
Table of Contents

Introduction	1.1
Chapter1. 스프링 속으로	1.2
Chapter2. 빈 와이어링(무기)	1.3
Chapter3. 고급 와이어링	1.4
Chapter4. 애스팩트 지향 스프링	1.5
Chapter5. 스프링 웹 애플리케이션 만들기	1.6
Chapter6. 웹 뷰 렌더링	1.7
Chapter10. 스프링과 JDBC를 사용하여 데이터베이스 사용하기	1.8
Chapter11. 객체 관계형 매핑을 통한 데이터 퍼시스팅	1.9
Chapter16. 스프링 MVC로 REST API 사용하기	1.10
Chapter21. 스프링 부트를 사용한 스프링 개발 간소화	1.11

스프링 스터디

- 목표: 교재 마스터하기
- 교재: 스프링 인 액션 4판
 - <http://www.yes24.com/24/goods/23542303>
- 시간: PM12 ~ PM2
- 장소: S동 6층 Green 구역 Cream 회의실
 - http://images.daumcorp.com/meetingroom/ALL_s06-1.jpg
- 일정: 10주간 주1회
- 기간: 2016-05-12.목 ~ 2016-07-14.목

GitBook

- Web: <https://www.gitbook.com/book/rebeccacho/spring-study-group>
- GitHub: <https://git.gitbook.com/rebeccacho/spring-study-group.git>

스터디 일정

- 2016-05-12.목: [Chapter1. 스프링 속으로](#) @adel.yang, @seed.sim
- 2016-05-19.목: [Chapter2. 빈 와이어링\(묵기\)](#) @rylan.kim, @adel.yang
- 2016-05-26.목: [Chapter3. 고급 와이어링](#) @poet.me, @sai.san
- 2016-06-02.목: [Chapter4. 애스팩트 지향 스프링](#) @seed.sim, @rebecca.cho
- 2016-06-09.목: [Chapter5. 스프링 웹 애플리케이션 만들기](#) @jayden.uk, @clint.cho
- 2016-06-16.목: [Chapter6. 웹 뷰 렌더링](#) @june.kim, @azalea.oh
- 2016-06-23.목: [Chapter10. 스프링과 JDBC를 사용하여 데이터베이스 사용하기](#) @rebecca.cho, @bryan.79
- 2016-06-30.목: [Chapter11. 객체 관계형 매핑을 통한 데이터 퍼시스팅](#) @clint.cho, @rylan.kim
- 2016-07-07.목: [Chapter16. 스프링 MVC로 REST API 사용하기](#) @june.kim, @poet.me
- 2016-07-14.목: [Chapter21. 스프링 부트를 사용한 스프링 개발 간소화](#) @bryan.79, @chris.song

Chapter1. 스프링 속으로

1.1 자바 개발 간소화

1.1.1 POJO 의 힘

POJO: Plain Old Java Object. 어느 컨벤션이나 프레임워크의 요구사항이 전혀 없는 보통 자바 오브젝트

ex) extends, implements, annotations, ... 등을 제거한 오브젝트

- POJO 를 이용한 가볍고(lightweight) 비침투적(non-invasive)인 개발

1.1.2 종속객체 주입

DI: Dependency Injection. 구성요소간의 종속관계를 외부에서 주입하는 패턴
Setter, Interface, 그리고 Constructor based injection 을 흔히 볼 수 있다.

- DI와 인터페이스 지향(interface orientation)을 통한 느슨한 결합도(loose coupling)

1.1.3 애스펙트 적용

AOP: Aspect-Oriented Programming. 비즈니스 로직에서 2차적 또는 보조 기능들을 분리시키는 개발 방법. 이렇게 분리시킨 모듈을 필요한 곳에 삽입하되, 코드 밖에서 설정한다.

로깅, 인증, 보안, 트랜잭션 관리, 메소드 성능검사, 예외반환, ... 같은 공동 모듈

- 애스펙트와 공통 규약을 통한 선언적(declarative) 프로그래밍

1.1.4 템플릿을 이용한 상투적인 코드 제거

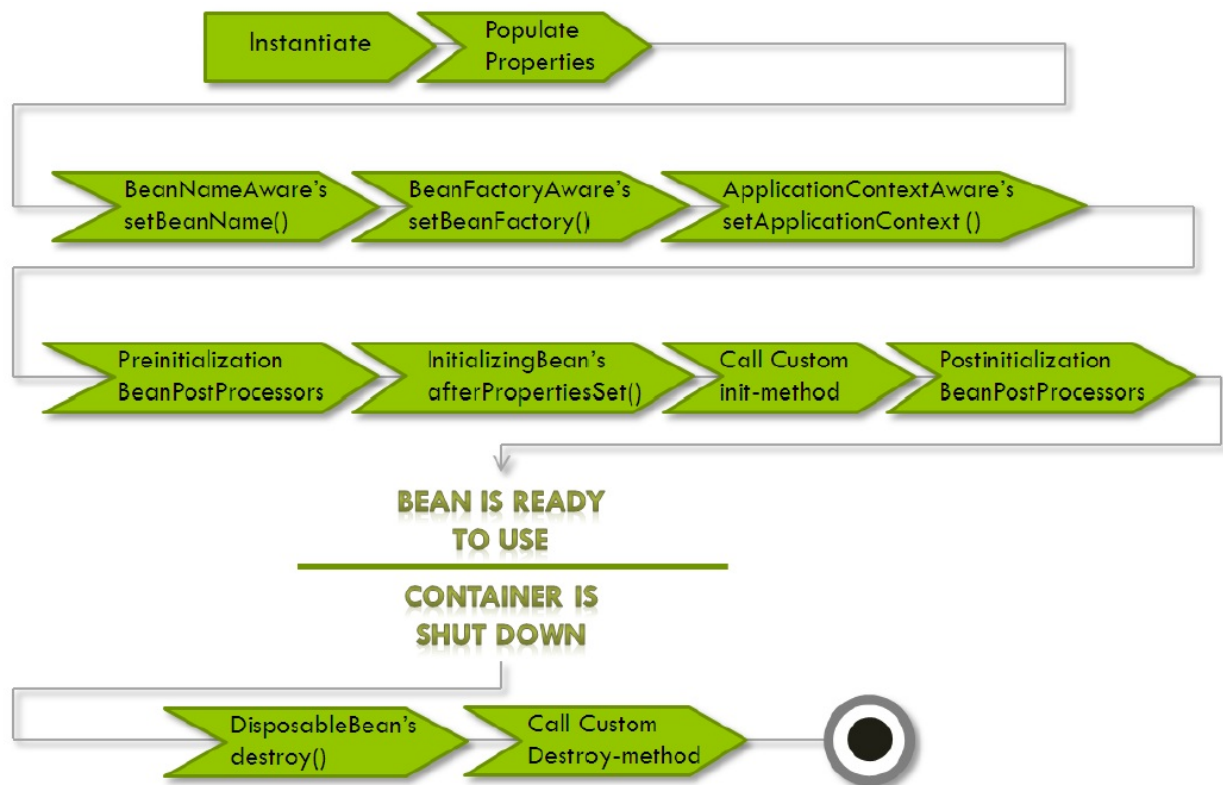
- 애스펙트와 템플릿(template)을 통한 반복적인 코드 제거

1.2 빈을 담는 그릇, 컨테이너

- 스프링 기반 애플리케이션에서는 스프링 컨테이너(Spring Container) 안에서 객체가 태어나고, 자라고, 소멸한다.
 - DI 을 이용해서 애플리케이션을 구성하는 컴포넌트를 관리.
- 크게 2가지로 분류된다.
 - 빈 팩토리(BeanFactory)
 - 애플리케이션 컨텍스트(Application Context)

1.2.1 또 하나의 컨테이너, 애플리케이션 컨텍스트

1.2.2 빈의 일생

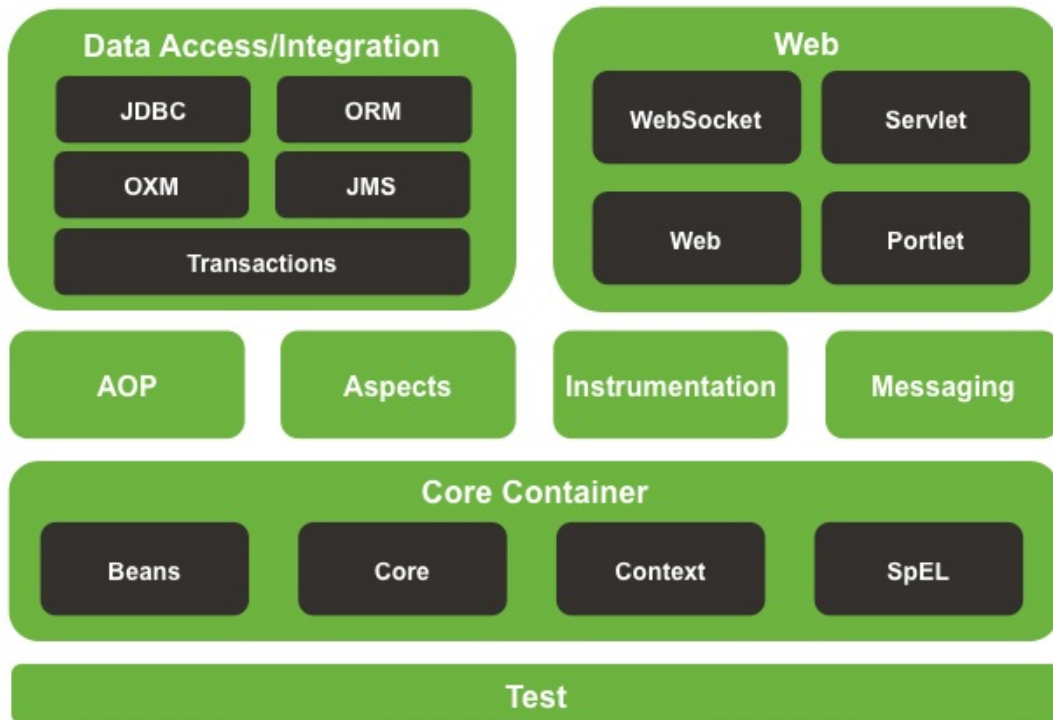


1.3 스프링 현황

1.3.1 스프링 모듈



Spring Framework Runtime



1. Core container

어플리케이션에서 빈의 생성, 설정 그리고 처리방법을 관리하는 컨테이너. 이 모듈 내에서 DI를 제공하는 스프링 빈 팩토리를 확인한다. 이 외에도 이메일, JNDI 액세스, EJB통합, 스케줄링 등 다양한 엔터프라이즈 서비스도 제공한다. 모든 스프링 모듈은 코어 컨테이너 위에 구축되므로 암묵적으로 코어모듈을 사용한다.

1. Data Access/Integration

◦ JDBC : spring-jdbc

데이터 access시 커넥션을 얻어오고, 질의객체를 생성하고 결과집합을 처리하고 커넥션을 닫는 작업을 반복한다. JDBC와 DAO 모듈은 이렇게 반복되는 코드를 추상화하고 리소스 관리를 해준다.

◦ ORM : spring-orm

ORM모듈은 JDBC보다 객체관계 매핑 도구를 선호하는 사람들을 위한 모듈이다. 직접 ORM 솔루션 구현보다는 hibernate, java persistence, mybatis와 같은 ORM 프레임워크와의 연결고리를 제공해준다.

- OXM : spring-oxm

객체 - XML 매핑

- JMS : spring-jms
- Transaction : spring-tx

AOP모듈을 이용하여 객체들의 트랜잭션을 관리한다.

2. web

- Web
- Web-MVC
- Web-Socket
- Web-Portlet

3. etc

- AOP
- Aspects

AOP 모듈을 통해 aspect 지향 프로그래밍을 풍부하게 지원한다. AOP는 주로 애플리케이션의 전체 걸친 관심사 (트랜잭션, 보안, 로깅 etc)와 객체 간의 결합도를 낮추는데 이용된다.

- Instrumentation

> JVM에 에이전트를 추가하는 기능을 제공한다. 정확히는 톰캣용 위빙 에이전트 제공. (좀더 찾아보자)

- Messaging
- Test

> 단위테스트를 위한 모의객체 구현이나 어플리케이션 컨텍스트에서 빈을 로드하고 이 컨텍스트에 있는 빈과의 작업을 지원한다.

1.3.2 스프링 포트폴리오

1. 스프링 웹 플로

웹 어플리케이션의 네비게이션 흐름제어 구축을 지원합니다.

2. 스프링 웹 서비스

SOAP 웹 서비스 개발을 용이하게 합니다.

3. 스프링 시큐리티

포괄적이고 확장 가능한 인증 및 인가를 지원하여 애플리케이션을 보호합니다.

4. 스프링 인티그레이션

엔터프라이즈 인티그레이션 패턴을 지원합니다.

5. 스프링 배치

대량 배치작업에 대해 간결하고 최적화된 처리를 합니다.

6. 스프링 데이터

데이터 접근에 대한 일관된 방식을 제공합니다. 관계형, 비관계형, 맵리듀스 등

7. 스프링 소셜

페이스북, 트위터, 링크드인과 같은 서드파티 API와 쉽게 연결해줍니다.

8. 스프링 모바일

장비 탐지 및 진보적인 랜더링 옵션을 통해 모바일 웹 어플리케이션 개발을 간편하게 합니다.

9. 안드로이드 스프링

안드로이드 개발에 사용되는 스프링 컴포넌트를 제공합니다.

10. 스프링 부트

스프링 애플리케이션을 구축하고 빠르게 실행 가능한 독단적인 뷰를 제공합니다.

Spring Framework Artifacts

GroupId	ArtifactId	Description
org.springframework	spring-aop	Proxy 기반 AOP 지원
org.springframework	spring-aspects	AspectJ 기반 스프링 aspect
org.springframework	spring-beans	Bean 지원
org.springframework	spring-context	애플리케이션 컨텍스트의 런타임 구현, 스케줄

		지원
org.springframework	spring-context-support	스프링과 서드파티 라이브러리간의 통합 지원
org.springframework	spring-core	Core 유틸리티
org.springframework	spring-expression	스프링 표현 언어 (SpEL)
org.springframework	spring-instrument	JVM 부트 스트래핑을 위한 instrumentation agent
org.springframework	spring-instrument-tomcat	Instrumentation agent for Tomcat
org.springframework	spring-jdbc	데이터 소스 설정과 JDBC 액세스 지원하는 JDBC 패키지
org.springframework	spring-jms	동기식 JMS 액세스와 메시지 리스너 컨테이너를 지원
org.springframework	spring-messaging	메시징 아키텍처와 프로토콜 지원
org.springframework	spring-orm	JPA, hibernate 등의 ORM 지원
org.springframework	spring-oxm	Object/XML 매핑
org.springframework	spring-test	유닛테스트 또는 integration 테스트 지원
org.springframework	spring-web	web 기능 지원
org.springframework	spring-webmvc	웹 어플리케이션을 위한 REST웹 서비스 및 MVC 구현체
org.springframework	spring-webmvc-portlet	포틀릿 환경의 MVC 구현체
org.springframework	spring-websocket	WebSocket, sockJS 구현체 (STOMP 지원)
org.springframework.webflow	spring-webflow	스프링 웹 플로우
org.springframework.ws	spring-ws-core	스프링 웹 서비스
org.springframework.security	spring-security-web	스프링 시큐리티 (spring aop)

org.springframework.integration	spring-integration-core	스프링 인티그레이션
org.springframework.batch	spring-batch-core	스프링 배치
org.springframework.data	spring-data-releasetrain	스프링 데이터
org.springframework.social	spring-social-facebook, spring-social-twitter	스프링 소셜
org.springframework.mobile	spring-mobile-device	스프링 모바일
org.springframework.android	spring-android-rest-template	스프링 소셜
org.springframework.boot	spring-boot-starter-web	스프링 부트

