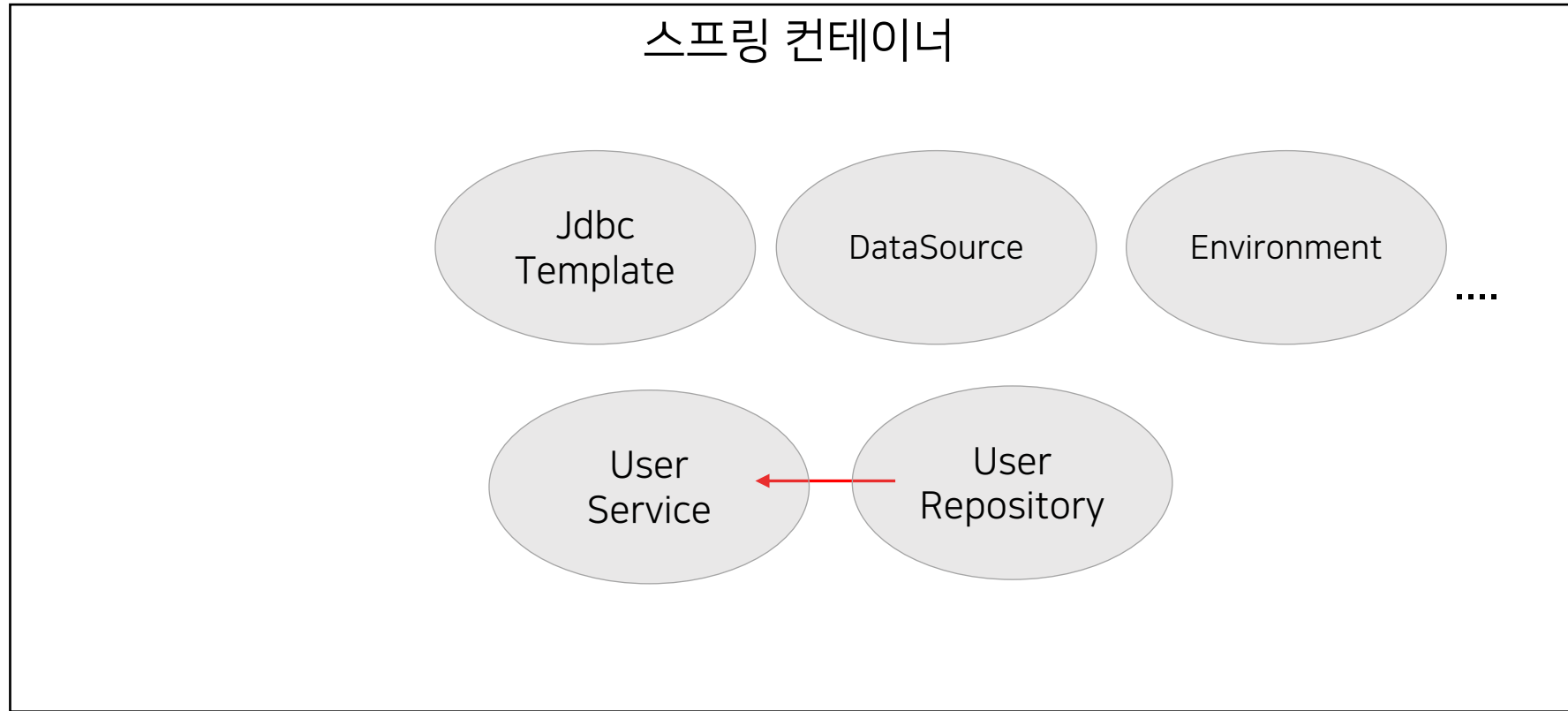


# 이제 3개의 클래스는 서버가 시작할 때 다음 순서를 따른다!



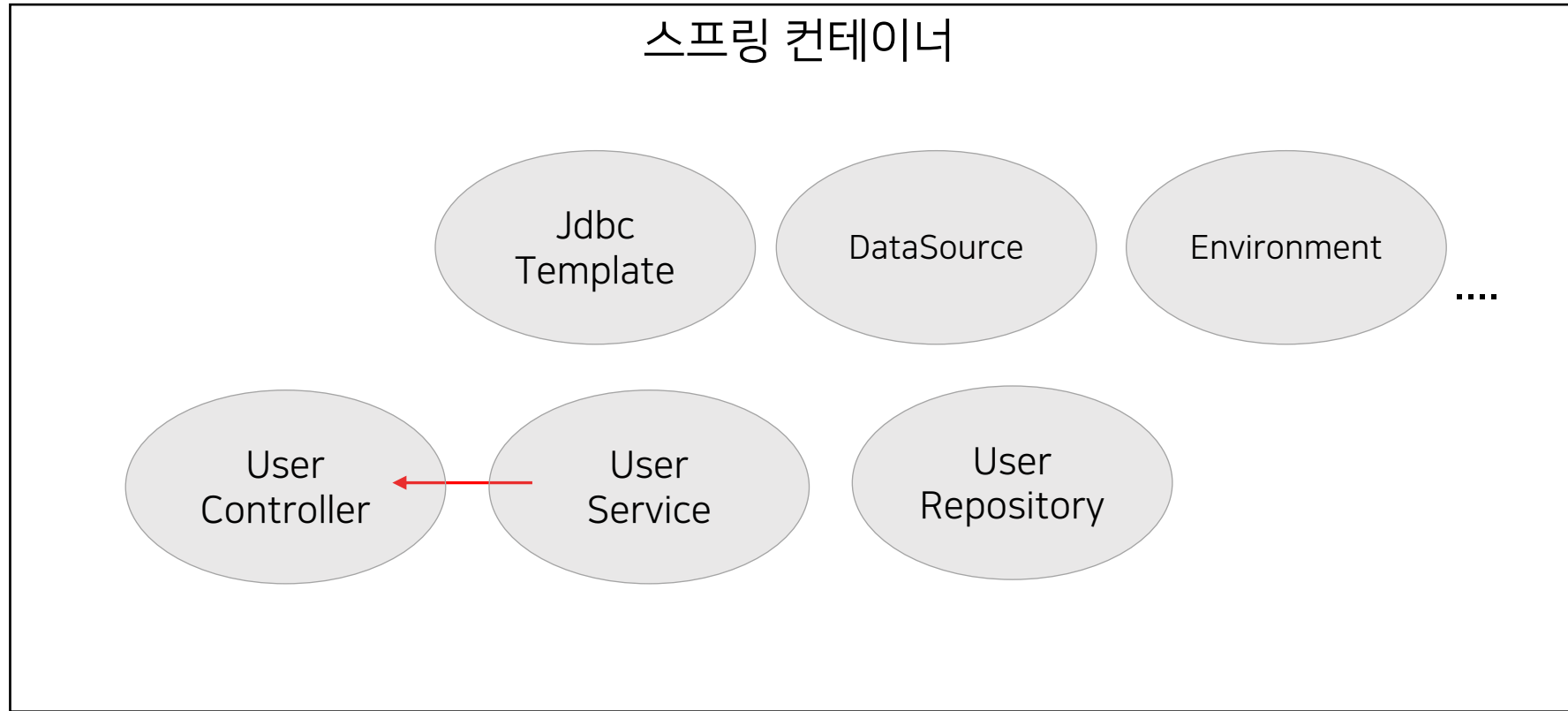
JdbcTemplate을 의존하는 UserRepository가  
스프링 빈으로 등록된다.

