Spring – 자바의 프레임 워크

Intellij를 기반으로 작성.

Start.spring.io 에서 기본적인 프로젝트를 다운받는다.(spring boot)

스프링기반의 프로젝트를 만들어주는 사이트

Maven, gradle – 필요한 라이브러리를 가져와주고, 빌드 라이프? 사이클을 관리해주는 툴

과거에는 메이븐이었지만 요즘에는 그레이들을 선호

그룹명은 보통 기업 도메인. 우리는 아무거나 적어도 된다. 예제는 hello

Artifact는 빌드된 결과물(프로젝트명 같은것). Hello-spring

Dependencies는 어떠한 라이브러리를 사용할 것인지를 묻는 것.

우리는 web프로젝트 이므로 spring web을 선택한다.

또, thymeleaf라는 템플릿 엔진을 사용한다.

템플릿 엔진 - HTML 코드에서 고정적으로 사용되는 부분은 템플릿으로 만들어두고 동적으로 생성되는 부분만 템플릿 특정 장소에 끼워넣는 방식으로 동작할 수 있도록 해준다.

이후 generate하면 프로젝트를 다운받는다.

Build.grdadle을 프로젝트로 열기로 연다.

Main->java->hello-spring의 hellospringApplication.java의 main메소드를 빌드하면 8080포트로 서버가 열린다.

접속에 성공하면 기본적인 설정이 끝이난다.

Spring은 내장 tomcat이 있어서 자동으로 서버를 열게 된다.

intellij설정에서 빌드를 gradle이 아닌 intellij를 통해 빌드되도록 수정한다.

기본설정 끝

System.print.out 보다는 log출력을 할 것.

Src- main-java-hello-spring밑에 controller라는 패키지를 만든다.

패키지 안에 HelloController.java파일을 만들고

@Controller  
public class HelloController {  
 @GetMapping("hello")// /hello라는 호출이 들어오면 이하의 메소드 실행  
 public String hello(Model model){  
 model.addAttribute("data","hello");  
 return "hello";  
 }  
}

이렇게 수정한다.

여기서의 controller와 model은 MVC모델의 그 controller와 model이다.

GetMapping의 Get은 Get과 Post방식이다.

이 코드는 Get방식으로 /hello가 넘어오면 Model객체에 Data키에 hello라는 value를 생성하고

hello라는 템플릿 페이지를 return 한다. 해당 페이지는 아래에서 설명한다.

src – main – resources – templates에 hello.html파일을 만든다.

<!DOCTYPE HTML>  
<html xmlns:th="http://www.thymeleaf.org">  
<head>  
 <title>Hello</title>  
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />  
</head>  
<body>  
<p th:text="'안녕하세요. ' + ${data}" >안녕하세요. 손님</p>  
</body>  
</html>

여기서 th는 thymeleaf이며

<html xmlns:th="http://www.thymeleaf.org">

는 타임리프 엔진이 템플릿 엔진이 선언되어 있다.

따라서 타임리프 문법을 사용할 수 있다.

p테그의 th:text는 값이 있다면 해당 내용으로 출력되지만 없다면 p태그의 내용이 출력된다.

빌드하기

인텔리j에 git bash를 터미널로 설정한 상태를 기준으로 설명한다.

프로젝트 폴더에서 ./gradlew build 입력

이후 cd build/libs 이동

Java -jar hello-spring-0.0.1-SNAPSHOT.jar 실행 확인

서버 배포시 해당 파일만 복사해서 서버에 넣어서 실행하면 된다.

웹 개발 방식

1. 정적 컨텐츠

서버에서 하는 것 없이 파일을 그대로 내려준다

1. MVC와 템플릿 엔진

서버상에서 동적으로 프로그래밍 해서 주는 것.

1. API

안드로이드나 아이폰 등의 개발시 json등의 데이터 형식으로 전달.

뷰,리엑트등이나 서버간 통신에서 사용

스프링은 정적 컨텐츠를 기본으로 지원한다. static폴더상의 파일에 대해서.

MVC

@GetMapping("hello-mvc")//get방식으로 hello-mvc접속  
public String helloMvc(@RequestParam("name") String name, Model model){//입력된 파라미터를 name키에 name value로 저장  
 model.addAttribute("name",name);//모델에 name키에 name 값으로 저장  
 return "hello-template";//리턴  
}

접속시 주소를 localhost:8080/hello-mvc?name=spring!으로 해보자.

template폴더의 hello-template.html 파일

<html xmlns:th="http://www.thymeleaf.org">  
<body>  
<p th:text="'hello ' + ${name}">hello! empty</p>  
</body>  
</html>

받아온 name객체의 내용을 표시한다.

Api

@GetMapping("hello-string")//get방식으로 hello-mvc접속  
@ResponseBody //html이 아닌 http의 body에 직접 내용 주입  
public String helloString(@RequestParam("name") String name){//입력된 파라미터를 name키에 name value로 저장  
 return "hello" + name;//리턴  
}

이후 localhost:8080/hello-string?name=hello 로 접속하면

화면상에 hello hello 라고 출력되지만 소스보기를 누르면 아무런 html태그없이 글자만 있다는 것을 알 수 있다.

@GetMapping("hello-api")//get방식으로 hello-api접속  
@ResponseBody //html이 아닌 http의 body에 직접 내용 주입  
public Hello helloApi(@RequestParam("name") String name){//입력된 파라미터를 name키에 name value로 저장  
 Hello hello = new Hello();  
 hello.setName(name);  
 return hello;  
}  
  
static class Hello{//클래스 내부에서 사용할 수 있는 클래스 객체  
 private String name;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
}

스태틱 클래스로 클래스를 만들어 미리 만든 기능을 사용할 수 있다.

