스터디 2주차(2021.08.22)

섹션 3. 회원 관리 예제 - 백엔드 개발

섹션 4. 스프링 빈과 의존 관계

섹션 5. 회원 관리 예제 - 웹 MVC 개발

Annotation 정리와 예제

1. Annotation이란?

> 자바 소스 코드에 추가하여 사용할 수 있는 메타데이터의 일종
>
> "출처", 위키백과

- `Annotation(@)`
- 사전적 의미는 **주석**

2. 지금까지 배운 Annotation 종류와 역할

@Component

- @Controller, @Service, @Repository 등이 포함하고 있는 어노테이션
- @Component가 붙은 객체를 스캔해서 스프링 컨테이너에 생성해 **스프링 빈**으로 자동 등록
    - 컴포넌트 스캔(Component Scan)

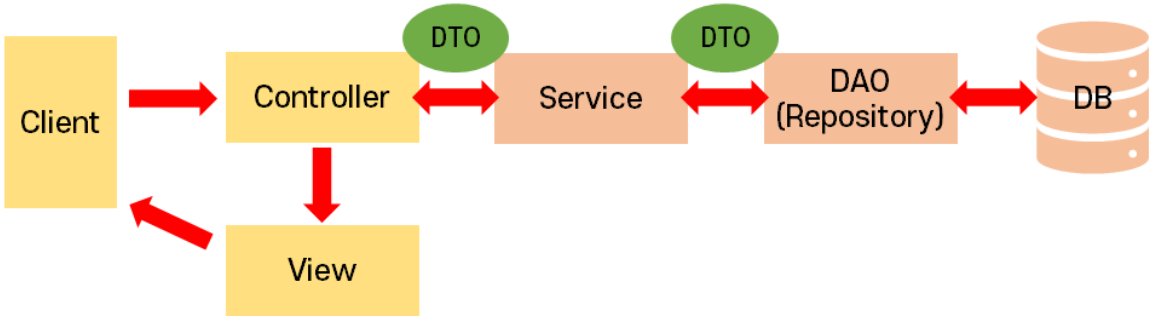
컴포넌트 스캔

- @SpringBootApplication이 있는 파일의 패키지 아래를 스캔한다.
- 컴포넌트 스캔을 통해 스프링 빈으로 등록할 때, 기본으로 '**싱글톤**'으로 등록
    - 공유하기 위해서
- 메모리를 절약할 수 있다.

@Autowired

- 생성자에 사용하면 객체 생성 시점에 스프링 컨테이너에서 **해당 스프링 빈을 찾아서** 주입한다.
    - **DI (Dependency Injection),** 의존성 주입

      : 객체 의존관계를 외부에서 넣어주는 것
- 외존성 주입이 필요한 파일의 생성자에게 사용
- 생성자가 1개만 있으면 생략 가능



- DTO(Data Transfer Object)

  : 계층간 데이터 교환을 위한 객체(Java Beans)

  : VO(value Object)와 유사한 특징이 있다.

@Controller

- 스프링에게 해당 클래스가 Controller 역할을 수행한다고 알려주기 위해 사용
- **MVC의 컨트롤러**에서 사용

```
@Controller
public class MemberController {
    private final MemberService memberService;

    @Autowired // MemberService와 의존관계의 주입
    public MemberController(MemberService memberService) {
        this.memberService = memberService;
    }
    // ... 생략 ...
}
```

@Service

- 스프링에게 해당 클래스가 서비스 비즈니스 역할을 수행한다고 알려주기 위해 사용
  - 서비스란, 비즈니스 도메인 객체을 이용하여 핵심 비즈니스 로직이 동작하도록 구현한 계층
- 스프링 **비즈니스 로직**에서 사용

```
//@Service

public class MemberService {
    private final MemberRepository memberRepository;

    @Autowired //MemberRepository와의 의존관계 주입
    public MemberService(MemberRepository memberRepository) {
        this.memberRepository = memberRepository;
    }
    // ... 생략 ...
}
```

