1. API 개발 기본

#1.인강/jpa활용편/활용편2/강의

- /회원 등록 API
- /회원 수정 API
- /회원 조회 API

회원 등록 API

준비

- postman 설치 (https://www.getpostman.com)

주의! - 스프링 부트 3.x 버전 선택 필수

start.spring.io 사이트에서 스프링 부트 2.x에 대한 지원이 종료되어서 더는 선택할 수 없습니다.
이제는 스프링 부트 3.0 이상을 선택해주세요.

스프링 부트 3.0을 선택하게 되면 다음 부분을 꼭 확인해주세요.

- **1. Java 17 이상**을 사용해야 합니다.
- **2. javax 패키지 이름을 jakarta로 변경**해야 합니다.
  - 오라클과 자바 라이센스 문제로 모든 `javax` 패키지를 `jakarta`로 변경하기로 했습니다.
- **3. H2 데이터베이스를 2.1.214 버전 이상 사용해주세요.**

패키지 이름 변경 예)

- **JPA 애노테이션**
  - javax.persistence.Entity ➜ jakarta.persistence.Entity
- **스프링에서 자주 사용하는 @PostConstruct 애노테이션**
  - javax.annotation.PostConstruct ➜ jakarta.annotation.PostConstruct
- **스프링에서 자주 사용하는 검증 애노테이션**
  - javax.validation ➜ jakarta.validation

스프링 부트 3.0 관련 자세한 내용은 다음 링크를 확인해주세요: https://bit.ly/springboot3

회원 등록 API

```
package jpabook.jpashop.api;

import jpabook.jpashop.domain.Member;
import jpabook.jpashop.service.MemberService;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.RequiredArgsConstructor;
import org.springframework.web.bind.annotation.*;

import javax.validation.Valid;
import java.util.List;
import java.util.stream.Collectors;

@RestController
@RequiredArgsConstructor
public class MemberApiController {

    private final MemberService memberService;

    /**
     * 등록 V1: 요청 값으로 Member 엔티티를 직접 받는다.
     * 문제점
     * - 엔티티에 프레젠테이션 계층을 위한 로직이 추가된다.
     *    - 엔티티에 API 검증을 위한 로직이 들어간다. (@NotEmpty 등등)
     *    - 실무에서는 회원 엔티티를 위한 API가 다양하게 만들어지는데, 한 엔티티에 각각의 API를 위한
모든 요청 요구사항을 담기는 어렵다.
     * - 엔티티가 변경되면 API 스펙이 변한다.
     * 결론
     * - API 요청 스펙에 맞추어 별도의 DTO를 파라미터로 받는다.
     */
    @PostMapping("/api/v1/members")
    public CreateMemberResponse saveMemberV1(@RequestBody @Valid Member member)
{
        Long id = memberService.join(member);
        return new CreateMemberResponse(id);
    }

    @Data
    static class CreateMemberRequest {
        private String name;
    }
```

```
    @Data
    static class CreateMemberResponse {
        private Long id;

        public CreateMemberResponse(Long id) {
            this.id = id;
        }
    }
}
```

V1 엔티티를 Request Body에 직접 매핑

- 문제점
  - 엔티티에 프레젠테이션 계층을 위한 로직이 추가된다.
  - 엔티티에 API 검증을 위한 로직이 들어간다. (@NotEmpty 등등)
  - 실무에서는 회원 엔티티를 위한 API가 다양하게 만들어지는데, 한 엔티티에 각각의 API를 위한 모든 요청 요구사항을 담기는 어렵다.
  - 엔티티가 변경되면 API 스펙이 변한다.
- 결론
  - API 요청 스펙에 맞추어 별도의 DTO를 파라미터로 받는다.

V2 엔티티 대신에 DTO를 RequestBody에 매핑

```
    /**
     * 등록 V2: 요청 값으로 Member 엔티티 대신에 별도의 DTO를 받는다.
     */
    @PostMapping("/api/v2/members")
    public CreateMemberResponse saveMemberV2(@RequestBody @Valid
CreateMemberRequest request) {

        Member member = new Member();
        member.setName(request.getName());

        Long id = memberService.join(member);
        return new CreateMemberResponse(id);
    }
```

- `CreateMemberRequest`를 `Member`엔티티 대신에 RequestBody와 매핑한다.
- 엔티티와 프레젠테이션 계층을 위한 로직을 분리할 수 있다.
- 엔티티와 API 스펙을 명확하게 분리할 수 있다.
- 엔티티가 변해도 API 스펙이 변하지 않는다.

> 참고: 실무에서는 엔티티를 API 스펙에 노출하면 안된다!