# 이제 3개의 클래스는 서버가 시작할 때 다음 순서를 따른다!



UserRepository를 의존하는 UserService가  
스프링 빈으로 등록된다.

# 이제 3개의 클래스는 서버가 시작할 때 다음 순서를 따른다!



UserService를 의존하는 UserController가  
스프링 빈으로 등록된다.

**자 그런데 한 가지 궁금한 점이 자연스럽게 떠오른다.**

뭔가 더 좋아진 것 같긴한데...

스프링 컨테이너를 왜 사용하는 걸까?! 그냥 new 연산자를 쓰면 안되나?!

# 20강. 스프링 컨테이너를 왜 사용할까?!

## 다음 요구사항을 생각해보자.

책 이름을 메모리에 저장하는 API를 매우 간단하게 구현하라!  
Service, Repository는 Spring Bean이 아니어야 한다.

# 구현된 그림

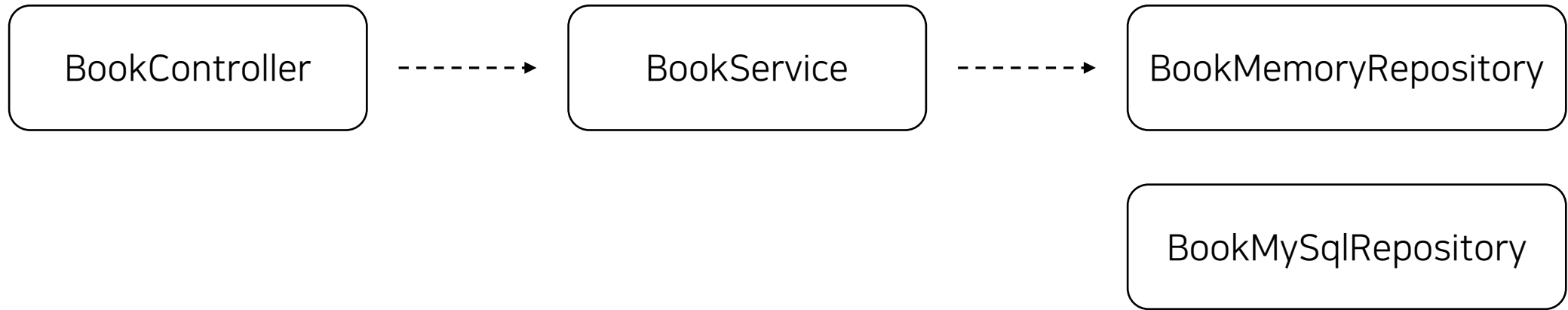




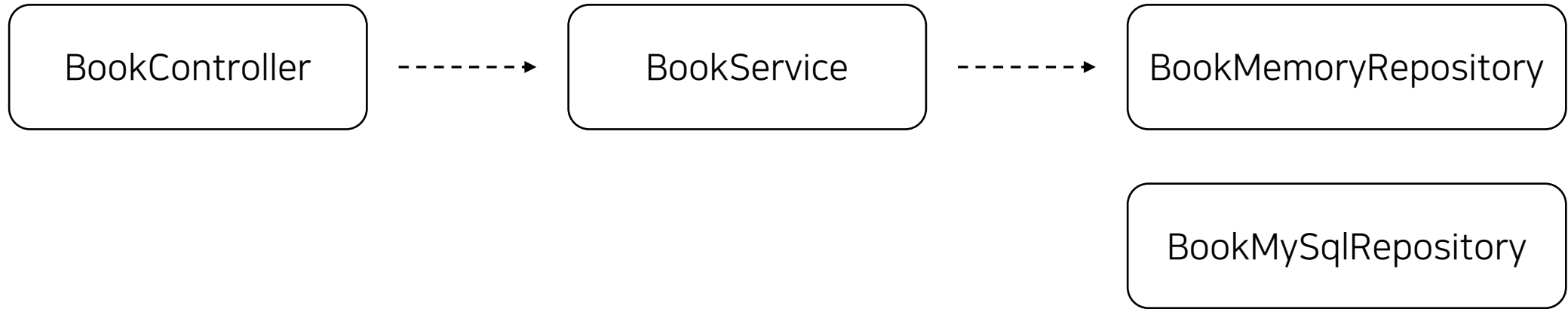
# 추가 요구사항을 구현하는 상상을 해보자!

이제 Memory가 아니라 MySQL과 같은 DB를 사용해야 한다!  
JdbcTemplate은 Repository가 바로 설정할 수 있다고 해보자!

# 구현된 그림

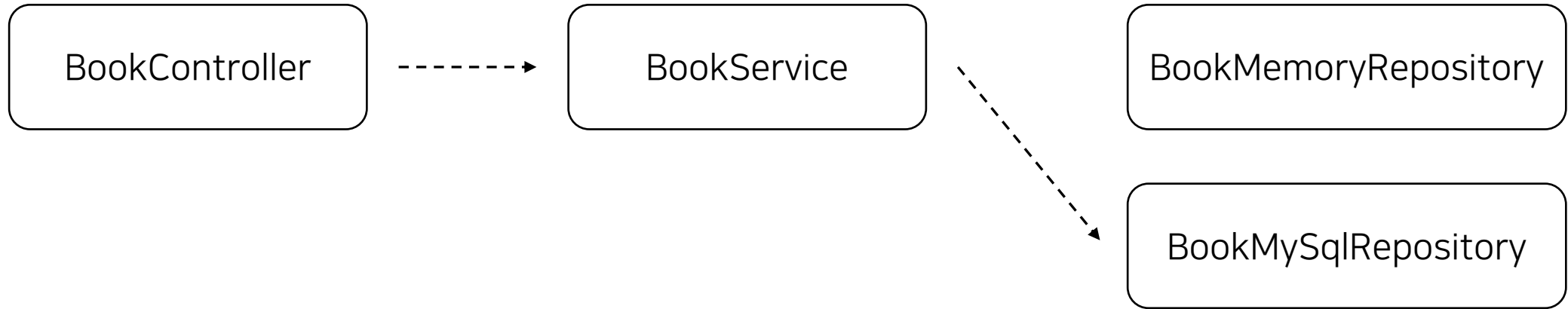


# 구현된 그림



```
public class BookService {  
    private final BookMemoryRepository bookRepository = new BookMemoryRepository();  
}
```

# 구현된 그림



```
public class BookService {  
  
    // private final BookMemoryRepository bookRepository = new BookMemoryRepository();  
    private final BookMySqlRepository bookRepository = new BookMySqlRepository();  
  
}
```

# 잘 하긴 했는데... 어딘가 부족하다!!

데이터를 메모리에 저장할지~ MySQL에 저장할지~  
Repository의 역할에 관련된 것만 바꾸고 싶은데  
BookService까지 바꿔야 한다!