spring life cycle

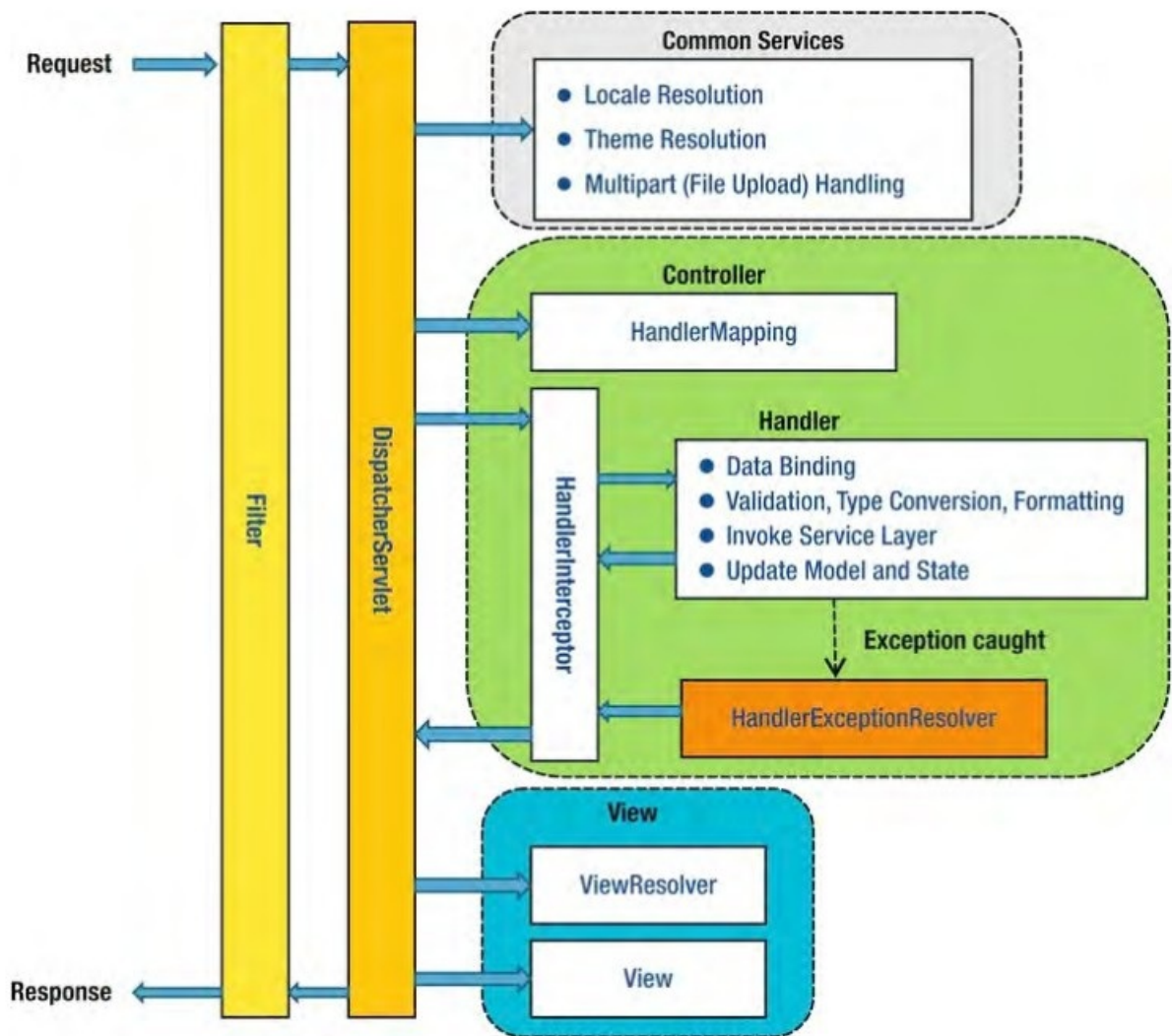
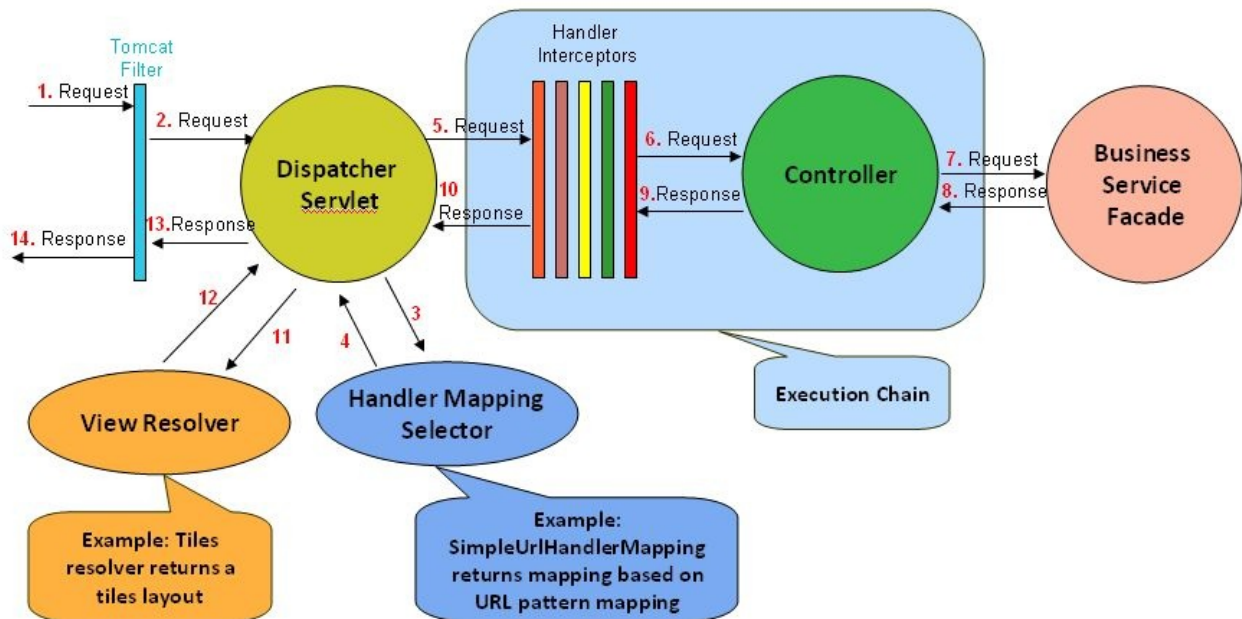


Figure 17-3. Spring MVC request life cycle

spring 처리 흐름



1.4 스프링 새로운 기능

1.4.1 Spring 3.1

1. Cache Abstraction

- Cache enable

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:cache="http://www.springframework.org/schema/cache"
xsi:schemaLocation="http://www.springframework.org/schem
a/beans http://www.springframework.org/schema/beans/spring-b
eans.xsd
        http://www.springframework.org/schema/cache http://
www.springframework.org/schema/cache/spring-cache.xsd">

    <cache:annotation-driven />
    ...
</beans>
```

- @Cacheable , @CacheEvict

```
@Cacheable("books")
public Book findBook(ISBN isbn) {...}

// use property 'rawNumber' on isbn argument as key
@Cacheable(value="book", key="#isbn.rawNumber")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

// cache only names shorter than 32 chars
@Cacheable(value="book", condition="#name.length < 32")
public Book findBook(String name)

// evict all cache entries
@CacheEvict(value = "books", allEntries=true)
public void loadBooks(InputStream batch)
```

2. Bean Definition Profiles

```
@Configuration
@Profile("production")
public class JndiDataConfig {

    @Bean
    public DataSource dataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}
```

3. Environment Abstraction

- datasource-config.xml

```
<beans profile="dev">
    <jdbc:embedded-database id="dataSource">
        <jdbc:script location="classpath:com/bank/config/s
ql/schema.sql"/>
        <jdbc:script location="classpath:com/bank/config/s
ql/test-data.sql"/>
    </jdbc:embedded-database>
</beans>
<beans profile="production">
    <jee:jndi-lookup id="dataSource" jndi-name="java:comp/
env/jdbc/datasource"/>
</beans>
```

- web.xml

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

- JVM option

```
-Dspring.profiles.active="production"
```

4. PropertySource Abstraction

```
<context:property-placeholder location="com/bank/config/data
source.properties"/>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicD
ataSource" destroy-method="close">
    <property name="driverClass" value="${database.driver}"
/>
    <property name="jdbcUrl" value="${database.url}"/>
    <property name="username" value="${database.username}"/>

    <property name="password" value="${database.password}"/>

</bean>
```

5. Code equivalents for Spring's XML namespaces

자주 사용하는 <context:component-scan/>, <tx:annotation-drive
n/>, <mvc:annotation-driven>의

스프링 XML 네임스페이스 엘리먼트와 동일한 기능을 대부분 @Enable 어노테
이션의 형식으로 사용할 수 있다.

이 기능은 스프링 3.0에서 도입된 @Configuration 클래스와 결합해서 사
용하도록 설계했다.

6. TestContext framework support for @Configuration classes and bean definition profiles

@ContextConfiguration 어노테이션은 이제 스프링 TestContext를 설정
하는 @Configuration 클래스를 제공한다.

새로 추가된 @ActiveProfiles 어노테이션은 ApplicationContext 통합
테스트에서 액티브 빈 선언 프로파일을
선언적으로 설정을 지원한다.

7. c: namespace for more concise constructor injection

```

<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:c="http://www.springframework.org/schema/c"
        xsi:schemaLocation="http://www.springframework.org/schema
/beans
        http://www.springframework.org/schema/beans/spring-be
ans.xsd">

    <bean id="bar" class="x.y.Bar"/>
    <bean id="baz" class="x.y.Baz"/>

    <!-- 'traditional' declaration -->
    <bean id="foo" class="x.y.Foo">
        <constructor-arg ref="bar"/>
        <constructor-arg ref="baz"/>
        <constructor-arg value="foo@bar.com"/>
    </bean>

    <!-- 'c-namespace' declaration -->
    <bean id="foo" class="x.y.Foo" c:bar-ref="bar" c:baz-ref=
"baz" c:email="foo@bar.com">

</beans>

```

8. Support for Servlet 3 code-based configuration of Servlet Container

- 전통적인 web.xml을 프로그래밍적으로 대체하는 서블릿 3.0의 ServletContainerInitializer에 기반을 둔 WebApplicationInitializer를 새로 추가했다

9. New HandlerMethod-based Support Classes For Annotated Controller Processing

- RequestMappingHandlerMapping <-- DefaultAnnotationHandlerMapping
- RequestMappingHandlerAdapter <-- AnnotationMethodHandlerAdapter
- ExceptionHandlerExceptionResolver <--- AnnotationMethodHandlerExceptionResolver

10. "consumes" and "produces" conditions in @RequestMapping

- 'Accept'헤더로 지정된 타입을 만드는 것(produce)과 마찬가지로 'Content-

Type'헤더로 지정된 미디어타입을 메서드로 소비(consume)하는 것에 대한 지원이 개선

11. Flash Attributes and RedirectAttributes
12. Support for injection against non-standard JavaBeans setters
13. JPA EntityManagerFactory bootstrapping without persistence.xml
14. Support for Servlet 3 MultipartResolver
15. Support for Hibernate 4.x
16. URI Template Variable Enhancements
17. @RequestPart Annotation On Controller Method Arguments
18. @Valid On @RequestBody Controller Method Arguments
19. UriComponentsBuilder and UriComponents
20. [3.1-release-notes](#) [참고](#)

1.4.2 Spring 3.2

1. Support for Servlet 3 based asynchronous request processing
2. Spring MVC Test framework
3. Content negotiation improvements
4. @ControllerAdvice annotation

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(CustomGenericException.class)
    public ModelAndView handleCustomException(CustomGeneric
Exception ex) {

        ModelAndView model = new ModelAndView("error/generi
c_error");
        model.addObject("errCode", ex.getErrCode());
        model.addObject("errMsg", ex.getErrMsg());
        return model;
    }

    @ExceptionHandler(Exception.class)
    public ModelAndView handleAllException(Exception ex) {

        ModelAndView model = new ModelAndView("error/generi
c_error");
        model.addObject("errMsg", "this is Exception.class"
);
        return model;
    }
}
```

5. Matrix variables

```
// GET /owners/42;q=11;r=12/pets/21;q=22;s=23

@RequestMapping(value = "/owners/{ownerId}/pets/{petId}")
public void findPet(
    @MatrixVariable Map<String, String> matrixVars) {

    // matrixVars: ["q" : [11,22], "r" : 12, "s" : 23]
}
```

6. Abstract base class for code-based Servlet 3+ container initialization

```
public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null;
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] { MyWebConfig.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

7. ResponseEntityExceptionHandler class

```
@ControllerAdvice
public class ResthubExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler(value={
        IllegalArgumentException.class,
        ValidationException.class,
        NotFoundException.class,
        NotImplementedException.class
    })
    public ResponseEntity<Object> handleCustomException(Exception ex, WebRequest request) {

        HttpHeaders headers = new HttpHeaders();
        HttpStatus status;

        if (ex instanceof IllegalArgumentException) {
            status = HttpStatus.BAD_REQUEST;
        }
    }
}
```

```
        } else if (ex instanceof ValidationException) {
            status = HttpStatus.BAD_REQUEST;
        } else if (ex instanceof NotFoundException) {
            status = HttpStatus.NOT_FOUND;
        } else if (ex instanceof NotImplementedException) {
            status = HttpStatus.NOT_IMPLEMENTED;
        } else {
            logger.warn("Unknown exception type: " + ex.getClass().getName());
            status = HttpStatus.INTERNAL_SERVER_ERROR;
            return handleExceptionInternal(ex, null, headers, status, request);
        }
        return handleExceptionInternal(ex, buildRestError(ex, status), headers, status, request);
    }

    private RestError buildRestError(Exception ex, HttpStatus status) {
        RestError.Builder builder = new RestError.Builder();
        ;
        builder.setCode(status.value()).setStatus(status.getReasonPhrase()).setThrowable(ex);
        return builder.build();
    }
}
```

8. Support for generic types in the RestTemplate and in @RequestBody arguments
9. Jackson JSON 2 and related improvements
10. Tiles 3
11. @RequestBody improvements
12. HTTP PATCH method
13. Excluded patterns in mapped interceptors
14. Using meta-annotations for injection points and for bean definition methods
15. Initial support for JCache 0.5
16. Support for @DateTimeFormat without Joda Time & Global date & time formatting

- 17. New Testing Features
- 18. Concurrency refinements across the framework
- 19. New Gradle-based build and move to GitHub
- 20. Refined Java SE 7 / OpenJDK 7 support
- 21. [3.2-release-notes](#) [참고](#)

1.4.3 Spring 4.0

- 1. Improved Getting Started Experience
- 2. Removed Deprecated Packages and Methods
- 3. Java 8 (as well as 6 and 7) & Java EE 6 and 7
- 4. Groovy Bean Definition DSL

```
def reader = new GroovyBeanDefinitionReader(myApplicationCon
text)
reader.beans {
    dataSource(BasicDataSource) {
        driverClassName = "org.hsqldb.jdbcDriver"
        url = "jdbc:hsqldb:mem:grailsDB"
        username = "sa"
        password = ""
        settings = [mynew:"setting"]
    }
    sessionFactory(SessionFactory) {
        dataSource = dataSource
    }
    myService(MyService) {
        nestedBean = { AnotherBean bean ->
            dataSource = dataSource
        }
    }
}
```

- 5. Core Container Improvements
- 6. General Web Improvements
- 7. WebSocket, SockJS, and STOMP Messaging
- 8. Testing Improvements

- Meta-Annotation Support for Testing

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
public @interface TransactionalDevTest { }

@RunWith(SpringJUnit4ClassRunner.class)
@TransactionalDevTest
public class OrderRepositoryTests { }
```

9. [4.0-release-notes](#) 참고

1.5 Sample Project

```
$ git clone git@github.daumkakao.com:adel-yang/SpringStudy.git
$ gradle jettyRun
```

Access <http://localhost:8080/>

Chapter2. 빈 와이어링(묵기)

2.1 스프링 설정 옵션 알아보기

- XML에서의 명시적 설정
- 자바에서의 명시적 설정
- 내재되어 있는 빈을 찾아 자동으로 와이어링 하기

2.2 자동으로 Bean 와이어링하기

- 컴포넌트 스캐닝 : @Component, @ComponentScan
- 오토 와이어링 : @Autowired

2.2.1 발견 가능한 빈 만들기

```
@Component
public class SgtPeppers implements CompactDisc {
    ...
}

@Configuration
@ComponentScan
public class CDPlayerConfig {
    ...
}
```

2.2.2 컴포넌트 스캔된 빈 명명하기

```
@Component("beanName") //Spring 종속적인 어노테이션
or
@Named("beanName")      //JSR-330의 어노테이션이며 Spring의 @Component, @Qualifier와 동일하다.
```

