

05. DI와 테스트, 디자인 패턴

테스트 코드를 이용한 테스트

보낸 요청에 대해 header나 body가 잘왔는지 등 정상적으로 작동하는지 테스트해보기 위함 → spring에서는 TestRestTemplate을 사용함

테스트 코드를 통해 기능을 테스트해보고 문제가 없는지 검증

테스트 코드를 이용한 테스트 - 개발자 테스트

HelloApiTest.java

```
package tobyspring.helloboot;

import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

import static org.assertj.core.api.Assertions.*; // Assertions.assertThat() -> assertThat()

// JUnit5
public class HelloApiTest {
    @Test
    void helloApi(){
        // <HTTPPie> http localhost:8080/hello?name=Spring
        TestRestTemplate rest = new TestRestTemplate(); //API      ->      (, RestTemplate      throw )
        ResponseEntity<String> res = rest.getForEntity( // ResponseEntity:      , <String> String body
            "http://localhost:8080/hello?name={name}",
            String.class, // response body binding (      )
            "Spring" // {name}
        );
        //
        // 1. status == 200 ?
        assertThat(res.getStatusCode()).isEqualTo(HttpStatus.OK);
        // 2. header (content-type) == text/plain ?
        assertThat(res.getHeaders().getFirst(HttpHeaders.CONTENT_TYPE)).startsWith(MediaType.TEXT_PLAIN_VALUE);
        // .isEqualTo(MediaType.TEXT_PLAIN_VALUE); // expected: "text/plain" but was: "text/plain; charset=ISO-8859-1"
        // 3. body == Hello Spring ?
        assertThat(res.getBody()).isEqualTo("Hello Spring");
    }
}
```

DI와 단위 테스트

HelloServiceTest.java

```
package tobyspring.helloboot;

import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;

public class HelloServiceTest {
    @Test
    void simpleHelloService(){ // service -> http      ->
        // http request      body java object
        // and
        //
        SimpleHelloService helloService = new SimpleHelloService();
        String ret = helloService.sayHello("Test");

        // live template
        Assertions.assertThat(ret).isEqualTo("Hello Test");
    }
}
```

HelloController.java

```
package tobyspring.helloboot;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

import java.util.Objects;

@RestController
public class HelloController {
    private final HelloService helloService;
    public HelloController(HelloService helloService) { // 2. ( )
        this.helloService = helloService;
    }

    @GetMapping("/hello")
    @ResponseBody
    public String hello(String name){
        // name null,
        if(name == null || name.trim().length() == 0) throw new IllegalArgumentException();

        return helloService.sayHello(name);
    }

    /**
     */
    // @RequestMapping
```

HelloControllerTest

```
package tobyspring.helloboot;

import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;

public class HelloControllerTest {
    @Test
    void helloController(){
        // hello controller  hello service  ? ->
        // -> DI
        HelloController helloController = new HelloController((HelloService) name -> name); //      -> ( )

        String ret = helloController.hello("Test");
        Assertions.assertThat(ret).isEqualTo("Test");
    }
    @Test
    void failHelloController(){ // null exception
        HelloController helloController = new HelloController((HelloService) name->name); // test stub      DI
        ( DI)

        Assertions.assertThatThrownBy(()->{ //      exception
            helloController.hello(null);
        }).isInstanceOf(IllegalArgumentException.class);

        //
        Assertions.assertThatThrownBy(()->{ //      exception
            helloController.hello("");
        }).isInstanceOf(IllegalArgumentException.class);
    }
}
```