**잘 하긴 했는데... 어딘가 부족하다!!**

어떻게 하면 Repository를 다른 Class로 바꾸더라도  
BookService를 변경하지 않을 수 있을까?!

**잘 하긴 했는데... 어딘가 부족하다!!**

Java의 interface를 활용해보자!!

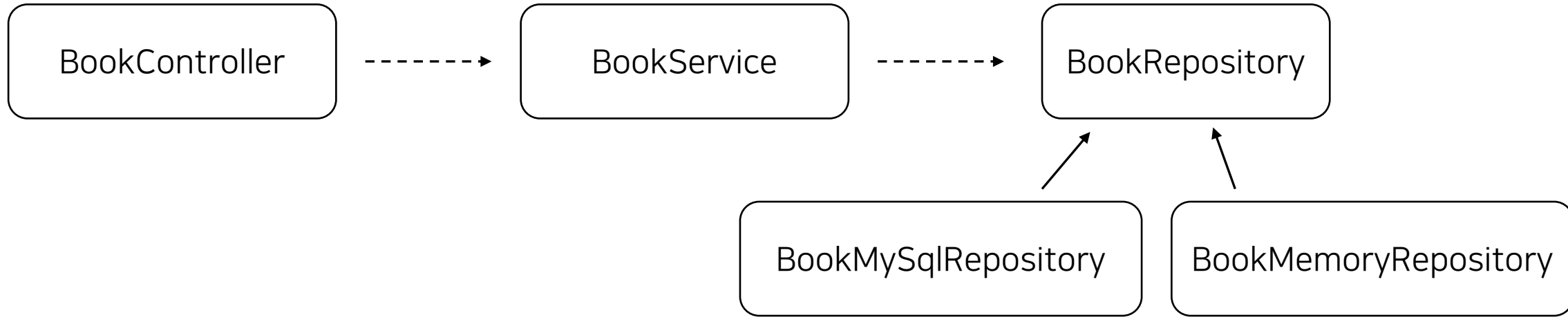
# 구현된 그림



```
public class BookService {  
  
    private final BookRepository bookRepository = new BookMemoryRepository();  
  
}
```



# 구현된 그림



```
public class BookService {  
  
    private final BookRepository bookRepository = new BookMySqlRepository(); // new BookMemoryRepository();  
  
}
```

**발전 했지만... 어딘가 부족하다!!**

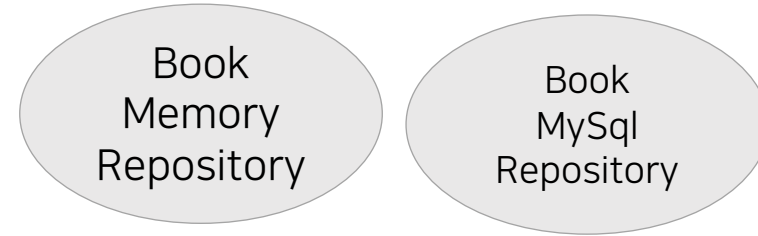
BookService의 변경 범위가 줄어들긴 했지만, 아직은 아쉽다!

