**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.

**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**

