**CHAPTER1 스프링 개발 툴**

mavne이나 gradle(<https://gradle.org/releases/>) 을 이용하여 환경구성

기존 manve project에서 gradle로 변경 하려면 해당 명령어 사용하면 된다.

**gradle init --type pom**

**CHAPTER2 스프링 코어**

**레시피 2-1 자바로 POJO 구성하기**

**기본**

public class SequenceGenerator {  
 private String prefix;  
 private String suffix;  
 private int initial;  
 private final AtomicInteger counter = new AtomicInteger();  
  
 public SequenceGenerator() {  
 }  
  
 public void setPrefix(String prefix) {  
 this.prefix = prefix;  
 }  
  
 public void setSuffix(String suffix) {  
 this.suffix = suffix;  
 }  
  
 public void setInitial(int initial) {  
 this.initial = initial;  
 }  
  
 public String getSequence() {  
 StringBuilder builder = new StringBuilder();  
 builder.append(prefix)  
 .append(initial)  
 .append(counter.getAndIncrement())  
 .append(suffix);  
 return builder.toString();  
 }  
}

**구성 클래스에서 @Configuration과 @Bean을 붙여 자바 POJO 생성**

@Configuration  
public class SequenceGeneratorConfiguration {  
 @Bean  
 public SequenceGenerator sequenceGenerator() {  
 SequenceGenerator seqgen = new SequenceGenerator();  
 seqgen.setPrefix("30");  
 seqgen.setSuffix("A");  
 seqgen.setInitial(100000);  
 return seqgen;  
 }  
}

🡺@Configuration은 이 클래스가 구성 클래스임을 스프링에게 알린다.

🡺@Bean을 붙이면 그 메서드와 동일한 이름의 빈이 생성된다.

**IoC 컨테이너를 초기화하여 애너테이션 스캐닝하기**

-Ioc 컨테이너를 인스턴스화 해야지 @Configuration, @Bean을 발견하고 나중에 Ioc 컨테이너에서 빈 인스턴스를 가져올 수 있다.

-bean factory와 application context 두 가지 Ioc 컨테이너를 제공한다.

-애플리케이션 컨텍스트는 빈 팩토리보다 발전된 기능을 지니고 있다.

**ApplicationContext**

ApplicationContext context = new AnnotationConfigApplicationContext(SequenceGeneratorConfiguration.class);

**AnnotationConfigApplicationContext (권장)**

public AnnotationConfigApplicationContext(Class<?>... annotatedClasses) {  
 this();  
 register(annotatedClasses);  
 refresh();  
}

🡺초기화 할 때 refresh 호출하면서 preloading 한다.

**IoC 컨테이너에서 POJO 인스턴스/빈 가져오기**

ApplicationContext context = new AnnotationConfigApplicationContext(SequenceGeneratorConfiguration.class);  
SequenceGenerator generator = context.getBean(SequenceGenerator.class);

🡺getBean을 이용해서 빈을 가지고 올 수 있다.

**POJO 클래스에 @Component를 붙여 DAO 빈 생성하기**

**DAO Interface**

public interface SequenceDao {  
 public Sequence getSequence(String sequenceId);  
  
 public int getNextValue(String sequenceId);  
}

**DAO 구현부**

@Component("sequenceDao")  
public class SequenceDaoImpl implements SequenceDao {  
 private final Map<String, Sequence> sequences = new HashMap<>();  
 private final Map<String, AtomicInteger> values = new HashMap<>();  
  
 public SequenceDaoImpl() {  
 sequences.put("IT", new Sequence("IT", "30", "A"));  
 values.put("IT", new AtomicInteger(10000));  
 }  
  
 @Override public Sequence getSequence(String sequenceId) {  
 return sequences.get(sequenceId);  
 }  
  
 @Override public int getNextValue(String sequenceId) {  
 AtomicInteger value = values.get(sequenceId);  
 return value.getAndIncrement();  
 }  
}

🡺@Component(“sequenceDao”)를 붙이면 스프링은 이 클래스를 이용해 POJO를 생성한다.

🡺value 없으면 기본값 비규격 (sequenceDaoImpl)

-@Component는 스프링이 발견할 수 있게 POJO에 붙이는 범용 애너테이션

-POJO의 쓰임새가 명확하지 않을 땐 그냥 @Component를 붙여도 되지만 명시적으로 하는 것이 좋다. (@Repository, @Service, @Controller)

**애너테이션을 스캐닝하는 필터로 IoC 컨테이너 초기화하기**

-기본적으로 스프링은 @Configuration, @Bean, @Component, @Repository, @Service, @Controller가 달린 클래스를 모두 감지한다.

-이 때 하나 이상의 포함/제외 필터를 적용해서 스캐닝 과정을 커스터마이징 가능

-스프링 지원 필터(4가지)

* annotation, assignable : 필터 대상 애너테이션 타입 및 클래스/인터페이스를 지정
* regex, aspectj는 정규식표현식과 AspectJ 포인트컷 표현식으로 클래스를 매치

@Configuration  
@ComponentScan(  
 includeFilters = {  
 @ComponentScan.Filter(  
 type = FilterType.*REGEX*,  
 pattern = { "chapter2.part1\_2.sequence.\*Dao",  
 "chapter2.part1\_2.sequence.\*Service" }  
 )  
 },  
 excludeFilters = {  
 @ComponentScan.Filter(  
 type = FilterType.*ANNOTATION*,  
 classes = { org.springframework.stereotype.Controller.class }  
 )  
 }  
)  
public class SequenceGeneratorConfiguration {  
}

🡺sequence 패키지에 속한 클래스 중 이름에 Dao나 Service가 포함된 것들은 모두 넣고 @Controller를 붙인 클래스는 뺀다.

**IoC 컨테이너에서 POJO 인스턴스/빈 가져오기**

ApplicationContext context = new AnnotationConfigApplicationContext("chapter2.part1\_2.sequence");  
SequenceDao sequenceDao = context.getBean(SequenceDao.class);  
  
System.*out*.println(sequenceDao.getNextValue("IT"));  
System.*out*.println(sequenceDao.getNextValue("IT"));

**레시피 2-2 생성자 호출해서 POJO 생성하기**

public class ShopConfiguration {  
 @Bean  
 public Product aaa() {  
 Battery p1 = new Battery("AAAA", 2.5);  
 p1.setRechargeable(true);  
 return p1;  
 }  
  
 @Bean  
 public Product cdrw() {  
 Disc p2 = new Disc("CD-RW", 1.5);  
 p2.setCapacity(700);  
 return p2;  
 }  
}

🡺생성자로 POJO 인스턴스/빈을 생성하는 자바 구성 클래스를 작성

ApplicationContext context = new AnnotationConfigApplicationContext(ShopConfiguration.class);  
  
Product aaa = context.getBean("aaa", Product.class);  
Product cdrw = context.getBean("cdrw", Product.class);  
  
System.*out*.println(aaa);  
System.*out*.println(cdrw);

**레시피 2-3 POJO 레퍼런스와 자동 연결을 이용해 다른 POJO와 상호 작용하기**

-필터, 세터 메서드, 생성자, 또는 다른 아무 메서드에 @Autowired를 붙이면 POJO 레퍼런스를 자동 연결해 쓸 수 있다.