**그래서 등장한 스프링 컨테이너!**

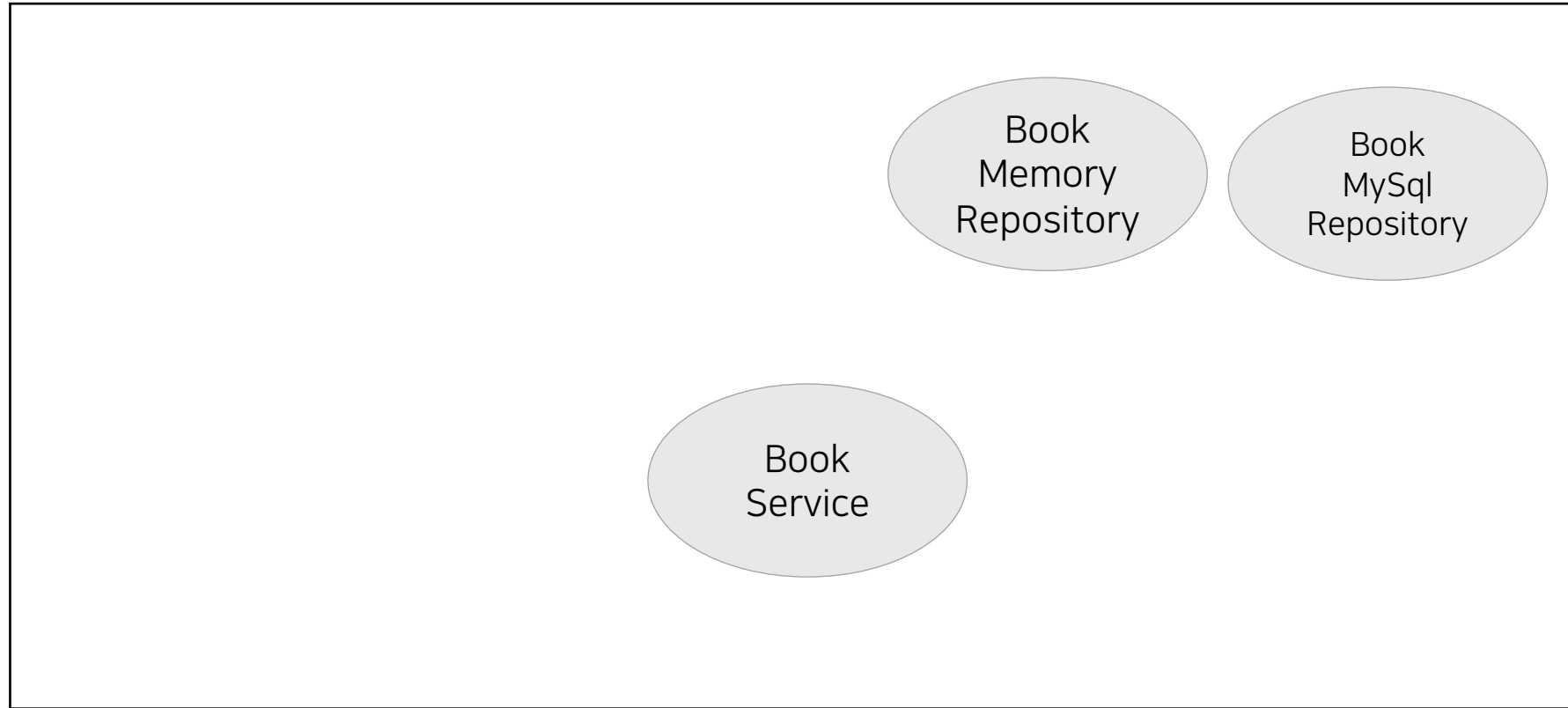
# 스프링 컨테이너를 사용하면

컨테이너가 BookService를 대신 인스턴스화 하고,  
그 때 알아서 BookRepository를 결정해준다!

# 스프링 컨테이너를 사용하면

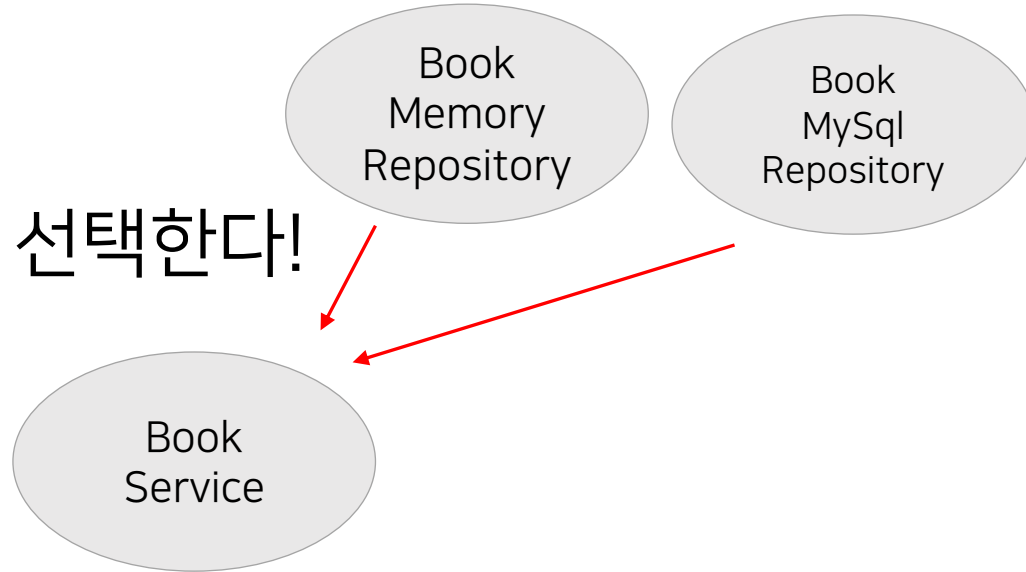


# 스프링 컨테이너를 사용하면

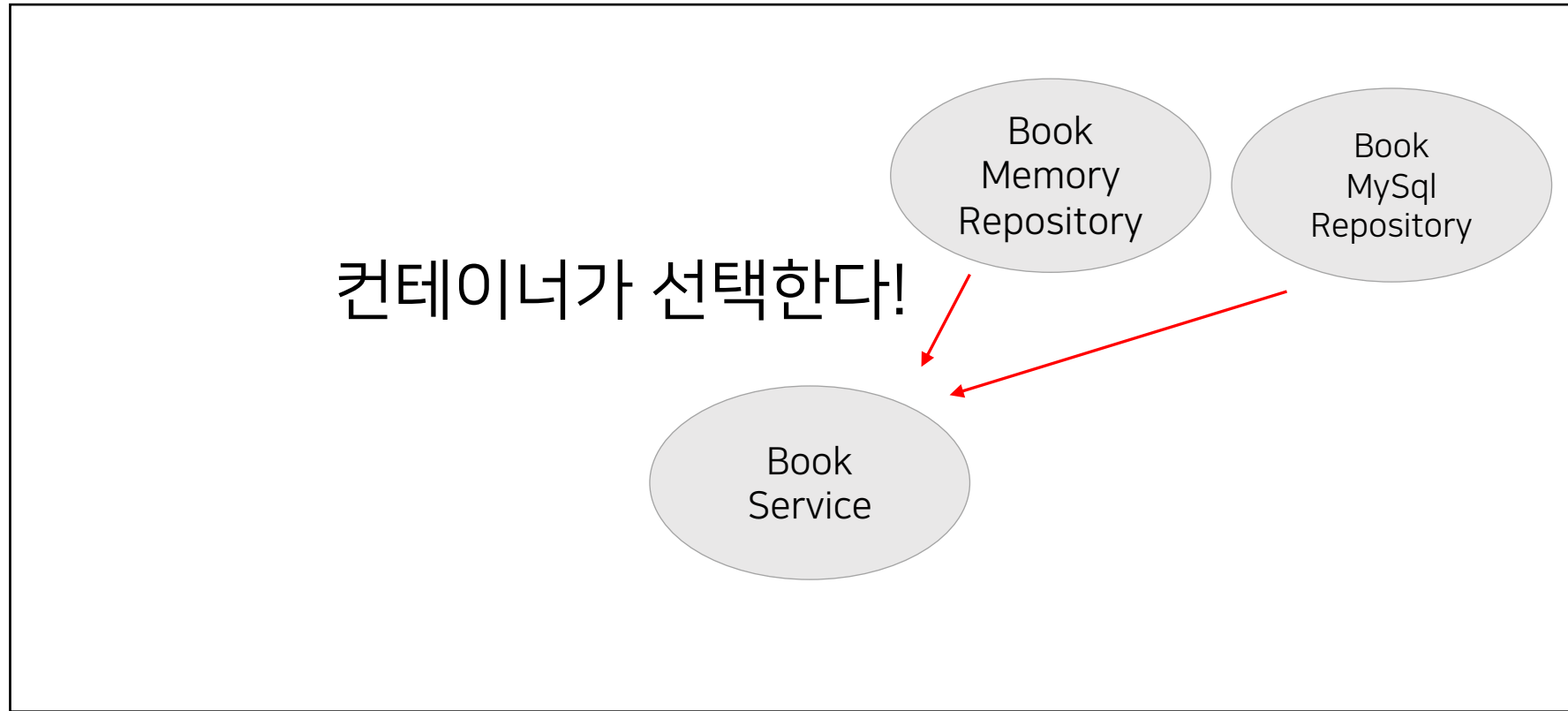


# 스프링 컨테이너를 사용하면

컨테이너가 선택한다!



