섹션 3. 역할의 분리와 스프링 컨테이너

이번 섹션의 목표

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.

17강. 좋은 코드(Clean Code)는 왜 중요한가?!

이번 시간에는 좋은 코드(Clean Code)가 왜 중요한지, Clean Code 관점에서 한 기능을 한 함수가 모두 구현하는 것이 왜 좋지 않은지 살펴보도록 하자.

먼저, 우리가 작성하고 있는 코드란 요구사항을 표현하는 언어이다. 개발자는 SW로써 달성해야 하는 요구사항을 구현하기 위해 코드를 읽고 작성한다. 여기서 흥미로운 포인트는, 코드를 작성하는 시간보다 읽는 시간이 훨씬 많다는 점이다. 특히나 팀으로 함께 협업하는 환경에서는 다른 사람이 작성한 코드를 읽는 경우가 많고, 내가 6개월 전, 1년 전에 작성했던 기억나지 않는 코드를 읽을 때도 많다. 그리고 코드를 읽어야만 맥락을 이해하고 기존 코드를 수정하거나 새로운 코드를 추가할 수 있다.

생각해 보자. 우리는 소설책을 읽더라도 개연성이 떨어지거나, 앞뒤 상황이 맞지 않거나, 너무 늘어진다거나 하면 더 스트레스를 받기 전에 그 소설책을 덮고 차라리 유튜브를 볼 것이다. 하지만 개발을 할 때 코드를 덮을 수는 없다!!! 😢😢 코드를 읽는 직업을 가지고 있으니 어떻게든 읽어야 하는 것이다!

다음 두 코드를 비교해 보자.

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

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

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

public class User {
  private int age;
  private int height;

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

두 코드 모두 같은 기능을 하는 코드이지만 전자는 코드만 보고 도저히 무슨 의미인지 이해하기 어려운 반면, 후자는 쉽게 코드를 읽고 이해할 수 있다. 그렇다. 좋은 코드가 왜 중요한지 감이 오기 시작한다. 👍

책 제목부터가 클린 코드(좋은 코드)인 저서 <Clean Code>에서는 좋지 않은 코드가 쌓이면 어떤 일이 일어나는지 간단하지만 오싹한 사례를 들고 있다.

- 옛날에 매우 인기 있는 앱을 개발한 회사가 있었다.
- 이 회사의 제품은 커다란 인기를 끌었고 수많은 전문가들이 구매해 사용했다.
- 그런데 제품 출시 주기가 점차 늘어지기 시작했다.
- 이전 버전에 있었던 버그가 다음 버전에도 그대로 남아 있고, 프로그램이 느려졌으며, 아예 동작하지 않는 일도 생기기 시작했다.
- 그렇게 점차 사용자들이 앱을 사용하지 않기 시작했고 회사는 얼마 못가 망하게 되었다.

- 후일담으로 전해진 내부 사정은 이렇다.
- 다음 버전 출시가 바빠 코드를 마구 작성하였으며, 기능을 추가할수록 코드는 엉망이 되어갔고 결국 감당이 불가능한 수준에 이르렀다고 한다.
- 나쁜 코드 때문에 회사가 망한 것이다.

안좋은 코드가 쌓이면, 시간이 지날 수록 생산성이 낮아진다!

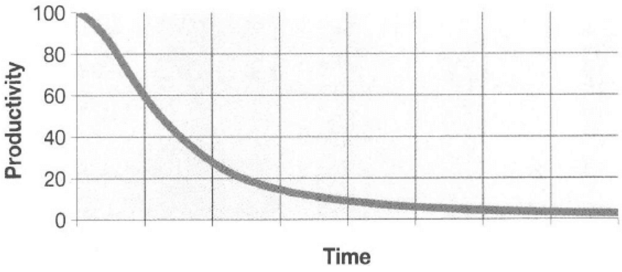


그림 1.1 생산성 대 시간

다시 우리의 관심사로 돌아와 Controller에서 모든 기능을 구현하면 안 되는 이유에 대해 조금 더 이야기해 보자.

앞서 언급한 <Clean Code>에서는 좋은 코드에 대한 정말 다양한 가이드라인을 제시하고 있다. 그중 함수 부분에서는 '함수는 최대한 작게 만들고 한 가지 일만 수행하는 것이 좋다'라는 내용을 담고 있으며, 클래스 부분에서는 '클래스는 작아야 하며 하나의 책임만을 가져야 한다'라는 내용을 담고 있다. 왜 그럴까?!

이런 경우를 생각해 보자. 우리가 어떤 기능을 개발하였다!! 이 기능은 매우 크고 복잡해서 코드 줄 수가 3000줄이 넘어간다. 그런데.. 하나의 함수로만 구현되어 있다. 상상만으로 살짝 눈물이 나는데~ 😢 구체적으로 다음의 문제들이 발생하게 된다.

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.
4. 너무 큰 기능이기 때문에 테스트도 힘들다.
5. 종합적으로 유지 보수성이 매우 떨어진다.