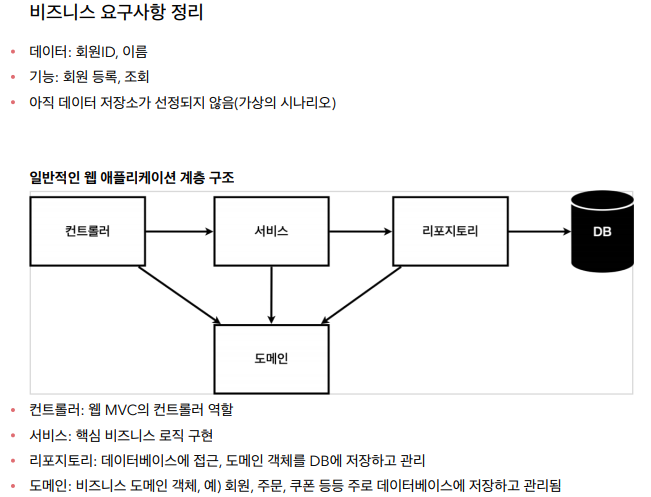
Localhost:8080/hello-api?name=tt로 접속할 경우

String name값이 Hello타입의 hello의 name에 저장된다.(set name메소드)

그리고 hello 객체를 리턴하면(클래스가 Hello이므로 Hello객체 리턴)

Json 형식으로 {name:tt}가 출력 되는 것을 볼 수 있다.

회원관리 예제



도메인 패키지를 만들어 회원정보를 관리한다.

리포지토리 패키지를 만들어 DB를 관리한다. 단 현 예제에서는 db를 결정하지 못하였다는 설정이므로 인터페이스로 기능을 설정한 후 우선 파일 형식으로 관리한다.

도메인

public class Member {  
 private Long id;//회원이 저장하는 아이디가 아닌 시스템에서 저장하는 아이디  
 private String name;  
  
 public Long getId() {  
 return id;  
 }  
  
 public void setId(Long id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
}

Member.java 라는 멤버관리를 위한 파일이다.

아이디와 이름으로 구성되며, getter와 setter를 이용하여 값을 가져오고 지정한다.

리포지토리

public interface MemberRepository {  
 Member save(Member member);//저장기능 저장하면 저장된 회원 반환  
 Optional<Member> findById(Long id);  
 Optional<Member> findByName(String name);  
 List<Member> findAll();//모든 회원 리스트 반환  
}

MemberRepository라는 인터페이스를 만든다.

아직 db형식을 결정하기 전이므로 어떤 방식에서도 사용할 수 있도록 틀을 잡는다.

인터페이스에서는 자료형, 기능의 이름, 인자 등만을 지정한다.

Optional은 null 처리를 위한자료형이다.

우선 예제에서는 메모리(파일)로 저장한다.

public class MemoryMemberRepository implements MemberRepository{  
  
 private static Map<Long,Member> *store* = new HashMap<>();  
 private static long *sequence* = 0L;//키값을 생성  
  
 @Override  
 public Member save(Member member) {  
 member.setId(++*sequence*);  
 *store*.put(member.getId(),member);  
 return member;  
 }  
  
 @Override  
 public Optional<Member> findById(Long id) {  
 return Optional.*ofNullable*(*store*.get(id));//NULL이 반환되더라도 optional로 감싸서 반환 가능 추후처리 용의  
 }  
  
 @Override  
 public Optional<Member> findByName(String name) {  
 return *store*.values().stream()  
 .filter(member -> member.getName().equals(name))  
 .findAny();  
 }  
  
 @Override  
 public List<Member> findAll() {  
 return new ArrayList<>(*store*.values());  
 }  
}

위의 인터페이스를 implemets하고 각 기능을 구현한다.

Map은 키와 value를 갖는 자료형으로 객체과 비슷하다고 생각한다.

Map형식 변수 store는 Key로 Long형식으로 구분하고, value로 Member형을 갖는다.

즉 id를 이용해서 회원을 관리한다고 볼 수 있다.

Long sequence는 간단하게 키값을 생성하기 위한 숫자형이고, long형임을 지정하기 위해 0L로 지정한다. 0으로 그냥 쓰면 int형으로 인식된다.

save기능에서 member를 인자로 받는다.

여기서 회원의 id는 회원이 직접 입력 하는 것이 아닌 시스템이 부여한다고 가정한다.

Sequence의 값을 증가시켜 해당 값을 id로 해당 회원에게 부여한다.

그 후 store변수에 해당 회원 정보를 추가한뒤 회원정보를 반환한다.

findById기능을 구현한다.

store변수에서 get메소드를 통해 입력받은 찾으려는 id를 찾아내어 value인 회원정보를 반환한다. 하지만 null일수 있기 때문에 Optional의 메소드인 ofNullable로 감싸 null이 반환이 가능하게 만든다.

findByName을 구현한다.

Key가 아닌 value를 기준으로 찾는 것이기 때문에 모든 회원정보를 확인해야 한다.

반복문 대신 store에 저장된 값들을 가져오는 메소드인 values를 이용하여 저장된 모든 Member객체들을 가져오고, stream메소드를 통해 전체를 loop한다.

Filter를 통해서 stream을 통해 순회하는 값을 member로 지정하고 -> 다음에 걸러내려는 기준을 적용한다. 해당 member의 이름을 가져와서 찾으려는 String name과 같은지 확인하기위해

-> member.getName().equals(name)을 사용한다. 이 값이 참인 어떠한 값이든 반환하기 위해 마지막에 .findAny()를 붙인다.