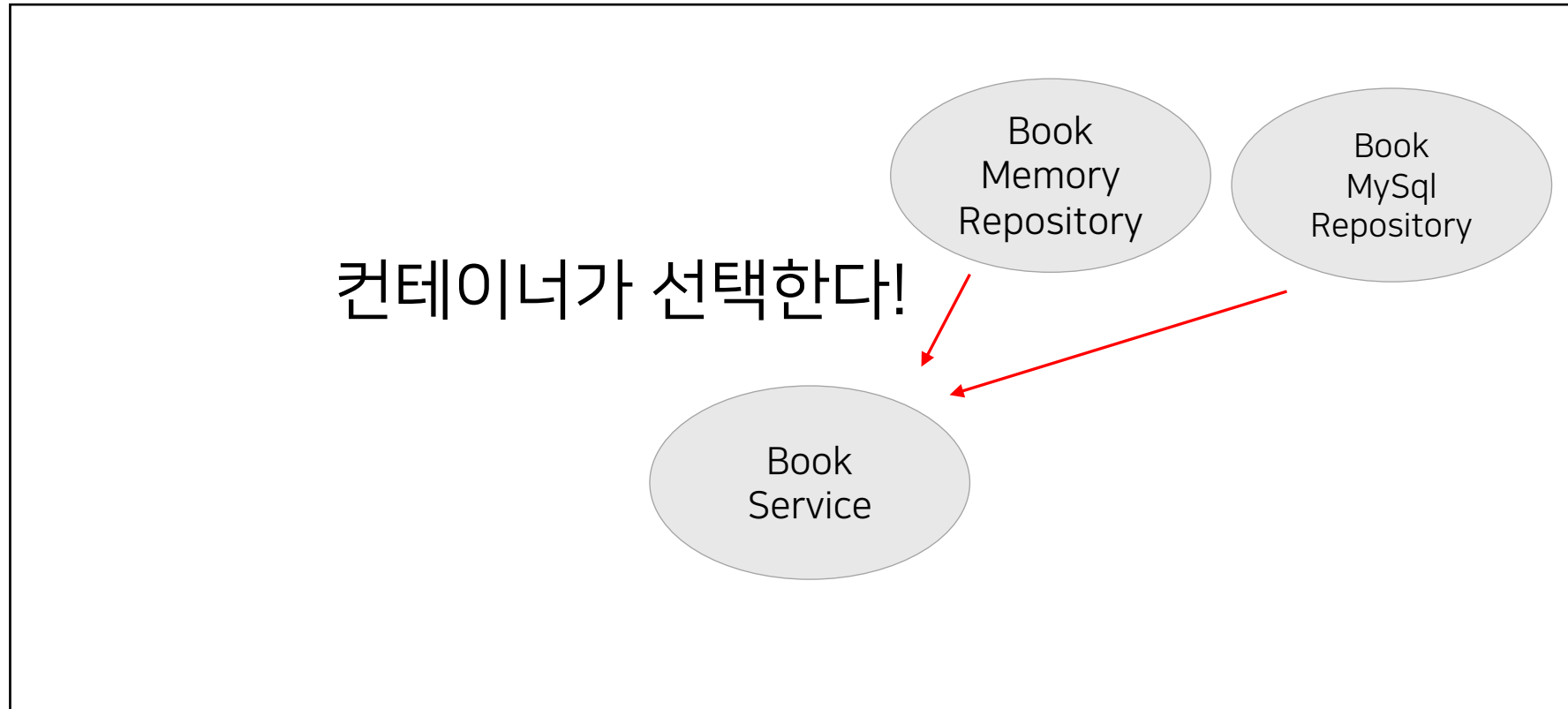
# 스프링 컨테이너를 사용하면



이런 방식을 **제어의 역전**(IoC, Inversion of Control)이라 한다.



# 스프링 컨테이너를 사용하면



컨테이너가 선택해 BookService에 넣어주는 과정을  
**의존성 주입**(DI, Dependency Injection)라고 한다.

**그렇다면 둘 중 어떤 Repository가 주입될까?!**

우리가 @Primary를 활용해서 조절할 수 있다!!

# 그렇다면 둘 중 어떤 Repository가 주입될까?!

@Primary : 우선권을 결정하는 어노테이션

# 21강. 스프링 컨테이너를 다루는 방법

# 빈을 등록하는 방법

@Configuration

- 클래스에 붙이는 어노테이션
- @Bean을 사용할 때 함께 사용해 주어야 한다!

@Bean

# 빈을 등록하는 방법

## @Configuration

- 클래스에 붙이는 어노테이션
- @Bean을 사용할 때 함께 사용해 주어야 한다!

## @Bean

- 메소드에 붙이는 어노테이션
- 메소드에서 반환되는 객체를 스프링 빈에 등록한다.

**UserRepository에 @Bean을 사용해보자!**

**언제 @Service, @Repository를 사용해야 할까?!**



# 언제 @Service, @Repository를 사용해야 할까?!

개발자가 직접 만든 클래스를 스프링 빈으로 등록할 때!

# 언제 @Configuration + @Bean을 사용해야 할까?!

외부 라이브러리, 프레임워크에서 만든 클래스를 등록할 때!

# 그렇다면 UserService, UserRepository는?!

@Service, @Repository를 사용하는 것이 좋다.

# 다음으로 살펴볼 어노테이션은

@Component

- 주어진 클래스를 '컴포넌트'로 간주한다.
- 이 클래스들은 스프링 서버가 뜰 때 자동으로 감지된다.

**사실 이 @Component는 지금까지 숨어있었다!!**

@Component 덕분에 우리가 사용했던 어노테이션이 자동감지 되었다!

# @Component는 언제 사용하는가?!

- 1) 컨트롤러, 서비스, 리포지토리가 모두 아니고
- 2) 개발자가 직접 작성한 클래스를  
스프링 빈으로 등록할 때 사용되기도 한다.