@Repository

- 스프링에게 해당 클래스가 리포지토리 역할을 수행한다고 알려주기 위해 사용
    - 리포지토리란
      : DAO(Data Access Object)
      : **DB에 접근**하고 도메인 객체를 DB에 저장하고 관리한다.
- 스프링 **데이터 접근 계층**에서 사용

```
@Repository
public class MemoryMemberRepository implements MemberRepository {
    //... 생략 ...
}
```

@Configuration

- 설정파일을 만들고 Bean을 등록하기 위한 에노테이션
- 스프링 **설정 정보**에서 사용

자바 코드로 직접 스프링 빈 등록

- 정형화 되지 않거나 상황에 따라 구현 클래스 변경해야 할 때 사용한다.
    - 구현 클래스 MemoryMemberRepository를 다른 구현 클래스 MemberRepository로 변경하려고 할 때,
      기존의 운영중인 코드는 수정하지 않고 아래와 같은 부분만 수정하고 다른 코드는 그대로 사용할 수 있도록 해준다.

```
//public class SpringConfig...
@Bean
public MemberRepository memberRepository() {
    return new DBMemberRepository(); //이 부분만 수정
}
```

@Bean

- **스프링 컨테이너가 관리**하는 자바 객체인 빈(bean)을 의미

```
@Configuration
public class SpringConfig {
     @Bean
    public MemberService memberService() {
        return new MemberService(memberRepository());
    }

    @Bean
    public MemberRepository memberRepository() {
        return new MemoryMemberRepository();
    }
}
```

자바 컨테이너

- 자바 객체의 생명 주기를 관리하며, 생성된 자바 객체(=bean)들에게 추가적인 기능을 제공하는 역할
- 객체 생성, 소멸과 같은 제어 흐름 관리

@Test

- JUnit에서 테스트를 수행하는 메소드

```
class MemberServiceTest {
    //...생략...
    @Test
    public void 중복_회원_에러() {
        //given
        Member member1 = new Member();
        member1.setName("spring");

        Member member2 = new Member();
        member2.setName("spring");

        //when
        memberService.join(member1);    //회원가입
        IllegalStateException e = assertThrows(IllegalStateException.class, () -
> memberService.join(member2));
        // member2를 넣을 때, IllegalStateException.class가 발생해야 한다,

        //then
        assertThat(e.getMessage()).isEqualTo("이미 존재하는 회원입니다."); //출력 에러
가 동일한지 검증
    }
}
```

테스트 케이스 작성

- 작성한 코드가 제대로 작동을 하는지 확인하기 위한 검증 방법으로 사용
- JUnit 프레임워크이기 때문에 각각의 테스트가 서로에게 의존관계없이 **독립적으로 실행**된다.
- 메소드명을 한글로 적어도 된다. 직관적으로 이해할 수 있을 정도로만 작성한다.
- 처음 테스트를 작성할 때는 given - when - then 순서대로 작성하는 것을 권장

테스트 주도 개발(TDD)

- 검증할 수 있는 테스트를 먼저 만들어두고 작품이 완성되었을 때 이 틀에 알맞게 들어가는 지를 확인

@AfterEach

- 각 메소드가 끝날 때마다 동작을 하는 메소드 정의
- 각 테스트들은 서로 독립적으로 실행되어야 하기 때문에 저장소, 공용 데이터를 지운다.

```
class MemberServiceTest {
    MemoryMemberRepository memberRepository;

    @AfterEach
    public void afterEach() {
        memberRepository.clearStore();
    }
    //...생략...
}
```

@BeforeEach

- 각 테스트 메소드 실행 전에 호출된다.
- 테스트가 서로 영향이 없도록 항상 새로운 객체를 생성하고, 의존관계도 새로 맺어준다

```
class MemberServiceTest {
    @BeforeEach
    public void beforeEach() {
        memberRepository = new MemoryMemberRepository();
        memberService = new MemberService(memberRepository);
    }
    //...생략...
}
```

@GetMapping()

요청한 localhost의 URL 내용과 `@GetMapping(String value)`의 value가 일치하면 해당 메소드 실행

```
@Controller
public class MemberController {
    // -- 생략 --
    @GetMapping("/members")
    public String list(Model model) {
        List<Member> members = memberService.findMember();
        model.addAttribute("members", members);
        return "members/memberList";    //templates/members/memberList.html로 넘어
간다.
    }
}
```

Model 객체란?

