1일차 : 9:10 ~ 17:51분 (8시간)

2일차 : 9:10 ~ 17:51분 (8시간)

휴식시간 50분 ~ 정각

점심시간 11시50분 ~ 13시

QR : 입실 / 확인(13시) / 퇴실 (17:51분 이후)

환경설정

1. sts3.9-bundle.zip 파일을 c:\ 아래에 압축을 풀어주세요

: 디렉토리명 변경 가능합니다. (디렉토리명에 공백이나 한글이 포함되지 않도록 해주세요)

: 변경 후에 C:\sts3.9-bundle\sts-3.9.12.RELEASE\sts.ini 파일 내의 java경로를 수정해주세요

2. MariaDB 설치

root password

: maria / maria

: charset utf-8 반드시 체크

sts3.9-bundle.zip 압축을 풀면

구글 드라이브에 있는 압축파일 3개를 다운로드 해주세요

springfwxml.zip

: Spring Configuration XML + Annotation 혼용

springfwconfig.zip ( No xml )

: Java Config + Annotation

myspringfw.zip - 수업중에 코드를 작성할 예정

: Maven Project Template

수업중 작성 프로젝트는 GitHub에 공유할 예정입니다.

<https://github.com/mysoyul/SpringFW5_App>

J2EE (Java 2 Enterprise Edition) => Jakarta EE

J2SE + J2EE 8

Eclipse 재단에 기부

팩키지 명이 변경됨 javax.\* => jakarta.\*

spring framework 6 버전으로 버전업 하신다면 jakarta.\* 로 모두 변경해야 합니다.

infrastructural support / "plumbing"

: 인증, 로깅, Tx 처리, 확장에 대비한 디자인 패턴을 적용

* [Core technologies](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html): dependency injection, events, resources, i18n, validation, data binding, type conversion, SpEL, AOP.
* [Testing](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/testing.html): mock objects, TestContext framework, Spring MVC Test, WebTestClient.
* [Data Access](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/data-access.html): transactions, DAO support, JDBC, ORM, Marshalling XML.
* [Spring MVC](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html) and [Spring WebFlux](https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html) web frameworks.

spring mvc 와 spring webflux의 차이점

- spring mvc : Synchronous & Blocking, Tomcat 엔진

- spring webflux : Asynchronous & Non-Blocking, Netty 엔진 ( Event Loop기능 제공)

Nodejs - Asynchronous & Non-Blocking, Event-Loop

POJO (Plain Old Java Object) : Regular Java Object

<https://martinfowler.com/bliki/POJO.html>

EJB(Enterprise Java Beans)

프레임워크와 라이브러리의 차이점은 ??

호출흐름의 제어권을 누가 가지고 있느냐?

: 라이브러리는 클래스의 제어권을 개발자가 가진다.

: 프레임워크는 개발자가 작성한 클래스에 대한 제어권을 프레임워크가 가져 갔음 ( 역 호출 )

왜 프레임워크가 제어권을 가지고 갔을까요?

UserDao userDao = new UserDaoImpl(); (X)

userDao.insertUser(sql);

개발자가 작성한 클래스(예를 들어 Dao, Service, Controller)에 대한 제어권이 없다.

//main(String[] args)

BeanFactory와 ApplicationContext는 Bean Container 역할을 하는 인터페이스입니다.

BeanFactory factory = new GenericXmlApplicationContext(“classpath:spring-beans.xml “);

ApplicationContext context = new GenericXmlApplicationContext(“classpath:spring-beans.xml “);

UserDao userDao = (UserDao) context.getBean(“userDao”);

Spring In Action Pdf 링크 (6th Edition)

<https://dl.ebooksworld.ir/books/Spring.in.Action.6th.Edition.Craig.Walls.Manning.9781617297571.EBooksWorld.ir.pdf>

Spring In Action Source 링크 (6th Edition)

<https://github.com/habuma/spring-in-action-6-samples>

Spring 프레임워크는 기반이 되는 기술 EJB(Enterprise Java Beans)