**이제 스프링 빈을 주입 받는 방법을 살펴보자!**

# 스프링 빈을 주입 받는 몇 가지 방법

(가장 권장) 생성자를 이용해 주입받는 방식



# 스프링 빈을 주입 받는 몇 가지 방법

두 번째 방법 - setter와 @Autowired 사용

# 스프링 빈을 주입 받는 몇 가지 방법

```
private JdbcTemplate jdbcTemplate;  
  
@Autowired  
public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {  
    this.jdbcTemplate = jdbcTemplate;  
}
```

# 스프링 빈을 주입 받는 몇 가지 방법

세 번째 방법 - 필드에 직접 @Autowired 사용

# 스프링 빈을 주입 받는 몇 가지 방법

```
@Autowired  
private JdbcTemplate jdbcTemplate;
```

# 스프링 빈을 주입 받는 몇 가지 방법 정리

1. 생성자 사용 (@Autowired 생략 가능)
2. setter 사용
3. 필드에 바로 사용

# 스프링 빈을 주입 받는 몇 가지 방법 정리

1. 생성자 사용 (@Autowired 생략 가능)
2. setter 사용
3. 필드에 바로 사용

# 스프링 빈을 주입 받는 몇 가지 방법 정리

1. 생성자 사용 (@Autowired 생략 가능)
2. setter 사용 : 누군가 setter를 사용하면 오작동할 수 있다.
3. 필드에 바로 사용

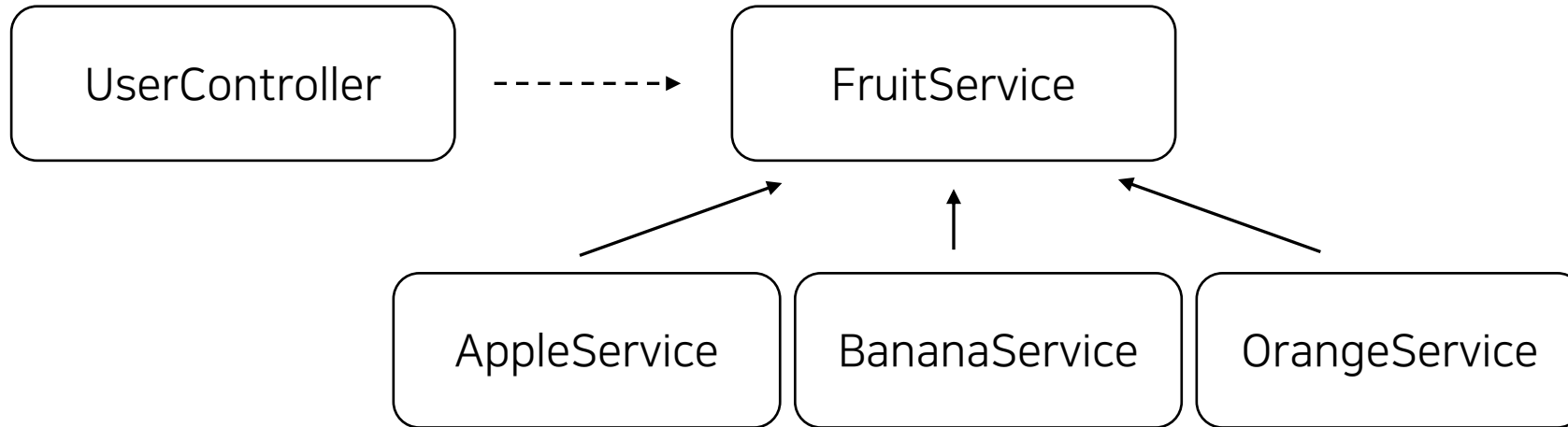
# 스프링 빈을 주입 받는 몇 가지 방법 정리

1. 생성자 사용 (@Autowired 생략 가능)
2. setter 사용 : 누군가 setter를 사용하면 오작동할 수 있다.
3. 필드에 바로 사용 : 테스트를 어렵게 만드는 요인이다.