- 참고 : @Component는 어떤 스프링이 관리하는 컴포넌트를 나타내는 일반적인 스테레오 타입이다. 좀더 세부적인 유스 케이스들을 위하여 @Component의 구체화된 형태로 @Repository, @Service, @Controller들이 있다.
 - @Repository : 퍼시스턴스(persistence) 레이어, 영속성을 가지는 속성(파일, 데이터베이스 등)
 - @Service : 서비스 레이어 (비즈니스 레이어)
 - @Controller : spring mvc 기반의 프리젠테이션 레이어

2.2.3 컴포넌트 스캐닝을 위한 베이스 패키지 셋팅

```
@ComponentScan("com.kakao.game.helloworld.service")
or
@ComponentScan(basePackages = "com.kakao.game.helloworld.service"
)
or
@ComponentScan(basePackages = {"com.kakao.game.helloworld.servic
e", "com.kakao.game.helloworld.support"})
or
@ComponentScan(basePackageClasses = {CDPlayer.class, DVDPlayer.c
lass})
```

2.2.4 오토와이어링 되는 빈의 에너지테이션

- 생성자 주입


```
@Component
public class CDPlayer implements MediaPlayer {
    private CompactDisc cd;

    @Autowired
    public CDPlayer(CompactDisc cd) {
        this.cd = cd;
    }
    public void play() {
        cd.play();
    }
}
```

- property setter method

```
@Autowired
public void setCompactDisc(CompactDisc cd) {
    this.cd = cd;
}
```

- 일반 method
- @Autowired 과 @Inject 및 @Resource 비슷하지만 차이가 있음.
 - @Autowired : spring 전용, 타입에 맞춰 연결
 - @Inject : java(JSR)에서 지원, 타입에 맞춰 연결
 - @Resource : java(JSR)에서 지원, 이름으로 연결

```

@Autowired
private Chicken penguin; //Chicken 타입으로 연결됩니다.

@Inject
private Penguin chicken; //Penguin 타입으로 연결됩니다.

@Resource
private Chicken penguin; //penguin 타입으로 연결되지만, Chicken
클래스를 자료형으로 두었기에 캐스팅이 되지 않아 에러가 납니다

@Resource
private Bird penguin; //penguin 타입으로 연결되어 호출해보면 penguin
클래스의 값을 호출하는 것을 볼 수 있습니다.

@Autowired
@Qualifier("chicken")
private Bird penguin; //타입이나 이름에 상관없이 chicken타입으로
연결

```

2.2.5 자동 설정 검증하기

2.3 자바로 Bean 와이어링하기

2.3.1 설정 클래스 생성 : @Configuration

```

@Configuration
@ComponentScan(basePackageClasses = {CDPlayer.class, SgtPeppers.class})
public class CDPlayerConfig {
}

```

2.3.2 간단한 Bean 선언하기 : @Bean

```
@Configuration
public class CDPlayerConfig {
    @Bean
    public CompactDisc compactDisc() {
        return new SgtPeppers();
    }

    @Bean
    public CDPlayer cdPlayer(CompactDisc compactDisc) {
        return new CDPlayer(compactDisc);
    }
}
```

2.3.3 JavaConfig 주입

- 스프링의 모든 빈은 싱글톤으로 생성되므로, 서로 다른 빈에서 같은 인스턴스를 참조할 수 있다.

```
@Bean
public CDPlayer cdPlayer1() {
    return new CDPlayer(sgtPeppers());
}

@Bean
public CDPlayer cdPlayer2() {
    return new CDPlayer(sgtPeppers());
}

@Bean
public CDPlayer cdPlayer3(CompactDisc compactDisc) {
    return new CDPlayer(compactDisc);
}

@Bean
public CDPlayer cdPlayer4(CompactDisc compactDisc) {
    CDPlayer cdPlayer = new CDPlayer();
    cdPlayer.setCompactDisc(compactDisc);
    return cdPlayer;
}
```

2.4 빈을 XML로 와이어링하기

XML을 단독으로 사용할 때 특징

- 생성되는 모든 빈을 XML에서 확인 가능하다.
- 설정 정보를 코드로부터 분리할 수 있다.
- 스프링이 제공하는 모든 종류의 빈 설정 메타정보 항목을 지정할 수 있는 유일한 방법이다.
- 빈의 개수가 많아지면 관리하기 어렵다.

예제

1. XML 설정 생성하기

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
</beans>
```

2. 빈 등록하기

```
<bean id="compactDisc" class="soundsystem.SgtPeppers" />
```

3. 생성자 주입하기

- 추후에 argument의 이름이 바뀔 수 있으니 주의해야 한다.

1) constructor-arg

```

<bean id="cdPlayer" class="soundsystem.CDPlayer">
  <constructor-arg ref="compactDisc" /> // 빈
  <constructor-arg value="Sgt. Pepper's Lonely Hearts Club Band"
/> // 리터럴
  <constructor-arg> // 빈 컬렉션
    <list>
      <ref bean="sgtPeppers" />
      <ref bean="whiteAlbum" />
      <ref bean="hardDaysNight" />
      ...
    </list>
  </constructor-arg>
  <constructor-arg> // 리터럴 컬렉션
    <list>
      <value>Sgt. Pepper's Lonely Hearts Club Band</value>
      <value>With a Little Help from My Friends</value>
      <value>Lucy in the Sky with Diamonds</value>
      ...
    </list>
  </constructor-arg>

  <constructor-arg index="0" ref="compactDisc" /> // 순서
  <constructor-arg type="soundsystem.CompactDisc" ref="compactDisc" /> // 타입
  <constructor-arg name="compactDisc" ref="compactDisc" /> // 이름
</bean>

```

2) c-namespace

```
<bean id="cdPlayer" class="soundsystem.CDPlayer" c:cd-ref="compactDisc" /> // 이름으로 와이어링
<bean id="compactDisc"
      class="soundsystem.BlankDisc"
      c:_title="Sgt. Pepper's Lonely Hearts Club Band"
      c:_artist="The Beatles" />

<bean id="cdPlayer" class="soundsystem.CDPlayer" c:_0-ref="compactDisc" /> // 순서로 와이어링
<bean id="compactDisc"
      class="soundsystem.BlankDisc"
      c:_0="Sgt. Pepper's Lonely Hearts Club Band"
      c:_1="The Beatles" />

<bean id="cdPlayer" class="soundsystem.CDPlayer" c:_-ref="compactDisc" /> // 하나만 있을 때
<bean id="compactDisc"
      class="soundsystem.BlankDisc"
      c:_="Sgt. Pepper's Lonely Hearts Club Band" />
```

4. 프로퍼티 주입하기

1) property

```
<bean id="cdPlayer" class="soundsystem.properties.CDPlayer">
  <property name="compactDisc" ref="compactDisc" />

  <property name="title" value="Sgt. Pepper's Lonely Hearts Club
Band" />
  <property name="artist" value="The Beatles" />

  <property name="tracks">
    <list>
      <value>Sgt. Pepper's Lonely Hearts Club Band</value>
      <value>With a Little Help from My Friends</value>
      <value>Lucy in the Sky with Diamonds</value>
      ...
    </list>
  </property>
</bean>
```

2) p-namespace


```
<bean id="cdPlayer" class="soundsystem.properties.CDPlayer"
      p:compactDisc-ref="compactDisc" />

<bean id="compactDisc" class="sound system.BlankDisc">
  <p:title="Sgt. Pepper's Lonely Hearts Club Band" />
  <p:artist="The Beatles" />
</bean>

<util:list id="trackList">
  <value>Sgt. Pepper's Lonely Hearts Club Band</value>
  <value>With a Little Help from My Friends</value>
  <value>Lucy in the Sky with Diamonds</value>
  ...
</util:list>

<bean id="compactDisc"
      class="soundsystem.properties.BlankDisc"
      p:title="Sgt. Pepper's Lonely Hearts Club Band"
      p:artist="The Beatles"
      p:tracks-ref="trackList" />
```

5. 자동 와이어링

다음과 같은 클래스가 있다고 하자.

```
public class Hello {
  ...
  public Hello(String name, Printer printer) {
    this.name = name;
    this.printer = printer;
  }
}
```

1) 이름으로 자동 와이어링

```
<bean id="printer" class="...StringPrinter" />
<bean id="hello" class="...Hello" autowire="byName"> //
  <property name="name" value="Spring" />
  // <property name="printer" ref="printer" />가 생략되었다. 빈의 이
  름과 프로퍼티의 이름이 같기 때문에 자동와이어링이 된다.
</bean>
```

- 오타 조심

2) 타입으로 자동 와이어링

```
<bean id="mainPrinter" class="...StringPrinter" />
<bean id="hello" class="...Hello" autowire="byType">
  <property name="name" value="Spring" />
  // <property name="printer" ref="printer" />가 생략되었다. 빈의 타
  입과 프로퍼티의 타입이 같기 때문에 자동와이어링이 된다.
</bean>
```

- 타입이 같은 빈이 두 개 이상 존재하면 쓸 수 없다.
- 타입 비교가 이름 비교보다 느리다.

3) 여러 빈을 자동 와이어링

```
<beans default-autowire="byName"> // 이 아래 모든 빈은 이름으로 자동
와이어링된다.
  <bean>...</bean>
  ...
</beans>
```

생성자 주입 vs 프로퍼티 주입

- 프로퍼티 주입을 더 많이 쓰는 이유
 1. 역사적인 이유 - 초창기에 프로퍼티 주입에 좀 더 집중된 구조 - 때문에
 2. 프로퍼티 주입으로 더 많은 설정이 가능하기 때문에
- 유효한 객체를 만들기 위해서는 생성자 주입을 더 신경써주어야 한다.

2.5 설정 가져오기와 믹싱하기

1) JavaConfig끼리 참조하기

```
@Import({CDConfig.class, CDPlayerConfig.class})
```

2) XML끼리 참조하기

```
<import resource="cd-config.xml" />
```

3) JavaConfig로 XML 설정 참조하기

```
@ImportResource("classpath:cd-config.xml")
```

4) XML 설정에서 JavaConfig 참조하기

```
<bean class="soundsystem.CDConfig" />
```

Chapter3. 고급 와이어링

3.1 환경과 프로파일

- 개발환경별로 다른 데이터베이스 구성

```
//Develop
@Bean(destroyMethod = "shutdown")
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .addScript("classpath:schema.sql")
        .addScript("classpath:test-data.sql")
        .build();
}
```

```
//Production
@Bean
public DataSource dataSource() {
    JndiObjectFactoryBean jndiObjectFactoryBean = new JndiObjectFactoryBean();
    jndiObjectFactoryBean.setJndiName("jdbc/myDS");
    jndiObjectFactoryBean.setResourceRef(true);
    jndiObjectFactoryBean.setProxyInterface(javax.sql.DataSource.class);
    return (DataSource) jndiObjectFactoryBean.getObject();
}
```

```
//QA
@Bean(destroyMethod = "shutdown")
public DataSource dataSource() {
    BasicDataSource dataSource = new BasicDataSource();
    dataSource.setUrl("jdbc:h2:tcp://dbserver/~/test");
    dataSource.setDriverClassName("org.h2.Driver");
    dataSource.setUsername("sa");
    dataSource.setPassword("password");
    dataSource.setInitialSize(20);
    dataSource.setMaxActive(30);
    return dataSource;
}
```

- 각 환경에 맞게 빈 설정파일(클래스 또는 XML)을 달리해 빌드 타임때 해결할 순 있지만, 각 환경별로 설정파일을 만들어야 하므로 어려운 점이 있다.

3.1.1 빈 프로파일 설정하기

- 스프링은 빌드 시 빈 구성을 결정하지 않고 런타임때까지 기다린다
- `@Profile` 어노테이션으로 사용하고, 해당 프로파일이 활성화되어 있는 경우에만 해당 빈을 사용할 수 있다.
- 프로파일이 지정되지 않은 모든 빈은 항상 활성화된다.

```
@Configuration
public class DataSourceConfig {
    @Bean(destroyMethod = "shutdown")
    @Profile("dev")
    public DataSource embeddedDataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.H2)
            .addScript("classpath:schema.sql")
            .addScript("classpath:test-data.sql")
            .build();
    }
    @Bean
    @Profile("prod")
    public DataSource jndiDataSource() {
        JndiObjectFactoryBean jndiObjectFactoryBean = new JndiObjectFactoryBean();
        jndiObjectFactoryBean.setJndiName("jdbc/myDS");
        jndiObjectFactoryBean.setResourceRef(true);
        jndiObjectFactoryBean.setProxyInterface(javax.sql.DataSource.class);
        return (DataSource) jndiObjectFactoryBean.getObject();
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:
jdbc="http://www.springframework.org/schema/jdbc"
xmlns:jee="http://www.springframework.org/schema/jee" xmlns:p
="http://www.springframework.org/schema/p"
xsi:schemaLocation="
http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee.xsd
http://www.springframework.org/schema/jdbc
http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.x
sd">

<beans profile="dev">
  <jdbc:embedded-database id="dataSource" type="H2">
    <jdbc:script location="classpath:schema.sql" />
    <jdbc:script location="classpath:test-data.sql" />
  </jdbc:embedded-database>
</beans>

<beans profile="prod">
  <jee:jndi-lookup id="dataSource"
    lazy-init="true"
    jndi-name="jdbc/myDatabase"
    resource-ref="true"
    proxy-interface="javax.sql.DataSource" />
</beans>
</beans>
```

3.1.2 프로파일 활성화하기

- 프로파일 활성화 상태를 결정하는 두 가지 프로퍼티
 - spring.profiles.active
 - spring.profiles.default
 - active 프로파일 설정이 우선함

- 동시에 여러 프로파일을 쉽표로 구분하여 사용가능
- 활성화 프로퍼티를 설정하기 위한 여러가지 방법
 - DispatcherServlet에 초기화된 파라미터
 - 웹 어플리케이션의 컨텍스트 파라미터
 - JNDI 엔트리
 - 환경 변수
 - JVM 시스템 프로퍼티
 - 통합 테스트 클래스에서 `@ActiveProfiles` 사용
- `@ActiveProfiles` 어노테이션을 통합테스트코드 예제

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=DataSourceConfig.class)
@ActiveProfiles("dev")
public static class DevDataSourceTest {
    @Autowired
    private DataSource dataSource;

    @Test
    public void shouldBeEmbeddedDatasource() {
        assertNotNull(dataSource);
        JdbcTemplate jdbc = new JdbcTemplate(dataSource);
        List<String> results = jdbc.query("select id, name from
Things", new RowMapper<String>() {
            @Override
            public String mapRow(ResultSet rs, int rowNum) throws
SQLException {
                return rs.getLong("id") + ":" + rs.getString("name")
            }
        });

        assertEquals(1, results.size());
        assertEquals("1:A", results.get(0));
    }
}
```


3.2 조건부 빈

- 스프링 4.0에서는 @Bean을 적용할 수 있는 새로운 @Conditional 어노테이션이 소개되었다.

```
@Bean
@Conditional(MagicExistsCondition.class)
public MagicBean magicBean() {
    return new MagicBean();
}
```

```
public class MagicExistsCondition implements Condition {

    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        Environment env = context.getEnvironment();
        return env.containsProperty("magic");
    }
}
```

- ConditionalContext 수행절차
 - getRegistry()반환을 통한 BeanDefinitionRegistry로 빈 정의 확인
 - getBeanFactory()에서 반환되는 ConfigurableListableBeanFactory를 통해 빈 프로퍼티 발굴
 - getEnvironment()로부터 얻은 Environment를 통해 환경 변수 값 확인
 - getResourceLoader()에서 반환된 ResourceLoader를 통해 로드된 자원 내용을 읽고 검사
 - getClassLoader()에서 반환된 ClassLoader()를 통해 클래스의 존재를 로드하고 확인
- AnnotatedTypeMetadata는 @Bean 메서드의 어노테이션을 검사할 수 있는 기회를 제공한다.
- @Profile 어노테이션은 @Conditional 및 Condition 인터페이스에 기초하여 리팩토링된다.

```

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@Documented
@Conditional(ProfileCondition.class)
public @interface Profile {
    String[] value();
}

```

```

class ProfileRegexCondition implements Condition {

    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        Environment environment = context.getEnvironment();
        if (environment != null) {
            String[] profiles = environment.getActiveProfiles();

            MultiValueMap<String, Object> attrs = metadata.getAllAnnotationAttributes(ProfileRegex.class.getName());
            if (attrs != null) {
                for (Object value : attrs.get("value")) {
                    for (String profile : profiles) {
                        boolean matches = Pattern.matches((String) value, profile);
                        if (matches) {
                            return true;
                        }
                    }
                }
                return false;
            }
        }
        return true;
    }
}

```

3.3 오토와이어링의 모호성

- 오토와이어링은 응용 프로그램 구성 요소를 조립하는 데 필요한 명시적 설정의 양을 감소시킨다.
- 정확히 하나의 빈이 원하는 결과와 일치할 때 오토와이어링은 동작한다.

```
@Autowired
public void setDessert(Dessert dessert) {
    this.dessert = dessert;
}

@Component
public class Cake implements Dessert {...}

@Component
public class Cookies implements Dessert {...}

@Component
public class IceCream implements Dessert {...}
```

- 스프링은 단일 후보 선택을 좁히기 위해 한정자를 사용한다.