HelloApiTest

```
package tobyspring.helloboot;

import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

import static org.assertj.core.api.Assertions.*; // Assertions.assertThat() -> assertThat()

// JUnit5
public class HelloApiTest {
    @Test
    void helloApi(){ // API
        // <HTTPPie> http localhost:8080/hello?name=Spring
        TestRestTemplate rest = new TestRestTemplate(); //API -> (, RestTemplate throw )

        ResponseEntity<String> res = rest.getForEntity( // ResponseEntity: , <String> String body
            "http://localhost:8080/hello?name={name}",
            String.class, // response body binding ( )
            "Spring" // {name}
        );

        //
        // 1. status == 200 ?
        assertThat(res.getStatusCode()).isEqualTo(HttpStatus.OK);
        // 2. header (content-type) == text/plain ?
        assertThat(res.getHeaders().getFirst(HttpHeaders.CONTENT_TYPE)).startsWith(MediaType.
TEXT_PLAIN_VALUE); // .isEqualTo(MediaType.TEXT_PLAIN_VALUE); // expected: "text/plain" but was: "text/plain;
charset=ISO-8859-1"
        // 3. body == Hello Spring ?
        assertThat(res.getBody()).isEqualTo("Hello Spring");
    }

    @Test
    void failsHelloApi(){ // API
        // <HTTPPie> http localhost:8080/hello?name=Spring
        TestRestTemplate rest = new TestRestTemplate(); //API -> (, RestTemplate throw )

        ResponseEntity<String> res = rest.getForEntity( // ResponseEntity: , <String> String body
            "http://localhost:8080/hello?name=",
            String.class // response body binding ( )
        );

        //
        // 1. status == 500 ?
        assertThat(res.getStatusCode()).isEqualTo(HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

DI를 이용한 Decorator, Proxy 패턴

DI 적용 이전

HelloController가 SimpleHelloService에 직접적으로 의존하는 상황

HelloController안에 SimpleHelloService를 선언해서 쓰는 상황을 예시로 들 수 있음

위에서 아래로 의존하는 느낌

DI 적용 이후

HelloController는 추상화된 HelloService만 의존하고 밑에 class들이 그 인터페이스를 구현하도록 만들어서 의존관계가 아래서 위로 올라가는 형태

- 장점
 - Hello Controller의 코드를 수정하지 않고도 Hello Service의 인터페이스를 구현한 클래스를 교체하면서 다양하게 적용하는게 가능함

이런 작업을 수행할때 필요한 것이 DI

DI가 뭐나?

코드 상으로는 Hello controller가 HelloService의 구현 클래스에 의존하고 있지 않지만 runtime에는 SimpleHelloService라는 클래스의 오브젝트를 사용하고 있어야하니 의존하는 것이 되는 것

즉 처음 스프링 컨테이너가 뜰 때 스프링 컨테이너가 DI의 어셈블러 역할을 해서 두 오브젝트를 엮어주는 역할을 함(DI)

Decorator??

HelloDecorator는 HelloService 인터페이스를 구현한 클래스

동시에 helloService 타입의 어떠한 오브젝트를 의존하고 있다. 마치 helloController 처럼.

그 말은 Hello decorator가 HelloService를 구현한 또 다른 오브젝트를 호출할 수 있다는 것

뭐하러 그렇게 만듬??

DI를 이용해서 런타임시에

요런 구조를 만들 수 있다.

SimpleHelloService는 메인이 되는 중요 로직을 가지고 있는 HelloService의 기능을 제공하는 오브젝트인데, 이 **오브젝트를 건드리지 않은 채로 여기에 어떠한 책임이나 기능을 추가하고 싶을때 Decorator를 중간에 삽입할 수 있다.**

즉 기존 오브젝트 코드를 수정하지 않고 DI구조와 decorator 패턴을 사용해서 **동적으로 새로운 기능을 추가할 수 있다.**

Hello Controller가 HelloDecorator를 의존하고 HelloDecorator가 SimpleHelloService를 의존하도록 오브젝트 레벨에서 어셈블러가 조합해주도록 컨테이너에게 구성정보를 우리가 제공하면 됨

Decorator 특

- 여러 개를 적용할 수 있다.

근데 우리가 HelloController에 SimpleHelloService를 지정해준게 아니라 HelloService만 지정했는데 어떻게 자동으로 Simple이가 들어갔지?

현재 container에 등록되어있는 빈 오브젝트들 중에서 HelloService 인터페이스를 구현한 클래스를 찾아보고, 만약 단 하나의 오브젝트만 그걸 구현하고 있다고 하면 그걸 의존대상이라고 생각하고 주입해줌 (단일 주입 후보)

자동으로 가져다가 연결한다고해서 autowiring이라고 함

그래서 옛날에는 생성자를 통해 주입하거나 프로퍼티의 setter 메소드를 이용해서 주입하든, 주입시킬 빈 오브젝트의 타입 정보를 가지고 후보를 찾아와서 자동으로 연결을 해달라고 @Autowired를 붙혔음

지금은 @Autowired를 생략해도 클래스에 생성자가 하나라면 클래스의 생성자를 보고 그 생성자의 파라미터 타입과 일치하는 빈을 찾아서 주입해주는 autowiring 해주는 룰을 만들었음

Decorator 어떻게 쓸까?

Hello Controller가 HelloDecorator를 의존하고 HelloDecorator가 SimpleHelloService를 의존하도록 구조를 구성해보자

HelloDecorator