물론 지금 우리가 작성한 코드는 이렇게까지 심각하지 않다! 하지만, 몇 줄 되지 않는 유저 수정 API를 보더라도 무려 3가지 역할을 수행하고 있다.

```
@PutMapping("/user")
public void updateUser(@RequestBody UserUpdateRequest request) {
  String readSql = "SELECT * FROM user WHERE id = ?";
  boolean isUserNotExist = jdbcTemplate.query(readSql, (rs, rowNum) -> 0, request.getI
d()).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 변환하고 있다.
2. 현재 유저가 있는지, 없는지 등을 확인하고 예외 처리를 해준다.
3. SQL을 사용해 실제 DB와의 통신을 담당한다.

무려 3가지 역할을 가지고 있기 때문에, DB와의 통신을 변경하려 해도 `updateUser()` 함수를 변경해야 하고, 예외 처리를 해주려고 해도 `updateUser()` 함수를 변경해야 하고, API와 관련된 내용을 바꾸려고 해도 `updateUser()` 함수를 변경해야 한다!!!!

앞서 이야기했던, '함수와 클래스는 작아야 하며 하나의 책임만을 가져야 한다'와는 거리가 있는 것이다.

매우 좋다~! 👍 우리는 Clean Code가 왜 중요하고, 우리의 코드가 가진 문제점이 무엇인지 확인하였다. 이제 다음 시간에는 이 함수를 역할에 맞게 3단 분리할 것이다!

18강. Controller를 3단 분리하기 - Service와 Repository

이번 시간에는 본격적으로 Controller를 3단 분리하려고 한다! 🔥 가장 복잡한 PUT API와 DELETE API를 분리하고, POST API와 GET API 순서로 진행할 예정이다.

첫 번째로 PUT API이다. 기존에 존재했던 UserController의 updateUser 메소드의 3가지 역할을 분리할 것이다. 3가지의 역할을 떠올려보면 다음과 같다.

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.

우선 `com.group.libraryapp.service.user`라는 패키지를 만들고 그 안에 `UserService` 클래스를 만들어 주자. 그리고 UserService 클래스 안에 updateUser라는 동일한 메소드를 만들어 다음과 같이 작성해주자.

우리는 Service 클래스에게 현재 유저가 있는지 없는지 등을 확인하고 예외 처리를 하는 역할을 부여할 것이다!

```
public class UserService {

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

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

}
```

그리고 `UserController.updateUser()`에서는 `UserService` 클래스의 `updateUser`를 호출하도록 변경하자. 그러기 위해서는 jdbcTemplate을 추가했던 것과 비슷하게, UserService를 필드로 가지고 있어야 한다. 이번에는 직접적으로 new 연산자를 호출해 UserSerivce를 가져오도록 하자.

```
@RestController
public class UserController {

  private final UserService userService = new UserService();

  // 나머지 부분 동일...

  @PutMapping("/user")
  public void updateUser(@RequestBody UserUpdateRequest request) {
    userService.updateUser(jdbcTemplate, request);
  }

}
```

매우 좋다~! 👍 UserService에는 `@PutMapping` 또는 `@RequestBody` 와 같이 API 진입 지점과 관련 있는 어노테이션이 붙어 있지 않고 기존의 기능을 정상적으로 수행할 수 있게 되었다.

이제 이어서 `com.group.libraryapp.repository.user`라는 패키지를 만들어 `UserRepository`라는 클래스를 만들어주자. `Repository`에게는 DB에 SQL을 날리는 역할, 저장장치와의 접근을 담당하도록 만들 것이다.

`UserRepository` 클래스를 다음과 같이 작성해 주도록 하자!!

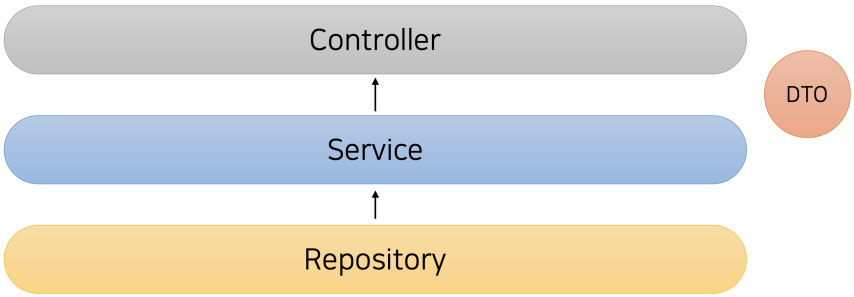
```
public class UserRepository {

  public boolean isUserNotExist(JdbcTemplate jdbcTemplate, long id) {
    String sql = "SELECT * FROM user WHERE id = ?";
    return jdbcTemplate.query(sql, (rs, rowNum) -> 0, id).isEmpty();
  }

  public void updateUserName(JdbcTemplate jdbcTemplate, String name, long id) {
    String sql = "UPDATE user SET name = ? WHERE id = ?";
    jdbcTemplate.update(sql, name, id);
  }

}
```

이제 UserService가 UserRepository를 활용하게 바꾼다면, UserService는 다음과 같이 변경된다.

```
public class UserService {

  private final UserRepository userRepository = new UserRepository();

  public void updateUser(JdbcTemplate jdbcTemplate, UserUpdateRequest request) {
    if (userRepository.isUserNotExist(jdbcTemplate, request.getId())) {
      throw new IllegalArgumentException();
    }

    userRepository.updateUserName(jdbcTemplate, request.getName(), request.getId());
  }

}
```

정말 좋다~!!! 👍 기존에 UserController에서 담당하던 세 가지 역할을 3개의 클래스로 적절히 분배하였다!! 이를 그림으로 표현해 보면 다음과 같다.

3가지 역할로 구분된 구조



이렇게 각 클래스가 각자의 역할을 가지고 겹겹이 쌓인 것을 어려운 말로 Layered Architecture라고 부른다.

자 아직 한 가지 불편한 것이 남아 있다! 바로 `JdbcTemplate` 이 파라미터를 통해 계속해서 전달하는 것이다. 이 문제 역시 해결해 코드를 더욱더 개선해 보자! 어떻게 하면 `JdbcTemplate` 을 제거할 수 있을까?

`UserRepository` 역시 `UserController` 처럼 `JdbcTemplate` 을 변수로 갖게 하고, 생성자를 만들어보자.

```
public class UserRepository {

  private final JdbcTemplate jdbcTemplate;

  public UserRepository(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
  }

}
```

그러면 자연스럽게 `UserService` 의 코드 역시 변경된다.