회원 수정 API

```
/**
 * 수정 API
 */
@PutMapping("/api/v2/members/{id}")
public UpdateMemberResponse updateMemberV2(@PathVariable("id") Long id,
@RequestBody @Valid UpdateMemberRequest request) {
    memberService.update(id, request.getName());
    Member findMember = memberService.findOne(id);
    return new UpdateMemberResponse(findMember.getId(), findMember.getName());
}

@Data
static class UpdateMemberRequest {
    private String name;
}

@Data
@AllArgsConstructor
static class UpdateMemberResponse {
    private Long id;
    private String name;
}
```

- 회원 수정도 DTO를 요청 파라미터에 매핑

```
public class MemberService {

    private final MemberRepository memberRepository;

    /**
     * 회원 수정
     */
    @Transactional
    public void update(Long id, String name) {
        Member member = memberRepository.findOne(id);
        member.setName(name);
```

```
    }

}
```

- 변경 감지를 사용해서 데이터를 수정

> 오류정정: 회원 수정 API `updateMemberV2`은 회원 정보를 부분 업데이트 한다. 여기서 PUT 방식을 사용했는데, PUT은 전체 업데이트를 할 때 사용하는 것이 맞다. 부분 업데이트를 하려면 PATCH를 사용하거나 POST를 사용하는 것이 REST 스타일에 맞다.

회원 조회 API

회원조회 V1: 응답 값으로 엔티티를 직접 외부에 노출

```
package jpabook.jpashop.api;

@RestController
@RequiredArgsConstructor
public class MemberApiController {

    private final MemberService memberService;

    /**
     * 조회 V1: 응답 값으로 엔티티를 직접 외부에 노출한다.
     * 문제점
     * - 엔티티에 프레젠테이션 계층을 위한 로직이 추가된다.
     *    - 기본적으로 엔티티의 모든 값이 노출된다.
     *    - 응답 스펙을 맞추기 위해 로직이 추가된다. (@JsonIgnore, 별도의 뷰 로직 등등)
     *    - 실무에서는 같은 엔티티에 대해 API가 용도에 따라 다양하게 만들어지는데, 한 엔티티에 각각의 API를 위한 프레젠테이션 응답 로직을 담기는 어렵다.
     * - 엔티티가 변경되면 API 스펙이 변한다.
     * - 추가로 컬렉션을 직접 반환하면 항후 API 스펙을 변경하기 어렵다.(별도의 Result 클래스 생성으로 해결)
     * 결론
     * - API 응답 스펙에 맞추어 별도의 DTO를 반환한다.
     */
    //조회 V1: 안 좋은 버전, 모든 엔티티가 노출, @JsonIgnore -> 이건 정말 최악, api가 이거 하나인가! 화면에 종속적이지 마라!
    @GetMapping("/api/v1/members")
    public List<Member> membersV1() {
        return memberService.findMembers();
    }
```

```
}
```

조회 V1: 응답 값으로 엔티티를 직접 외부에 노출한

- 문제점
  - 엔티티에 프레젠테이션 계층을 위한 로직이 추가된다.
  - 기본적으로 엔티티의 모든 값이 노출된다.
  - 응답 스펙을 맞추기 위해 로직이 추가된다. (@JsonIgnore, 별도의 뷰 로직 등등)
  - 실무에서는 같은 엔티티에 대해 API가 용도에 따라 다양하게 만들어지는데, 한 엔티티에 각각의 API를 위한 프레젠테이션 응답 로직을 담기는 어렵다.
  - 엔티티가 변경되면 API 스펙이 변한다.
  - 추가로 컬렉션을 직접 반환하면 항후 API 스펙을 변경하기 어렵다.(별도의 Result 클래스 생성으로 해결)
- 결론
  - API 응답 스펙에 맞추어 별도의 DTO를 반환한다.

> 참고: 엔티티를 외부에 노출하지 마세요!
> 실무에서는 `member` 엔티티의 데이터가 필요한 API가 계속 증가하게 된다. 어떤 API는 `name` 필드가 필요하지만, 어떤 API는 `name` 필드가 필요없을 수 있다. 결론적으로 엔티티 대신에 API 스펙에 맞는 별도의 DTO를 노출해야 한다.

회원조회 V2: 응답 값으로 엔티티가 아닌 별도의 DTO 사용

```
    /**
     * 조회 V2: 응답 값으로 엔티티가 아닌 별도의 DTO를 반환한다.
     */
    @GetMapping("/api/v2/members")
    public Result membersV2() {

        List<Member> findMembers = memberService.findMembers();
        //엔티티 -> DTO 변환
        List<MemberDto> collect = findMembers.stream()
                .map(m -> new MemberDto(m.getName()))
                .collect(Collectors.toList());

        return new Result(collect);
    }
```

```
@Data
@AllArgsConstructor
static class Result<T> {
    private T data;
}

@Data
@AllArgsConstructor
static class MemberDto {
    private String name;
}
```

- 엔티티를 DTO로 변환해서 반환한다.
- 엔티티가 변해도 API 스펙이 변경되지 않는다.
- 추가로 `Result` 클래스로 컬렉션을 감싸서 향후 필요한 필드를 추가할 수 있다.