**자바 구성 클래스에서 POJO 참조하기**

@Configuration  
public class SequenceConfiguration {  
 @Bean  
 public DatePrefixGenerator datePrefixGenerator() {  
 DatePrefixGenerator dpg = new DatePrefixGenerator();  
 dpg.setPattern("yyyyMMdd");  
 return dpg;  
 }  
  
 @Bean  
 public SequenceGenerator sequenceGenerator() {  
 SequenceGenerator sequence = new SequenceGenerator();  
 sequence.setInitial(100000);  
 sequence.setSuffix("A");  
 sequence.setPrefixGenerator(datePrefixGenerator());  
 return sequence;  
 }  
}

🡺sequenceGenerator 메서드에서 datePrefixGenerator 메서드를 참조할 수 있다.

**POJO 필드에 @Autowired를 붙여 자동 연결하기**

@Component  
public class SequenceService {  
  
 @Autowired  
 private SequenceDao sequenceDao;  
  
 public void setSequenceDao(SequenceDao sequenceDao) {  
 this.sequenceDao = sequenceDao;  
 }  
  
 public String generate(String sequenceId) {  
 Sequence sequence = sequenceDao.getSequence(sequenceId);  
 int value = sequenceDao.getNextValue(sequenceId);  
 return sequence.getPrefix() + value + sequence.getSuffix();  
 }  
}

🡺DAO를 직접 호출하는 대신 일종의 퍼사드를 둔다.

🡺@Component를 붙였기 때문에 스프링 빈으로 등록된다.

🡺@Autowired가 있기 때문에 sequenceDao 빈이 이 프로퍼티에 자동 연결된다.

@Autowired  
private PrefixGenerator[] prefixGenerators;

🡺prefixGenerator와 타입 호환되는 빈을 한 번에 모두 찾아 자동 연결

@Autowired  
private List<PrefixGenerator> prefixGenerators;

🡺이 컬렉션과 타입 호환되는 빈을 모두 찾아 자동 연결

@Autowired  
private Map<String, PrefixGenerator> prefixGeneratorMap;

🡺타입 호환되는 빈을 모두 찾아 빈 이름이 키인 맵에 추가

**@Autowired로 POJO 메서드와 생성자를 자동 연결하기, 자동 연결을 선택적으로 적용하기**

@Autowired  
public void setPrefixGenerator(PrefixGenerator prefixGenerator) {  
 this.prefixGenerator = prefixGenerator;  
}

🡺세터 메서드로 빈 연결

@Autowired(required = false)  
public void setPrefixGenerator(PrefixGenerator prefixGenerator) {  
 this.prefixGenerator = prefixGenerator;  
}

🡺required false 설정하면 스프링이 빈을 못 찾더라도 예외 안남

@Autowired  
public void myOwnCustomInjectionName(PrefixGenerator prefixGenerator) {  
 this.prefixGenerator = prefixGenerator;  
}

🡺메서드 인수의 이름과 개수에 상관없이 적용할 수 있다.

@Autowired

public SequenceGenerator(PrefixGenerator prefixGenerator) {  
 this.prefixGenerator = prefixGenerator;  
}

🡺생성자에도 자동 연결할 수 있음

**애너테이션으로 모호한 자동 연결 명시하기**

-IoC 컨테이너에 호환 타입이 여럿 존재하거나 프로퍼티가 (배열, 리스트, 맵 등의) 그룹형이 아닐 경우 제대로 연결이 되지 않는데, **@Primary, @Qualifier**로 해결할 수 있다.

**@Primary로 모호한 자동 연결 명시하기**

@Component  
@Primary  
public class DatePrefixGenerator implements PrefixGenerator {  
  
 @Override  
 public String getPrefix() {  
 DateFormat formatter = new SimpleDateFormat("yyyyMMdd");  
 return formatter.format(new Date());  
 }  
}

🡺@Primary를 붙이면 여러 빈이 자동 연결 대상일 때 우선권을 부여한다.

**@Qualifier로 모호한 자동 연결 명시하기**

@Autowired  
@Qualifier("datePrefixGenerator")  
private PrefixGenerator prefixGenerator;

🡺@Qualifier로 후보 빈을 명시할 수 있다.

**여러 곳에 분산된 POJO 참조 문제 해결하기**

-규모가 커질수록 POJO 설정을 하나의 클래스에 담아주기 어렵다. 그래서 나누어 관리한다.

🡺여럿이 공존하면 참조하기 어려움.

@Configuration  
@Import(PrefixConfiguration.class)  
public class SequenceConfiguration {  
  
 @Value("#{datePrefixGenerator}")  
 private PrefixGenerator prefixGenerator;  
  
 @Bean  
 public SequenceGenerator sequenceGenerator() {  
 SequenceGenerator sequence = new SequenceGenerator();  
 sequence.setInitial(100000);  
 sequence.setSuffix("A");  
 sequence.setPrefixGenerator(prefixGenerator);  
 return sequence;  
 }  
}

🡺datePrefixGenerator를 설정하기 위해 @Import(PrefixConfiguration.class) 를 설정했다.

**레시피 2-4 @Resource와 @Inject를 붙여 POJO 자동 연결 하기**

-@Resource는 JSR-250에 규정된 애너테이션으로, 이름으로 POJO 레퍼런스를 찾아 연결한다.

-@Inject는 JSR-330에 규정된 애너테이션으로 타입으로 POJO 레퍼런스를 찾아 연결한다.

**@Resource로 POJO 자동 연결하기**

public class SequenceGenerator {  
 @Resource(name = "datePrefixGenerator")  
 private PrefixGenerator prefixGenerator;

…

}

🡺타입이 같은 POJO가 여럿일 때 @Autowired를 사용할 때 @Qualifier를 사용해야 하는 불편함이 있다.

🡺@Resource는 @Autowired + @Qualifier를 합한 것이라고 생각하면 된다.