: POJO(Plain Old Java Object) 객체 중심

Spring Framework Api doc

<https://docs.spring.io/spring-framework/docs/current/javadoc-api/>

https://docs.spring.io/spring-framework/docs/5.3.28/javadoc-api/

Spring FW 5.2.24 버전

<https://docs.spring.io/spring-framework/docs/5.2.24.RELEASE/javadoc-api/>

HikariCP Api doc

<https://javadoc.io/doc/com.zaxxer/HikariCP/4.0.2/index.html>

MyBatis Api doc

<https://mybatis.org/mybatis-3/apidocs/index.html>

MyBatis-Spring Api doc

<https://mybatis.org/spring/apidocs/index.html>

AspectJ Runtime Api doc

<https://javadoc.io/doc/org.aspectj/aspectjrt/latest/index.html>

Spring Bean Definition 메타정보 설정 방법

1. 전략1 - 모든 Configuration(설정)을 XML로 설정 (ver 2.5 이전)
2. 전략2 - XML 과 Annotation을 혼용해서 설정 (ver 2.5 이후) <component-scan />
3. 전략3 - JavaConfig 와 Annotation을 사용해서 설정(ver 3.0 이후) @Configuration

@Component 와 @Bean 어노테이션의 차이점

* Spring Bean을 나타내는 어노테이션
* @Component 는 클래스 위에 선언 하고, @Bean은 메소드 위에 선언한다.
* 외부라이브러리에서 제공하는 클래스를 SpringBean으로 설정하는 경우에는 @Bean 어노테이션을 사용한다.

@Bean

public Docket api() {

return new Docket(DocumentationType.SWAGGER\_2)

.select()

.apis(RequestHandlerSelectors.any())

.paths(PathSelectors.any())

.build();

}

IoC - DL (Dependency Lookup) : JDNI(Java Directory Naming Interface)

DI (Dependency Injection)

<bean>

<property name=”name” value=”spring” />

</bean>

<bean p:name=”spring” />

------------------------------------------------------------------------------------------

Eclipse ShortCut (단축키)

ctrl + space : code assist (자동완성)

ctrl + s : code save

ctrl + shift + f : code format

ctrl + shift + o(오우) : auto import

alt + shift + l(엘) : return type 자동생성

ctrl + / (슬래시) : java line comment toggle

ctrl + shift + c : xml, html comment

ctrl + shift + t : open type

f3 : 선택한 클래스나 메소드 로 이동

ctrl + f11 : java run, test case run

alt + shift + s

alt + shift + s + o : Generate Constructor using field

--------------------------------------------------------------------------------------------

해당 프로젝트의 오른쪽 메뉴

Maven -> Update Project -> Force Update 체크박스 선택

setMyFirstName(String name)

<property name=”myFirstName” value=”스프링” />

property는 setter method를 호출 하는 Tag

// Container 객체 생성하는 방법1

BeanFactory factory = new GenericXmlApplicationContext(“classpath:springBeans.xml”);

Hello hello = (Hello)factory.getBean(“hello”);

// Container 객체 생성하는 방법2

ApplicationContext context1 = new GenericXmlApplicationContext(“classpath:springBeans.xml”);

Hello hello2 = context1.getBean(“hello”, Hello.class);

<bean /> 태그의 scope 속성

Attribute : scope

The scope of this bean: typically "singleton" (one shared instance, which will be returned by all

calls to getBean with the given id), or "prototype" (independent instance resulting from each

call to getBean). By default, a bean will be a singleton, unless the bean has a parent bean

definition in which case it will inherit the parent's scope. Singletons are most commonly used,

and are ideal for multi-threaded service objects. Further scopes, such as "request" or

"session", might be supported by extended bean factories (e.g. in a web environment). Inner

bean definitions inherit the scope of their containing bean definition, unless explicitly specified:

The inner bean will be a singleton if the containing bean is a singleton, and a prototype if the

containing bean is a prototype, etc.

<context:component-scan base-package="myspring.di.annot">

Element : component-scan

Scans the classpath for annotated components that will be auto-registered as Spring beans.

By default, the Spring-provided @Component, @Repository, @Service, @Controller,

@RestController, @ControllerAdvice, and @Configuration stereotypes will be detected.

Note:

This tag implies the effects of the 'annotation-config' tag, activating @Required, @Autowired,

@PostConstruct, @PreDestroy, @Resource, @PersistenceContext and @PersistenceUnit

annotations in the component classes, which is usually desired for autodetected components

(without external configuration).

Turn off the 'annotation-config' attribute to deactivate this default behavior, for example in order to use custom BeanPostProcessor definitions for handling those annotations. Note: You may use placeholders in package paths, but only resolved against system properties (analogous to resource paths).