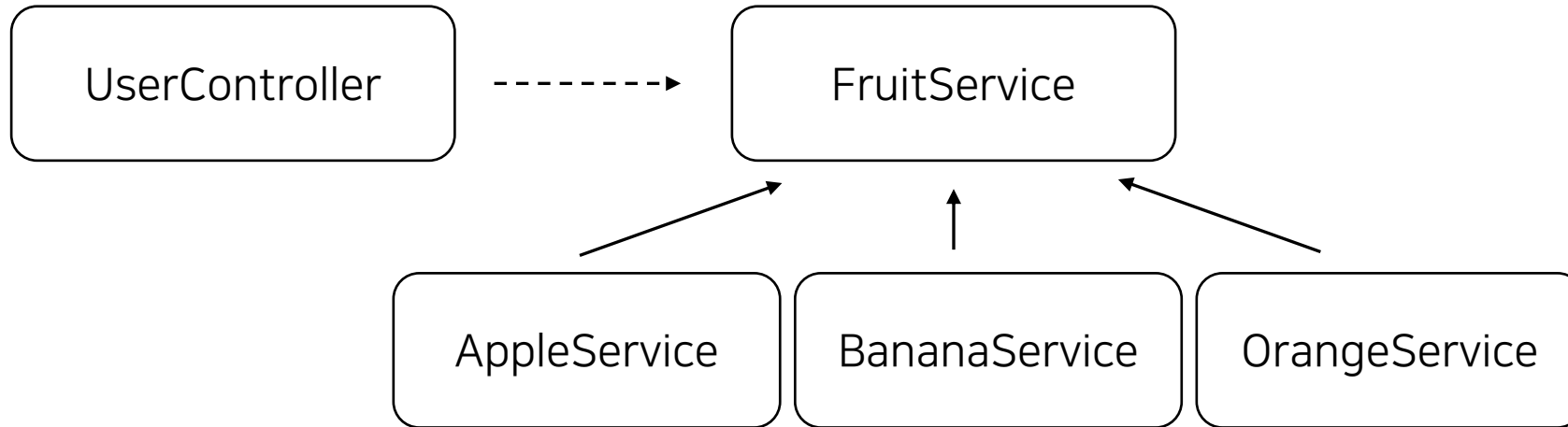
**마지막으로 @Qualifier를 알아보자!**

# 마지막으로 @Qualifier를 알아보자!



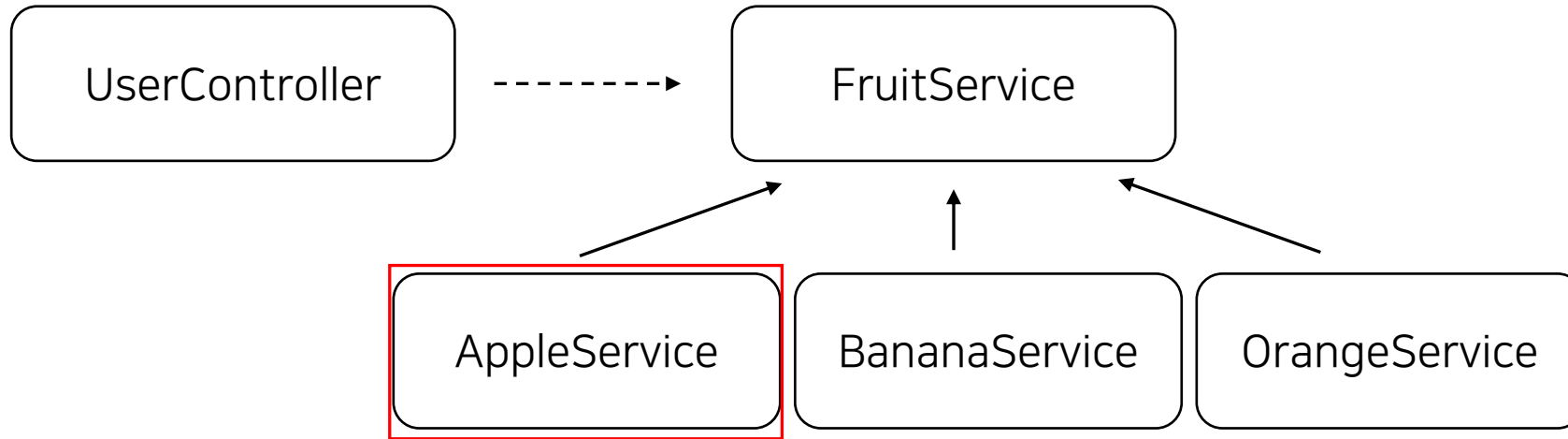
# 마지막으로 @Qualifier를 알아보자!

@Qualifier("appleService")

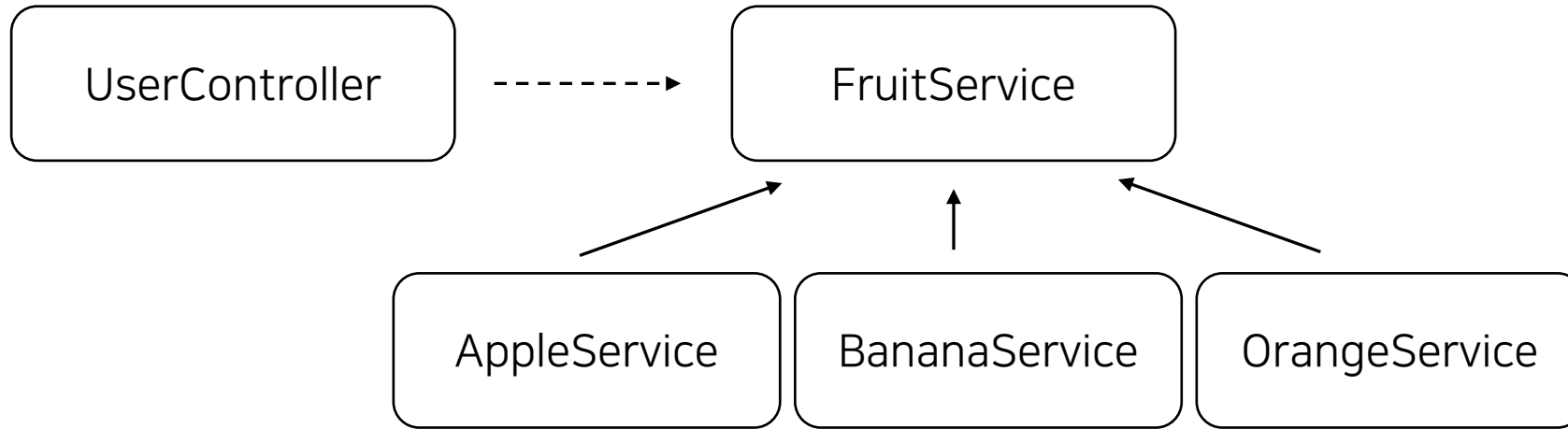


# 마지막으로 @Qualifier를 알아보자!

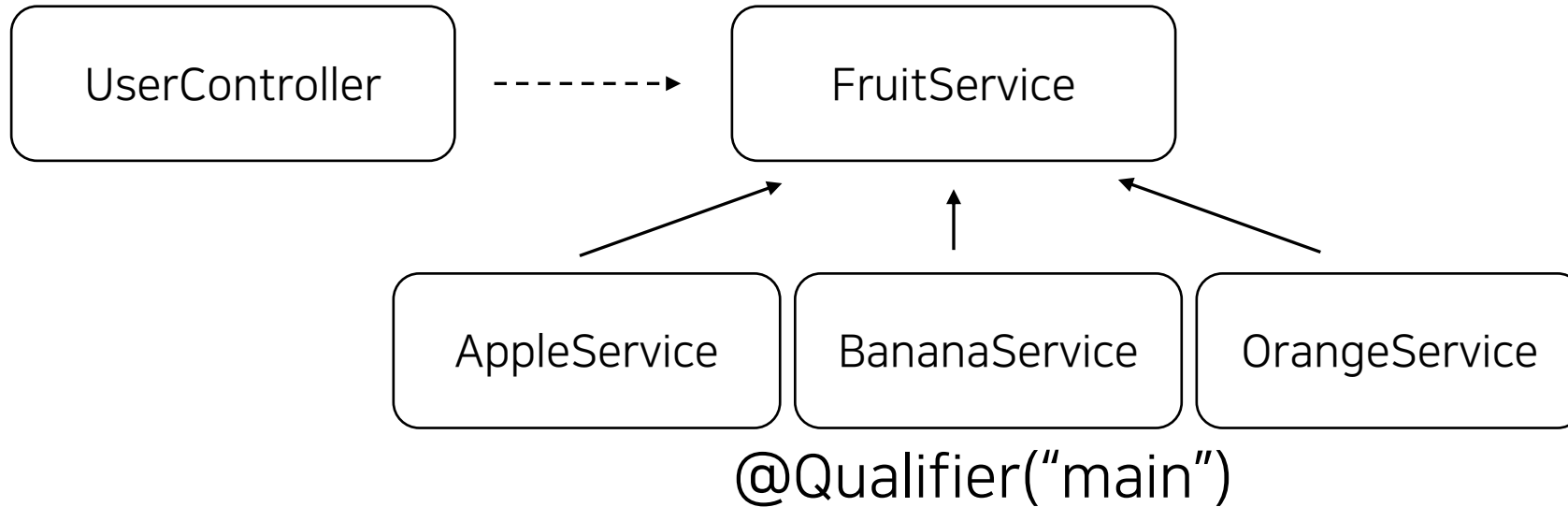
@Qualifier("appleService")