**@Inject로 POJO 자동 연결하기**

public class SequenceGenerator {  
  
 @Inject  
 private PrefixGenerator prefixGenerator;

…

}

🡺타입이 같은 POJO가 여럿일 때엔 다른 방법을 사용해야 한다. (커스텀 애너테이션 작성)

@Qualifier  
@Target({ElementType.*TYPE*, ElementType.*FIELD*, ElementType.*PARAMETER*})  
@Documented  
@Retention(RetentionPolicy.*RUNTIME*)  
public @interface DatePrefixAnnotation {  
}

🡺@Qualifier는 스프링에서 쓰는 @qualifier와는 전혀 다른 @Inject와 동일 패키지(javax.inject) 애너에티션이다.

@DatePrefixAnnotation  
public class DatePrefixGenerator implements PrefixGenerator {

…

}

🡺실제 주입할 클래스에 DatePrefixGenerator(커스텀 어노테이션)을 붙여 주었다………………

public class SequenceGenerator {  
  
 @Inject  
 @DatePrefixAnnotation  
 private PrefixGenerator prefixGenerator;

…

}

🡺실제 사용 부

**결론**

-@Autowired, @Resource, @Inject 셋다 결과는 동일하다.

-@Autowired는 스프링에, @Resource와 @Inject는 자바 표준에 근거한 해법이라는 차이만 있다.

🡺다중POJO인 경우 @Resource가 가장 간편하다.

**레시피 2-5 @Scope를 붙여 POJO 스코프 지정하기**

-@Scopre는 빈 스코프를 지정하는 애너테이션이다.

|  |  |
| --- | --- |
| **스코프** | **설명** |
| singleton | IoC 컨테이너당 빈 인스턴스 하나를 생성 |
| prototype | 요청할 때마다 빈 인스턴스를 새로 생성 |
| request | HTTP 요청당 하나의 빈 인스턴스를 생성 (웹 애플리케이션 컨텍스트) |
| session | HTTP 세션당 빈 인스턴스 하나를 생성 (웹 애플리케이션 컨텍스트) |
| globalSession | 전역 HTTP 세션당 빈 인스턴스 하나를 생성 (포털 애플리케이션 컨텍스트) |

**카트 클래스**

@Component  
public class ShoppingCart {  
  
 private List<Product> items = new ArrayList<>();  
  
 public void addItem(Product item) {  
 items.add(item);  
 }  
  
 public List<Product> getItems() {  
 return items;  
 }  
}

@Configuration  
@ComponentScan("chapter2.part5\_1.shop")  
public class ShopConfiguration {  
 @Bean  
 public Product aaa() {  
 Battery p1 = new Battery();  
 p1.setName("AAA");  
 p1.setPrice(2.5);  
 p1.setRechargeable(true);  
 return p1;  
 }  
  
 @Bean  
 public Product cdrw() {  
 Disc p2 = new Disc("CD-RW", 1.5);  
 p2.setCapacity(700);  
 return p2;  
 }  
  
 @Bean  
 public Product dvdrw() {  
 Disc p2 = new Disc("DVD-RW", 3.0);  
 p2.setCapacity(700);  
 return p2;  
 }  
}

🡺상품 빈

public static void main(String[] args) throws Exception {  
 ApplicationContext context = new AnnotationConfigApplicationContext(ShopConfiguration.class);  
  
 Product aaa = context.getBean("aaa", Product.class);  
 Product cdrw = context.getBean("cdrw", Product.class);  
 Product dvdrw = context.getBean("dvdrw", Product.class);  
  
 ShoppingCart cart1 = context.getBean("shoppingCart", ShoppingCart.class);  
 cart1.addItem(aaa);  
 cart1.addItem(cdrw);  
 System.*out*.println("Shopping cart 1 contains " + cart1.getItems());  
  
 ShoppingCart cart2 = context.getBean("shoppingCart", ShoppingCart.class);  
 cart2.addItem(dvdrw);  
 System.*out*.println("Shopping cart 2 contains " + cart2.getItems());  
}

🡺서로 다른 카트에 아이템을 넣었지만 결과는? (동일 객체기 때문에 누적된다.)

**결과**



@Component  
@Scope("prototype")  
public class ShoppingCart {

…

}

🡺Scope를 prototype으로 설정

**결과**



**레시피 2-6 외부 리소스(텍스트, XML, 프로퍼티, 이미지 파일)의 데이터 사용하기**

-@PropertySource를 이용하면 빈 프로퍼티 구성용 .properties 파일을 읽어들일 수 있다.

-스프링 Resource 인터페이스에 @Value를 곁들이면 어느 파일이라도 읽을 수 있다.

**프로퍼티 파일 데이터를 이용해 POJO 초깃값 설정하기**

**discounts.properties**

specialcustomer.discount=0.1  
summer.discount=0.15  
endofyear.discount=0.2

@Configuration  
@PropertySource("classpath:discounts.properties")  
@ComponentScan("chapter2.part6\_1.shop")  
public class ShopConfiguration {  
  
 @Value("${endofyear.discount:0}")  
 private double specialEndofyearDiscountField;  
  
 @Bean  
 public static PropertySourcesPlaceholderConfigurer propertySourcesPlaceholderConfigurer() {  
 return new PropertySourcesPlaceholderConfigurer();  
 }

@Bean  
 public Product dvdrw() {  
 Disc p2 = new Disc("DVD-RW", 3.0, specialEndofyearDiscountField);  
 p2.setCapacity(700);  
 return p2;  
 }  
}

🡺@PropertySource를 사용해서 discounts.properteis 정보를 가지고 왔다.

🡺PropertySource를 사용하려면 PropertySourcePlaceholderConfigurer 빈을 선언해야 된다. (중요)

(난 왜 안붙여도 되지 …)

🡺@Value에 placeholer 표현식을 넣어 프로퍼티값을 자바 변수에 할당한다.

🡺프로퍼티 파일 데이터를 빈 프로퍼티 구성 외의 다른 용도로 쓰려면 Resource 메커니즘을 이용한다.

**POJO에서 외부 리소스 파일 데이터를 가져와 사용하기**

public class BannerLoader {  
  
 private Resource banner;  
  
 public void setBanner(Resource banner) {  
 this.banner = banner;  
 }  
  
 @PostConstruct  
 public void showBanner() throws IOException {  
 Files.*lines*(Paths.*get*(banner.getURI()), Charset.*forName*("UTF-8"))  
 .forEachOrdered(System.*out*::println);  
 }  
}

🡺스프링 Resource 형으로 선언되었고, @PostConstruct를 붙여 빈 생성 후 초기시킨다.

@Configuration  
@PropertySource("classpath:discounts.properties")  
@ComponentScan("chapter2.part6\_2.shop")  
public class ShopConfiguration {  
  
 @Value("classpath:banner.txt")  
 private Resource banner;  
  
 @Bean  
 public static PropertySourcesPlaceholderConfigurer propertySourcesPlaceholderConfigurer() {  
 return new PropertySourcesPlaceholderConfigurer();  
 }  
  
 @Bean  
 public BannerLoader bannerLoader() {  
 BannerLoader bl = new BannerLoader();  
 bl.setBanner(banner);  
 return bl;  
 }  
}

🡺@Value(“classpath:banner.txt”)로 banner.txt 파일을 찾아 Resource에 주입했다.