A component scan results in new bean definitions being registered; Spring's PropertySourcesPlaceholderConfigurer will apply to those bean definitions just like to regular bean definitions, but it won't apply to the component scan settings themselves. See javadoc for

org.springframework.context.annotation.ComponentScan for information on code-based

alternatives to bootstrapping component-scanning.

Content Model : (include-filter\*, exclude-filter\*)

<context:component-scan base-package="x.y.z">

<context:include-filter type="regex" expression="(service|dao)\..\*"/>

</context:component-scan>

<context:property-placeholder location="classpath:values.properties" />

Element : property-placeholder

Activates replacement of ${...} placeholders by registering a

PropertySourcesPlaceholderConfigurer within the application context. Properties

will be resolved against the specified properties file or Properties object -- so

called "local properties", if any, and against the Spring Environment's current

set of PropertySources. Note that as of Spring 3.1 the system-properties-mode

attribute has been removed in favor of the more flexible PropertySources

mechanism. However, applications may continue to use the 3.0 (and older)

versions of the spring-context schema in order to preserve system-properties-

mode behavior. In this case, the traditional PropertyPlaceholderConfigurer

component will be registered instead of the newer

PropertySourcesPlaceholderConfigurer. See ConfigurableEnvironment javadoc

for more information on usage.

<aop:aspectj-autoproxy /> @EnableAspectJAutoProxy

Element : aspectj-autoproxy

Enables the use of the @AspectJ style of Spring AOP.

See org.springframework.context.annotation.EnableAspectJAutoProxy Javadoc for information on

code-based alternatives to this XML element.

Content Model : (include\*)

<tx:annotation-driven /> @EnableTransactionManager

Enables Spring's annotation-driven transaction management capability, similar to the support found in Spring's <tx:\*> XML namespace. To be used on @Configuration classes as follows:

@Configuration

@EnableTransactionManagement

public class AppConfig {

@Bean

public FooRepository fooRepository() {

// configure and return a class having @Transactional methods

return new JdbcFooRepository(dataSource());

}

@Bean

public DataSource dataSource() {

// configure and return the necessary JDBC DataSource

}

@Bean

public PlatformTransactionManager txManager() {

return new DataSourceTransactionManager(dataSource());

}

}

For reference, the example above can be compared to the following Spring XML configuration:

<beans>

<tx:annotation-driven/>

<bean id="fooRepository" class="com.foo.JdbcFooRepository">

<constructor-arg ref="dataSource"/>

</bean>

<bean id="dataSource" class="com.vendor.VendorDataSource"/>

<bean id="transactionManager" class="org.sfwk...DataSourceTransactionManager">

<constructor-arg ref="dataSource"/>

</bean>

</beans>

In both of the scenarios above, @EnableTransactionManagement and <tx:annotation-driven/> are responsible for registering the necessary Spring components that power annotation-driven transaction management, such as the TransactionInterceptor and the proxy- or AspectJ-based advice that weave the interceptor into the call stack when JdbcFooRepository's @Transactional methods are invoked.

A minor difference between the two examples lies in the naming of the PlatformTransactionManager bean: In the @Bean case, the name is *"txManager"* (per the name of the method); in the XML case, the name is *"transactionManager"*. The <tx:annotation-driven/> is hard-wired to look for a bean named "transactionManager" by default, however @EnableTransactionManagement is more flexible; it will fall back to a by-type lookup for any PlatformTransactionManager bean in the container. Thus the name can be "txManager", "transactionManager", or "tm": it simply does not matter.

For those that wish to establish a more direct relationship between @EnableTransactionManagement and the exact transaction manager bean to be used, the TransactionManagementConfigurer callback interface may be implemented - notice the implements clause and the @Override-annotated method below:

@Configuration

@EnableTransactionManagement

public class AppConfig implements TransactionManagementConfigurer {

@Bean

public FooRepository fooRepository() {

// configure and return a class having @Transactional methods

return new JdbcFooRepository(dataSource());

}

@Bean

public DataSource dataSource() {

// configure and return the necessary JDBC DataSource

}

@Bean

public PlatformTransactionManager txManager() {

return new DataSourceTransactionManager(dataSource());

}

@Override

public PlatformTransactionManager annotationDrivenTransactionManager() {

return txManager();

}

}

This approach may be desirable simply because it is more explicit, or it may be necessary in order to distinguish between two PlatformTransactionManager beans present in the same container. As the name suggests, the annotationDrivenTransactionManager() will be the one used for processing @Transactional methods. See TransactionManagementConfigurer Javadoc for further details.