Store의 value인 member가 Optional로 감싸져서 반환된다.

Findall 기능을 구현한다

모든 회원정보를 반환하기 위해 List형식을 사용하고(자료형 Member)

반환은 간단하게 return new ArrayList<>(store.values()); 로 한다.

Values로 인해 store의 모든 value인 member들이 array에 추가되어 반환된다.

Test case작성

테스트를 위해 main의 애플리케이션과 컨트롤러를 이용할 수 있지만 준비가 번거롭고 여러테스트를 하기 어렵다는 단점을 극복하기 junit이라는 프레임워크로 테스트를 실행한다.

main폴더가 아닌 test폴더의 hello.hellospring 에 main과 동일하게 repository패키지를 생성한다.

해당 패키지에 테스트 할 MemoryMemberRepository의 테스트인 MemoryMemberRepositoryTest클래스를 생성한다.

MemoryMemberRepository repository = new MemoryMemberRepository();  
  
@Test  
public void save(){  
 Member member = new Member();  
 member.setName("spring");  
  
 repository.save(member);  
 Member result = repository.findById(member.getId()).get();  
 //System.out.println("result : " + (result==member));  
  
 //Assertions.assertEquals(member,null); junit.jupiter  
   
 //Assertions.assertThat(member).isEqualTo(result);//assertj  
 *assertThat*(member).isEqualTo(result);//Assertions에서 alt+enter 하면 import static org.assertj.core.api.Assertions.\*; 추가됨  
  
}

@Test를 붙이고 import해두면 해당 클래스에서 테스트를 바로 수행할 수 있다.

save기능이 동작하는지 확인하기 위해 member인자를 만들고 findById를 통해 반환된 optional에서 get()으로 추출하고, 추출한 result와 원본인 member를 비교한다.

비교방식은 다양하며 주석으로 설명을 대체한다.



테스트 성공시 좌측 실패시 우측으로 나타나며

System out.println방식의 경우 콘솔창에 글씨로 나타나나 권장하지 않는다.

@Test  
public void findByName(){  
 Member member1 = new Member();  
 member1.setName("spring1");  
 repository.save((member1));  
  
 Member member2 = new Member();  
 member2.setName("spring2");  
 repository.save((member2));  
  
 Member result1 = repository.findByName("spring1").get();  
 *assertThat*(result1).isEqualTo(member1);  
}

이름찾기 테스트를 위해 두가지의 회원을 등록하고, 이름으로 찾은 정보와 실 정보가 맞는지 비교한다.

@Test  
public void findAll(){  
 Member member1 = new Member();  
 member1.setName("spring1");  
 repository.save((member1));  
  
 Member member2 = new Member();  
 member2.setName("spring2");  
 repository.save((member2));  
  
 List<Member> result = repository.findAll();  
 *assertThat*(result.size()).isEqualTo(2);  
}

Findall 을 위해 모든 정보를 반환하는 리스트의 크기를 비교한다.

모든 테스트는 한번에 실행될 때 별개로 순서없이 수행되므로 테스트간 간섭이 있을 수 있다. 따라서 모든 테스트는 수행 완료 후 리셋되어야 한다.

@AfterEach  
public void afterEach(){  
 repository.clearStore();  
}

테스트 코드 상단에 해당 코드를 삽입한다.

@AfterEach는 각 메소드가 수행을 완료할 때 자동으로 수행되는 메소드를 나타낸다.

MemoryMemberRepository에 메소드를 추가한다.

public void clearStore(){  
 *store*.clear();  
}

이 코드는 store의 저장된 모든 정보를 지운다.

회원 서비스 개발

Hello.hellospring에 서비스라는 패키지를 만든다.

MemberService라는 클래스를 생성한다.

public class MemberService {  
 private final MemberRepository memberRepository = new MemoryMemberRepository();  
  
 */\*\*  
 회원가입  
 \*/* public Long join(Member member){  
 //중복이름 X  
 */\*\*  
 \* Optional<Member> result = memberRepository.findByName(member.getName());  
 result.ifPresent(m->{  
 throw new IllegalStateException("이미 존재하는 이름입니다.");  
 });\*/  
 /\*\*memberRepository.findByName(member.getName()).ifPresent(m->{  
 throw new IllegalStateException("이미 존재하는 이름입니다.");  
 });\*/* validateDuplicateMember(member);  
 memberRepository.save(member);//증복회원 검증  
 return member.getId();  
 }  
  
 private void validateDuplicateMember(Member member) {  
 memberRepository.findByName(member.getName()).ifPresent(m->{  
 throw new IllegalStateException("이미 존재하는 이름입니다.");  
 });  
 }  
 public List<Member> findMembers(){  
 return memberRepository.findAll();  
 }  
 public Optional<Member> findOne(Long memberId){  
 return memberRepository.findById(memberId);  
 }  
}

우선 회원가입 부분을 만든다.

저장소에 가입기능이 있기 때문에 단순히 인자로 받아온 Member객체의 save기능을 이용하여 가입을 한 뒤 id를 리턴한다.

단, 중복이름을 방지하는 기능을 추가하기 위해 입력된 Member개체의 이름을 기준으로 findByName을 하여 반환값을 확인한다.

반환값을 확인하는 방법으로 Optional의 기능인 .ifPresent를 사용하면 반환값이 존재할 때 ( )내부의 기능을 실행한다.

반환값이 존재하는 경우 m으로 받고, 새 IllegalStateException 메시지를 던진다.