3.3.1 기본 빈 지정

- 기본 빈 지정 : @Primary 어노테이션

```
@Component
@Primary
public class IceCream implements Dessert {...}

@Bean
@Primary
public Dessert iceCream() {
    return new IceCream();
}
```

3.3.2 오토와이어링의 자격

- @Primary 한계점 : 하나의 명백한 옵션 선택을 하지 못한다

- 스프링의 수식은 결국 소정의 자격을 충족하는 모든 후보 빈에 적용되고, 단일 빈 대상으로 협소화 작업을 적용한다.

```
@Autowired
@Qualifier("iceCream") // "iceCream"은 주입할 빈의 ID
public void setDessert(Dessert dessert) {
    this.dessert = dessert;
}
```

- 클래스 이름을 변경하면 수식자는 잘못된 것으로 랜더링될 수 있다.
- 빈에 자신의 수식자를 지정할 수 있다.

```
@Component
@Qualifier("cold")
public class IceCream implements Dessert {...}

@Autowired
@Qualifier("cold")
public void setDessert(Dessert dessert) {
    this.dessert = dessert;
}

@Bean
@Qualifier("cold")
public Dessert iceCream() {
```

- 하지만 여전히 한정자가 "cold"인 디저트가 있다면 오토와이어링의 모호함에 직면한다
- 해결책은 양쪽의 주입지점과 빈 정의에 다른 @Qualifier를 고정시키는 방법을 사용하는 것이다.

```

@Component
@Qualifier("cold")
@Qualifier("creamy")
public class IceCream implements Dessert {...}

@Component
@Qualifier("cold")
@Qualifier("fruity")
public class Popsicle implements Dessert {...}

@Autowired
@Qualifier("cold")
@Qualifier("creamy")
public void setDessert(Dessert dessert) {
    this.dessert = dessert;
}

```

- 하지만, 자바는 동일한 유형의 여러 어노테이션이 같은 항목에 반복 될 수 없다.
- 그래서 사용자 지정 수식자 어노테이션을 사용한다.
- 맞춤형 어노테이션을 사용하는 것은 `@Qualifier` 어노테이션을 사용하는 것과 문자열로 수식을 지정하는 것보다 더 `type-safe`하다

```

@Target({ElementType.CONSTRUCTOR, ElementType.FIELD
        , ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Cold {}

```

```

@Target({ElementType.CONSTRUCTOR, ElementType.FIELD
        , ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Creamy {}

```

```

@Component
@Cold
@Creamy
public class IceCream implements Dessert {...}

@Component
@Cold
@Fruity
public class Popsicle implements Dessert {...}

@Autowired
@Cold
@Creamy
public void setDessert(Dessert dessert) {
    this.dessert = dessert;
}

```

3.4 빈 범위

기본적으로 스프링의 생성되는 빈은 싱글톤이다.

- 싱글톤(Singleton) : 전체 애플리케이션을 위해 생성되는 빈의 인스턴스 하나
- 프로토타입(Prototype) : 빈이 주입될 때마다 생성되거나 스프링 애플리케이션 컨텍스트에서 얻는 빈의 인스턴스 하나
- 세션(Session) : 웹 애플리케이션에서 각 세션용으로 생성되는 빈의 인스턴스 하나
- 요청(Request) : 웹 애플리케이션에서 각 요청요으로 생성되는 빈의 인스턴스 하나

```

@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class Notepad {
}

```

또는 `@Scope("prototype")` 을 사용할 수 있지만 TypeSafe 하지 않으므로 권장하지 않는다.

자바빈 설정은 다음처럼

```
@Bean
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public Notepad notepad() {
}
```

xml 은 다음처럼

```
<bean id="notepad" class="com.myapp.Notepad" scope="prototype" />
```

3.4.1 요청과 세션 범위 작업하기

```
@Component
@Scope(value=WebApplicationContext.SCOPE_SESSION, proxyMode=Scop
edProxyMode.INTERFACES)
public ShoppingCart cart() {
}
```

세션 내 에서 싱글톤임 범위 프록시는 요청과세션 범위 빈 주입을 연기한다.

3.4.2 xml 로 범위 프록시 선언하기

```
<bean id="cart" class="com.mysqpp.ShoppingCart" scope="session">
  <aop:scoped-proxy />
</bean>
```

기본적으로 타킷 클래스 프록시를 생성하기 위해 cglib 을 사용한다. 아래와 같이 하면 인터페이스 기반의 프락시를 사용한다.

```
<bean id="cart" class="com.mysqpp.ShoppingCart" scope="session">
  <aop:scoped-proxy proxy-target-class="false" />
</bean>
```

3.4.3 범위빈을 런타임시에 생성하기

스프링에 어노테이션은 범위빈 사용을 빈주입시에만 가능하다. 만약 런타임시마다 빈을 받고 싶다면 JavaEE6 의 `@Inject` 와 `javax.inject.Provider` 가 필요하다.

```
@Autowired
private Client client; // 빈 인젝션에서만 사용됨

@Inject
private Provider<Client> clientProvider; // 팩토리를 주입

public void useClient() {
    clientProvider.get().doSomething();
}
```

3.5 런타임 값 주입

- 프로퍼티 플레이스 홀더(Property placeholders)
- 스프링 표현 언어(SpEL, Spring Expression Language)

3.5.1 외부 값 주입

```
@Configuration
@PropertySource("classpath:/com/soundsystem/app.properties")
public class ExpressiveConfig {
    @Autowired
    Environment env;

    @Bean
    public BlankDisk disk() {
        return new BlankDisc(env.getProperty("disc.title"), env.getProperty("disc.artist"));
    }
}
```

app.properties


```
disc.title=Sgt. Peppers..
disc.arties=The Beatles
```

getProperty 오버로딩

- `String getProperty(String key)`
- `String getProperty(String key, String defaultValue)`
- `T getProperty(String key, Class<T> type)`
- `T getProperty(String key, class<T> type, T defaultValue)`

```
int connectionCount = env.getProperty("db.connection.count", Integer.class, 30);
```

프로퍼티 조회시에 값이 없을 경우 null 을 리턴한다. 강제하고 싶다면

`getRequiredProperty()` 를 사용한다.

프로퍼티가 정의되지 않으면 `IllegalStateException` 이 발생한다. 프로퍼티 존재를 확인해야할 경우 `containsProperty()` 를 호출한다.

```
Class<CompactDisc> cdClass = env.getPropertyasClass("dis.class", CompactDisc.class);
```

Environment 는 프로파일이 활성화되어있는지를 확인하는 몇가지 메소드를 제공한다.

- `String[] getActiveProfiles()` : 활성화된 프로파일 명 배열을 리턴
- `String[] getDefaultProfiles()` : 기본 프로파일 명 배열을 반환
- `boolean acceptProfiles(String... profiles)` : 환경에 주어진 프로파일을 지원하면 true 리턴

프로퍼티 플레이스홀더 처리하기

프로퍼티 와이어링(?)

```
<bean id="sgtPapers" class="soundssystem.BalnkDisc" c:_title="${disk.title}" c:_artist="${disc.artist}" />
```

```
public BalnkDisc(@Value("${disk.title}") String title, @Value("${disc.artist}")String artist) {

}
설정하기
```

PropertyPlaceholderConfiguraer 빈 또는
PropertySourcePlaceholderConfigurer 빈을 설정한다.

```
@Bean
public static PropertySourcesPlaceholderConfigurer placeholderCo
nfigurer() {
    return new PropertySourcesPlaceHolderConfigurer();
}
```

```
<context:property-placeholder />
```

3.5.2 스프링 표현식 와이어링

스프링3부터 SpEL 을 이용해 런타임시 주입하기 위한 값을 계산할 수 있다.

- ID 로 빈을 참조하는 기능
- 메소드 호출과 객체의 프로퍼티 액세스
- 값에서의 수학적인 동작, 관계와 관련된 동작, 논리연산 동작
- 정규 표현식 매칭
- 컬렉션 처리

SpEL 은 종속객체 주입보다는 다른 용도로 사용된다. 예를 들면 스프링 시큐리티는 SpEL 표현식을 포함하는 보안을 정의하는데 도움이된다. 스프링 MVC 애플리케이션의 뷰처럼 Tnymeleaf 템플릿승ㄴ 사용한다면 이 템플릿은 모델 데이터를 참조하기 위해 SpEL 표현식을 사용한다.

SpEL 의 표현식 : `#{ ... }` 형식, 프로퍼티 플레이스 홀더 : `${ ... }`

- `#{1}` : 상수 1
- `#{T(System).currentTimeMillis()}` : T 연산자는 타입을 평가하고, 메서드

를 수행

- `{sgtPeppers.artist}` : 빈 참조
- `{systemProperties['disc.title']}` : `systemProperties` 객체를 통해 시스템 프로퍼티 참조

빈 와이어링

```
public BalnkDisc(
    @Value("#{systemProperties['disk.title']}")String title,
    @Value("#{systemProperties['disk.artist']}")String artist
)
```

```
<bean id="sgtPapers" class="soundsystem.BalnkDisc" c:_title="#{systemProperties['disk.title']}" c:_artist="#{systemProperties['disc.artist']}" />
```

- `#{3.14159}`
- `#{9.87E4}`
- `#{'Hello'}`
- `#{false}`
- `#{artistSelector.selectArtist()}`
- `#{artistSelector.selectArtist().toUpperCase()}`
- `#{artistSelector.selectArtist()?.toUpperCase()}` : `NullPointerException` 방지
- `#{2 * T(java.lang.Math).PI * circle.radius}`
- `#{disc.title + ' by ' + dist.artist}`
- `#{counter.total == 100}`
- `#{scoreBoard.score > 1000 ? "winnder!" : "Looser"}`
- `#{disk.title ?: 'i am null'}`
- `#{admin.email matches '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.\com'}`
- `#{jukebox.songs[4].title}`
- `#{'this is a test'[3]}` : 문자열에서 한글자 추출
- `#{jukebox.songs.[artist eq 'Aerosmith']}` : 컬렉션에서 특정 프로퍼티가 일치하는 요소 찾기
- `#{jukebox.songs.^[artist eq 'Aerosmith']}`

- `{jukebox.songs.$[artist eq 'Aerosmith']}`
- `{jukebox.songs.![title]}` : 요소의 프로퍼티로 새로운 컬렉션 생성
- `{jukebox.songs.?[artist eq 'Aerosmith'].![title]}`

Operator

연산 유형	연산자
산술	+, -, *, /, %, ^
비교	<, lt, >, gt, ==, eq, <=, le, >=, ge
논리	and, or, not,
조건	?: (ternary), ?:(elvis)
정규표현식	matches

Chapter4. 애스팩트 지향 스프링

ex) 전략소비량 모니터링

횡단관심사 (cross-cutting concerns)

한 애플리케이션의 여러 부분에 걸쳐있는 기능

보통은 애플리케이션의 비즈니스 로직과 개념적으로 분리된다.

- 예: 로깅, 트랜잭션, 보안, 캐싱

애스팩트 지향 프로그래밍(AOP)

횡단관심사의 분리를 위한 것이다.

4.1 AOP란 무엇인가?

AOP(Aspect Oriented Programming)

주 목적은 횡단관심사의 모듈화에 있다.

애스팩트(Aspect) AOP에서 횡단 관심사를 애스팩트(Aspect)라는 특별한 클래스로 모듈화 한다.

애스팩트 vs 상속,위임

공통 기능을 재사용하기 위한 방법이라는 점에서는 동일하지만, 애스팩트는 상속이나 위임보다 더 깔끔한 해결책을 제공한다.

- 상속: 객체의 정적 구조에 의존하므로 복잡하고 깨지기 쉬운 구조가 되기 쉽다.
- 위임: 대상 객체에 대한 복잡한 호출로 인해 번거롭다.

애스팩트는 대상 클래스를 전혀 수정할 필요가 없다는 점이 큰 차이점이다.
상속이나 위임만으로는 불가능한 모듈화가 가능하다.

애스펙트 장점

1. 전체 코드 기반에 흩어져 있는 관심사항이 하나의 장소로 응집된다.
2. 서비스 모듈이 자신의 주요 관심사항(핵심기능)에 대한 코드만 포함한다. (코드가 깔끔해진다.)

4.1.1 AOP 용어 정의

어떤 애스펙트의 기능(어드바이스)은 하나 이상의 조인포인트 지점을 통해 프로그램의 실행(execution)에 위빙(weaving)된다.

어드바이스(advice)

애스펙트가 '무엇'을 '언제' 할지를 정의한다.

애스펙트가 해야 하는 작업과 언제 그 작업을 수행해야 하는지를 정의한 것이다.

어드바이스 종류	설명
이전(before)	어드바이스 대상 메소드가 호출되기 전에 어드바이스 기능을 수행한다.
이후(after)	결과에 상관없이 어드바이스 대상 메소드가 완료된 후에 어드바이스 기능을 수행한다.
반환이후(after-returning)	어드바이스 대상 메소드가 성공적으로 완료된 후에 어드바이스 기능을 수행한다.
예외발생이후(after-throwing)	어드비아스 대상 메소드가 예외를 던진 후에 어드바이스 기능을 수행한다.
주위(around)	어드바이스가 어드바이스 대상 메소드를 감싸서 어드바이스 대상 메소드 호출 전과후에 몇가지 기능을 제공한다.

조인포인트(join point)

어드바이스를 적용할 수 있는 곳

애스펙트를 끼워 넣을 수 있는 지점을 말한다.

- 예: 메소드 호출 지점, 예외 발생, 필드값 수정 등

포인트컷(pointcut)

애스펙트가 어드바이스를 '어디서' 할지를 정의한다.

애스펙트가 어드바이스할 조인포인트의 영역을 좁히는 역할을 한다.

어드바이스가 위빙(weaving)되어야 하는 하나 이상의 조인포인트를 정의한다.

- 예: 클래스나 메소드명, 정규표현식, 동적 포인트컷(싱הל중에 얻는 정보 이용) 등

애스펙트(aspect)

어드바이스와 포인트컷을 합친 것

어드바이스와 포인트컷을 합치면 애스펙트가 무엇을 언제 어디서 할지 필요한 정보가 모두 정의된다.

인트로덕션(introduction)

기존 클래스에 코드 변경 없이 새메소드나 멤버변수를 추가하는 기능이다.

위빙(weaving)

타깃 객체에 애스펙트를 적용해서 새로운 프록시 객체를 생성하는 절차이다.

애스펙트는 타깃 객체의 조인포인트로 위빙된다.

위빙시점	설명
컴파일시간 (compile time)	타깃 클래스가 컴파일 될 때 애스펙트가 위빙되며, 별도의 컴파일러가 필요하다. 예: AspectJ 위빙 컴파일러
클래스로 드시간 (classload time)	클래스가 JVM에 로드될 때 애스펙트가 위빙된다. 이렇게 하려면 애플리케이션에서 사용되기 전에 타깃 클래스의 바이트 코드를 enhance하는 특별한 ClassLoader가 필요하다. 예: AspectJ5의 로드시간위빙 기능
실행시간 (runtime)	애플리케이션 실행 중에 애스펙트가 위빙된다. 보통 타깃 객체에 호출을 위임하는 구조의 프록시 객체를 위빙중에 AOP 컨테이너가 동적으로 만들어낸다. 예: 스프링 AOP 애스펙트 위빙 방식

4.1.2 스프링의 AOP 지원

AOP 프레임워크마다 제공하는 조인포인트 모델 수준이 다르고, 애스펙트를 언제 어떻게 위빙하는지도 다르다.