The mode attribute controls how advice is applied: If the mode is AdviceMode.PROXY (the default), then the other attributes control the behavior of the proxying. Please note that proxy mode allows for interception of calls through the proxy only; local calls within the same class cannot get intercepted that way.

Note that if the mode is set to AdviceMode.ASPECTJ, then the value of the proxyTargetClass attribute will be ignored. Note also that in this case the spring-aspects module JAR must be present on the classpath, with compile-time weaving or load-time weaving applying the aspect to the affected classes. There is no proxy involved in such a scenario; local calls will be intercepted as well.

Since:

3.1

Author:

Chris Beams

Juergen Hoeller

See Also:

TransactionManagementConfigurer

TransactionManagementConfigurationSelector

ProxyTransactionManagementConfiguration

org.springframework.transaction.aspectj.AspectJTransactionManagementConfiguration

* **setMapperLocations**public void setMapperLocations(org.springframework.core.io.Resource... mapperLocations)  
  Set locations of MyBatis mapper files that are going to be merged into the SqlSessionFactory configuration at runtime.  
  This is an alternative to specifying "<sqlmapper>" entries in an MyBatis config file. This property being based on Spring's resource abstraction also allows for specifying resource patterns here: e.g. "classpath\*:sqlmap/\*-mapper.xml".  
  **Parameters:**mapperLocations - location of MyBatis mapper files

사용된 Spring Container 객체의 클래스 이름

1.Junit 사용해서 테스트 케이스 작성

: org.springframework.context.support.GenericXmlApplicationContext 클래스

2. Spring Test 사용해서 테스트 케이스 작성 ( xml 설정파일 사용)

: org.springframework.context.support.GenericApplicationContext

3. Spring Test 사용해서 테스트 케이스 작성 ( Config 설정클래스 사용)

: org.springframework.context.annotation.Class AnnotationConfigApplicationContext

AOP(Aspect Oriented Programming) 용어

1. Aspect ( = Advisor )

= Advice + PointCut

2. Advice

: 부가기능 (공통기능) 예를 들어 로깅,인증,Tx처리

3. PointCut

: Advice를 적용할 Target의 메서드를 선택(선별)하는 정규표현식(Regular Expression)

4. Target

: 핵심기능( Service Biz 로직, DAO Data Access Logic)

: Advice를 적용할 대상

5. JoinPoint

: PointCut에 의해서 선택된 Target의 메서드를 JoinPoint 라고 한다.

6. Weaving (Weaver)

: 런타임에 Target의 JoinPoint에 Advice가 삽입(적용)되는 행위

Jackson 이 직렬화/역직렬화를 처리한다.

Serialization (직렬화)

: Java Objects into JSON

DeSerialization(역직렬화)

: JSON into Java Objects

@RestController

= @Controller + @ResponseBody

@ResponseBody

: 변환된 Json 데이터를 Response(응답)의 Body에 담아서 전달 해주는 역할

: 서버 => 클라이언트에 응답을 줄때

@RequestBody

: 변환된 Java Object를 Request(요청)의 Body에 담아서, 컨트롤러 메서드의 아규먼트로 매핑 해주는 역할

ResponseEntity 클래스

: Body + Status Code + Header

전체 User 목록

GET

[http://localhost:8080/myspringfw/users](http://loccalhost:8087/myspringfw/users)

특정 User

GET

[http://localhost:8080/myspringfw/users/dooly](http://loccalhost:8087/myspringfw/users/dooly)

User 등록

POST

[http://localhost:8080/myspringfw/users](http://loccalhost:8087/myspringfw/users)

request header

content-type:application/json

{

"userId": "rest",

"name": "레스트",

"gender": "남",

"city": "경기"