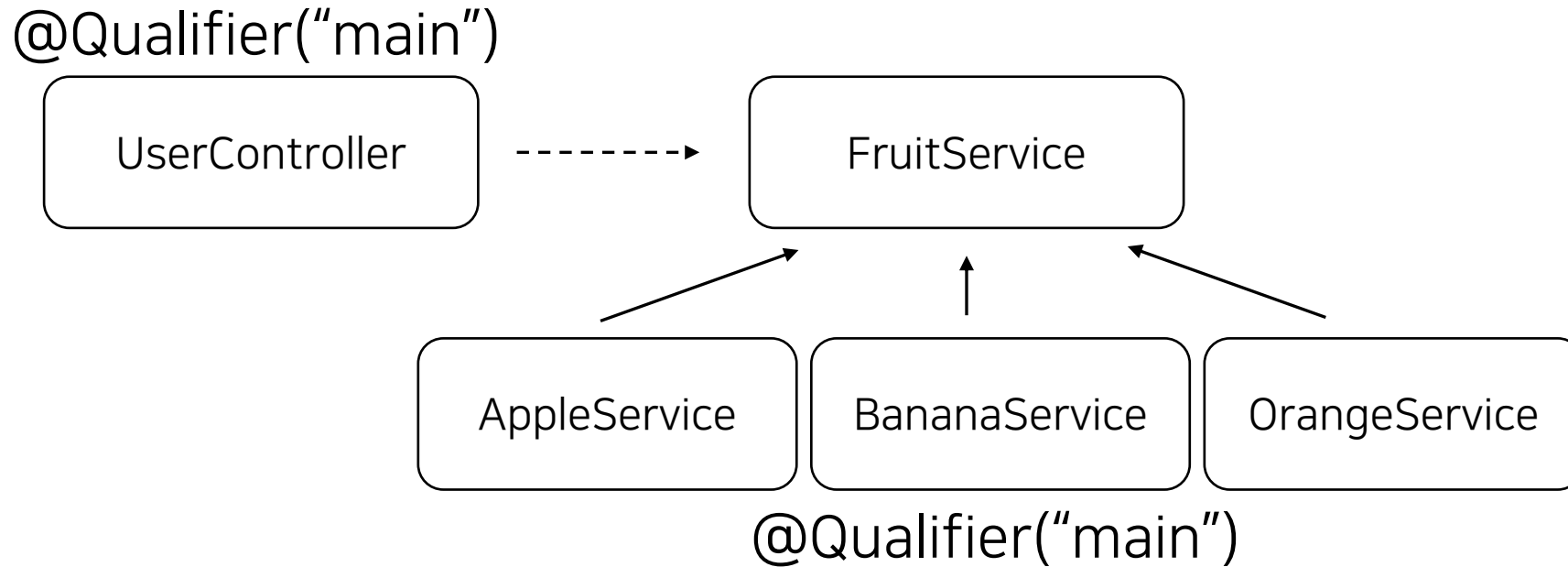
# 마지막으로 @Qualifier를 알아보자!



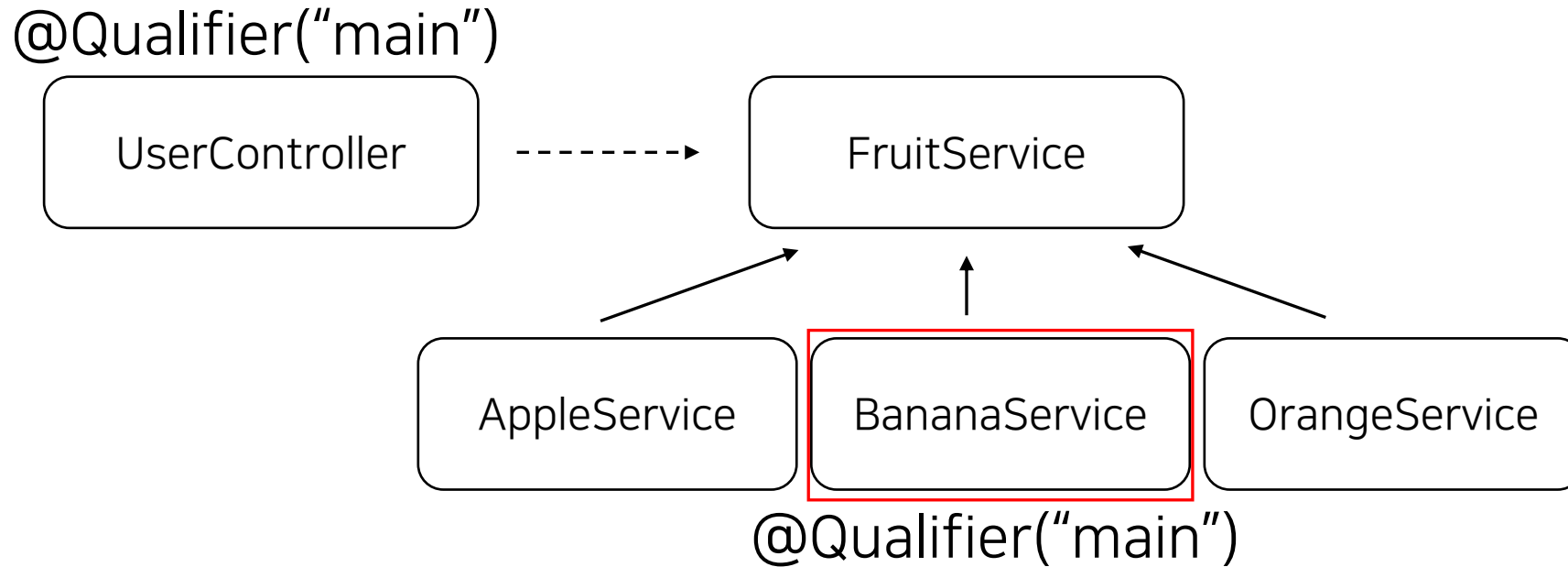
# 마지막으로 @Qualifier를 알아보자!



# 마지막으로 @Qualifier를 알아보자!



# 마지막으로 @Qualifier를 알아보자!





# 마지막으로 @Qualifier를 알아보자!

스프링 빈을 사용하는 쪽, 스프링 빈을 등록하는 쪽 모두 @Qualifier를 사용할 수 있다!

스프링 빈을 사용하는 쪽에서만 쓰면, 빈의 이름을 적어주어야 한다.

양쪽 모두 사용하면, @Qualifier 끼리 연결된다!

# @Primary vs @Qualifier

사용하는 쪽에서 직접 적어준  
@Qualifier가 이긴다!

# 22강. Section3 정리. 다음으로!

## Section 3. 역할의 분리와 스프링 컨테이너

1. 좋은 코드가 왜 중요한지 이해하고, 원래 있던 Controller 코드를 보다 좋은 코드로 리팩토링한다.
2. 스프링 컨테이너와 스프링 빈이 무엇인지 이해한다.
3. 스프링 컨테이너가 왜 필요한지, 좋은 코드와 어떻게 연관이 있는지 이해한다.
4. 스프링 빈을 다루는 여러 방법을 이해한다.

**이렇게 깔끔한 코드까지 적용해 보았습니다!**

**감사합니다**