그러나, 어드바이스를 어느 조인포인트에 위빙할지 정의하는 포인트컷을 생성한다는 개념은 같다.

스프링 AOP 지원

스프링에서 지원되던 AOP는 AspectJ프로젝트에서 많은 것을 가져왔다.

1. 고전적인(classic) 스프링 프록시 기반 AOP -> 너무 무겁고 지나치게 복잡하므로 여기서는 다루지 않는다.
2. Pure-POJO 애스펙트 -> 스프링의 aop 네임스페이스를 사용하여 POJO에서 애스펙트로 전환할 수 있다. XML 설정이 필요하지만 객체에서 애스펙트로 전환하는 가장 명확한 방법이다.
3. @AspectJ 애너테이션 기반 애스펙트 -> 스프링은 프록시 기반 AOP이지만 프로그래밍 모델에서 AspectJ로 애너테이션된 애스펙트를 사용할 수 있다. XML 설정이 필요없다.
4. AspectJ 애스펙트에 빈 주입 (스프링 모든버전에서 지원)

1,2,3번은 스프링 AOP 구현체에서 파생된 것이므로 동적 프록시 기반으로 만들어진다. 스프링의 AOP는 메소드 가로채기(interception)로만 제한된다.

메소드 가로채기 이상(생성자 또는 멤버변수에 대한 가로채기)의 능력이 필요하다면, 4번 방식인 AspectJ를 이용하여 애스펙트를 구현해야 한다.

스프링 어드바이스는 자바로 작성

스프링에서 생성하는 모든 어드바이스는 표준 자바 클래스로 작성한다.

포인트컷트는 보통 스프링 XML 설정 파일에 정의하게 된다.

AspectJ는 자바 언어를 확장한 형태로 구현되어 있다.

AOP를 위한 특별한 언어를 갖게 되어 더 강력하고 세밀한 제어가 가능하며 풍부한 AOP 도구모음을 제공한다.

하지만 새로운 도구와 문법을 배워야만 한다는 부담도 존재한다.

실행 시간에 만드는 스프링 어드바이스

스프링에서는 빈을 감싸는 프록시 객체를 실행시간에 생성함으로써 애스펙트가 스프링 관리 빈에 위빙된다.

- 프록시 객체는 타깃 객체로 위장해서 어드바이스 대상 메소드의 호출을 가로챈 후 타깃 객체로 호출을 전달(forward)
- 애스펙트의 로직은 프록시가 메소드 호출을 가로채고나서 실행되며, 그 다음에 타

깃 빈의 메소드가 호출된다.

스프링은 애플리케이션이 프록시 타깃 빈을 실제 필요로 할 때까지 프록시 타깃 객체를 생성하지 않는다.

스프링은 런타임시에 프록시를 생성하므로 위빙을 위한 별도의 컴파일러가 필요하지 않다.

스프링은 메소드 조인포인트만 지원

스프링은 동적 프록시를 기반으로 AOP를 구현하므로 메소드 조인포인트만 지원한다.

따라서, 객체 필드 값의 수정이나 객체 인스턴스화에 어드바이스를 적용할 수 없다.

그러나 메소드 조인포인트만으로도 필요한 대부분이 충족된다.

- AspectJ나 JBoss는 필드나 생성자 조인포인트까지 제공한다.

4.2 포인트커트를 이용한 조인포인트 선택

스프링 AOP에서 포인트커트는 AspectJ의 포인트커트 표현식 언어를 이용해 정의된다.

스프링은 AspectJ에서 사용할 수 있는 포인트커트 지정자(designator)에 속하는 것만 지원한다.

스프링에서 지원되는 **AspectJ의 포인트커트 표현식 언어**

AspectJ 지정자	설명
args()	인자가 주어진 타입의 인스턴스인 조인포인트 매칭을 정의
@args()	전달된 인자의 런타임 타입이 주어진 타입의 애너테이션을 갖는 조인포인트 매칭을 정의
execution()	메소드 실행 조인포인트와 일치시키는데 사용
this()	AOP 프록시의 빈 레퍼런스가 주어진 타입의 인스턴스를 갖는 조인포인트를 정의
target()	대상 객체가 주어진 타입을 갖는 조인포인트를 정의
@target()	수행 중인 객체의 클래스가 주어진 타입의 애너테이션을 갖는 조인포인트를 정의
within()	특정 타입에 속하는 조인포인트를 정의
@within()	주어진 애너테이션을 갖는 타입 내 조인포인트를 정의 (스프링 AOP를 사용할 때 주어진 애너테이션을 사용하는 타입으로 선언된 메소드 실행)
@annotation	조인포인트의 대상 객체가 주어진 애너테이션을 갖는 조인포인트를 정의

AspectJ의 다른 지정자를 사용하면 `IllegalArgumentException` 발생한다.

`execution` 지정자만 실제로 일치시키는 작업을 수행하고 나머지는 일치를 제한하는데 사용된다.

따라서, `execution` 지정자가 포인트컷의 기본 지정자이다.

4.2.1 포인트컷 작성

포인트컷 대상 정의

```
package concert;

public interface Performance {
    public void perform();
}
```

메소드 실행시 어드바이스하는 포인트컷 표현식

`perform` 메소드가 실행될 때마다 어드바이스를 하기 위한 포인트컷 표현식

- execution 지정자: Performance.perform 메소드를 선택
- *: 메소드 반환 타입이 무엇이든 상관없음
- (..): 인자 목록이 무엇이든지 간에 perform 메소드를 선택

```
execution(* concert.Performace.perform(..))
```

포인트컷트 범위 제한

포인트컷트 범위를 concert 패키지로만 제한한다.

- within 지정자: 포인트컷트의 범위를 제한
- && 연산자: execution과 within 지정자를 and 관계로 결합시킨다.
- || 연산자: or 관계를 나타낼 수 있다.
- ! 연산자: 지정자의 영향을 부정하는데 사용한다.
- XML 기반의 설정에서 포인트컷트를 지정할때는 && 대신 and를 사용한다. 마찬가지로, || 대신 or, ! 대신 not을 사용한다. (XML에서는 &이 특별한 의미가 있기때문)

```
execution(* concert.Performace.perform(..) && within(concert.*))
```

4.2.2 포인트컷트에서 빈 선택하기

bean() 지정자

포인트컷트 표현식 내에서 ID로 빈을 지정할 수 있다.

인자로 빈ID나 이름을 받고 특정 빈에 대한 포인트컷트의 영향을 제한한다.

특정 빈으로 포인트컷트를 좁히는 경우도 있지만, 특정 ID가 아닌 모든 빈에 적용하기 위해 부정(!)을 사용할 수도 있다.

ex1) Performance에 있는 perform() 메소드의 실행시 ID가 woodstock인 빈에 제한하여 어드바이스를 적용한다.

```
execution(* concert.Performace.perform(..) and bean('woodstock'))
```

ex2) 애스팩트의 어드바이스가 ID가 woodstock이 아닌 모든 빈에 위빙된다.

```
execution(* concert.Performace.perform(..) and !bean('woodstock'
))
```

4.3 애스펙트 애너테이션 만들기

애너테이션을 사용하여 애스펙트를 만들 수 있는 기능. (@AspectJ)

어드바이스: @After, @AfterReturning, @AfterThrowing, @Around, @Before

포인트컷: @Pointcut

오토프록싱: @EnableAspectJAutoProxy

인트로덕션: @DeclareParents

4.3.1 애스펙트 정의하기

```
// 애스펙트로 선언
@Aspect
public class Audience {
    // 포인트컷 execution(...)
    // 어드바이스 Before, silenceCellPhone() {...}
    @Before("execution(** concert.Performance.perform(..))")
    public void silenceCellPhone() {
        System.out.println("Silencing cell phones");
    }

    ...
}
```

```

@Configuration
// AspectJ 오토-프록싱 활성화
@EnableAspectJAutoProxy
@ComponentScan
public class ConcertConfig {
    @Bean
    public Audience audience() {
        return new Audience();
    }
}

```

4.3.2 around 어드바이스 만들기

```

@Around("performance()")
public void watchPerformance(ProceedingJoinPoint jp) {
    try {
        System.out.println("Silencing cell phones");
        System.out.println("Taking seats");
        // 반드시 proceed()를 호출해야 한다.
        jp.proceed();
        System.out.println("CLAP CLAP CLAP!!!");
    } catch (Throwable throwable) {
        System.out.println("Demanding a refund");
    }
}

```

4.3.3 어드바이스에서 파라미터 처리하기

```

// playTrack(int) int 인자
// args(trackNumber) 인자 스펙
@PointCut("execution(* soundsystem.CompactDisc.playTrack(int)) &
& args(trackNumber)")
public void trackPlayed(int trackNumber) {}

@Before("trackPlayed(trackNumber)")
public void countTrack(int trackNumber)

```

4.3.4 인트로덕션 애너테이션

```
@Aspect
public class EncoreableIntroducer {

    // value: 적용할 대상, Performance+ Performance 의 서브타입.
    // defaultImpl: 인트로덕션 구현체를 제공하는 클래스, DefaultEncoreable.class
    @DeclareParents(value="concert.Performance+", defaultImpl=DefaultEncoreable.class)
    public static Encoreable encoreable;
}
```

4.4 XML에서 애스펙트 선언하기

4.4.1 before 어드바이스와 after 어드바이스 선언하기

```
<aop:config>
  <!-- 빈 참조 -->
  <aop:aspect ref="audience">
    <!-- before 어드바이스 -->
    <aop:before
      <!-- 포인트컷 -->
      pointcut="execution(** concert.Performance.perform(..))"
      method="silenceCellPhones"/>
    ...
  </aop:aspect>
</aop:config>
```

4.4.2 around 어드바이스 선언

```
<aop:config>
  <aop:aspect ref="audience">
    <!-- around 어드바이스 -->
    <aop:pointcut
      id="performance"
      expression="execution(** concert.Performance.perform(..))"
    />
    <aop:around
      pointcut="performance"
      method="watchPerformance"/>
    ...
  </aop:aspect>
</aop:config>
```

4.4.3 어드바이스에 파라미터 전달

```
<aop:config>
  <aop:aspect ref="trackCounter">
    <!-- around 어드바이스 -->
    <aop:pointcut
      id="trackPlayed"
      expression="execution(* soundsystem.CompactDisc.playTrack(
int)) and args(trackNumber)"/>
    <aop:before
      pointcut-ref="trackPlayed"
      method="countTrack"/>
  </aop:aspect>
</aop:config>
```

4.4.4 애스펙트를 이용한 새로운 기능 도입

```

<aop:config>
  <aop:declare-parents
    types-matching="concert.Performance+"
    implement-interface="concert.Encoreable"
    default-impl="concert.DefaultEncoreable"
  />
</aop:config>

```

4.5 AspectJ 애스팩트 주입

Aspect 정의

```

public aspect CriticAspect {
    pointcut performance() : execution(* perform(..));

    after() : returning() : performance() {
        System.out.println(criticismEngine.getCriticism());
    }
    ...
}

```

CriticAspect - CriticismEngineImpl 와이어링.

```

<bean id="criticismEngine"
  class="com.springinaction.springidol.CriticismEngineImpl">
  <property name="criticisms">
    <list> ... </list>
  </property>
</bean>

<bean class="com.springinaction.springidol.CriticAspect"
  factory-method="aspectOf">
  <property name="criticEngine" ref="criticismEngine" />
</bean>

```

4.6 요약

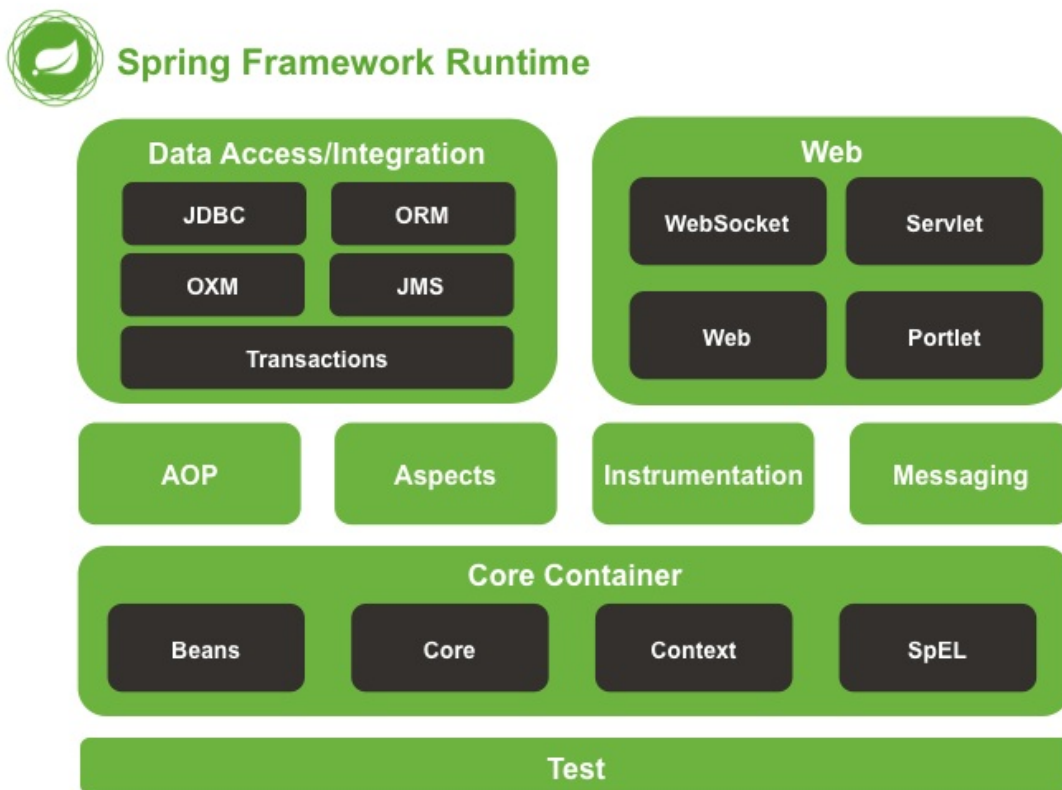
Github: <https://github.com/storyzero/SpringInAction>

Chapter5. 스프링 웹 애플리케이션 만들기

5.1 스프링 MVC 시작하기

Spring MVC

- MVC 패턴을 기반으로 유연한 웹 기반 어플리케이션 개발 모듈

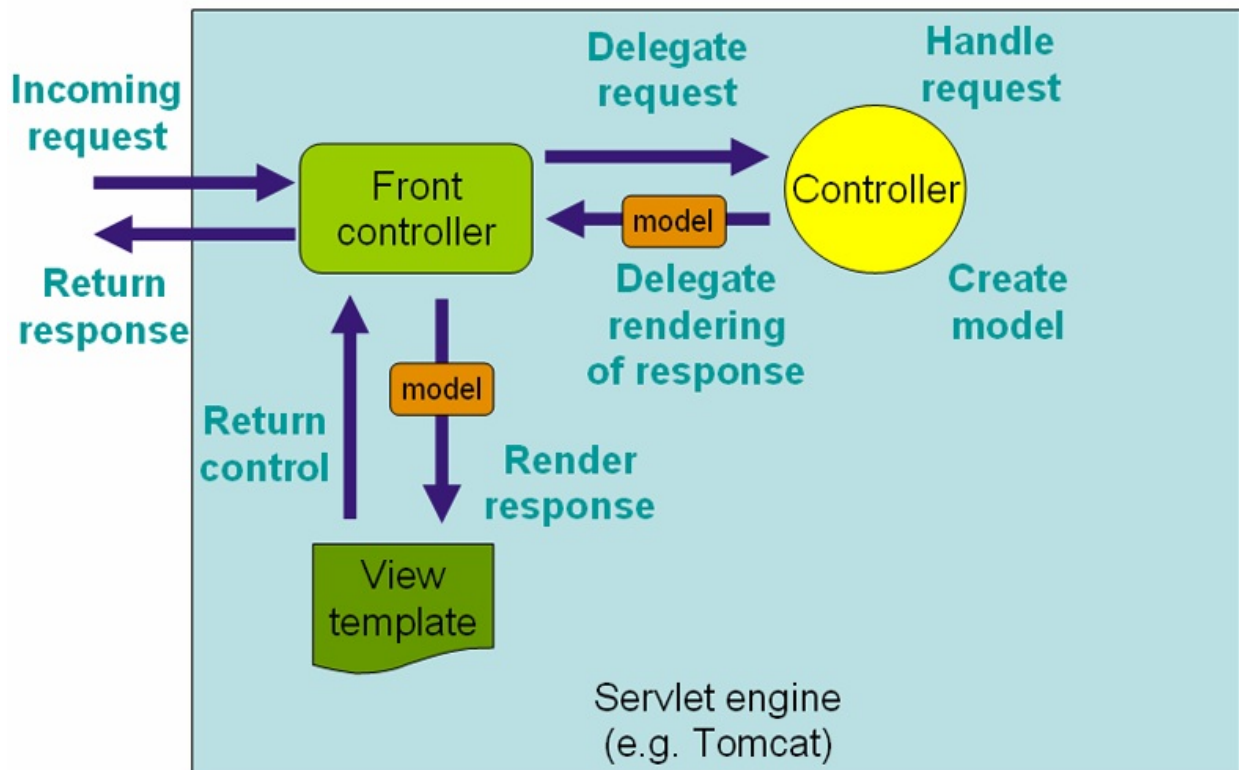


The Spring Web model-view-controller (MVC) framework is designed around a **DispatcherServlet** that dispatches requests to handlers, with configurable handler mappings, view resolution, locale, time zone and theme resolution as well as support for uploading files.