이미 반환값의 자료형을 알고 있기 때문에 변수를 선언하지 않고 바로 사용할 수 있으며, 해당 기능은 clt+alt+shift+T로 함수로추출하는 것이 좋다.

이후 모든 member객체를 확인하는 함수와 하나의 Member객체만 확인하는 함수를 생성한다.

회원 서비스 테스트

기존처럼 test 폴더 하위에 일일이 패키지와 클래스를 만들 필요 없이 ctl+shift+T 하면 테스트를 자동으로 만들어준다. Junit5와 이름, 테스트할 함수들을 확인한다.

테스트 코드에서 이름은 한글로 써도 무방하다.

테스트 코드를 작성할 때 given, when, then 문법 사용하기

주석으로 표기만 하는 것으로 충분하며, //given 주어질 때, //when실행했을 때, //then 결과

@Test  
void join() {  
 //given  
 Member member = new Member();  
 member.setName("hello");  
  
 //when  
 Long saveId = memberService.join(member);  
  
 //then  
 Member findMember = memberService.findOne(saveId).get();  
 *assertThat*(findMember.getName()).isEqualTo(member.getName());  
}

회원가입 테스트

새 회원 하나의 이름을 주고 서비스의 기능을 이용하여 가입을 시킨 뒤, 반환받은 id로 회원을 찾은 뒤 원본의 정보와 비교한다.

맞을 때의 정보만을 비교하지 않고 틀릴경우. 이 경우에는 중복 이름의 경우도 테스트 해야 한다.

@Test  
public void 중복\_회원\_예외(){  
 //given  
 Member member1 = new Member();  
 member1.setName("spring");  
 Member member2 = new Member();  
 member2.setName("spring");  
  
 //when  
 memberService.join(member1);  
 */\*\*try {  
 memberService.join(member2);  
 fail();  
 }  
 catch (IllegalStateException e){  
 assertThat(e.getMessage()).isEqualTo("이미 존재하는 이름입니다1.");  
 }\*/* IllegalStateException e = *assertThrows*(IllegalStateException.class, () -> memberService.join(member2));  
 *assertThat*(e.getMessage()).isEqualTo("이미 존재하는 이름입니다.");  
  
  
}

Try catch로 2번째 입력시 에러발생을 확인할 수 있다.

받은 에러메세지와 설정한 메시지가 동일한지 확인할 수 있는데, 이보다는 arrertThorws가 권장된다.

람다식안의 내용이 수행되면 에러를 받는데, 에러가 있다면 통과되고 에러를 반환한다.

에러 메시지가 같은지 확인하면 된다.

하지만 지금 테스트 환경에서는 테스트 종료시 초기화를 하지 않고 있으므로 테스트를 초기화하도록 코드를 추가한다.

MemoryMemberRepository memberRepository = new MemoryMemberRepository();  
  
@AfterEach  
public void afterEach(){  
 memberRepository.clearStore();  
}

테스트 클래스에서 저장소를 memberservice에서 하나, 본문에서 하나 만들어 사용하고 있다.

이는 테스트나 실 사용에서 문제가 될 수 있으므로 코드를 수정해야 한다.

Service의 코드를

private final MemberRepository memberRepository = new MemoryMemberRepository();

에서

Alt + insert를 통해 construct를 이용하여.

private final MemberRepository memberRepository;  
  
 public MemberService(MemberRepository memberRepository) {  
 this.memberRepository = memberRepository;  
 }

로 바꿔준다. 이는 초기화시 새로 만드는 것이 아닌 기존의 저장소를 받아와 연결한다.

이후 테스트 코드를 수정한다

MemberService memberService;  
MemoryMemberRepository memberRepository;  
  
@BeforeEach  
public void beforeEach()  
{  
 memberRepository = new MemoryMemberRepository();  
 memberService=new MemberService(memberRepository);  
}

이렇게 하면 각 테스트를 수행하기 전에 저장소를 새로 만들고 해당 저장소를 사용하는 서비스를 연결한다. 이를 DI = dependency injection라고 한다

스프링 빈

서비스와 저장소가 완성되었으니 화면에 보이기 위해서 컨트롤러와 뷰템플릿이 필요하다.

컨트롤러는 서비스를 통해 회원가입이 가능해야 하고, 조회할 수 있어야 한다.

이를 의존관계가 있다고 표현한다.

컨트롤러 패키지에 MemberController를 만든다.

@Controller  
public class MemberController {  
   
}

@Controller 를 하면

스프링 컨테이너에 MemberCotrolloer객체(빈)를 생성해서 컨테이너에 넣어둔다. 그 뒤 스프링이 관리한다.

스피링 빈은 스프링 컨테이너가 관리하는 자바 객체를 의미한다.

컨테이너에서 빈이 관리된다

스프링을 사용하면. 스프링 컨테이너에 객체들을 등록하고 꺼내써야한다.

@Controller  
public class MemberController {  
 private final MemberService memberService;  
  
 @Autowired  
 public MemberController(MemberService memberService) {  
 this.memberService = memberService;  
 }  
}

private final MemberService memberService;

를 new MemberService() 할 수 있지만 다른 컨트롤러에서 도 사용할 수 있고 하나만 만들어 공유하면 되기 때문에 컨테이너에 등록하기 위해 alt + insert 로 constructor를 불러온다.

@Autowired는, 스프링 컨테이너에 MemberController객체가 생성될 때 생성자(Autowired이하부분)가 호출 되는데 이때 Memberservice를 컨테이너 내부의 Meberservice와 연결해주는 역할을 한다.

이 때에 HelloSpringApplication을 실행시키면 MemberService를 찾을 수 없다고 나타난다.