🡺클래스패스에 위치한 리소스 접두어는 classpath:를 쓴다.

**Resource**

-ClassPathResource, FileSystemResource, UrlResource 등으로 가지고 올 수 있음.

**레시피 2-7 프로퍼티 파일에서 로케일마다 다른 다국어 메시지를 해석하기**

-MessageSource 인터페이스에는 리소스 번들 메시지를 처리하는 메서드가 몇가지 정의되어 있고, ResourceBundleMessageSource는 가장 많이 쓰이는 구현체이다.

**message\_ne\_us.properties**

alert.checkout=A shopping cart has been checked out.  
alert.inventory.checkout=A shopping cart with {0} has been checked out at {1}.

@Configuration  
public class ShopConfiguration {  
  
 @Bean  
 public ReloadableResourceBundleMessageSource messageSource() {  
 ReloadableResourceBundleMessageSource messageSource = new ReloadableResourceBundleMessageSource();  
 messageSource.setBasenames("classpath:messages");  
 messageSource.setCacheSeconds(1);  
 return messageSource;  
 }  
}

🡺ReloadableResourceBundleMessageSource 빈 파일을 구성하고 인스턴스는 반드시 messageSource라고 명명해야 알아서 감지한다.

🡺setBasenames() 메서드에 가변 문자열 인수를 넘겨 위치 지정

🡺setCacheSeconds로 캐시 주기 설정

public static void main(String[] args) throws Exception {  
 ApplicationContext context = new AnnotationConfigApplicationContext(ShopConfiguration.class);  
  
 String alert = context.getMessage("alert.checkout", null, Locale.*KOREA*);  
 String alert\_inventory = context.getMessage("alert.inventory.checkout",  
 new Object[] { "[DVD-RW 3.0]", new Date() },  
 Locale.*KOREA*);  
  
 System.*out*.println("The I18N message for alert.checkout is: " + alert);  
 System.*out*.println("The I18N message for alert.inventory.checkout is: " + alert\_inventory);  
}

🡺getMessage() 메서드로 메시지를 해석할 수 있다. (첫 번째는 메시지 키, 세 번째는 대상 로케일)

**레시피 2-8 애너테이션을 이용해 POJO 초기화/폐기 커스터마이징하기**

-자바 구성 클래스의 @Bean 정의부에 initMethod, destoryMEthod 속성을 설정하면 스프링은 이들을 각각 초기화, 폐기 콜백 메서드로 인지한다.

-@PostConstruct, PreDestory를 붙여도 마찬가지

@Configuration  
@ComponentScan("chapter2.part8\_1.shop")  
public class ShopConfiguration {  
 @Bean(initMethod = "openFile", destroyMethod = "closeFile")  
 public Cashier cashier() {  
 String path = System.*getProperty*("java.io.tmpdir") + "/cashier";  
 Cashier c1 = new Cashier();  
 c1.setFileName("checkout");  
 c1.setPath(path);  
 return c1;  
 }  
}

🡺Cashier 빈 생성 이전에 openFile() 메서드를, 폐기 직전에 closeFile() 메서드를 각각 실행하도록 설정한다.

**@PostConstruct와 @PreDestory로 POJO 초기화/폐기 메서드 지정하기**

@Component  
public class Cashier {  
 @Value("checkout")  
 private String fileName;  
  
 @Value("c:/Windows/Temp/cashier")  
 private String path;  
  
 private BufferedWriter writer;  
  
 public void setFileName(String fileName) {  
 this.fileName = fileName;  
 }  
  
 public void setPath(String path) {  
 this.path = path;  
 }  
  
 @PostConstruct  
 public void openFile() throws IOException {  
 File targetDir = new File(path);  
 if (!targetDir.exists()) {  
 targetDir.mkdir();  
 }  
 File checkoutFile = new File(path, fileName + ".txt");  
 if (!checkoutFile.exists()) {  
 checkoutFile.createNewFile();  
 }  
 writer = new BufferedWriter(new OutputStreamWriter(  
 new FileOutputStream(checkoutFile, true)));  
 }  
  
 public void checkout(ShoppingCart cart) throws IOException {  
 writer.write(new Date() + "\t" + cart.getItems() + "\r\n");  
 writer.flush();  
 }  
  
 @PreDestroy  
 public void closeFile() throws IOException {  
 writer.close();  
 }  
}

🡺@Component를 붙여 POJO 클래스 정의할 경우에는 @PostConstruct와 @PreDestroy를 해당 클래스에 붙여 초기화/폐기 메서드를 지정한다.

**참고**

component-scan 시 다음과 같은 어노테이션을 스프링 빈으로 등록한다.

@Component, @Repository, @Service,@Controller, @RestController,

@ControllerAdvice, @Configuration

**@Lazy로 느긋하게 POJO 초기화하기**

-네트워크 접속, 파일 처리 등 무거운 작업을 처리하는 POJO는 느긋한 초기화가 더 어울린다.

@Component  
@Scope("prototype")

@Lazy  
public class ShoppingCart {

…

}

🡺빈에 @Lazy를 붙이면 느긋한 초기화가 적용된다.

🡺애플리케이션이 요구하거나 다른 POJO가 참조하기 전까지 초기화 되지 않는다.

**@DependsOn으로 초기화 순서 정하기**

-여러 자바 구성 클래스에 분산 선언된 많은 POJO가 서로를 참조하다 보면 경합 조건(race condition)이 일어나기 쉽다.

@Configuration  
public class SequenceConfiguration {  
 @Bean  
 @DependsOn("datePrefixGenerator")  
 public SequenceGenerator sequenceGenerator() {  
 SequenceGenerator sequence = new SequenceGenerator();  
 sequence.setInitial(100000);  
 sequence.setSuffix("A");  
 return sequence;  
 }  
}

🡺@DependsOn(“datePrefixGenerator”)를 붙였기 때문에 datePrefixGenerator 빈은 sequenceGenerator 빈보다 반드시 먼저 생성된다.

**레시피 2-9 후처리기를 만들어 POJO 검증/수정하기**

-빈 후처리기를 이용하면 초기화 콜백 메서드(@Bean의 initMethod 속성에서 지정한 메서드나 @PostConstruct를 붙인 메서드) 전후에 원하는 로직을 빈에 적용할 수 있다.

-보통 빈 후처리기는 빈 프로퍼티가 올바른지 체크하거나 어떤 작업을 수행하는 용도로 사용

-기존 POJO 코드를 전혀 건드릴 필요가 없으므로 아주 이상적이다.