"Open for extension..." A key design principle in Spring Web MVC and in Spring in general is the "Open for extension, closed for modification" principle.

5.1.1 스프링 MVC를 이용한 요청 추적

"Front Controller" design pattern



Spring's web MVC framework is, like many other web MVC frameworks, request-driven, designed around a **central Servlet** that **dispatches** requests to controllers and offers other functionality that facilitates the development of web applications.

1. 요청이 전달되어 스프링의 DispatcherServlet(Front Controller)에 도착.
2. DispatcherServlet이 Handler Mapping을 참조하여 전달할 Controller를 선택.
3. DispatcherServlet은 선택한 컨트롤러에게 요청을 전달.(Service -> Repo -> ...)
4. Controller에서 처리된 결과를 사용자에게 전달해주기 위한 정보(Model)과 View 이름을 DispatcherServlet에게 되돌려줌.
5. DispatcherServlet는 View Resolver를 참조하여 View Mapping 정보를 획득.
6. DispatcherServlet는 렌더링하기 위한 View에 Model 데이터 전달.
7. View 렌더링 후, 응답객체에 의해 전달.

5.1.2 스프링 MVC 설정하기

1.DispatcherServlet 설정하기

XML(web.xml)

```
<web-app>
  <servlet>
    <servlet-name>example</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>example</servlet-name>
    <url-pattern>/example/*</url-pattern>
  </servlet-mapping>

</web-app>
```

Java Config

```
public class MyWebApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext container) {
        ServletRegistration.Dynamic registration = container.addServlet("dispatcher", new DispatcherServlet());
        registration.setLoadOnStartup(1);
        registration.addMapping("/example/*");
    }

}
```

WebApplicationInitializer is an interface provided by Spring MVC that ensures your code-based configuration is detected and automatically used to initialize any **Servlet 3 container**. An abstract base class implementation of **WebApplicationInitializer** named **AbstractDispatcherServletInitializer** makes it even **easier** to register the DispatcherServlet

- SpringBootServletInitializer/AbstractDispatcherServletInitializer/AbstractAnnotationConfigDispatcherServletInitializer implements **WebApplicationInitializer**

DispatcherServlet 설정

1. WebApplicationInitializer 직접 구현
2. AbstractDispatcherServletInitializer/AbstractAnnotationConfigDispatcherServletInitializer 상속

AbstractAnnotationConfigDispatcherServletInitializer

```
public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null;
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] { MyWebConfig.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }

}
```

AbstractDispatcherServletInitializer

```
public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

    @Override
    protected WebApplicationContext createRootApplicationContext() {
        return null;
    }

    @Override
    protected WebApplicationContext createServletApplicationContext() {
        XmlWebApplicationContext cxt = new XmlWebApplicationContext();
        cxt.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");
        return cxt;
    }

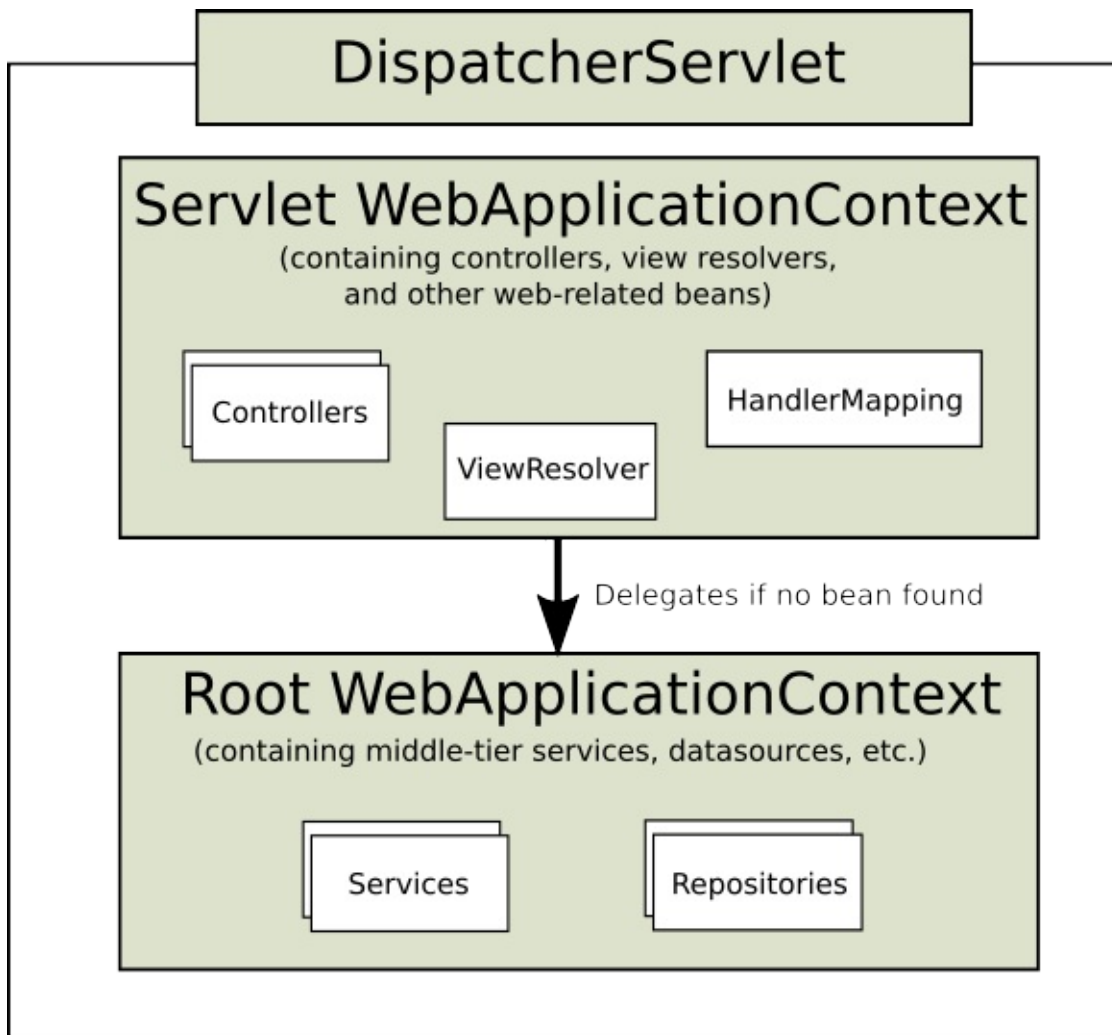
    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }

}
```

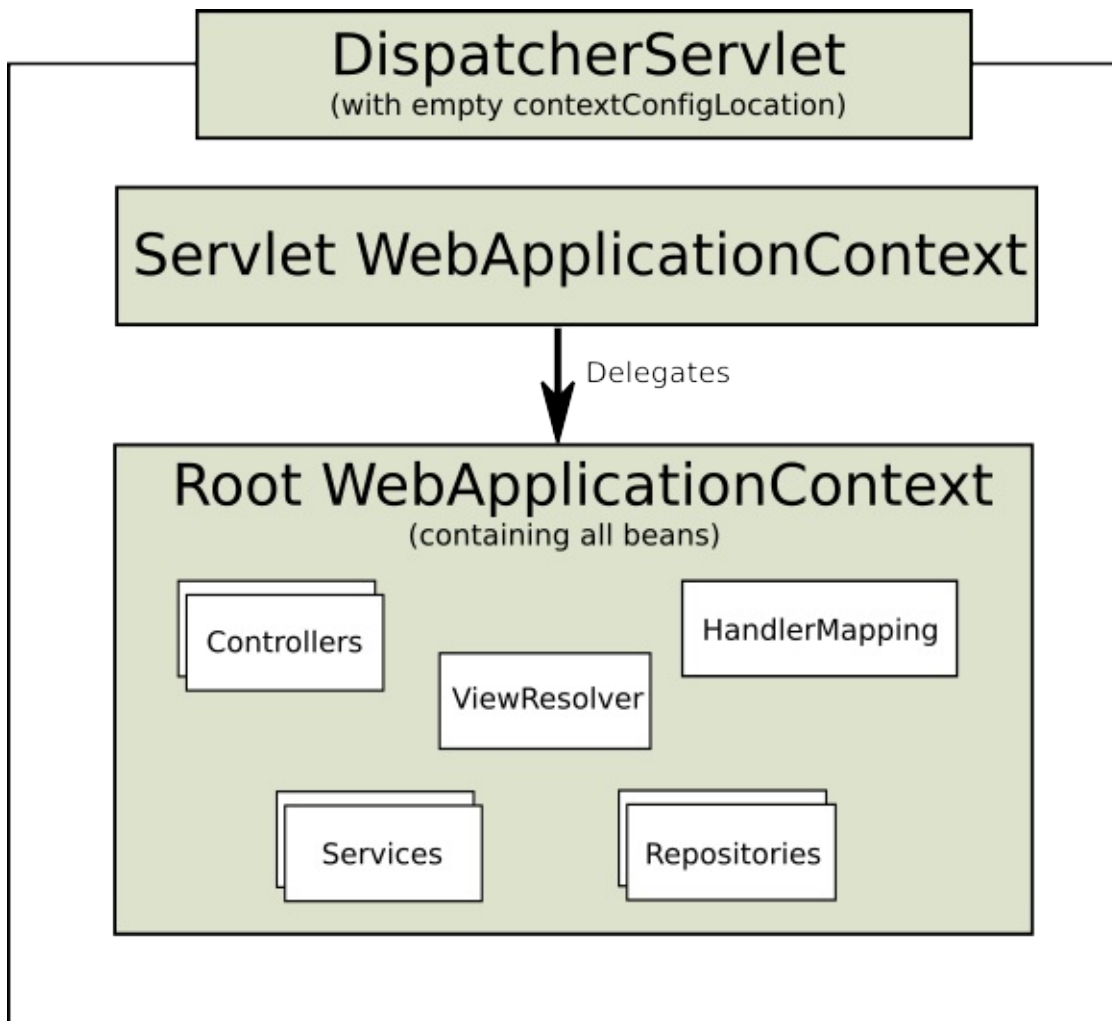
2.두 어플리케이션 컨텍스트에 대한 이야기

In the Web MVC framework, each DispatcherServlet has **its own WebApplicationContext**, which inherits all the beans already defined in **the root WebApplicationContext**

- DispatcherServlet이 시작되면서 스프링 어플리케이션 컨텍스트를 생성하고 이를 통해 클래스나 설정 파일로 선언된 빈으로 로딩하기 시작한다.



Single root context in Spring Web MVC



주의

- 클래스를 사용한 DispatcherServlet 설정방식은 톰캣7과 같이 서블릿 3.0이상을 지원하는 서버에만 적용가능

웹 애플리케이션의 컨텍스트 구성

- Servlet Context와 Root Context 계층구조
- Root Context 단일 구조
 - 스프링mvc를 안쓰고 서드파티 mvc 프레임워크를 사용할 경우등.
- **Servlet Context** 단일 구조

스프링 MVC 활성화하기

Java Config


```
@Configuration
@EnableWebMvc
public class WebConfig {

}
```

XML Config

```
<mvc:annotation-driven/>
```

위의 선언으로 다음과 같은 기능들이 활성화됨

- @RequestMapping
- @ExceptionHandler
- @Controller + @Valid
- @RequestBody
- @NumberFormat
- HttpMessageConverter 등등..

스프링 MVC 세부기능 커스터마이징

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    // Override configuration methods...

}
```

WebMvcConfigurerAdapter Configuration

- Conversion and Formatting
- Validation
- Interceptors
- Content Negotiation
- View Controllers
- View Resolvers

- Serving of Resources
- Path Matching
- Message Converters 등등..

5.1.3 Spitttr 애플리케이션 소개

5.2 간단한 컨트롤러 작성하기

Spring framework 4.3

<http://docs.spring.io/spring/docs/4.3.0.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/#new-in-4.3>

간단한 컨트롤러 작성

```
@Controller
public class HomeController {

    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String home(){
        return "home";
    }
}
```

- @Controller
 - @Component를 기반을 둔 어노테이션
 - 컴포넌트 스캔을 통해서 스프링 어플리케이션 컨텍스트의 빈으로 등록
- @RequestMapping
 - value
 - request path 명시
 - http method 기술
- return "home"
 - view의 이름을 리턴
 - DispatcherServlet은 ViewResolver에게 전달
 - InternalResourceViewResolver의 설정으로 /WEB-INF/view/home.jsp 적용

5.2.1 컨트롤러 테스트

테스트 코드 작성

```
public class HomeControllerTest {
    @Test
    public void testHomePage() {
        HomeController controller = new HomeController();
        MockMvc mockMvc = standaloneSetup(controller).build();
        mockMvc.perform(get("/"))
            .andExpect(view().name("home"))
    }
}
```

- standaloneSetup(controller).build() 인스턴스 생성
- get 요청으로 '/' 호출
- view() 를 통해 view의 이름에 대한 기대 값 설정

5.2.2 클래스 레벨 요청 처리 정의하기

@RequestMapping 분할하기

```
@Controller
@RequestMapping("/")
public class HomeController {

    @RequestMapping(method=GET)
    public String home(){
        return "home";
    }
}
```

- 컨트롤러 클래스에 붙은 @RequestMapping은 모든 메소드에 적용
- home() 메서드의 @RequestMapping은 클래스에 붙은 @RequestMapping과 조합하여 "/" 에 대한 GET요청을 처리

매핑 추가하기

```
@Controller
@RequestMapping("/{", "/homepage"})
public class HomeController {
    ...
}
```

- HomeController는 "/"와 "/homepage" 요청을 처리한다.

5.2.3 뷰에 모델 데이터 전달하기

Spitter

```
public class Spitter {

    private Long id;
    private String username;
    private String password;
    private String firstName;
    private String lastName;
    private String email;

    public Spitter() {}

    public Spitter(String username, String password, String firstName, String lastName, String email) {
        this(null, username, password, firstName, lastName, email);
    }

    public Spitter(Long id, String username, String password, String firstName, String lastName, String email) {
        this.id = id;
        this.username = username;
        this.password = password;
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
    }
}
```

```
}

...getter/setter

@Override
public boolean equals(Object that) {
    return EqualsBuilder.reflectionEquals(this, that, "firstName"
, "lastName", "username", "password", "email");
}

@Override
public int hashCode() {
    return HashCodeBuilder.reflectionHashCode(this, "firstName",
"lastName", "username", "password", "email");
}

}
```

- Apache Commons Lang
 - equals(), hashCode() 구현

/spittles의 GET요청 처리를 위한 SpittleController 테스트하기

```

@Test
public void shouldShowRecentSpittles() throws Exception {
    List<Spittle> expectedSpittles = createSpittleList(20);
    SpittleRepository mockRepository = mock(SpittleRepository.class); //Mock 저장소
    when(mockRepository.findSpittles(Long.MAX_VALUE, 20))
        .thenReturn(expectedSpittles);

    SpittleController controller = new SpittleController(mockRepository);
    MockMvc mockMvc = standaloneSetup(controller) //Mock 스프링 MVC
        .setSingleView(new InternalResourceView("/WEB-INF/views/spittles.jsp"))
        .build();

    mockMvc.perform(get("/spittles")) // /spittles 확보
        .andExpect(view().name("spittles"))
        .andExpect(model().attributeExists("spittleList"))
        .andExpect(model().attribute("spittleList", //Asset 예상
            hasItems(expectedSpittles.toArray())));
}

```

- 다른 컨트롤러와 달린 테스트에서는 MockMvc 빌더에서 setSingleView() 를 호출한다.

SpittleController