}

User 수정

PUT

[http://localhost:8080/myspringfw/users](http://loccalhost:8087/myspringfw/users)/dooly

request header

content-type:application/json

{

"gender": "여",

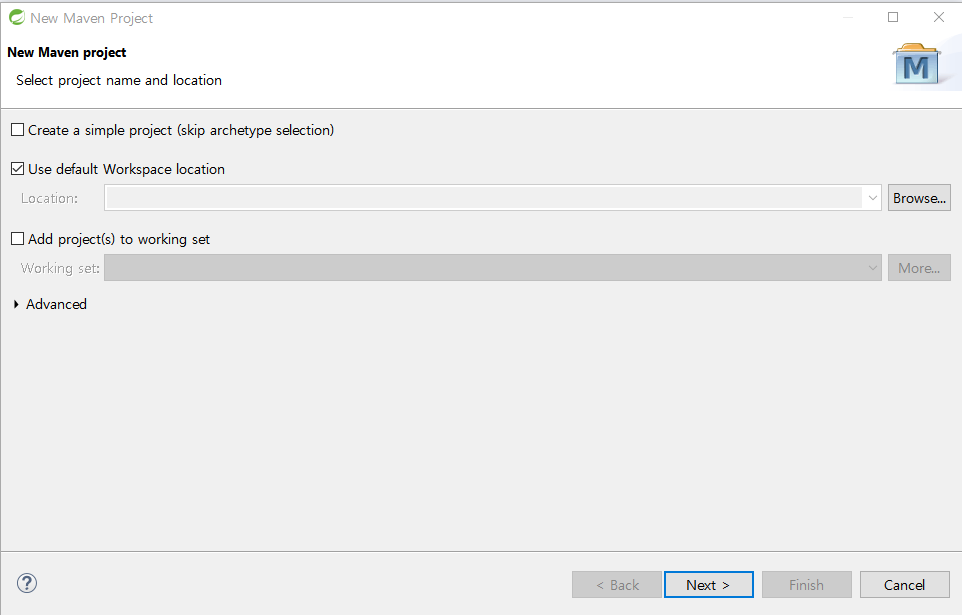
"city": "부산"

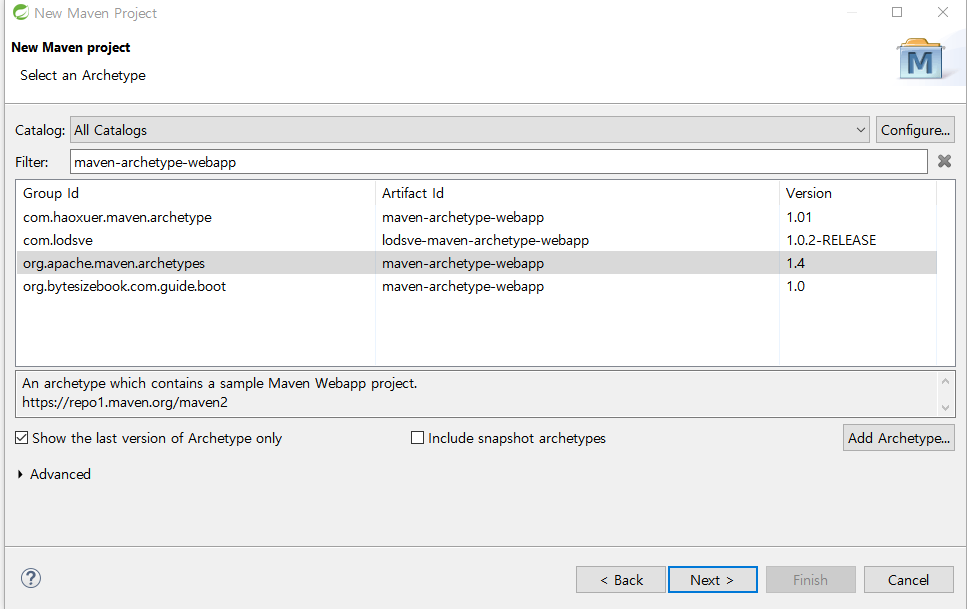
}

Java Object -> Xml 변환 처리는 JAXB(java api for xml binding)

STS에서 Maven 프로젝트 생성하기

[ 처음부터 Maven 프로젝트로 생성하기 ]





groupid : com.maven.spring

artifactid : MavenWebProject

1. JRE 변경

build path -> configure build path -> Libraries 탭에서

1.7 선택하고 edit 버튼 선택해서 1.8 로 변경하기

2. Servlet / JSP 라이브러리 추가하기

build path -> configure build path -> Libraries 탭에서 -> Add Library

-> Server Runtime -> Tomcat8.5 선택

3. src/main/resources 라는 source 폴더를 추가합니다.