**테스트와 로깅**

**로그시스템 – log4j2라이브러리 이용. Build.gradle로 설치. Log4j2.xml로 설정필요.**

Log4j2라이브러리의 log 객체를 사용하기 위해 @Log4j2 클래스레벨 어노테이션 적용.

Conosle에 출력하는 것처럼 log의 메서드를 이용해서 명시적으로 log시스템에 출력.   
log.info(…); log.warn(…); …

암시적으로 @Log4j2이 적용된 클래스에서 일어난 일들을 log4j2.xml에서 설정한 규칙, 형식대로 출력됨.

Src>main>resources>log4j2.xml에서 어떤 범위에서 일어난 일을 어떤 레벨부터 어디에다가 출력할 것인지 등 다양한 로그 출력 옵션을 제공한다.  
대략 log4j2.xml 설정 형식은   
<Loggers>  
 <Root level=”출력레벨지정”>//출력할 장소에 출력레벨 이상의 일만 출력한다.  
 <AppenderRef ref=”출력할장소지정” />  
 </Root>//Root는 모든 범위를 의미.   
  
 <Logger name=”감지하는패키지범위지정” level=”출력레벨지정” additivity=”중복출력여부지정”>  
 <AppenderRef ref=”출력할장소” />  
 </Logger>  
</Loggers>

**JDBC 로그시스템 – log4jdbc-log4j2라이브러리 사용. Build.gradle로 설치.  
 Resources>log4jdbc.log4j2.properties로 설정. Application.properties 알맞게 수정. 만하면 건들 일 없음.**

jdbc처리 관련해서 스프링 내부에서 사용하는 완성된 sql문을 확인해봐야할 때가 있다. 기존 log4j2 로그시스템으로는 보지는 못하고 로그를 가로채서 편집해 출력하는 **log4jdbc-log4j2라이브러리**가 필요하다.

**테스트 – 스프링용**

스프링이 적용된 것을 테스트하기 위해서는 스프링이 제공한 단위테스트를 연동해야함.  
@ExtendWith(SpringExtension.class) 클래스레벨 어노테이션을 테스트 클래스에 적용.

스프링이 적용된 것을 테스트할 때 어떤 컨텍스트 configruation이 해당 테스트 케이스에 적용되야하는지 명시해야한다.  
@ContextConfiguration(classes={ 스프링컨텍스트config클래스명.class들 나열 }

원래 테스트 원칙은 출력이나 로그 등을 사람의 눈으로 확인하는 과정이 있으면 안됨. 테스트는 자동화 되는 것이 원칙. 테스트의 통과/실패 여부만을 확인해야하는 것이 원칙. 그래도 실패시 로그를 봐야할 수 있으니 출력과 로그를 봐야하는 상황이 온다.  
테스트할때도 로그을 봐야할 수도 있으므로 @Log4j2 클래스레벨 어노테이션 테스트 클래스에 적용.  
테스트용 log4j2 설정을 따로 하고 싶다면 test>resources>log4j2.xml를 따로 설정하면 해당 설정이 테스트 할 때 적용됨. 없으면 실제 운영을 위한 설정이 테스트에 적용됨.

테스트 클래스에서 필요한 DI는 필드 멤버로 선언하고 @Autowired로 명시하는 것이 권장됨. 운영시에는 @Autowired는 권장되지 않음.   
혹은 필요한 DI를 final 멤버로 선언하고 final 멤버를 채우는 생성자를 생성하는 @RequiredArgsConstructor 클래스 어노테이션을 테스트 클래스에 적용하는 것도 권장됨.

예시)  
@ExtendWith(SpringExtension.class)  
@ContextConfiguration(classes={RootConfig.class, ServletConfig.class })  
@Log4j2  
class TestMapper{…}

**테스트 – 스프리용. 웹 요청이 필요한 테스트.**

웹 요청을 테스트할 때는 어떻게 할까? 실제로 브라우저에 일일이 url을 입력해가며 테스트할 수도 있지만 별로.  
스프링에서 제공하는 ‘실제 mvc 웹 요청을 발생시키는 것을 모방하는 가짜mvc’인 MockMvc객체를 이용하면 된다.  
MockMvc객체는 웹 서버에 요청을 보낸 것과 같은 효과를 내는 메서드를 제공한다.  
해당 메서드 호출로 웹 요청(get/post/update/delete 등)을 수행한다.

테스트 마다 새로운 MockMvc객체가 필요하므로. @BeforeEach가 적용된 메서드에서 매번 생성되게끔 하자.

MockMvc를 사용하기 위해선 @WebAppConfiguration 클래스레벨 어노테이션을 테스트 클래스에 적용해야 한다.

예시와 MockMvc 사용법)  
@ExtendWith(SpringExtension.class)  
@WebAppConfigurtation//웹앱컨텍스트 사용을 위한. MockMvc생성에 필요해서.  
@ContextConfiguration( classes={  
 RootConfig.class, ServletConfig.class  
})  
@Log4j2  
public class ControllerTest{  
 @Autowired  
 private WebApplicationContext ctx;//자동 등록된 웹앱컨텍스트 DI요청  
   
 private MockMvc mockMvc;  
  
 @BeforeEach  
 public void setup( ){  
 this.mockMvc=MockMvcBuilders.webAppContextSetup(ctx).build( );  
 //웹앱 컨텍스트가 적용되어 생성된 mockmvc객체.  
 }  
  
 @Test  
 public void TestMethod( ) throws Exception{  
 ResultActions ra=mockMvc.perform( MockMvcRequestBuilders.get(url형식) );  
 //먼저 요청url과 메서드로 MockMvcRequestBuilders로 요청을 생성  
 //mockMvc객체의 perform메서드로 해당 요청 실행.  
 //우리가 만든 컨트롤러가 매핑된 메서드를 실행하여 결과 도출.  
  
 //ResultActions 객체를 통해서 결과를 반환 받거나 추가작업, 다음 핸들러조 전달 등 다양한 연산 가능.  
  
 MvcResult mr=ra.andReturn( );//컨트롤러의 처리 결과를 반환 받음.  
 //컨트롤러에 의해 구성된 model(정보저장) 및 view(갈 페이지)에 대한 정보 및 처리결과(상태코드)등을 가짐.  
   
 //다양한 정보에서 모델과 뷰만을 가져오는 메소드  
 ModelAndView mav=mr.getModelAndView( );  
  
 //그 중 모델만을 가져오라는 메소드  
 Model model=mav.getModelMap( );// 해당 객체를 통해 컨트롤러가 model에 뭘 넣어서 줬는지 확인!  
  
 //get요청시 쿼리스트링이 있는 것과 post요청시 바디에 쿼리스트링이 있는 것은  
 //MockMvcRequestBuilders로 요청을 만들 때 .param(키,값)을 체이닝하면 된다.  
 //MockMvcRequestBuilders.post(url문자열).param(“title”,”new titledfdfd”).param(“content”,”내용”)  
 }  
}