**모든 빈 인스턴스를 처리하는 후처리기 생성하기**

-빈 후처리기는 BeanPostProcessor 인터페이스를 구현한 객체다.

@Component  
public class AuditCheckBeanPostProcessor implements BeanPostProcessor {  
  
 @Override  
 public Object postProcessBeforeInitialization(Object bean, String beanName)  
 throws BeansException {  
 System.*out*.println("In AuditCheckBeanPostProcessor.postProcessBeforeInitialization, processing bean type: " + bean.getClass());  
 return bean;  
 }  
  
 @Override  
 public Object postProcessAfterInitialization(Object bean, String beanName)  
 throws BeansException {  
 return bean;  
 }  
}

🡺중요한건 두 메서드 다, 원본 bean을 반환 해야 된다.

🡺클래스 레벨에 @Component를 붙이면 애플리케이션 컨텍스트에 빈 후처리기로 등록된다.

🡺구현 빈을 감지해 컨테이너 안에 있는 다른 빈 인스턴스에 일괄 적용한다.

**주어진 빈 인스턴스만 처리하는 후처리기 생성하기**

@Override  
public Object postProcessBeforeInitialization(Object bean, String beanName)  
 throws BeansException {  
 if (bean instanceof Product) {  
 String productName = ((Product) bean).getName();  
 System.*out*.println("In ProductCheckBeanPostProcessor.postProcessBeforeInitialization, processing Product: " + productName);  
 }  
 return bean;  
}

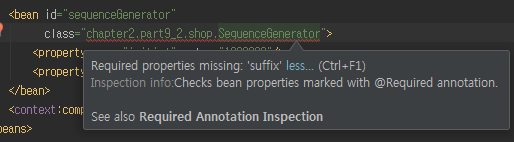
🡺instanceof 사용

**@Required로 프로퍼티 검사하기**

-특정 빈 프로퍼티가 설정되었는지 체크하고 싶은 경우에는 커스텀 후처리기를 작성하고 해당 프로퍼티에 @Required를 붙인다.

public class SequenceGenerator {  
  
 private PrefixGenerator prefixGenerator;  
 private String suffix;  
  
 @Required  
 public void setPrefixGenerator(PrefixGenerator prefixGenerator) {  
 this.prefixGenerator = prefixGenerator;  
 }  
  
 @Required  
 public void setSuffix(String suffix) {  
 this.suffix = suffix;  
 }  
}

🡺@Required를 붙인 프로퍼티는 스프링이 감지해서 값의 존재 여부를 조사하고 프로퍼티값이 없으면 BeanInitializationException 예외를 던진다.



XML에서는 컴파일 시점에 에러가 난다.

**레시피 2-10 팩토리(정적 메서드, 인스턴스 메서드, 스프링 FactoryBean)로 POJO 생성하기**

-IoC 컨테이너 안에서 팩토리 빈은 다른 빈을 찍어내는 공장 역할을 한다.

-스프링은 FactoryBean 인터페이스를 상속한 템플릿 클래스 AbstractFactoryBean을 제공한다.

**정적 팩토리 메서드로 POJO 생성하기**

**정적 팩토리 메서드**

public class ProductCreator {  
 public static Product createProduct(String productId) {  
 if ("aaa".equals(productId)) {  
 return new Battery("AAA", 2.5);  
 } else if ("cdrw".equals(productId)) {  
 return new Disc("CD-RW", 1.5);  
 } else if ("dvdrw".equals(productId)) {  
 return new Disc("DVD-RW", 3.0);  
 }  
 throw new IllegalArgumentException("Unknow product");  
 }  
}

**정적 팩토리 메서드 호출**

@Configuration  
@ComponentScan("chapter2.part10\_1.shop")  
public class ShopConfiguration {  
 @Bean  
 public Product aaa() {  
 return ProductCreator.*createProduct*("aaa");  
 }  
  
 @Bean  
 public Product cdrw() {  
 return ProductCreator.*createProduct*("cdrw");  
 }  
  
 @Bean  
 public Product dvdrw() {  
 return ProductCreator.*createProduct*("dvdrw");  
 }  
}

**인스턴스 팩토리 메서드로 POJO 생성하기**

public class ProductCreator {  
 private Map<String, Product> products;  
  
 public void setProducts(Map<String, Product> products) {  
 this.products = products;  
 }  
  
 public Product createProduct(String productId) {  
 Product product = products.get(productId);  
 if (product != null) {  
 return product;  
 }  
 throw new IllegalArgumentException("Unknow product");  
 }  
}

🡺Map을 이용해서 상품을 담아둘 수도 있따.

@Configuration  
@ComponentScan("chapter2.part10\_2.shop")  
public class ShopConfiguration {  
 @Bean  
 public ProductCreator productCreatorFactory() {  
 ProductCreator factory = new ProductCreator();  
 Map<String, Product> products = new HashMap<>();  
 products.put("aaa", new Battery("AAA", 2.5));  
 products.put("cdrw", new Disc("CD-RW", 1.5));  
 products.put("dvdrw", new Disc("DVD-RW", 3.0));  
 factory.setProducts(products);  
 return factory;  
 }  
  
 @Bean  
 public Product aaa() {  
 return productCreatorFactory().createProduct("aaa");  
 }  
  
 @Bean  
 public Product cdrw() {  
 return productCreatorFactory().createProduct("cdrw");  
 }  
  
 @Bean  
 public Product dvdrw() {  
 return productCreatorFactory().createProduct("dvdrw");  
 }  
}

🡺createProduct을 통해서 다른 빈들을 인스턴스화 한다.

**스프링 팩토리 빈으로 POJO 생성하기**

public class DiscountFactory extends AbstractFactoryBean<Product> {  
 private Product product;  
 private double discount;  
  
 public void setProduct(Product product) {  
 this.product = product;  
 }  
  
 public void setDiscount(double discount) {  
 this.discount = discount;  
 }  
  
 @Override public Class<?> getObjectType() {  
 return product.getClass();  
 }  
  
 @Override protected Product createInstance() throws Exception {  
 product.setPrice(product.getPrice() \* (1 - discount));  
 return product;  
 }  
}

🡺팩토리 빈은 제네릭 클래서 AbstractFactoryBean<T>을 상속하고, createInstance() 메서드를 오버라이드해 대상 빈 인스턴스를 생성한다.

🡺자동 연결 기능이 작동하도록 getObjectType() 메서드로 대상 빈 타입을 반환한다.

🡺팩토리빈은 AbstractFactoryBean<T>를 상속하고 createInstance() 메서드를 오버라이드해 대상 빈 인스턴스를 생성한다.