```
@Controller
@RequestMapping("/spittles")
public class SpittleController {

    private static final String MAX_LONG_AS_STRING = "922337203685
4775807";

    private SpittleRepository spittleRepository;

    @Autowired
    public SpittleController(SpittleRepository spittleRepository)
    {
        this.spittleRepository = spittleRepository;
    }

    @RequestMapping(method=RequestMethod.GET)
    public List<Spittle> spittles(
        @RequestParam(value="max", defaultValue=MAX_LONG_AS_STRING)
        long max,
        @RequestParam(value="count", defaultValue="20") int count)
    {
        return spittleRepository.findSpittles(max, count);
    }
}
```

- @Autowired
 - 생성자 @Autowired를 통해서 SpittleRepository 객체 주입
- Model
 - Model을 통해서 Spittle 리스트로 모델을 채우는 것이 가능해짐.
 - Map 자료구조를 이용해서 (key-value) View에 넘겨져서 클라이언트에 렌더링된다.
 - java.util.Map으로 모델을 넘길 수 있다.
- View Name
 - View 이름으로 "spittle"를 반환

Request Path를 통한 View Name 추론

```
@RequestMapping(value="/{spittleId}", method=RequestMethod.GET)
public List<Spittle> spittle(
    @PathVariable("spittleId") long spittleId,
    Model model) {
    return spittleRepository.findOne(spittleId);
}
```

- 메소드의 이름을 통해서 View 이름을 처리한다.
 - 위의 경우 메소드이름 spittle이 View 이름 spittle이 된다.
-

5.3 요청 입력받기

스프링 MVC에서 클라이언트에서 서버(컨트롤러의 핸들러 메소드)로 데이터 전달 방법

- 쿼리 파라미터
 - 폼 파라미터
 - 패스 변수
-


```

@Test
public void shouldShowRecentSpittles() throws Exception {
    List<Spittle> expectedSpittles = createSpittleList(20);
    SpittleRepository mockRepository = mock(SpittleRepository.class);
    when(mockRepository.findSpittles(Long.MAX_VALUE, 20))
        .thenReturn(expectedSpittles);

    SpittleController controller = new SpittleController(mockRepository);
    MockMvc mockMvc = standaloneSetup(controller)
        .setSingleView(new InternalResourceView("/WEB-INF/views/spittles.jsp"))
        .build();

    mockMvc.perform(get("/spittles"))
        .andExpect(view().name("spittles"))
        .andExpect(model().attributeExists("spittleList"))
        .andExpect(model().attribute("spittleList",
            hasItems(expectedSpittles.toArray())));
}

```

5.3.1 쿼리 파라미터 입력받기

@RequestParam

- request 파라미터를 컨트롤러의 메소드 파라미터의 값으로 채워준다.
- Optional Element
 - **defaultValue**(String) - 파라미터가 값이 없을 경우 기본값으로 사용.
 - **name**(String) - request 파라미터의 이름.
 - **required**(boolean) - 필수 입력 값인지 여부.(default : true)
 - **value**(String) - name의 별칭.
- 메소드 파라미터가 `Map<String, String>` or `MultiValueMap<String, String>` 일 경우, 이름을 명시할 필요가 없다.

[springframework 4.2.6 API 참조](#)

예제 코드

Spittr 애플리케이션에서 페이지화된 Spittles 리스트 요청 기능 구현

- **[TEST]** 페이징 처리를 위한 테스트 메소드

```
@Test
public void shouldShowPagedSpittles() throws Exception {
    List<Spittle> expectedSpittles = createSpittleList(50);
    SpittleRepository mockRepository = mock(SpittleRepository.
class);
    when(mockRepository.findSpittles(238900, 50))
        .thenReturn(expectedSpittles);

    SpittleController controller = new SpittleController(mockR
epository);
    MockMvc mockMvc = standaloneSetup(controller)
        .setSingleView(new InternalResourceView("/WEB-INF/view
s/spittles.jsp"))
        .build();

    mockMvc.perform(get("/spittles?max=238900&count=50"))
        .andExpect(view().name("spittles"))
        .andExpect(model().attributeExists("spittleList"))
        .andExpect(model().attribute("spittleList",
            hasItems(expectedSpittles.toArray())));
}
```

- **[Controller]** 페이징 처리를 위한 컨트롤러 메소드

```
@Controller
@RequestMapping("/spittles")
public class SpittleController {

    @RequestMapping(method=RequestMethod.GET)
    public List<Spittle> spittles(
        @RequestParam(value="max") long max,
        @RequestParam(value="count") int count) {
        return spittleRepository.findSpittles(max, count);
    }
}
```

- **[Controller]** 페이징 처리를 위한 컨트롤러 메소드(default value)

```
private static final String MAX_LONG_AS_STRING = "9223372036
854775807";
//private static final String MAX_LONG_AS_STRING = Long.toSt
ring(Long.MAX_VALUE);

@RequestMapping(method=RequestMethod.GET)
public List<Spittle> spittles(
    @RequestParam(value="max", defaultValue=MAX_LONG_AS_STRI
NG) long max,
    @RequestParam(value="count", defaultValue="20") int coun
t) {
    return spittleRepository.findSpittles(max, count);
}
```

private static final String MAX_LONG_AS_STRING =
Long.toString(Long.MAX_VALUE) 를 defaultValue에 사용할 경우 "attribute
value must be constant" compile error 발생.

5.3.2 패스 파라미터를 통한 입력받기

@PathVariable

- URI template 변수를 컨트롤러의 메소드 파라미터의 값으로 채워준다.
- Optional Element
 - **value(String)** - The URI template 변수의 이름.
- 메소드 파라미터가 `Map<String, String>` or `MultiValueMap<String, String>` 일 경우, URI path의 모든 template 변수들이 map으로 값이 채워진다.

[springframework 4.2.6 API 참조](#)

예제 코드

Spittr 어플리케이션에서 주어진 ID에 대해 하나의 Spittle 요청 기능 구현.

쿼리 파라미터를 이용한 처리(`/spittles/show?spittle_id=12345`)는 리소스 지향 관점에서 보면 이상적인 방식이 아님.

URL 패스에 의해 식별되는 방법(`/spittles/12345`)이 이상적인 방식.

- **[TEST]** 패스 변수로 ID를 명시하는 Spittle의 요청에 대한 테스트

```
@Test
public void testSpittle() throws Exception {
    Spittle expectedSpittle = new Spittle("Hello", new Date())
;
    SpittleRepository mockRepository = mock(SpittleRepository.
class);
    when(mockRepository.findOne(12345)).thenReturn(expectedSpi
ttle);

    SpittleController controller = new SpittleController(mockR
epository);
    MockMvc mockMvc = standaloneSetup(controller).build();

    mockMvc.perform(get("/spittles/12345"))
        .andExpect(view().name("spittle"))
        .andExpect(model().attributeExists("spittle"))
        .andExpect(model().attribute("spittle", expectedSpittle)
);
}
```

- **[Controller]** 패스 변수로 ID를 명시하는 Spittle 요청 컨트롤러 메소드

```
@RequestMapping(value="/{spittleId}", method=RequestMethod.G
ET)
public String spittle(
    @PathVariable("spittleId") long spittleId,
    Model model) {
    model.addAttribute(spittleRepository.findOne(spittleId));
    return "spittle";
}
```

- **[Controller]** 패스 변수로 ID를 명시하는 Spittle 요청 컨트롤러 메소드(value 파라미터 생략)

```
@RequestMapping(value="/{spittleId}", method=RequestMethod.GET)
public String spittle(@PathVariable long spittleId, Model model) {
    model.addAttribute(spittleRepository.findOne(spittleId));
    return "spittle";
}
```

5.4 폼 처리하기

폼 처리 과정은 폼 보여 주기와 폼을 통해 제출한 데이터 처리하기 두 가지 과정으로 나눌 수 있다.

예제 코드

Spittr 애플리케이션에서 새로운 사용자가 애플리케이션에 등록을 하기 위한 폼 구현.

- **[TEST]** 폼을 표시하는 컨트롤러 메소드 테스트

```
@Test
public void shouldShowRegistration() throws Exception {
    SpitterRepository mockRepository = mock(SpitterRepository.class);
    SpitterController controller = new SpitterController(mockRepository);
    MockMvc mockMvc = standaloneSetup(controller).build();
    mockMvc.perform(get("/spitter/register"))
        .andExpect(view().name("registerForm"));
}
```

- **[Controller]** 사용자가 랩에 가입할 수 있는 폼을 표시

```
@RequestMapping("/spitter")
public class SpitterController {

    @RequestMapping(value="/register", method=GET)
    public String showRegistrationForm() {
        return "registerForm";
    }
}
```

5.4.1 폼 처리 컨트롤러 작성

request 파라미터의 이름과 동일한 이름의 프로퍼티를 가진 객체를 메소드 파라미터로 사용해서 동일 이름의 request 파라미터 값들이 채워지도록 한다.

InternalResourceViewResolver는 뷰 명세에서 접두사 redirect:, forward: 를 뷰 이름이 아닌 별도의 redirect, forward로 처리한다.

예제 코드

등록 폼에서 POST 요청을 처리할 때, 폼 데이터를 받고 Spitter 객체로 저장하는 기능 구현

- **[TEST]** 폼 처리 컨트롤러를 테스트하는 메소드

```
@Test
public void shouldProcessRegistration() throws Exception {
    SpitterRepository mockRepository = mock(SpitterRepository.
class);
    Spitter unsaved = new Spitter("jbauer", "24hours", "Jack",
"Bauer", "jbauer@ctu.gov");
    Spitter saved = new Spitter(24L, "jbauer", "24hours", "Jac
k", "Bauer", "jbauer@ctu.gov");
    when(mockRepository.save(unsaved)).thenReturn(saved);

    SpitterController controller = new SpitterController(mockR
epository);
    MockMvc mockMvc = standaloneSetup(controller).build();

    mockMvc.perform(post("/spitter/register")
        .param("firstName", "Jack")
        .param("lastName", "Bauer")
        .param("username", "jbauer")
        .param("password", "24hours")
        .param("email", "jbauer@ctu.gov"))
        .andExpect(redirectedUrl("/spitter/jbauer")));

    verify(mockRepository, atLeastOnce()).save(unsaved);
}
```

- **[Controller]** 새로운 사용자를 등록하기 위한 폼을 제출하기

```
@Controller
@RequestMapping("/spitter")
public class SpitterController {

    private SpitterRepository spitterRepository;

    @Autowired
    public SpitterController(SpitterRepository spitterRepository) {
        this.spitterRepository = spitterRepository;
    }

    @RequestMapping(value="/register", method=GET)
    public String showRegistrationForm() {
        return "registerForm";
    }

    @RequestMapping(value="/register", method=POST)
    public String processRegistration(Spitter spitter) {

        spitterRepository.save(spitter);
        return "redirect:/spitter/" + spitter.getUsername();
    }

    @RequestMapping(value="/{username}", method=GET)
    public String showSpitterProfile(@PathVariable String username, Model model) {
        Spitter spitter = spitterRepository.findByUsername(username);
        model.addAttribute(spitter);
        return "profile";
    }
}
```

public String processRegistration(Spitter spitter) 메소드에서 파라미터로 Spitter 객체를 받는다. 이 객체는 firstName, lastName, username, password 프로퍼티를 갖고, 같은 이름의 request 파라미터로부터 값이 채워지게 된다.

5.4.2 폼 검증하기

폼 데이터를 검증하기 위한 방법으로 스프링에서는 자바 인증(Validation) API(a.k.a JSR-303)을 제공(스프링 3.0 부터 지원)

추가적인 설정 없이 자바 인증이 스프링 MVC에서 동작하지만, 별도의 자바 인증 API 구현체가 필요(e.g. Hibernate Validator, Apache BVal ...)

- 자바 인증 API에서 제공되는 검증 Annotation 목록

Annotation	Description
@AssertFalse	Boolean 타입에 false 값이어야만 하는 요소에 붙이는 Annotation
@AssertTrue	Boolean 타입에 true 값이어야만 하는 요소에 붙이는 Annotation
@DecimalMax	값이 BigDecimalString 보다 작거나 같은 값을 갖는 숫자여야 하는 요소에 붙는 Annotation
@DecimalMin	값이 BigDecimalString 보다 크거나 같은 값을 갖는 숫자여야 하는 요소에 붙는 Annotation
@Digits	십진수 값을 갖는 숫자여야 하는 요소에 붙는 Annotation
@Future	값이 미래의 날짜여야 하는 요소에 붙는 Annotation
@Max	주어진 값보다 작거나 같은 값을 갖는 숫자여야 하는 요소에 붙는 Annotation
@Min	주어진 값보다 크거나 같은 값을 갖는 숫자여야 하는 요소에 붙는 Annotation
@NotNull	값이 null이 아니어야 하는 요소에 붙는 Annotation
@Null	값이 null이어야 하는 요소에 붙는 Annotation
@Past	값이 과거의 날짜여야 하는 요소에 붙는 Annotation
@Pattern	값이 주어진 정규 표현식에 맞아야 하는 요소에 붙는 Annotation
@Size	크기가 주어진 값과 같은 String, collection, array이어야 하는 요소에 붙는 Annotation

예제 코드

SpittleForm validation 처리 기능 추가

- **[Spitter]** SpittleForm: SpittlePost 요청에서 제출된 필드만 전달하기

```
public class Spitter {  
  
    private Long id;  
  
    @NotNull  
    @Size(min=5, max=16)  
    private String username;  
  
    @NotNull  
    @Size(min=5, max=25)  
    private String password;  
  
    @NotNull  
    @Size(min=2, max=30)  
    private String firstName;  
  
    @NotNull  
    @Size(min=2, max=30)  
    private String lastName;  
  
    .  
    .  
    ...  
}
```

- **[Controller]** processRegistration(): 제출된 데이터의 적합성 확인

```
@RequestMapping(value="/register", method=POST)
public String processRegistration(
    @Valid Spitter spitter,
    Errors errors) {
    if (errors.hasErrors()) {
        return "registerForm";
    }

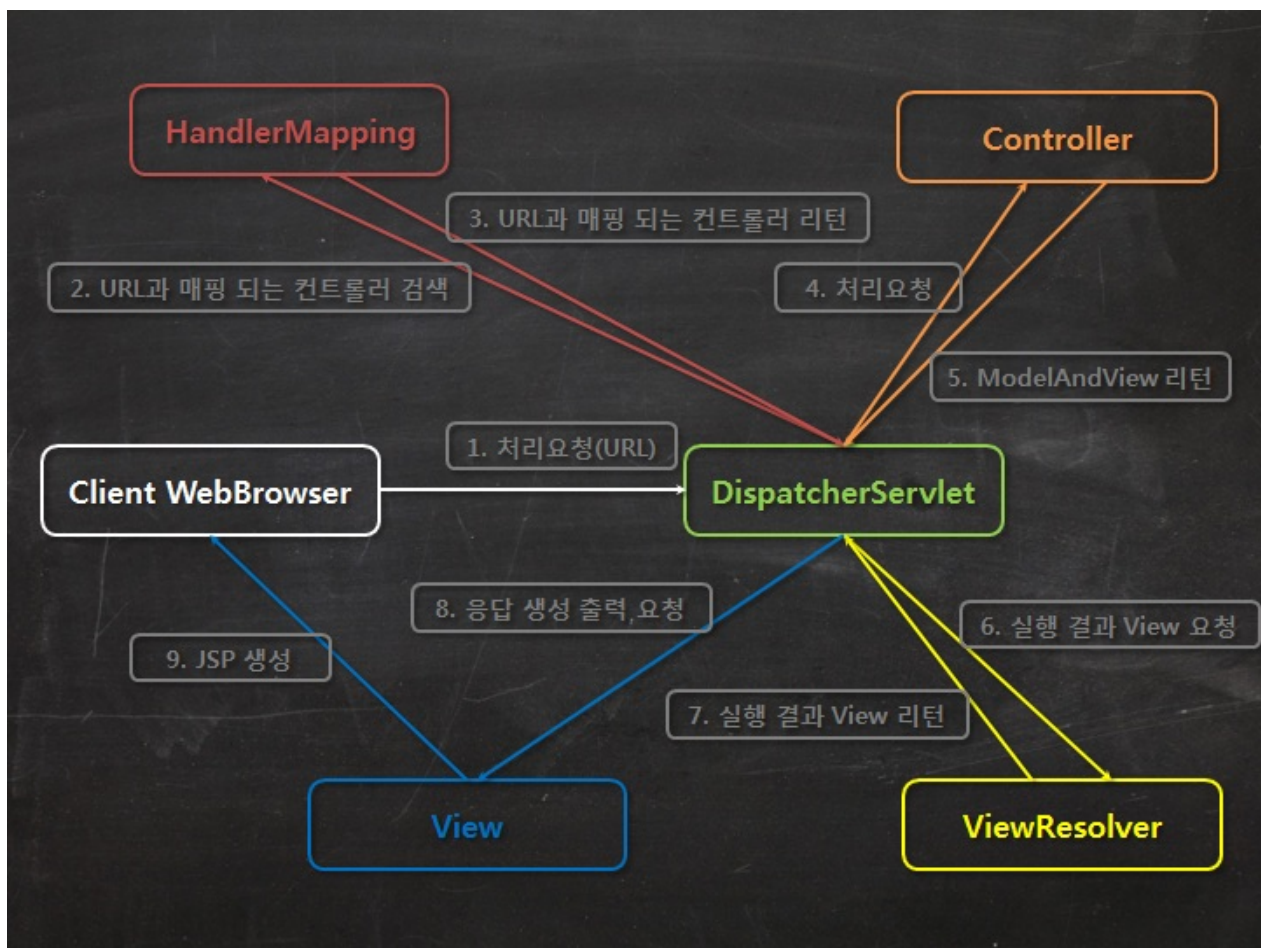
    spitterRepository.save(spitter);
    return "redirect:/spitter/" + spitter.getUsername();
}
```

Chapter6. 웹 뷰 렌더링

6.1 뷰 리졸루션 이해하기

뷰 리졸버의 역할

모델을 실제로 렌더링하기 위해 구현되어있는 뷰를 결정하는 역할.



스프링MVC는 아래와 같은 내용의 **ViewResolver** 인터페이스를 정의한다.