이는 컨트롤러에서는 @Autowired를 통해 컨테이너에서 memverservice를 가져오도록 되어있지만

Memberservice는 컨테이너에 등록되어있지 않기 때문이다.

마잔가지로 MemoryMemberRepository(구현체) 또한 순수한 java상태이기 때문에 두 파일의 최상단에

@Service와 @Repository를 붙인다.

Controller, Service, Repository는 정형화된 패턴이다

Controller를 통해 외부 요청을 받고, Service에서 비즈니스 로직을 만들고 Repository에서 데이터를 저장하는게 정형화된 패턴으로 존재한다.

스프링이 시작될 때 Controller, service,repo를 다 가져와서, controller가 필요로하는 service를 연결시켜주는데 (autowired) 이것이 DI dependency injection 의존성 주입이다.

Service에서도 마찬가지로 Auto wired를 해줘야 저장소를 연결시켜줄 수 있다.

@Service  
public class MemberService {  
 private final MemberRepository memberRepository;  
  
 @Autowired  
 public MemberService(MemberRepository memberRepository) {  
 this.memberRepository = memberRepository;  
 }

이후 정상적으로 동작하는 것을 알 수 있다.

스프링 빈을 등록하는 방법은 2가지가 있다.

1. 컴포넌트 스캔과 자동 의존관계 설정
2. 자바코드로 직접 스프링 빈 등록하기.

1번인 컴포넌트 스캔과 자동 의존관계 설정이 지금까지의 방법이다.

사실 컴포넌트 스캔은 @Service가 아닌 @Component해야 하는데 @Service, @Controller,@Repository 내부에는 @Component가 내장되어있다.

이런식으로 @Component가 있는 객체들을 스프링이 컨테이너에 객체를 만들어 등록을 한다

@Autowired는 연관관계를 연결해준다.

컴포넌트스캔은 실행되는 위치부터 하위위치만 해당되고 그 이외는 안된다.

스프링빈은 기본적으로 싱글톤으로 등록되어 하나만 등록해서 공유한다.

자바 코드로 스프링 빈 직접 등록하기

컨트롤러는 그냥 그대로 두고 service와repository의 @Service,Repository와 @Autowired를 지운다.

최 상위 폴더에 SpringConfig를 만들고

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

실행될 때 반환값인 서비스와 저장소가 빈으로 등록이 되고, 서비스는 등록되어있는 저장소가 자동으로 연결된다.

컨트롤러와 컨트롤러에서의 Autowired는 어쩔 수 없이 드대로 사용한다.

각 방식의 장단점

1. XML로 설정하는 방식도 있지만 최근에는 잘 사용하지 않으므로 생략한다
2. DI에는 필드 주입, setter 주입, 생성자 주입 이렇게 3가지 방법이 있다. 의존관계가 실행중에 동적으로 변하는 경우는 거의 없으므로 생성자 주입을 권장한다.
3. 실무에서는 주로 정형화된 컨트롤러, 서비스, 리포지토리 같은 코드는 컴포넌트 스캔을 사용한다. 그리고 정형화 되지 않거나, 상황에 따라 구현 클래스를 변경해야 하면 설정을 통해 스프링 빈으로 등록한다.
4. @Autowired 를 통한 DI는 helloConroller , memberService 등과 같이 스프링이 관리하는 객체에서만 동작한다.(@Service, @Bean등) 스프링 빈으로 등록하지 않고 내가 직접 생성한 객체에서는 동작하지 않는다.

직접 설정파일을 통해 등록하는 경우 간단하게 교체가 가능하나, autocomponent방식은 번거롭다.

홈화면 만들기.

Controller에서 HomeController를 만든다.

@Controller//컨트롤러임 명시  
public class HomeControlloer {  
  
 @GetMapping("/")// 기본페이지 접속시  
 public String home(){  
 return "home"; //home글자 반환. 반환된 home은 templates에서 home이라는 파일을 수행한다.  
 //정확히는 home이라는 템플릿 페이지를 생성?  
 }  
}

이때 index.html이 수행되지 않았는데 이유는.

요청이 들어올 때 톰캣 내장 서버가 스프링컨테이너에 “/”에 대한 컨트롤러를 먼저 찾는다. 존재하지 않을 때 static에서 찾기 때문에 home.html이 먼저 수행된다.

회원 등록 페이지 만들기.

회원가입 링크는 localhost:8080/members/new 이다.

따라서 membercontroller에 해당 페이지에 대한 라우트?를 만든다.

@GetMapping("/members/new")  
public String createForm(){  
 return "members/createMemberForm";  
}

이렇게하면 템플릿 폴더에 members폴더 밑에 createMemberForm을 열게된다.

템플릿 폴더에 members폴더를 만들고 createMemberForm파일을 만들어둔다.

해당 파일에서 post방식으로 반환한다.

@PostMapping("/members/new")  
public String memberForm(MemberForm form)  
{  
 Member member = new Member();  
 member.setName(form.getName());  
 memberService.join(member);  
 return "redirect:/";  
}

따라서 다시 member controller에 라우트를 만든다.

@GetMapping이 아닌 PostMapping임에 유의.

public class MemberForm {  
 private String name;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
}

MemberForm 클래스를 만들어 둔다.

Getter setter를 해두면 인자로 들어간 form에 자동으로 submit 한 name이 String name으로 들어간다.

멤버 에 해당 정보를 저장하고 회원가입 시킨 뒤 메인화면으로 리다이렉트 한다.

회원 리스트 출력

회원 목록을 나타내는 주소는 localhost:8080/members이다.