```
package tobyspring.helloboot;

import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Service;

@Service
@Primary // @Primary      ->
public class HelloDecorator implements HelloService {
    private final HelloService helloService;

    public HelloDecorator(HelloService helloService) { //  HelloService
        //  HelloService      ?
        //
        this.helloService = helloService;
    }

    @Override
    public String sayHello(String name) {
        return "*" + helloService.sayHello(name) + "*"; //  HelloService  +
    }
}
```

HelloController.java

```
package tobyspring.helloboot;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

import java.util.Objects;

@RestController
public class HelloController {
    private final HelloService helloService;
    public HelloController(HelloService helloService) {
        // HelloService ?
        //
        //
        // @Primary
        this.helloService = helloService;
    }

    @GetMapping("/hello")
    @ResponseBody
    public String hello(String name){
        if(name == null || name.trim().length() == 0) throw new IllegalArgumentException();

        return helloService.sayHello(name);
    }
}

//@RequestMapping
```

HelloServiceTest.java

```
package tobyspring.helloboot;

import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;

public class HelloServiceTest {
    @Test
    void simpleHelloService(){
        SimpleHelloService helloService = new SimpleHelloService();
        String ret = helloService.sayHello("Test");

        Assertions.assertThat(ret).isEqualTo("Hello Test");
    }

    @Test
    void helloDecorator(){ //Decorator test
        HelloDecorator helloDecorator = new HelloDecorator(name -> name);
        String ret = helloDecorator.sayHello("Test");
        Assertions.assertThat(ret).isEqualTo("*Tets*");
    }
}
```

Proxy pattern

어떤 실체가 존재하는데 실체 대신 그 앞에 원기를 대신할 수 있는 것을 가져다 놓는다는 느낌

예를 들어, 프록시는

HelloController가 보기에는 HelloService를 구현한 오브젝트인 것처럼 생긴 부가적인 효과를 제공하는 대리자인 거임

- 비용이 많이 드는 오브젝트를 생성하는 것을 onDemand일때만 한다든지 최대한 지연시키는 Lazy Loading에서 사용
- local에 hello service를 구현한 오브젝트가 있다고 생각했지만 사실 API를 타고 네트워크 너머 서버에 존재하는 오브젝트를 사용해야하는 경우가 있을 수 있음
 - 이때 helloController는 hello service라는 인터페이스를 통해 '아 나는 그냥 로컬에 있는 오브젝트를 호출한다'라는 느낌으로 사용하면 프록시가 API를 호출해서 remote에 있는 오브젝트를 수행시켜 결과를 받은 후 return 해주는 기능을 담당 가능
- 보안 기능 적용
- access control 기능

하지만 프록시가 중간에서 대리자 역할을 하기 때문에 controller는 HelloService의 내부 사정은 전혀 신경쓰지 않아도 된다는 것

결론

DI 가 가능한 구조로 코드를 설계해두면 굉장히 다양한 유형의 유연함을 코드에 제공해줄수 있고

Spring 은 DI 의 container로써 이런 작업을 수행하는데에 필요로하는 굉장히 편리한 기능을 많이 제공한다.