```
public class UserService {

  private final UserRepository userRepository;

  public UserService(JdbcTemplate jdbcTemplate) {
    this.userRepository = new UserRepository(jdbcTemplate);
  }

}
```

생성자가 생기고, UserRepository 타입의 필드를 만들어 줄 때 JdbcTemplate을 받아 UserRepository로 전달하는 것이다. 그러면~ 다시 한번 자연스럽게 `UserController`의 코드도 변경된다.

```
@RestController
public class UserController {

  private final JdbcTemplate jdbcTemplate;
  private final UserService userService;

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

}
```

매우 좋다~!!! 👍 이제 UserRepository 클래스 자체가 JdbcTemplate을 가지고 있으니, 파라미터를 통해 매번 넘겨주지 않아도 괜찮다!!! 😊 코드를 마저 변경해 보자.

먼저 DELETE API이다.

```
// Controller
@DeleteMapping("/user")
public void deleteUser(@RequestParam String name) {
  userService.deleteUser(name);
}

// Service
public void deleteUser(String name) {
  if (userRepository.isUserNotExist(name)) {
    throw new IllegalArgumentException();
  }

  userRepository.deleteUserByName(name);
}

// Repository
public void deleteUserByName(String name) {
  String sql = "DELETE FROM user WHERE name = ?";
  jdbcTemplate.update(sql, name);
}
```

다음으로는 POST API이다.

```
// Controller
@PostMapping("/user")
public void saveUser(@RequestBody UserCreateRequest request) {
  userService.saveUser(request);
}

// Service
public void saveUser(UserCreateRequest request) {
  userRepository.saveUser(request.getName(), request.getAge());
}

// Repository
public void saveUser(String name, Integer age) {
  String sql = "INSERT INTO user(name, age) VALUES(?, ?)";
  jdbcTemplate.update(sql, name, age);
}
```

마지막으로 GET API이다.

```
// Controller
@GetMapping("/user")
public List<UserResponse> getUsers() {
  return userService.getUsers();
}

// Service
public List<UserResponse> getUsers() {
  return userRepository.getUserResponses();
}

// Repository
public List<UserResponse> getUserResponses() {
  String sql = "SELECT * FROM user";
  return jdbcTemplate.query(sql, (rs, rowNum) -> {
    long id = rs.getLong("id");
    String name = rs.getString("name");
    int age = rs.getInt("age");
    return new UserResponse(id, name, age);
  });
}
```

이렇게 리팩토링을 마무리한 후, 다시 웹 UI를 열어 기능을 간단히 확인해 보면 모든 기능이 정상 동작하는 것을 확인할 수 있다~! 👍 우리는 기존에 Controller에서 처리하던 여러 역할을 적절한 단위로 나눔으로써 훨씬 더 좋은 코드로 변경했다!!!! 😊

그런데 한 가지 궁금한 것이 있다. Controller에서 `JdbcTemplate`은 어떻게 가져온 것일까?! Controller는 어차피 `JdbcTemplate` 을 전달해 주는 역할만 하고 있는데, Repository에서 바로 `JdbcTemplate` 을 가져올 수는 없을까?! 그 비밀이 다음 시간에 밝혀진다~!! 😄

19강. UserController와 스프링 컨테이너

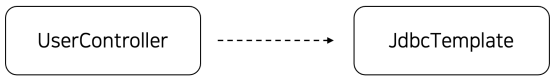
> 💡 강의 영상과 PPT를 함께 보시면 **더욱** 좋습니다 😊

이번 시간에는 Controller와 JdbcTemplate의 비밀을 파헤쳐 보자. 사실 이 UserController는 의아한 점이 두 가지나 존재한다.

```
@RestController
public class UserController {

  private final UserService userService;

