7/13 ~ 7/17 2주차 - 9

테스트 코드에 대해서 (2)

- 스프링은 다양한 테스트 방법을 제공한다.
- 대표적으로 Slice Test 라는 것으로 특정 레이어에 대해서 Bean을 최소한으로 등록시켜 테스트 하고자 하는 부분에 최대한 단위 테스트를 지원한다.
- 다양하게 지원해주는만큼 테스트 코드를 통일성 있게 관리 하는 것이 중요하다.

더 안전하고 통일성 있게 테스트를 진행하는 법알아보자.

테스트 전략

어노테이션	설명	부모 클래스	Bean
@SpringBootTest	통합 테스트, 전체	IntegrationTest	Bean 전체
@WebMvcTest	단위 테스트, Mvc 테스트	MockApiTest	MVC 관련된 Bean
@DataJpaTest	단위 테스트, Jpa 테스트	RepositoryTest	JPA 관련 Bean
None	단위 테스트, Service 테스트	MockTest	None
None	POJO, 도메인 테스트	None	None

통합 테스트

장점

- 모든 Bean을 올리고 테스트를 진행하기 때문에 쉽게 테스트 진행가능
- 모든 Bean을 올리고 테스트를 진행하기 때문에 운영환경과 가장 유사하게 테스트 가능

- API를 테스트할 경우 요청부터 응답까지 전체적인 테스트 진행가능

단점

- 모든 Bean을 올리고 테스트를 진행하기 때문에 테스트 시간이 오래 걸림
- 테스트의 단위가 크기 때문에 테스트 실패시 디버깅이 어려움
- 외부 API 콜 같은 Rollback 처리가 안되는 테스트 진행을 하기 어려움

Junit 4 & Spring Test을 이용한 TDD 환경 세팅

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = { RootContextConfig.class },
        loader = AnnotationConfigWebContextLoader.class)
@WebAppConfiguration
public class SpringTestSupport {

    @Autowired
    protected WebApplicationContext wac;

    protected MockMvc mockMvc;

    @Before
    public void setup() throws Exception {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build();
    }

    @Test
    public void isWebAppContextLoadingProperly() {
        ServletContext servletContext = wac.getServletContext();

        Assert.assertNotNull(servletContext);
        Assert.assertTrue(servletContext instanceof MockServletContext);
        Assert.assertNotNull(wac.getBean("rootContextConfig"));
    }

}
```

@RunWith(SpringJUnit4ClassRunner.class)

→이 애너테이션을 붙여 줘아 스프링 테스트를 JUnit으로 돌릴 수 있다.

@ContextConfiguration(classes = { RootContextConfig.class }, loader= AnnotationConfigWebContextLoader.class)

→ RootContextConfig.class를 spring context의 빈 설정 파일로 사용한다는 의미.

@WebAppConfiguration

→ 이 애너테이션을 붙이면 Controller 및 web환경에 사용되는 빈들을 자동으로 생성하여 등록하게 됨

Controller 테스트 예제

```
public class UserControllerTest extends SpringTestSupport {
```

```
    @Test
    public void issueTokenApiTest() throws Exception {
        MvcResult mvcResult = this.mockMvc.perform(MockMvcRequestBuilders.get("/v1/issue-token"))
                .andDo(MockMvcResultHandlers.print())
                .andExpect(MockMvcResultMatchers.status().is(HttpStatus.OK.value()))
                .andExpect(MockMvcResultMatchers.content().contentType(MediaType.APPLICATION_JSON_UTF8_VALUE))
                .andReturn();

        Assert.assertEquals(MediaType.APPLICATION_JSON_UTF8_VALUE,
                mvcResult.getResponse()
                        .getContentType());

        Assert.assertEquals(HttpStatus.OK.value(),
                mvcResult.getResponse()
                        .getStatus());
    }

}
```

spring integration test 참고 : https://www.baeldung.com/integration-testing-in-spring

Bean Mocking Test

- Controller를 테스트 할 경우 Service layer까지 내려가게 됨. Controller만 테스트하고 싶다면 Service를 Mocking 하면 된다.
- mocktio framework을 이용하여 적용해보자.

```
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
```

```
public class UserControllerTest extends SpringTestSupport {

    @Mock
    private UserService userService;

    @InjectMocks
    private UserController userController;

    @Test
    public void issueTokenApiIntegrateTest() throws Exception {
        this.mockMvc.perform(get("/v1/issue-token"))
                .andDo(print())
                .andExpect(status().is(HttpStatus.OK.value()))
                .andExpect(content().contentType(MediaType.APPLICATION_JSON_UTF8_VALUE))
                .andExpect(jsonPath("$.result.token").exists())
                .andExpect(jsonPath("$.result.created_at").exists());
    }

    @Test
    public void issueTokenApiUnitTest() {
        MockitoAnnotations.initMocks(this);
        LocalDateTime testTime = LocalDateTime.now();
        givenUserService(testTime);
```

```
        ApiResponseModel<AccessToken> result = userController.issueToken();

        Assert.assertEquals(result.getCode(), HttpStatus.OK.value());
        Assert.assertEquals(result.getMsg(), HttpStatus.OK.getReasonPhrase());
        Assert.assertEquals(result.getResult().getToken(), "this is sample access-token");
        Assert.assertEquals(result.getResult().getCreatedAt(), testTime);
    }

    private void givenUserService(LocalDateTime createdAt) {
        User mockUser = new User();
        mockUser.setAccessToken(new AccessToken("this is sample access-token", createdAt));
        when(userService.registerUser()).thenReturn(mockUser);
    }

}
```

- Mocking전의 테스트 코드와 비교하여 달라진 점은 @Mock, @InjectMocks와 givenUserService 메서드가 추가 되었고
- MockitoAnnotations.initMocks(this); 줄이 포함되었다.
  1. @Mock → Mocking할 빈을 설정 한다.
  2. @InjectMocks → UserController빈을 Context에 등록할 때 연관관계를 가진 빈 대신 Mocking된 빈을 집어 넣는다.
  3. givenUserService에서 when 메서드에 userService의 registerUser()메서드가 호출되면 mockUser를 return하게 만든다.

     given, when, then 패턴을 알아두자.
  4. MockitoAnnotations.initMocks(this); → 현재 테스트 클래스의 @Mock이 달린 객체를 초기화하는 작업이다.
  5. 테스트 클래스는 @Test가 붙은 메서드마다 객체가 새로 만들어지게 되며 issueTokenApiUnitTest메서드가 호출될 때 새로 생성된 테스트 클래스의 객체를 파라미터로 넣어준다.