@Configuration  
@ComponentScan("chapter2.part10\_3.shop")  
public class ShopConfiguration {  
 @Bean  
 public Product aaa() {  
 Battery aaa = new Battery("AAA", 2.5);  
 return aaa;  
 }  
  
 @Bean  
 public Product cdrw() {  
 Disc aaa = new Disc("CD-RW", 1.5);  
 return aaa;  
 }  
  
 @Bean  
 public Disc dvdrw() {  
 Disc aaa = new Disc("DVD-RW", 3.0);  
 return aaa;  
 }  
  
 @Bean  
 public DiscountFactoryBean discountFactoryBeanAAA() {  
 DiscountFactoryBean factory = new DiscountFactoryBean();  
 factory.setProduct(aaa());  
 factory.setDiscount(0.2);  
 return factory;  
 }  
  
 @Bean  
 public DiscountFactoryBean discountFactoryBeanCDRW() {  
 DiscountFactoryBean factory = new DiscountFactoryBean();  
 factory.setProduct(cdrw());  
 factory.setDiscount(0.1);  
 return factory;  
 }  
  
 @Bean  
 public DiscountFactoryBean discountFactoryBeanDVDRW() {  
 DiscountFactoryBean factory = new DiscountFactoryBean();  
 factory.setProduct(dvdrw());  
 factory.setDiscount(0.1);  
 return factory;  
 }  
}

🡺Bean을 붙여 자동 적용

**레시피 2-11 스프링 환경 및 프로파일마다 다른 POJO 로드하기**

-POJO 초깃값은 애플리케이션 시나리오마다 달라질 수 있다. (개발 -> 테스트 -> 운영)

@Configuration  
@Profile("spring")  
public class ShopConfigurationSpr {  
  
 @Bean  
 public Product aaa() {

…  
 }  
  
 @Bean  
 public Product cdrw() {  
 …  
 }  
  
 @Bean  
 public Product dvdrw() {  
 …  
 }  
}

@Configuration  
@Profile("summer")  
public class ShopConfigurationSummer {  
  
 @Bean  
 public Product aaa() {

…  
 }  
  
 @Bean  
 public Product cdrw() {  
 …  
 }  
  
 @Bean  
 public Product dvdrw() {  
 …  
 }  
}

🡺@Profile로 여러 버전으로 나누었다.

🡺@Profile 이름이 여러 개면 CSV 형식으로 {}로 감싸 넣는다. ex) {“summer”, “winter”}

**프로파일을 환경에 로드하기**

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
context.getEnvironment().setActiveProfiles("global", "autumn");  
context.scan("chapter2.part11\_1.shop");  
context.refresh();

🡺프로파일을 로드하려면 setActiveProfiles() 메서드를 호출한다.

-자바 런타임 플래그로 로드할 프로파일을 명시하려면?

🡺-Dspring.profiles.active=global,winter

**기본 프로파일 지정하기**

-setActiveProfiles() 대신 setDefaultProfiles() 메서드를 사용한다.

**보충**

<https://www.lesstif.com/pages/viewpage.action?pageId=18220309>

**1) jvm property**

-Dspring.profiles.active=dev

**2) web.xml**

<context-param>  
 <param-name>spring.profiles.active</param-name>  
 <param-value>dev</param-value>  
</context-param>

**3) ApplicatinoContext 설정**

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  
context.getEnvironment().setActiveProfiles("global", "autumn");

**4) @ActiveProfiles annotation**

@RunWith(SpringJUnit4ClassRunner.class)  
// ApplicationContext will be loaded from "classpath:/app-config.xml"  
@ContextConfiguration("/app-config.xml")  
@ActiveProfiles("dev")  
public class TransferServiceTest {  
 @Autowired  
 private TransferService transferService;  
  
 @Test  
 public void testTransferService() {  
 // test the transferService  
 }  
}