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

}
```

1. 우리는 UserController에 존재하는 메소드를 API 진입 지점으로 사용하고 있다. 그런데 생각해 보면, 클래스 안에 있는 함수를 사용하기 위해서는 인스턴스화가 필요하다!!!! 하지만 우리는 UserController를 현재 인스턴스화하고 있지 않다. 어떻게 된 것일까?
2. UserController의 생성자는 JdbcTemplate이라는 클래스를 필요로 하고 있다. 이것을 어려운 말로 '의존한다'라고 한다. 우리는 JdbcTemplate에 대해 처리한 적이 한 번도 없다. 어떻게 UserController는 `JdbcTemplate` 을 가져올 수 있었을까?!

UserController는 JdbcTemplate에 의존하고 있다.



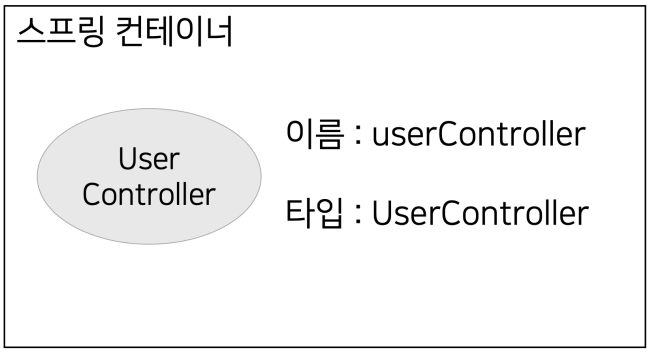
UserController는 JdbcTemplate이 없으면 동작하지 않는다!

비밀은 바로 `@RestController`에 있다!! 사실 이 어노테이션은 UserController 클래스를 API의 진입 지점으로 만들어 줄 뿐만 아니라, 이 클래스를 **스프링 빈**으로 등록시킨다.

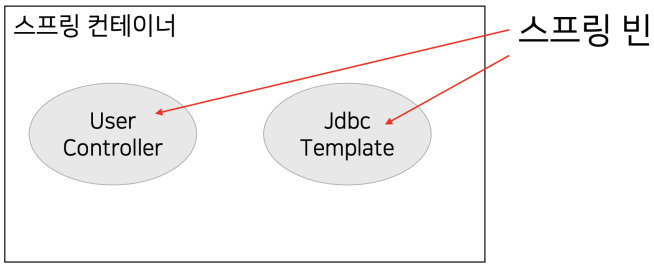
잠깐! 클래스를 **스프링 빈**으로 등록시킨다는 게 도대체 무슨 말인지 조금 더 천천히 살펴보자.

2강에서 나왔던 기억을, 우리가 서버를 시작할 때 `@SpringBootApplication`이 다양한 설정들을 모두 자동으로 해준다는 것을 언급한 적이 있다. 이때 자동으로 해주는 것 중 하나가 스프링 서버 내부에 거대한 **컨테이너**를 만드는 것이다. 우리가 흔히 떠올리는 컨테이너를 생각하면 된다.

그럼, 이 컨테이너 안에는 무엇이 들어갈까? 바로 클래스가 들어가게 된다! 그리고 이렇게 들어간 클래스를 **스프링 빈**이라고 부른다. 클래스가 들어갈 때는 이 빈을 식별할 수 있는 이름 및 타입과 함께 다양한 정보들이 저장된다. 그리고 이때 인스턴스화도 함께 이뤄진다.



잠깐, 그런데 UserController를 인스턴스화하려면 `JdbcTemplate` 이 필요하다. 이 `JdbcTempalte` 은 어디서 가져오는 것일까?! 사실은 `JdbcTemplate` 역시 **스프링 빈**으로 등록되어 있다. 우리가 build.gradle에 설정했던 `spring-boot-starter-data-jpa` 의존성이 `JdbcTemplate` 을 **스프링 빈**으로 미리 등록해 준 것이다.

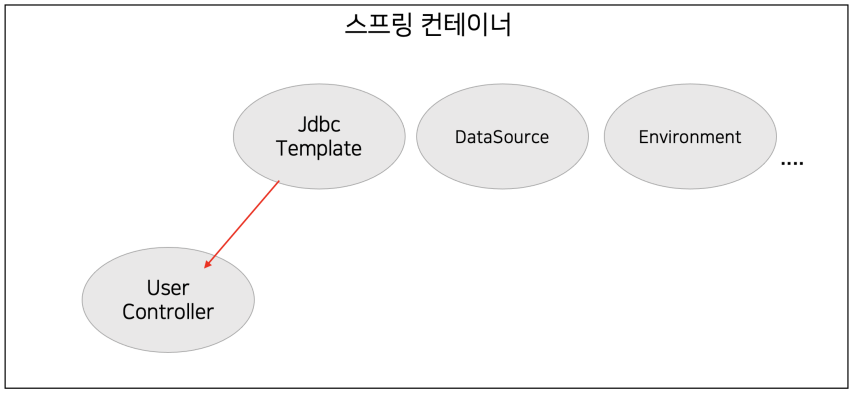


스프링 컨테이너는 UserController를 인스턴스화할 때, UserController가 필요하다고 하는 `JdbcTemplate` 을 컨테이너 내부에서 찾아 인스턴스화를 진행해 주게 된다. 그리고 만약 이때

`JdbcTemplate` 이 없으면 에러가 난다! 실제로 의존성을 제거하고, gradle refresh를 진행한 후 다시 서버를 실행시켜 보면 다음과 같은 에러가 난다!

즉, 우리가 서버를 시작하게 되면 다음과 같은 일이 순차적으로 일어난다.

1. 컨테이너가 시작된다.
2. 컨테이너에 기본적으로 많은 스프링 빈이 등록된다. (예를 들어, `JdbcTemplate`이 등록된다)
3. 스프링을 사용하는 개발자 설정해 준 스프링 빈이 등록된다. (예를 들어, `UserController`가 등록된다)
4. 이때 필요한 의존성이 자동으로 설정된다. (예를 들어 `UserController`를 만들 때 `JdbcTemplate`을 알아서 넣어준다)



좋다. 이제 스프링 빈이 무엇이고 스프링 컨테이너가 무엇인지 살펴보았다. 이제 우리는 지금의 `UserRepository`가 `JdbcTemplate`을 바로 가져오지 못하는 이유를 설명할 수 있다. `UserController`는 스프링 빈이기 때문에 같은 스프링 빈인 `JdbcTemplate`을 가져올 수 있지만, `UserRepository`는 스프링 빈이 아니기 때문이다.

그럼 이제 우리가 직접 인스턴스화를 해주었던 `UserService`와 `UserRepository` 모두 스프링 빈으로 등록해 보자!

먼저 Repository를 스프링 빈으로 등록할 때는 `@Repository` 어노테이션을 사용하면 된다.

```
@Repository
public class UserRepository {

  private final JdbcTemplate jdbcTemplate;
```

```
  public UserRepository(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
  }

}
```

다음으로 Service를 스프링 빈으로 등록할 때는 `@Service` 어노테이션을 사용하면 된다.