- 화면에서 전달 받은 데이터를 담고 있는 객체
- Controller에서 생성한 데이터를 담아서 View로 전달한다.
- addAttribute("key", "value") 메소드 사용해서 model 객체에 저장한다.

@PostMapping()

뷰에서 전달한 URL 내용과 `@PostMapping(String value)`의 value가 일치하면 해당 메소드 실행

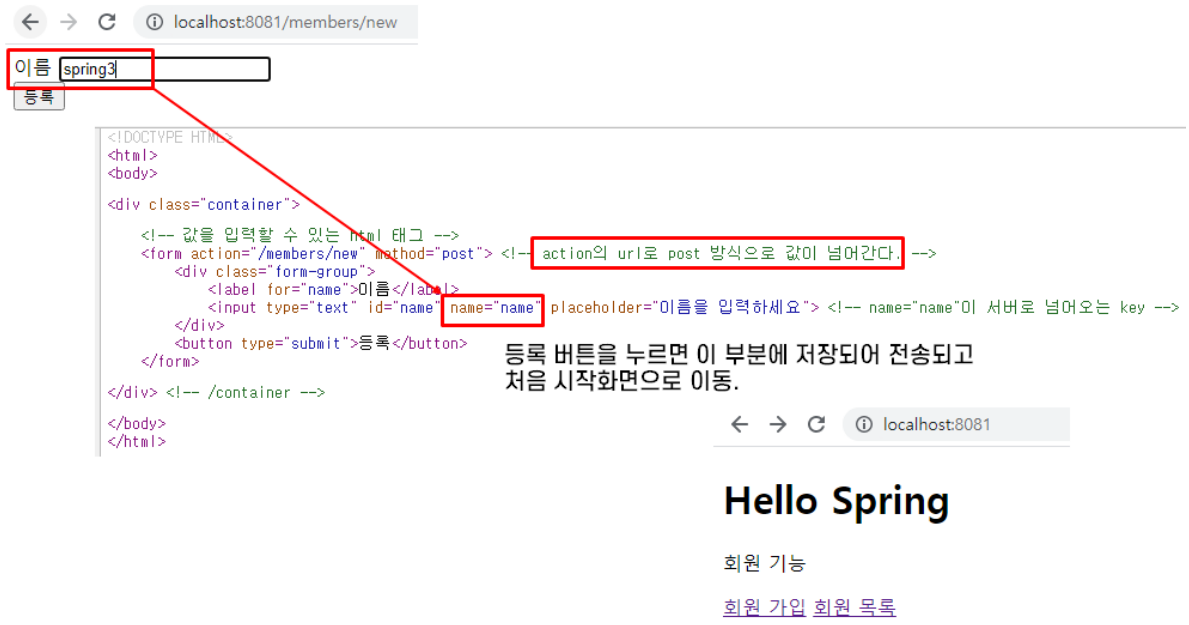
```
@Controller
public class MemberController {
    @PostMapping("/members/new")
    public String create(MemberForm form) { //MemberForm클래스에 받아온 value가
form에 저장
        Member member = new Member();
        member.setName(form.getName());
        memberService.join(member); //회원가입
        return "redirect:/";    // 시작화면 html로 이동
    }
}
```