**레시피 2-12 POJO에게 IoC 컨테이너 리소스 알려주기**

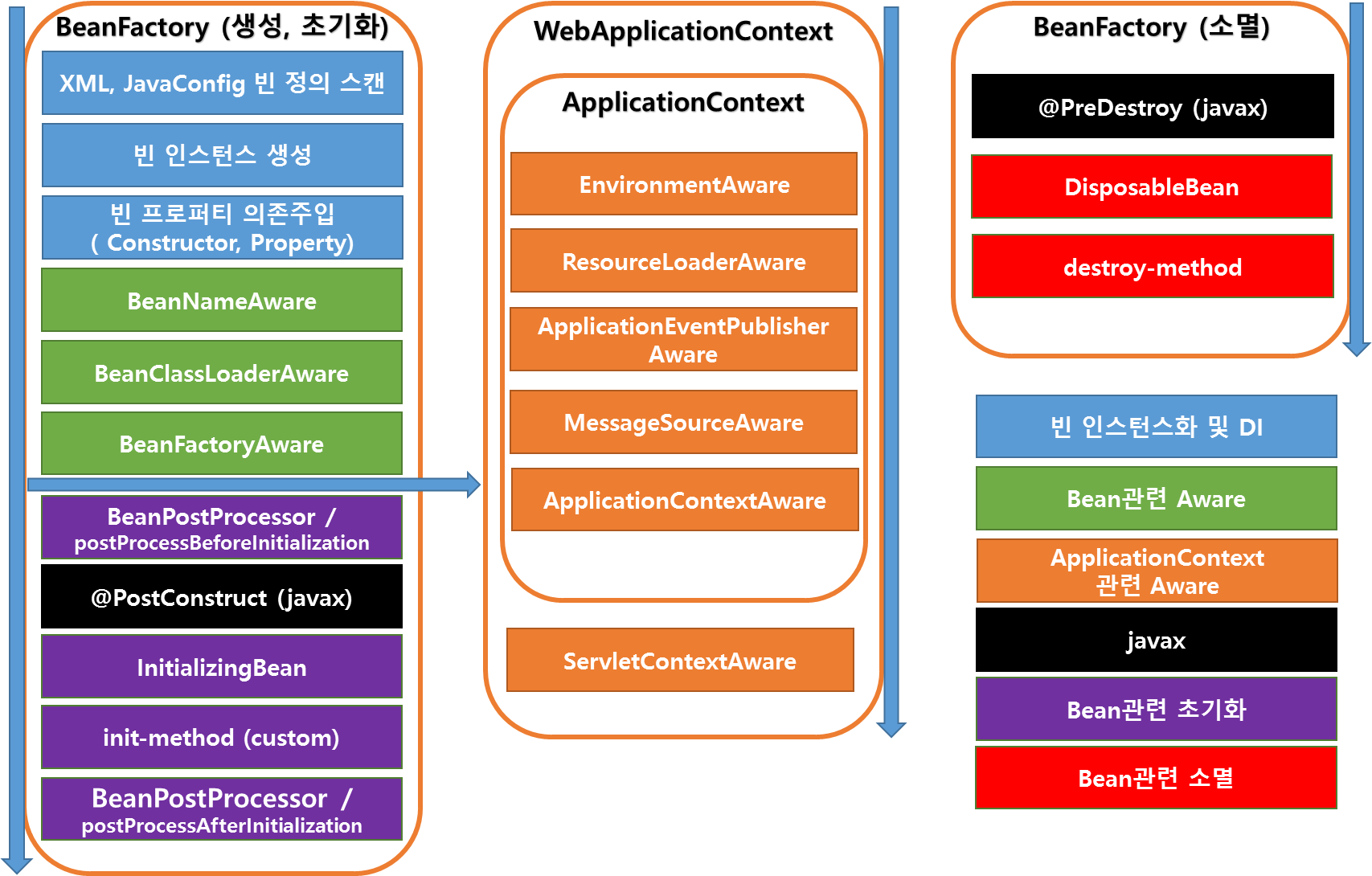
**Aware ?**

Bean이 Spring 컨테이너와 리소스에 대한 정보를 얻어 오기 위해 Aware 인터페이스 제공

우리는 단지 setter 메소드만 구현하면 정보를 받아올 수 있다.

|  |  |
| --- | --- |
| **Aware 인터페이스** | **대상 리소스 타입** |
| BeanNameAware | IoC 컨테이너에 구성한 인스턴스의 빈 이름 |
| BeanFactoryAwre | 현재 빈 팩토리, 컨테이너 서비스를 호출하는데 쓰임 |
| ApplicationContextAware | 현재 애플리케이션 컨텍스트, 컨테이너 서비스를 호출하는데 쓰임 |
| MessageSourceAware | 메시지 소스, 텍스트 메시지를 해석하는데 쓰임 |
| ApplicationEvent PublisherAware | 애플리케이션 이벤트 발행기(퍼블리셔), 애플리케이션 이벤트를 발생하는데 쓰임 |
| ResourceLoaderAware | 리소스 로더, 외부 리소스를 로드하는데 쓰임 |
| EnvironmentAware | ApplicationCotnext 인터페이스에 묶인 Environment 인스턴스 |

**BeanFactory와 ApplicationContext Aware 구분**



-Aware 인터페이스의 세터 메서드는 스프링이 빈 프로퍼티 설정 이후, 초기화 콜백 메서드를 호출하기 이전에 호출 (@PostConstruct, init-method 등)

**lifeCycle**

1) 생성자나 팩토리 메서드를 호출해 빈 인스턴스 생성

2) 빈 프로퍼티에 값, 빈 레퍼런스를 설정

3) Aware 인터페이스에 정의한 세터 메서드 호출

4) 빈 인스턴스를 각 후처리기에 있는 postProcessBeforeInitialization() 메서드로 넘겨 추기화 콜백 메서드 호출

5) 빈 초기화 콜백 메서드

6) 빈 인스턴스를 각 후처리기 postProcessAfterInitialization() 메서드로 넘김. 이 시점에 빈 사용준비 끝~

7) 컨테이너가 종료되면 폐기 콜백 메서드(@PreDestroy destroy-method 등 )를 호출한다.

-Aware는 스프링 lifecycle과 연관되기 때문에 IoC 컨테이너 외부에서는 제대로 작동하지 않음.

🡺POJO가 아닌 듯

**CHAPTER4 스프링 REST**

-REST는 HTTP를 기본으로 웹 서비스를 구현하는 아키텍처로 각광받고 있다.

**레시피 4-1 REST 서비스로 XML 발행하기**

-스프링 mvc에서 애플리케이션 데이터를 REST 서비스로 발행하는 주역은 @RequestMapping과 @PathVariable 두 애너테이션이다.

**MarshallingVeiw로 XML 만들기**

@RequestMapping("/members")  
public String getRestMembers(Model model) {  
 Members members = new Members();  
 members.addMembers(memberService.findAll());  
 model.addAttribute("members", members);  
 return "membertemplate";  
}

🡺host\_name/애플리케이션명/members URL로 접근 가능

@Bean  
public View membertemplate() {  
 return new MarshallingView(jaxb2Marshaller());  
}

🡺위의 뷰는 MarshallingView 형으로 정의 (마샬러를 이용해 렌더링 하는 범용 클래스)

-마샬링은 메모리에 있는 객체를 특정한 데이터 형식으로 변환하는 과정

@Bean  
public Marshaller jaxb2Marshaller() {  
 Jaxb2Marshaller marshaller = new Jaxb2Marshaller();  
 marshaller.setClassesToBeBound(Members.class, Member.class);  
 return marshaller;  
}

🡺classesToBeBound는 XML로 변환할 대상 클래스

@XmlRootElement  
public class Member {  
 private String name;  
 private String phone;  
 private String email;

}

🡺@XmlRootElement는 Jaxb2Marshaller가 클래스 필드를 자동으로 감지해서 XML 데이터로 변경

**콘텐트 협상**

-http://[호스트명]/[애플리케이션명]/memebrs 처럼 확장 없이 URL을 요청할 경우 MVC 뷰의 해석 로직에 달려 있다. 이런 경우 URL 확장자 또는 HTTP 헤더를 보고 뷰를 결정한다.

ex) Accept: application/xml

**@ResponseBody로 XML 만들기**

-여러 뷰로 보여주기 위해 뷰를 하나씩 추가하는건 너무 번거롭기 때문에 이런 경우 HttpMessageConverter를 이용하면 된다.

@RequestMapping("/members")  
@ResponseBody  
public Members getRestMembers(Model model) {  
 Members members = new Members();  
 members.addMembers(memberService.findAll());  
 return members;  
}

🡺@ResponseBody를 사용해서 응답 본문으로 취급

**결 과**

<http://localhost:8080/members.xml>

<members>  
 <member>  
 <email>marten@deinum.biz</email>  
 <name>Marten Deinum</name>  
 <phone>00-31-1234567890</phone>  
 </member>  
 <member>  
 <email>john@doe.com</email>  
 <name>John Doe</name>  
 <phone>1-800-800-800</phone>  
 </member>  
 <member>  
 <email>jane@doe.com</email>  
 <name>Jane Doe</name>  
 <phone>1-801-802-803</phone>  
 </member>  
</members>

<http://localhost:8080/members>

marten@deinum.bizMarten Deinum00-31-1234567890john@doe.comJohn Doe1-800-800-800jane@doe.comJane Doe1-801-802-803

**@PathVariable로 결과 거르기**

@RequestMapping("/member/{memberid}")  
@ResponseBody  
public Member getMember(@PathVariable("memberid") long memberID) {  
 return memberService.find(memberID);  
}

🡺@PathVariable을 이용해서 매개변수 사용하기

@RequestMapping("/member/\*/{memberid}")

🡺와일드카드 사용할 경우 /member/심준보/1 이런 url 호출도 가능

**ResponseEntity로 클라이언트에게 알려주기**

@RequestMapping("/member/{memberid}")  
@ResponseBody  
public ResponseEntity<Member> getMember(@PathVariable("memberid") long memberID) {  
 Member member = memberService.find(memberID);  
 if (member != null) {  
 return new ResponseEntity<Member>(member, HttpStatus.*OK*);  
 }  
 return new ResponseEntity(HttpStatus.*NOT\_FOUND*);  
}

🡺ResponseEntity는 결과 본문을 HTTP 상태 코드와 함께 집어넣은 래퍼 클래스이다.