```
@Service
public class UserService {

  private final UserRepository userRepository;

  public UserService(UserRepository userRepository) {
    this.userRepository = userRepository;
  }

}
```

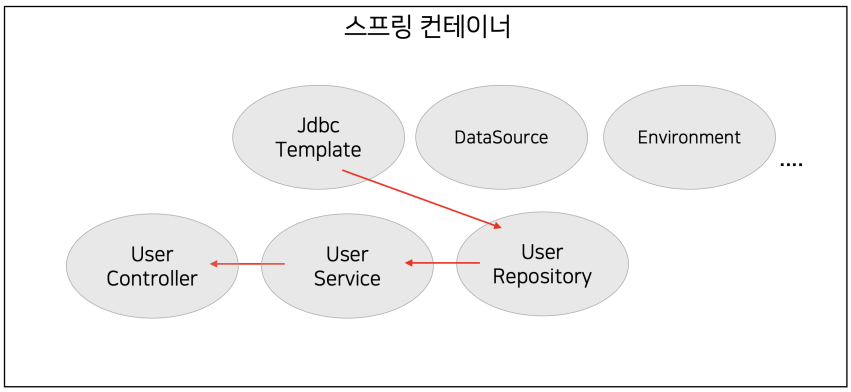
그럼 이제 `UserController` 입장에서도 `UserService`가 스프링 빈이니 굳이 직접 `new` 연산자를 통해 인스턴스화해줄 필요가 없다! 컨테이너가 알아서 처리해 줄 것이다. 또한 `UserRepository`가 `JdbcTemplate`을 직접 가지고 있기 때문에 `JdbcTemplate` 도 가지고 있을 필요가 없어진다. 때문에 `UserController`의 코드는 다음과 같이 변경된다.

```
@RestController
public class UserController {

  private final UserService userService;

  public UserController(UserService userService) {
    this.userService = userService;
  }

}
```

이제 우리가 작성한 `UserController` - `UserService` - `UserRepository` 클래스들은 서버가 시작할 때 다음과 같이 처리될 것이다.

1. 미리 존재하던 JdbcTemplate을 이용해 UserRepository가 스프링 빈으로 등록된다. (인스턴스화도 이루어진다!)
2. UserRepository를 의존하는 UserService가 스프링 빈으로 등록된다.

3. UserService를 의존하는 UserController가 스프링 빈으로 등록된다.
4. 이렇게 3개의 클래스 모두 스프링 빈으로 등록된다!



매우 좋다~!!! 👍 이번 시간에 우리는 스프링 빈이 무엇인지, 스프링 컨테이너가 무엇인지 알아보고 우리가 작성했던 새로운 클래스들을 스프링 빈으로 리팩토링해보았다.

그런데 한 가지 궁금한 점이 자연스럽게 떠오른다. 이 스프링 컨테이너를 왜 사용하는 것일까?! 그냥 `new` 연산자를 모두 사용하면 안 될까?! 다음 시간에는 스프링 컨테이너를 사용하는 이유에 대해 알아보도록 하자!

20강. 스프링 컨테이너를 왜 사용할까?!

> 💡 강의 영상과 PPT를 함께 보시면 **더욱** 좋습니다 😊

이번 시간에는 스프링의 핵심이라고 하는, 스프링 컨테이너를 사용하는 이유에 대해 알아보자.

Controller, Service, Repository를 이용해 책을 메모리에 저장하는 API를 매우 간단하게 만들어보자. Controller만 스프링 빈으로 등록하고 Service와 Repository는 이전처럼 구현할 것이다. 책은 간단하게 이름만 받도록 하자.

```
@RestController
public class BookController {

  private final BookService = new BookService();

  @PostMapping("/book")
  public void saveBook(@RequestBody SaveBookRequest request) {
```

```
    bookService.saveBook(request);
  }

}
```

```
public class BookService {

  private final BookMemoryRepository bookRepository = new BookMemoryRepository();

  public void saveBook(SaveBookRequest request) {
    bookRepository.save(request.getName();
  }

}
```

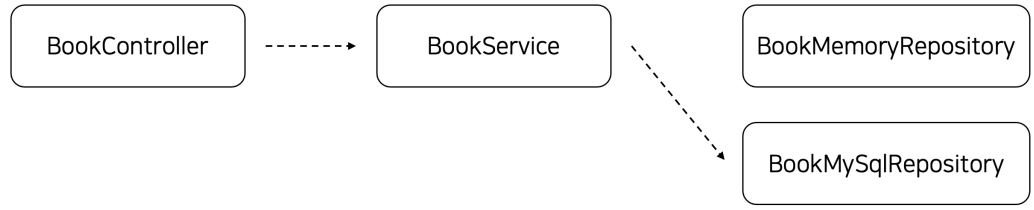
```
public class BookMemoryRepository {

  private final List<String> books = new ArrayList();

  public void save(String bookName) {
    books.add(bookName);
  }

}
```

매우 좋다~ 👍 이제 이런 요구사항은 익숙해져 쉽게 구현할 수 있다!!!

매우 좋다~~ 이제 이 상황에서 MySQL을 사용하도록 다시 한번 변경한다고 해보자! 머리로만 상상을 해보는 것이다! 무슨 일이 일어날까?!

1. 가장 먼저 `BookMemoryRepository` 대신 `BookMySqlRepository`가 새로 생길 것이다. JdbcTemplate을 생성자로 받을 수도 있지만, 이번에는 어떻게 어떻게 `BookMySqlRepository` 가 직접 설정해 준다고 가정하자.
2. BookService도 변경될 것이다.
    a. `BookMemoryRepository()` 대신 `BookMySqlRepository()`를 사용하게 될 것이다.

바로 이 부분이 문제이다. 분명 우리는 `Repository`의 역할인 데이터를 메모리에 저장할지~ MySQL에 저장할지에 대해서만 변경하고 싶지만, BookService까지 필연적으로 변경이 일어나게 된다.




```
public class BookService {

  // private final BookMemoryRepository bookRepository = new BookMemoryRepository();
  private final BookMySqlRepository bookRepository = new BookMySqlRepository();

}
```

어떻게 하면 BookService의 변경을 최소화할 수 있을까?!! 가장 먼저 도입해 볼 수 있는 것은 Java에 존재하는 interface이다. interface를 도입하게 되면 코드는 다음과 같이 변경된다.