따라서

@GetMapping("/members")  
public String list(Model model){  
 List<Member> members = memberService.findMembers();  
 model.addAttribute("members",members);  
 return "members/memberList";  
}

Model 객체는 컨트롤러에서 view로 전송할 때 주로 사용된다.

따라서 Model 인자에 키로 members를 사용하는 회원 리스트를 저장하고, 템플릿을 반환한다.

해당 템플릿을 만든다.

<tr th:each="member : ${members}">  
 <td th:text="${member.id}"></td>  
 <td th:text="${member.name}"></td>  
</tr>

Th는 템플릿 엔진 문법으로 each는 반복한다는 것으로, model로부터 전달받은 members키에 해당 하는 값을 member에 담고, id와 name을 꺼낸다. 아마 getter setter에 접근해서 get으로 가져오는데, 무료버전에선 잘 동작 안할지도

DB연결하기

예제에서는 가벼운 H2를 사용해본다.

설치 이후 경로로가서 bin 밑에 있는 H2.bat 을 실행시킨다. 아마 DB서버인듯..?

이후 연결을 눌러 DB를 하나 만들고 현재 JDBC URL이 파일이므로 동시접근을위해

jdbc:h2:tcp://localhost/~/test로 바꾼다.

create table member(id bigint generated by default as identity, name varchar(255),primary key(id));

를 입력해서 member테이블을 만든다.

Big int 가 Long이고 generated by default as identity는 Id가 입력이 안된 경우 자동으로 넣어준다는 의미이다.

순수 JDBC

Build.gradle 에

Implementation 'org.springframework.boot:spring-boot-starter-jdbc' runtimeOnly 'com.h2database:h2'를 추가한다.

Java는 db를 이용하려면 jdbc 드라이버가 필수이다. 또, db에서 제공하는 클라이언트가 필요하다('com.h2database:h2)

이후 db 접속정보를 application properties에 입력한다

무료버전이라 자동으로 연결을 안해줘서 안될 것 같지만 된다.

spring.datasource.url=jdbc:h2:tcp://localhost/~/test

spring.datasource.driver-class-name=org.h2.Driver

spring.datasource.username=sa

DB접근에 대한 준비는 끝났다.

JDBC를 이용하여 저장하기

Repository에 JDBCMemberRepository를 만든다.

public class JDBCMemeberRepository implements MemberRepository

기본 인터페이스에 대한 구현체 이므로 implements 해준다.

이 이하는 의미 없으므로 복붙하고 넘어간다. 20년도 더된 방식이라 알고있을 필요가 크게 없다.

그 다음 Spring Config를 수정한다.

@Bean  
public MemberRepository memberRepository(){  
  
 //return new MemoryMemberRepository();  
 return new JDBCMemberRepository(dataSource);  
}

로 기본 저장소를 교체하고

dataSource를 필요로 하므로 상단에

private DataSource dataSource;  
  
@Autowired  
public SpringConfig(DataSource dataSource) {  
 this.dataSource = dataSource;  
}

를 추가한다.

이후 db와 연결되어 동작하는 것을 확인할 수 있다.

통합 테스트

JDBC로 DB를 연결하게 되면서 기존의 순수 java코드로 진행한 테스트는(메모리방식 이므로) 사용하기 어려워졌다.

따라서 스프링과 연결된 테스트로 수정한다.

@SpringBootTest : 스프링 컨테이너와 테스트를 함께 실행한다.

@Transactional : 테스트 케이스에 이 애노테이션이 있으면, 테스트 시작 전에 트랜잭션을 시작하고, 테스트 완료 후에 항상 롤백한다. 이렇게 하면 DB에 데이터가 남지 않으므로 다음 테스트에 영향을 주지 않는다.

@BeforeEach  
public void beforeEach()  
{  
 memberRepository = new MemoryMemberRepository();  
 memberService=new MemberService(memberRepository);  
}

이 코드를 지우고 컨테이너로부터 주입받는 방식으로 수정한다.

@SpringBootTest  
@Transactional  
class MemberServiceIntegrationTest {  
 @Autowired MemberService memberService;  
 @Autowired MemberRepository memberRepository;

메모리를 사용하지 않으므로

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

이 코드도 지워 주도록 한다.

@Transactional덕분에 롤백이 되므로 일일이 지우는 코드를 넣을 필요 없다.

Spring JDBC Template

Jdbc 템플릿을 이용하여 db에 접근한다.

우선 JdbcTemplateMemberRepository를 만든다.

인터페이스를 상속한 후

private final JdbcTemplate jdbcTemplate;  
  
public JdbcTemplateMemeberRepository(JdbcTemplate jdbcTemplate) {  
 this.jdbcTemplate = jdbcTemplate;  
}

jdbctemplate를 생성한다.

Injection을 받을 수 없으므로 인자를 변경한다.

@Autowired  
public JdbcTemplateMemeberRepository(DataSourc  
jdbcTemplate = new JdbcTemplate(dataSource  
}

다음과 같이 수정한다.

사실 생성자가 하나 인 경우에는 @Autowired를 생략해도 자동으로 Bean 등록된다.

조회하기 findById

@Override  
public Optional<Member> findById(Long id) {  
 List<Member> result = jdbcTemplate.query("select \* from member where id = ?", memberRowMapper(),id);  
 return result.stream().findAny();  
}

jdbcTemplate를 통해 쿼리문을 전송한다.