**레시피 4-2 REST 서비스로 JSON 발행하기**

**JSON 형식으로 처리하는 이유**

🡺어느 브라우저든 자바스크립트 언어 해석기는 다 장착되어 있기 때문에

**MappingJackson2JsonView로 JSON 만들기**

@RequestMapping("/members")  
public String getRestMembers(Model model) {  
 Members members = new Members();  
 members.addMembers(memberService.findAll());  
 model.addAttribute("members", members);  
 return "jsonmembertemplate";  
}

@Bean  
public View jsonmembertemplate() {  
 MappingJackson2JsonView view = new MappingJackson2JsonView();  
 view.setPrettyPrint(true);  
 return view;  
}

🡺MappingJackson2JsonView 뷰는 잭슨2 라이브러리를 이용해 객체를 JSON으로 바꾼다. (내부적으로는 ObjectMapper 사용)

**Xml, Json 별도 뷰**

@RequestMapping(value = "/members", produces = MediaType.*APPLICATION\_XML\_VALUE*)  
public String getRestMembersByXml(Model model) {  
 Members members = new Members();  
 members.addMembers(memberService.findAll());  
 model.addAttribute("members", members);  
 return "xmlmembertemplate";  
}  
  
@RequestMapping(value = "/members", produces = MediaType.*APPLICATION\_JSON\_VALUE*)  
public String getRestMembersByJson(Model model) {  
 Members members = new Members();  
 members.addMembers(memberService.findAll());  
 model.addAttribute("members", members);  
 return "jsonmembertemplate";  
}

🡺하지만 이런 방식은 뷰 별로 중복이 되버린다.

**/members.xml**



**/members.json**



**@ResponseBody로 Json 만들기**

@RequestMapping(value = "/members")  
@ResponseBody  
public Members getRestMembersByXml(Model model) {  
 Members members = new Members();  
 members.addMembers(memberService.findAll());  
 return members;  
}

@Configuration  
@EnableWebMvc  
@ComponentScan(basePackages = "chapter4.part7")  
public class CourtRestConfiguration {  
  
}

위 **@ResponseBody로 XML 만들기** 와 동일

**왜 이런일들이 가능할까?**

-스프링 MVC가 클래스패스에 있는 것을 자동 감지하기 때문에…

-JAXB2, 잭슨 등의 라이브러리가 발견되면 해당 기술에 적합한 HttpMessageConvert를 알아서 등록

**GSON으로 JSON만들기**

compile group: 'com.google.code.gson', name: 'gson', version: '2.8.0'

Jackson 대신 해당 라이브러리 추가

**레시피 4-3 스프링으로 REST 서비스 액세스하기**

-페이로드 : 사용에 있어서 전송되는 데이터를 뜻한다. (헤더와 메타데이터 제외)

-페이로드를 쉽게 찾아 쓰게 할 목적으로 WADL(Web Application Description Language)를 제공하는 REST 서비스도 있다.

public static void main(String[] args) {  
 final String uri = "http://localhost:8080/members.json";  
 RestTemplate restTemplate = new RestTemplate();  
 String result = restTemplate.getForObject(uri, String.class);  
 System.*out*.println(result);  
}

🡺RestTeamplte 을 사용해서 서비스에 액세스 했다.

**매개변수화한 URL에서 데이터 가져오기**

private static void placeholders() {  
 final String uri = "http://localhost:8080/member/{memberId}";  
 Map<String, String> params = new HashMap<>();  
 params.put("memberId", "1");  
  
 RestTemplate restTemplate = new RestTemplate();  
 String result = restTemplate.getForObject(uri, String.class, params);  
 System.*out*.println(result);  
}

🡺HashMap을 이용하여 getForObject 메서드에 넣어서 호출한다.

**데이터를 매핑된 객체로 가져오기**

final String uri = "http://localhost:8080/members.xml";  
RestTemplate restTemplate = new RestTemplate();  
Members result = restTemplate.getForObject(uri, Members.class);

🡺결과를 Members.class로 지정

**레시피 4-4 RSS/아톰 피드 발행하기**

-RSS/아톰 피드는 정보를 발행하는데 널리 쓰이는 수단이다.

**RSS/아톰 공통적 특징**

-피드 내용을 서술하는 메타데이터 영역이 있다. (아톰 <authr>와 <title>, RSS의 <description>과 <pubDate>)

-순환 엘리먼트로 복수의 정보를 나타낼 수 있다.(dㅏ톰 <entity>, RSS의 <item>)

-버전이 다양하다.

**Atom**

public class AtomFeedView extends AbstractAtomFeedView {  
  
 @Override  
 protected void buildFeedMetadata(Map<String, Object> model, Feed feed, HttpServletRequest request) {

}  
  
 @Override  
 protected List<Entry> buildFeedEntries(Map<String, Object> model, HttpServletRequest request, HttpServletResponse response) {

}

🡺buildFeedMetadata() 메서드는 피드 데이터가 담긴 Map 객체(핸들러 메서드에 할당된 데이터)

, 피드를 처리하는데 필요한 ROME의 Feed, HTTP 요청 시 필요한 HttpServletRequest

🡺buildFeedEntries() 메서드는 피드 데이터가 담긴 Map 객체, HTTP 요청을 다루어야 할 때 필요한 HttpServletRequest, HTTP 응답을 다루어야 할 때 필요한 HttpServletResponse

**/atomfeed.atom**



**RSS**

public class RSSFeedView extends AbstractRssFeedView {  
  
 @Override  
 protected void buildFeedMetadata(Map<String, Object> model, Channel feed, HttpServletRequest request) {  
  
 }  
  
 @Override  
 protected List<Item> buildFeedItems(Map<String, Object> model, HttpServletRequest request, HttpServletResponse response)  
 throws Exception {  
  
 }  
}

🡺buildFeedMetadata() 메서드는 Channel 객체를 사용해서 RSS 피드를 만든다.

🡺buildFeedItems()는 Atom의 **buildFeedEntries**와 용도가 같다.

**/rssfeed.rss**