Get vs Post

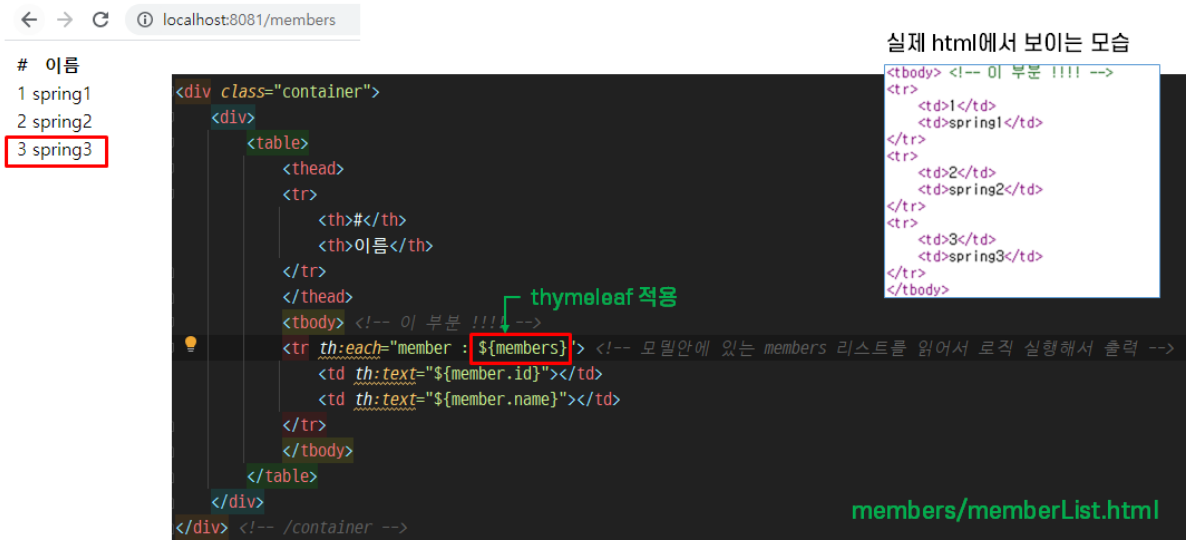
- GET 이란?
  - 클라이언트에서 서버로 **정보를 요청**하기 위해 사용되는 메서드
  - 조회할 때 주로 사용한다.
    - 예> 게시판의 게시물 조회
  - GET을 통한 요청은 **쿼리 스트링(queryString) 방식**으로 url 끝에 파라미터를 통해 전송된다.
    - ex) localhost:8080/hello-mvc?**name=spring**
  - 길이제한이 있음
  - 중요한 정보는 다루면 안된다.
    - 요청 내용이 파라미터에 다 노출되기 때문
  - 데이터를 요청할때만 사용한다.
- Post 란?
  - 클라이언트에서 서버로 데이터를 보낼 때 사용되는 메서드
  - 전송할 데이터를 **HTTP 메시지 body 부분**에 담아서 서버로 전송
  - 길이 제한이 없어서 용량이 큰 데이터 전송 가능
  - body 부분에 담아서 전송하기 때문에 보안이 필요한 부분에서 많이 사용
    - 개인정보, 비밀번호 데이터 전송

위 코드 실행 모습

1. 데이터 입력 및 전송 – POST 방식 사용



2. 회원 목록 조회 요청 – GET 방식 사용



@ResponseBody

- 객체를 HTTP Response Body 부분에 바로 전송한다는 의미
- 이 어노테이션을 사용하면 뷰 리졸버(`viewResolver`)를 사용하지 않는다.
- **JSON 구조**로 반환한다.
  - key : value 로 이루어짐.

@RequestParam()

- @RequestParam(value="원하는 파라미터 명", required=true)
- **get방식**으로 넘어온 **url의 파라미터**를 받아온다.
- 쿼리 스트링 부분을 읽는다.
- required의 기본값은 true이기 때문에 @RequestParam 사용 시 작성하지 않아도 된다.
  단, 선언한 파라미터에는 반드시 있어야 한다. -> 없으면 에러 발생

```java
@Controller
public class HelloController {
    //...생략...
    @GetMapping("hello-api") //일반적인 API 방식
    @ResponseBody
    public Hello helloApi(@RequestParam("name") String name) {
        Hello hello = new Hello(); // 객체 생성
        hello.setName(name);
        return hello; //JSON 구조로 반환된다.
    }
}
```