쿼리문의 반환값이 데이터 형식으로 나오기 때문에 RowMapper로 매핑해줘야 한다.

private RowMapper<Member> memberRowMapper(){  
 return new RowMapper<Member>() {  
 @Override  
 public Member mapRow(ResultSet rs, int rowNum) throws SQLException {  
 Member member = new Member();  
 member.setId(rs.getLong("id"));  
 member.setName(rs.getString("name"));  
 return member;  
 }  
 }  
}

찾으려는 회원 정보가 rs에 있으므로 회원변수를 하나 생성하여 rs로부터 값을 저장한뒤 반환한다.

다시 조회로 넘어와서 반환한 값은 member지만 RowMapper라는 리스트 형식으로 반환된다.

리스트에는 하나의 member만 있을 것이므로, 리스트에서 stream으로 반복조회하고 findany로 하나만 있을 값을 찾자마자 optional객체를 반환한다(findby id가 optional 형 이므로)

save기능

public Member save(Member member) {  
 SimpleJdbcInsert jdbcInsert = new SimpleJdbcInsert(jdbcTemplate);//쿼리문을 대신 만드는 무언가  
 jdbcInsert.withTableName("member").usingGeneratedKeyColumns("id");//테이블 명과 key속성을 알려준다  
 Map<String, Object> parameters = new HashMap<>();//name키와 전달받은 회원 이름을 넣을 객체 hashmap을 만든다  
 parameters.put("name", member.getName());  
 Number key = jdbcInsert.executeAndReturnKey(new MapSqlParameterSource(parameters)); //뭔지 몰라도 insert쿼리를 대신 날린다? 그리고 생성된 키를 반환  
 member.setId(key.longValue());//가입된 해당 회원 id넣어주고 해당 회원 반환..  
 return member;  
}

이 부분은 설명을 안해줬다…

@Override  
public Optional<Member> findByName(String name) {  
 List<Member> result = jdbcTemplate.query("select \* from member where name = ?", memberRowMapper(),name);  
 return result.stream().findAny();  
}  
  
@Override  
public List<Member> findAll() {  
 return jdbcTemplate.query("select \* from member", memberRowMapper());  
}

이름으로 찾기와 전체 목록은 쉽게 만들 수 있다.

저장소 변경

Spring confing에서 저장소코드 한줄만 수정하면 완성.

이후 기존의 통합 테스트를 실행하면 잘 돌아가는 것을 알 수 있다.

JPA

db연결 방식을 JPA를 이용한 방식으로 바꾼다.

Build.gradle에

//implementation 'org.springframework.boot:spring-boot-starter-jdbc'  
implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

로 수정한다.

이후 application properties에 코드를 추가한다

spring.jpa.show-sql=true  
spring.jpa.hibernate.ddl-auto=none

f

위 코드는 jpa가 sql을 볼 수 있게 하는 것이고,

두번재는 jpa가 객체를 보고 테이블을 자동으로 만드는 기능인데, 예제이서는 이미 만든 테이블을 사용할 것이기 때문에 꺼놓는다.

이후 domain/member에 엔티티를 매핑한다.

Jpa는 인터페이스를 제공하고, 구현 bander들이 제공된다.

Jpa 는 객체와 ORM(Object Relational Mapping)기술

@Entity

라고 쓰면 JPA가 관리하는 Entity임을 명시하는 것이다.

@Id @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
private Long id;//회원이 저장하는 아이디가 아닌 시스템에서 저장하는 아이디  
  
//@Column(name="username")  
private String name;

여기서 @Id는 primary key를 알려주는 것이고 @GenratedValue의 내용은 우리가 테이블을 설정할 때 자동으로 생성되는 것을 설정한 것을 알려준 것이다.

name부분의 @Colum은 만약 db에서의 속성이름이 username일 경우 저렇게 매핑을 해주는 것이다.

jpa저장소 생성

JpaMemberRepository 를 만들고 상속까지 완료한뒤 코드를 삽입한다.

private final EntityManager em;  
  
public JpaMemberRepository(EntityManager em) {  
 this.em = em;  
}

처음에

implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

이 라이브러리를 설치하면

스프링부트가 자동으로 entity manager를 생성해서 db와 연결해준다.

우리는 그 것을 생성자로 injection받기만 하면된다.

Jpa의 모든 행동이 entitymanager를 통해 이루어진다.

private final EntityManager em;  
  
public JpaMemberRepository(EntityManager em) {  
 this.em = em;  
}  
  
@Override  
public Member save(Member member) {  
 em.persist(member);//member를 insert 단 반환값 없음  
 return member;  
}  
  
@Override  
public Optional<Member> findById(Long id) {  
 Member member = em.find(Member.class,id);//domain 맴버 클래스에서 해당 아이디를 이용하여 찾는다.  
 return Optional.*ofNullable*(member);//optional로 감싸서 반환  
}  
  
@Override  
public Optional<Member> findByName(String name) {  
 List<Member> result = em.createQuery("select m from member m where m.name = :name", Member.class)  
 .setParameter("name", name).getResultList();  
 //특정값을 찾는 것은 jpql쿼리문을 작성해야 한다.  
 //jpql은 테이블이 아닌 엔티티 객체를 대상으로 한다.  
 //member 를 m 으로 alias하고, m에서 m의 이름이 변수 name인 것을 찾는다.  
 //다음 인자로 엔티티 클래스를 준다.  
 // 그 후 변수 name의 값을 setParameter로 실제 스트링 name과 연결하고, List 형식으로 반환한다.  
 return result.stream().findAny();  
}  
  
@Override  
public List<Member> findAll() {  
 return em.createQuery("select m from member m", Member.class)  
 .getResultList();  
}

주석 참조

정확히는 PK(Primary Key)를 기반으로 하는 값이 아닌 경우에는 쿼리문 작성이 필요하다.