```
public class BookService {

  private final BookRepository bookRepository = new BookMemoryRepository();

}
```

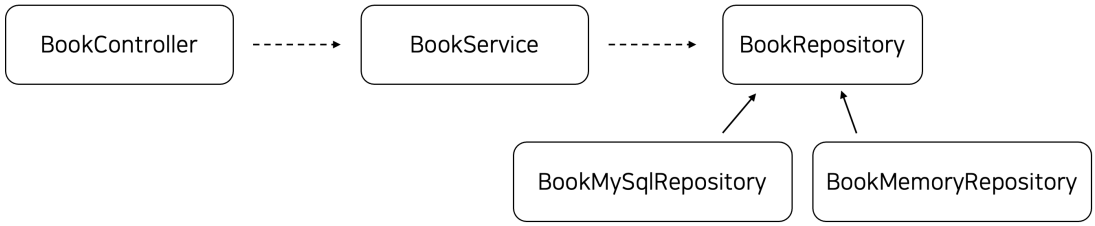
```
public interface BookRepository {

  public void save(String bookName);

}
```

```
public class BookMemoryRepository implements BookRepository {

  private final List<String> books = new ArrayList();

  @Override
  public void save(String bookName) {
    books.add(bookName);
  }

}
```

```
public class BookMySqlRepository implements BookRepository {

  @Override
```

```
    public void save(String bookName) {
      // jdbcTemplate.....
    }

}
```

매우 좋다~ interface를 도입함으로써 `private final BookRepository =` 까지는 변경하지 않아도 괜찮아졌다. 하지만 여전히 `new BookMemoryRepository()`를 사용할지, `new BookMySqlRepository()`를 사용할지는 `BookService`를 수정해 주어야 했다.



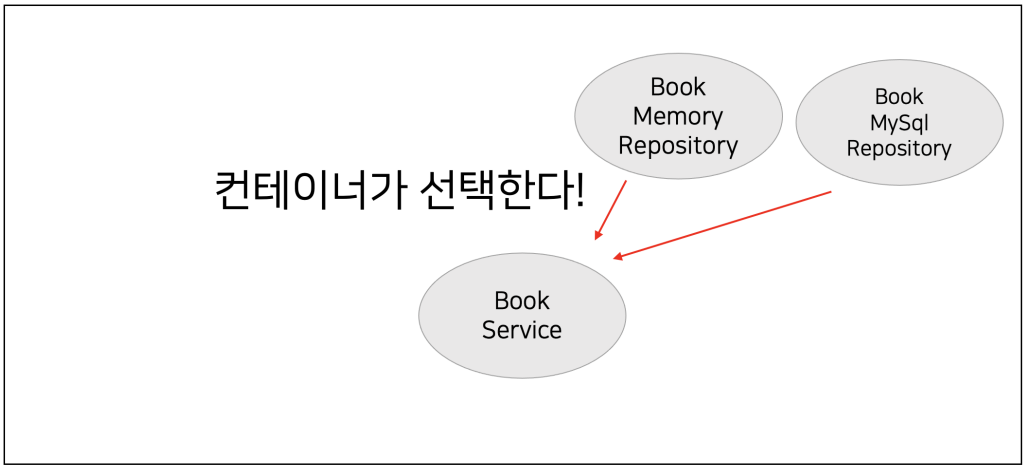

```
public class BookService {

  private final BookRepository bookRepository = new BookMySqlRepository(); // new BookMemoryRepository();

}
```

지금이야 `BookRepository`에 의존하는, 즉 `BookRepository`를 사용하는 코드가 한 군데 밖에 없지만 프로젝트가 커져서 `BookRepository`를 100군데도 넘게 사용하고 있다면, 모두 변경하는 것은 정말 서글플 것이다. 😢 (그날은 야근 당첨이다!! 😢)

그렇다면, 어떻게 해야 `BookService` 코드를 완전히 바꾸지 않은 채 `BookRepository`만을 온전하게 변경할 수 있을까?! 이 고민에 대한 해결책이 바로 **스프링 컨테이너**이다.

스프링 컨테이너를 사용한다고 해보자. 그러면 컨테이너가 `BookMemoryRepository` 혹은 `BookMySqlRepository` 중 하나를 선택한 후, BookService를 만들어줄 것이다. 이런 방식을 어려운 말로 제어의 역전 (IoC, **I**nversion **o**f **C**ontrol)이라 부른다.

또한 이때 컨테이너가 `BookService`를 만들어 줄 때 `BookMemoryRepository`와 `BookMySqlRepository` 중 하나를 선택해서 넣어주는 과정을 의존성 주입(**D**ependency **I**njection)이라고 한다.



그렇다면 둘 중 무엇을 넣어줄까?! 우리는 `@Primary` 어노테이션을 이용해 우선권을 제어할 수 있다!! 이제 코드를 다음과 같이 변경해 확인해 보도록 하자!

```
@Service
public class BookService {

  private final BookRepository bookRepository;

  public BookService(BookRepository bookRepository) {
    this.bookRepository = bookRepository;
  }

}
```

```
public interface BookRepository {

  public void save(String bookName);

}
```

```
@Repository
public class BookMemoryRepository implements BookRepository {

  @Override
  public void save(String bookName) {
    println("Memory Repository " + bookName);
  }

}
```

```
@Repository
@Primary // 우선권을 부여하는 어노테이션!!
public class BookMySqlRepository implements BookRepository {

  @Override
  public void save(String bookName) {
    println("MySQL Repository " + bookName);
  }