Jpa는 모든 데이터 입력이 transaction안에서 이루어져야 한다.

트랜잭션이란 질의어sql을 통해 db에 접근하는 것을 의미한다.

Member Service 상단에 @Transaction을 추가한다.

test일때와는 다르게 동작한다.

이후 springconfig를 수정한다.

@Configuration  
public class SpringConfig {  
 */\*\*private DataSource dataSource;* ***@Autowired*** *public SpringConfig(DataSource dataSource) {  
 this.dataSource = dataSource;  
 }\*/* private final EntityManager em;  
 @Autowired  
 public SpringConfig(EntityManager em) {  
 this.em = em;  
 }  
  
 @Bean  
 public MemberService memberService(){  
 return new MemberService(memberRepository());  
 }  
 @Bean  
 public MemberRepository memberRepository(){  
  
 //return new MemoryMemberRepository();  
 //return new JDBCMemberRepository(dataSource);  
 //return new JdbcTemplateMemeberRepository(dataSource);  
 return new JpaMemberRepository(em);  
 }  
}

이후 통합 테스트를 실행해본다.

Spring data jpa

먼저 저장소를 만드는데, 인터페이스로만든다

public interface SpringDataJpaMemberRepository extends JpaRepository<Member,Long>, MemberRepository{  
 @Override  
 Optional<Member> findByName(String name);  
}

인터페이스에서 인터페이스를 불러오는 경우에는 implements가 아닌 extends

JpaRepository에 객체와, PK자료형, 그리고 기본 저장소 인터페이스를 가져온다.

public class SpringConfig {  
   
  
 private final MemberRepository memberRepository;  
 @Autowired  
 public SpringConfig(MemberRepository memberRepository) {  
 this.memberRepository = memberRepository;  
 }  
  
 @Bean  
 public MemberService memberService(){  
 return new MemberService(memberRepository);  
 }  
 */\*\*****@Bean*** *public MemberRepository memberRepository(){  
  
 //return new MemoryMemberRepository();  
 //return new JDBCMemberRepository(dataSource);  
 //return new JdbcTemplateMemeberRepository(dataSource);  
 return new JpaMemberRepository(em);  
 }\*/*}

그 후 springconfig 를 다음과 같이 수정하면 끝난다.

원리는

저장소가 호출될 때 등록되어있는 저장소가 없지만 인터페이스로 연결된 data jpa가 자동으로 객체를 생성하여 bean에 등록된다. 따라서 자동으로 주입된 memberrepository를 생성자에서 받아서 memberservice로 넣는 것 만으로 끝난다.

이후 기능들은 사실 키를 이용하거나 CRUD 등의 공통 기능들은 JpaReopsitory에 구현되어있기 때문에 메소드를 호출하기만 하면 수행이 된다.(예제에서 반환형 등을 이상하게 쓴 이유도 이 형식에 맞춰두었기 때문)

findByName 등의 별도의 코드만 따로 작성하면 되는데 이 조차도 모두 구현할 필요없이, 작성시 규칙만 따른다면 나머지는 자동으로 생성되어 동작한다.

예시

@Override

Optional<Member> findeByNameAndId(String name,Long id);

이후 통합테스트가 동작하는 것을 확인할 수 있다.

모든 메소드의 호출 시간을 알고싶어진 경우에 각 메소드를 수정하여 알 수 있다.



다음과 같이 모든 메소드를 일일이 수정하고 관리하는 것은 비효율적이고 어려운 일이다

시간을 측정하는 기능은 핵심 기능이 아니고, 공통의 관심사항이다.

시간을 측정하는 로직과 비즈니스 로직이 뒤섞여 유지보수가 어렵다.

또, 시간을 측정하는 로직을 별도의 공통 로직으로 만들기 매우 어렵다.

시간을 측정하는 로직을 변경할 때 모든 메소드마다 로직을 변경해야 한다.

이 때 필요한 것이 aop이다.

Aop를 이용하여 별도의 시간측정 로직을 만들어 원하는 곳에 적용하기만 하면 된다.

aop패키지를 만들고, TimeTraceAop클래스를 작성한다.

Aop로 사용하기 위해서는

@Aspect

를 붙여야 한다.

스프링 빈에 등록하기 위해 @Component 하거나 springconfig에

@Bean  
public TimeTraceAop TimeTraceAop(){  
 return new TimeTraceAop();  
}

해도 되지만 예제에서는 @Component로 해당 클래스에서 사용하기로 한다.

@Component  
@Aspect  
public class TimeTraceAop {  
 @Around("execution(\* hello.hellospring..\*(..))")  
 public Object execute(ProceedingJoinPoint joinPoint) throws Throwable {  
 long start = System.*currentTimeMillis*();  
 System.*out*.println("START: " + joinPoint.toString());  
 try {  
 return joinPoint.proceed();  
 } finally {  
 long finish = System.*currentTimeMillis*();  
 long timeMs = finish - start;  
 System.*out*.println("END: " + joinPoint.toString()+ " " + timeMs +  
 "ms");  
 }  
 }  
}

@Around는 적용 범위로 hellospring 이하의 모든 메소드에 적용된다는 뜻이다.

Jointpoint는 끼어드는 지점이고, .proceed 하면 시작되는 것이라고 볼 수 있다.

나머지는 시간 측정에 관한 내용이고, .tostring은 어떤 메소드에서 수행중인지를 볼 수 있게 해준다.

AOP는 실제 메소드를 호출할 때 가짜 서비스를 만들어서 가짜가 끝나면 진짜를 호출한다.