}
```

실제 `@Primary` 어노테이션을 `BookMemoryRepository`에 붙였다가, `BookMySqlRepository`에 붙였다가 변경해 보면, `BookService` 코드 변경 없이 다른 Spring Bean이 사용됨을 확인할 수 있다!! 😊 🎉

매우 좋다~! 👍 이제 다음 시간에는 우리가 사용했던 `@RestController`, `@Service`, `@Repository` 외에도 스프링 컨테이너에 빈을 다루는 방법을 알아자.

21강. 스프링 컨테이너를 다루는 방법

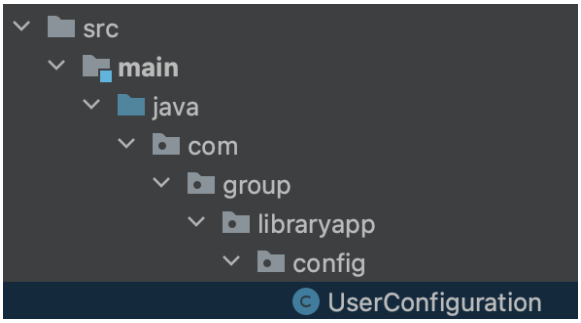
이번 시간에는 스프링 컨테이너에 빈을 등록하는 방법, 그리고 가져오는 방법 몇 가지에 대해 알아보자. 먼저 빈을 등록하는 방법이다.

우리가 Service, Repository 클래스에 대해 `@Service`, `@Repository`를 사용하는 대신 다음 두 개의 어노테이션을 활용해 스프링 빈으로 등록할 수도 있다.

- @Configuration : 클래스에 붙이는 어노테이션으로, @Bean을 사용할 때 함께 사용해 주어야 한다.
- @Bean : 메소드에 붙이는 어노테이션으로, 메소드에서 반환되는 객체를 스프링 빈에 등록한다.

예를 들어 `UserRepository`를 `@Configuration`과 `@Bean` 을 활용해 바꿔보자!

우선 `com.group.libraryapp.config.user` 패키지를 만들고, 그 안에 `UserConfiguration` 클래스를 만들자!



그 다음 아래와 같이 타이핑 하면 된다.

```
@Configuration
public class UserConfiguration {

  @Bean
  public UserRepository userRepository(JdbcTemplate jdbcTemplate) {
    return new UserRepository(jdbcTemplate);
  }

}

// @Repository // 주석처리
public class UserRepository {
```

`UserRepository`의 `@Repository`를 주석 처리했을 때는 `UserService`에 빨간 줄이 나왔지만 `@Configuration`과 `@Bean`을 알맞게 사용해 주니 다시 빨간 줄이 사라졌다.

`@Bean` 어노테이션이 붙은 메소드의 파라미터 역시 스프링 컨테이너가 필요한 의존성을 주입해 줄 수 있다. 예를 들어 위의 코드에서는 `JdbcTemplate`을 스프링 컨테이너가 넣어준다.

좋다~! 이제 `UserService` 역시 `@Configuration`과 `@Bean`을 사용하도록 변경해 보자!

```
@Configuration
public class UserConfiguration {

  @Bean
  public UserRepository userRepository(JdbcTemplate jdbcTemplate) {
    return new UserRepository(jdbcTemplate);
  }

  @Bean
  public UserService userService(UserRepository userRepository) {
    return new UserService(userRepository);
  }

}
```

매우 간단하다~!! 😊

그렇다면 언제 `@Service` 나 `@Repository`를 사용해야 하고 언제 `@Configuration` + `@Bean`을 사용해야 할까?

정답은 없지만 일반적으로 개발자가 직접 만든 클래스를 스프링 빈으로 등록할 때에는 `@Service` 나 `@Repository`를 사용하고, 외부 라이브러리나 프레임워크에 만들어져 있는 클래스를 스프링 빈으로 등록할 때에는 `@Configuration` + `@Bean` 조합을 많이 사용하게 된다.

다음으로 살펴볼 어노테이션은 `@Component`이다! 사실 지금까지 우리가 Class에 붙였던 4가지 어노테이션 `@RestController` / `@Service` / `@Repository` / `@Configuration`은 모두 `@Component` 어노테이션을 가지고 있다!!!

즉 `@Component` 어노테이션을 붙이면 주어진 클래스를 '컴포넌트'로 간주하고, 컴포넌트들은 스프링 스프링 서버가 뜰 때 자동으로 감지된다. `@Component` 덕분에 지금까지 우리가 사용했던 어노테이션들이 모두 자동으로 감지되었던 것이다!! 😲

이 `@Component` 어노테이션은 컨트롤러, 서비스, 리포지토리가 모두 아닌 추가적인 클래스를 스프링 빈으로 등록할 때 종종 사용되곤 한다.

이제 스프링 빈으로 등록하는 방법을 살펴보았으니, 스프링 빈을 주입받는 방법을 살펴보자!

가장 간단하고 권장되는 방법은 생성자를 이용해 주입받는 방법이다. 지금까지 우리가 계속 해서 사용해왔다.

```
@Repository
public class UserRepository {

  private final JdbcTemplate jdbcTemplate;

  // 생성자에 JdbcTemplate이 있으므로 스프링 컨테이너가 넣어준다.
  public UserRepository(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
  }

}
```

스프링 빈을 주입받는 또 다른 방법은 setter를 사용하는 것이다! setter를 사용하기 위해서는 `final` 키워드를 제거해야 하고 setter 메소드에 `@Autowired` 어노테이션을 붙여야 한다.

```
@Repository
public class UserRepository {

  private JdbcTemplate jdbcTemplate;
```

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

}
```

`@Autowired` 연산자가 있어야지만 스프링 컨테이너에 있는 스프링 빈을 찾아 setter에 넣어 주게 된다.

또 다른 방법은 필드에 직접적으로 주입하는 방법이다. 필드 위에 바로 `@Autowired`를 적어주면 된다.

```
@Repository
public class UserRepository {

  @Autowired
  private JdbcTemplate jdbcTemplate;

}
```

setter를 사용하는 방법이나 필드에 바로 주입하는 방법은 기본적으로 권장되지 않는다. setter를 사용하게 되면 혹시 누군가가 setter를 사용해 다른 인스턴스로 교체해 동작에 문제가 생길 수도 있고, 필드에 바로 주입하게 되면 테스트가 어렵기 때문이다.

마지막으로 살펴볼 어노테이션은 `@Qualifier` 어노테이션이다. `@Qualifier` 어노테이션은 `@Primary` 어노테이션이 없는 상황에서 주입받는 쪽에서 특정 스프링 빈을 선택할 수 있게 해준다.

```
public interface FruitService {
}
```

```
@Service
public class AppleService {

}
```

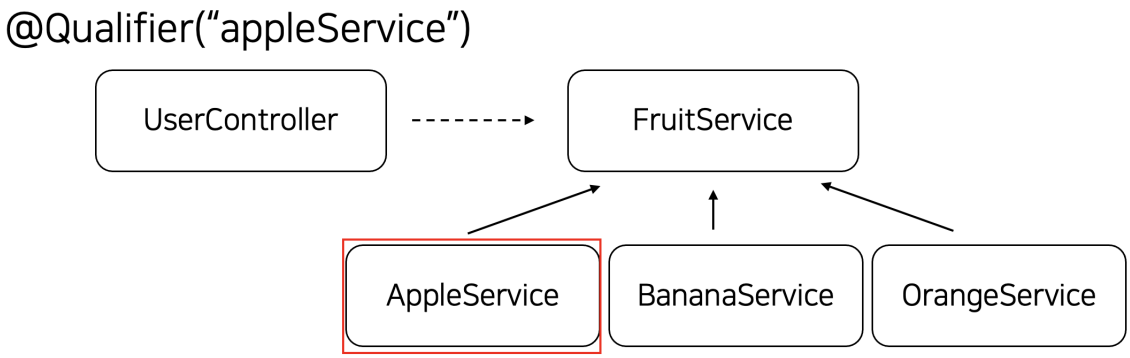
```
@Service
public class BananaService {
```

```
}
```

```
@Service
public class OrangeService {

}
```

```
@RestController
public class UserController {

  private final UserService userService;
  private final FruitService fruitService;

  public UserController(
    UserService userService,
    @Qualifier("appleService") FruitService fruitService
  ) {
    this.userService = userService;
    this.fruitService = fruitService;
  }

}
```

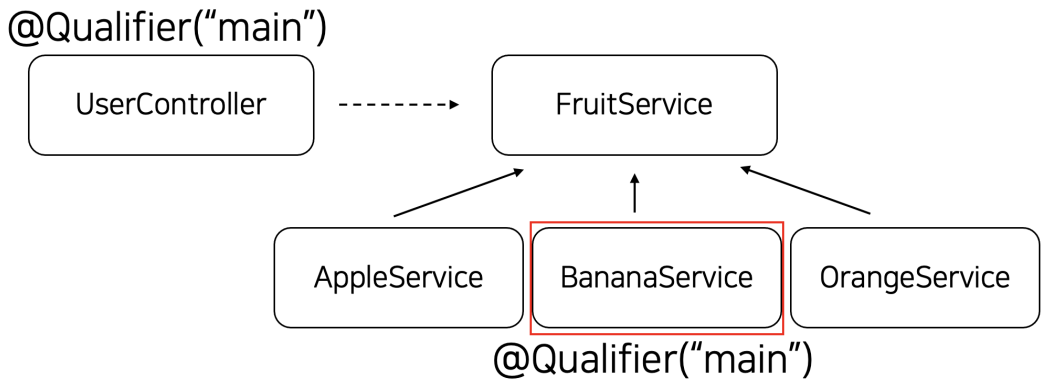
- `FruitService`에는 `AppleService`가 들어오게 된다!



또한, `@Qualifier` 어노테이션은 스프링 빈을 사용하는 쪽과 스프링 빈을 등록하는 쪽 모두 사용할 수 있다. 이 경우에는 `@Qualifier` 어노테이션에 적어준 값이 같은 것끼리 연결된다!!!

```
@Service
@Qualifier("main")
public class BananaService {

}
```

```
@RestController
public class UserController {

  private final UserService userService;
  private final FruitService fruitService;

  public UserController(
    UserService userService,
    @Qualifier("main") FruitService fruitService
  ) {
    this.userService = userService;
    this.fruitService = fruitService;
  }

}
```



그렇다면 만약 `@Primary` 와 `@Qualifier`를 둘 다 사용하고 있으면 어떻게 될까?! 예를 들어 `OrangeService`에 `@Primary`가 붙어 있는 상황에서 UserController는 `@Qualifier("main")`을 사용한 것이다!

정답은, `@Qualifier`의 우선순위가 높다는 것이다! 스프링 빈을 사용하는 쪽에서 특정 빈을 지정해 준 것의 우선순위를 더 높게 간주하는 것이다.

이렇게 이번 시간에는 `@Configuration` 과 `@Bean`, `@Component`, `@Autowired`, `@Qualifier`에 대해 알아보았다! 다음 시간에는 이번 Section을 정리해 보자!!! 😊 👍

22강. Section3 정리

이번 Section을 지나며 우리는 모든 기능을 처리하던 한 함수를 3가지 계층으로 분리하며 더 좋은 구조를 만들어냈다! 또한 이 과정에서 아래의 내용들을 익힐 수 있었다.

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.

하지만~~ 여전히 추가적으로 개선할 수 있는 부분이 분명 존재한다. 과연 어떤 부분을 어떻게 개선할 수 있을지 다음 섹션에서 JPA와 함께 탐구해 보도록 하자!!! 🔥 🏃