```
public interface ViewResolver {
    View resolveViewName(String viewName, Locale locale) throws Exception;
}
```

`resolveViewName()` 메소드는 view이름, locale을 넘겨받고 view인스턴스를 return한다.

view의 다른 인터페이스는 다음과 같다.

```
public interface View {
    String getContentType();
    void render(Map<String,?> model, HttpServletRequest request, HttpServletResponse response) throws Exception;
}
```

view 인터페이스는 모델, 서블릿 요청, 응답객체를 전달받아 결과를 응답 내의 결과에 렌더링한다.

스프링은 상황에 맞게 바로 사용할 수 있는 13가지 **ViewResolver**를 제공한다. (p.201)

- **InternalResourceResolver**는 일반적으로 **JSP**에 사용되고,
- **TilesViewResolver**는 아파치 타일즈 뷰에 사용되고,
- **FreeMarkerViewResolver**는 FreeMarker 템플릿 뷰,
- **VelocityViewResolver**는 Velocity템플릿 뷰에 대응한다.

6.2 JSP 뷰 만들기

스프링은 두가지 방식으로 **JSP**를 지원한다

1. **InternalResourceViewResolver**는 JSP파일에 뷰 이름을 결정하기 위해서 사용한다. JSP페이지의 JSTL을 사용하는 경우 **InternalResourceViewResolver**는 **JstlView**에 의한 JSP파일로 뷰 이름을 결정하고, JSTL포맷과 메시지 태그에 JSTL locale, resource bundle변수를 대입한다.
2. 스프링은 form-to-model 바인딩을 위한 것과 일반적인 유틸리티 기능을 제공하는 두 개의 JSP 태그 라이브러리를 제공한다.

6.2.1 JSP-ready 뷰 리졸버 설정하기

1) 기본적인 **InternalResourceViewResolver**

InternalResourceViewResolver는 웹 애플리케이션 뷰 리소스의 물리적 path를 결정하기 위해 prefix, suffix를 뷰 이름에 붙이는 규칙을 따른다.

일반적으로 JSP파일들은 직접적인 접근을 막기 위해 웹애플리케이션의 WEB-INF폴더에 넣어둔다.

메소드에서 설정

```
@Bean
public ViewResolver viewResolver() {
    InternalResourceViewResolver resolver = new InternalResourceVi
ewResolver();
    resolver.setPrefix("/WEB-INF/views/");
    resolver.setSuffix(".jsp");
    return resolver;
}
```

XML에서 설정

```
<bean class="org.springframework.web.servlet.view.InternalResour
ceViewResolver">
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
</bean>
```

2) JSTL 뷰 결정하기

여기까지는 기본적인 InternalResourceViewResolver를 설정했다. 여기서 추가적으로 JSTL을 사용하려면 다음과 같이 추가하면 된다.

JSTL : 메소드에서 설정

```
@Bean
public ViewResolver viewResolver() {
    InternalResourceViewResolver resolver = new InternalResourceVi
ewResolver();
    resolver.setPrefix("/WEB-INF/views/");
    resolver.setSuffix(".jsp");
    resolver.setViewClass(org.springframework.web.servlet.view.Jst
lView.class);
    return resolver;
}
```

JSTL : XML에서 설정

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/views/" />
  <property name="suffix" value=".jsp" />
  <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
</bean>
```

6.2.2 스프링 JSP라이브러리 사용하기

1) 폼에 모델 바인딩하기

스프링의 폼 바인딩 JSP태그 라이브러리에는 14개의 태그가 있다. 이들 대부분은 HTML폼 태그를 렌더링한다. 이 태그들이 HTML태그들과 다른점은 **모델의 객체로 바인딩된다는 점**과 모델 객체의 프로버티로부터 값이 입력되는 것이 가능하다는 점이다.

폼바인딩 태그를 사용하기 위해서는 JSP페이지 안에 아래의 내용을 선언한다.

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="sf"%>
```

이 책에서는 간결하고 입력하기 쉬운 이유로 prefix를 Spring Forms의 축약 sf로 가정한다. 폼 바인딩 태그 라이브러리가 선언되면 (p.206) 표에 나와있는 14개의 태그를 사용한다.

Example

```
<sf:checkbox>, <sf:input>, <sf:label>, ...
```

Example

```
<sf:form method="POST" commandName="user">
  First Name : <sf:input path="firstName"/>
  Last Name : <sf:input path="lastName"/>
</sf:form>
```

2) 오류 표시하기

3) 스프링의 일반 태그 라이브러리

4) 다국어 메시지 표시하기

5) URL만들기

6) 콘텐츠 이스케이핑

6.3 아파치 타일즈 뷰로 레이아웃 정의하기

아파치 타일즈는 레이아웃 엔진중 하나. 레이아웃 엔진은 말그대로 레이아웃 잡는 것을 도와 준다.

예를 들어 구현해야 할 페이지가 많을 때 공통적으로 사용하는 header 영역, footer 영역을 어떻게 관리 할 것인가? JSP 코드에서 매번 include 해 준다? 번거롭다. 레이아웃 엔진으로 미리 잡아 주면 main 영역 코드만 짜면 된다.

Apache Tiles 기본 사용법

설정 (ex. tiles.xml)


```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC
    "-//Apache Software Foundation//DTD Tiles Configuration 3
    .0//EN"
    "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">
<tiles-definitions>

    <definition name="base" template="/WEB-INF/layout/page.jsp"> /
/ 기본 타일 정의
    <put-attribute name="header" value="/WEB-INF/layout/header.j
    sp" />
    <put-attribute name="footer" value="/WEB-INF/layout/footer.j
    sp" />
    </definition>

    <definition name="home" extends="base"> // 기본 타일 확장
    <put-attribute name="body" value="/WEB-INF/views/home.jsp" />

    </definition>

</tiles-definitions>
```

base 레이아웃으로 page.jsp 를 사용 했다. home 은 base 를 상속 받아 body 에 home.jsp 를 넣는다.

레이아웃 파일. page.jsp

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="s"
%>
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="t" %
>
<%@ page session="false" %>
<!DOCTYPE html>
<html>
  <head>
    <title>June Kim's World!</title>
    ...
  <body>
    <t:insertAttribute name="header" />

    <div class="container">
      <t:insertAttribute name="body" />
    </div><!-- /.container -->

    <t:insertAttribute name="footer" />
  </body>
</html>
```

spring 에서 사용 하려면

일단 디펜던시 추가 해 주자. 기존 JSP 설정에 tiles-jsp:3.0.4 를 추가 하면 된다.

```
compile('org.apache.tomcat.embed:tomcat-embed-jasper')
compile('org.apache.tiles:tiles-jsp:3.0.4')
compile('javax.servlet:jstl')
```

Config 에 다음의 내용을 추가 한다. WebConfig.java

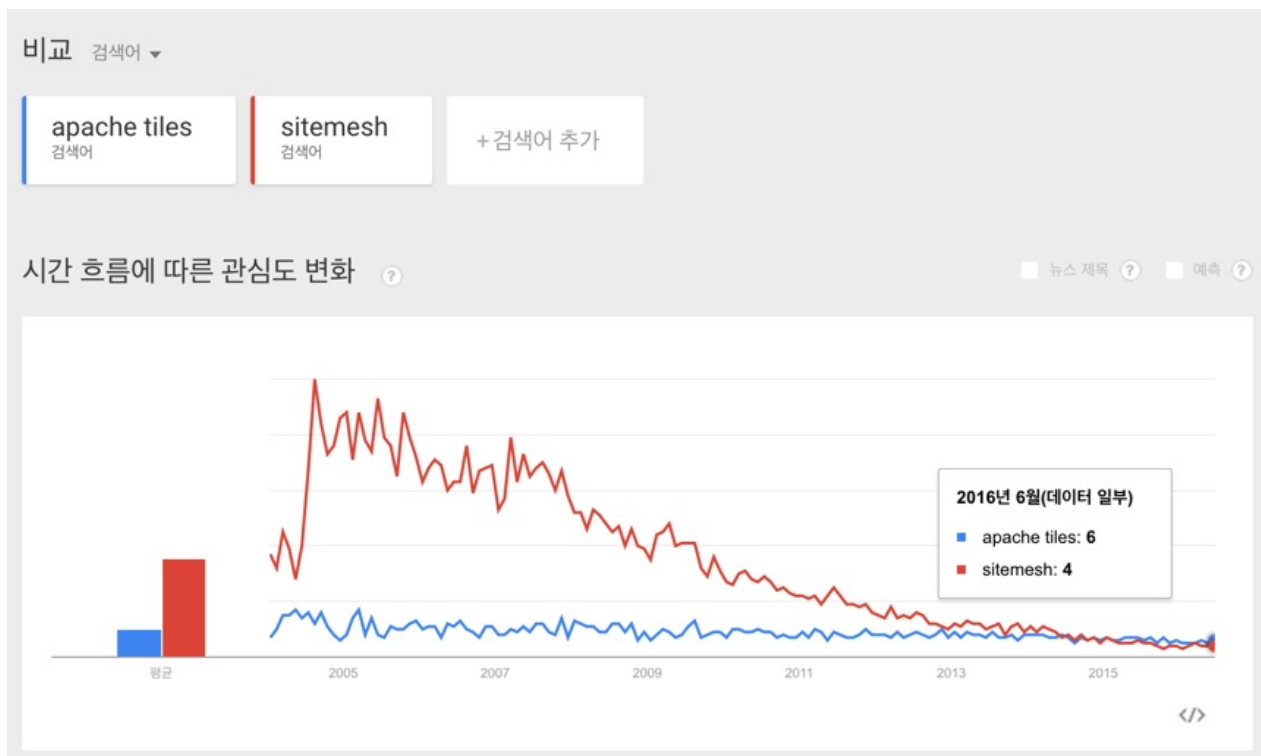
```
@Bean
public TilesConfigurer tilesConfigurer() {
    final TilesConfigurer configurator = new TilesConfigurer()
; // 타일 정의
    configurator.setDefinitions(new String[] { "WEB-INF/layout
/tiles.xml" }); // 설정 파일 위치
    configurator.setCheckRefresh(true); // 리프레시를 가능하게 설정
    return configurator;
}

@Bean
public TilesViewResolver tilesViewResolver() {
    final TilesViewResolver resolver = new TilesViewResolver
();
    resolver.setViewClass(TilesView.class);
    return resolver;
}
```

자세한 설정은 <http://docs.spring.io/spring/docs/4.2.6.RELEASE/spring-framework-reference/html/view.html#view-tiles-integrate> 를 참고 한다.

레이아웃 템플릿 엔진. Tiles, Sitemesh

대표적으로 Tiles, Sitemesh



<https://www.google.com/trends/explore#q=apache%20tiles%2C%20sitemesh>

2011 KSUG 글 https://groups.google.com/forum/#!topic/ksug/fuLQvWC_ikw

박** 전 tiles 1 < sitemesh < tiles 2 순입니다. 사실 둘 다 마음에 안 듭니다.

참고자료

- 타일즈(Tiles) 프레임워크 2015.07.28 <http://expitly.tistory.com/13>
 - Tiles 는 Composite View 패턴
 - Sitemesh 는 Decorator 패턴
- Tiles VS SiteMesh 2014.01.21 <http://sojw.tistory.com/1169742017>
- Tiles VS SiteMesh 2009.05.01 <http://whiteship.tistory.com/2215>
- Tiles와 SiteMesh 차이 2009.05.01 <http://whiteship.tistory.com/2216>
- Tiles <https://tiles.apache.org/> <https://github.com/apache/tiles>

<https://github.com/apache/tiles/releases> 3.0.5 가 2014.09.22

- Sitemesh <http://wiki.sitemesh.org/wiki/display/sitemesh/Home>
<https://github.com/sitemesh/sitemesh3/releases> 3.0.1 이 2015.05.29

6.4 Thymeleaf로 작업하기

JSP 의 문제점

내부가 HTML 또는 XML 의 형태로 되어 있지만, 실상 어느쪽도 아님

```
<input type="text" value="<c:out value="${thing.name}"/> />
```

html 이지만 html 이 아니다. 브라우저에서 열어 볼 수 없다. 그렇다고 보기 좋지도 않다.

템플릿 코드 자체가 HTML 이라 브라우저에서 바로 열어 볼 수 있는 템플릿을 Natural Template 이라고 함.

서블릿 스펙과 강하게 결합되어 있다. 서블릿 기반 웹 애플리케이션의 웹 뷰에 사용된다는 것을 의미. 이메일 형식 등 다른 목적의 템플릿이나 서블릿 기반이 아닌 웹 애플리케이션에는 좋은 선택이 아님

thymeleaf

왜인지는 몰라도 spring 에서 thymeleaf 를 밀고 있다.

장점: Natural Template 이라 브라우저에서 바로 열어 볼 수 있다. 다만 완벽하지는 않음. th:href 때문에 링크 동작 안함.

thymeleaf 기본 예

```

<table>
  <thead>
    <tr>
      <th th:text="#{msgs.headers.name}">Name</th>
      <th th:text="#{msgs.headers.price}">Price</th>
    </tr>
  </thead>
  <tbody>
    <tr th:each="prod: ${allProducts}">
      <td th:text="${prod.name}">Oranges</td>
      <td th:text="${#numbers.formatDecimal(prod.price, 1, 2)}">
0.99</td>
    </tr>
  </tbody>
</table>

```

th:each, th:text 등 약간 익숙하지 않은 속성이 보이지만 JSP 에 비해 깔끔하다.

`${}` 변수 표현식

```

${session.user.name}
<span th:text="${book.author.name}">
<li th:each="book : ${books}">

```

`*{}` 선택 표현식

th:object="\${spitter}" 후 th:field="*{firstName}" 하면 Spitter 객체의 firstName 프로퍼티

```

<div th:object="${book}">
  ...
  <span th:text="*{title}">...</span>
  ...
</div>

```

자세한 문법은 <http://www.thymeleaf.org/doc/articles/standarddialect5minutes.html>의 내용을 참고 한다.

thymeleaf 사용해 보기

dependency 추가

```
compile "org.thymeleaf:thymeleaf-spring4:$thymeleafVersion"
```

config

```
@Bean
public ViewResolver viewResolver(SpringTemplateEngine template
Engine) {
    ThymeleafViewResolver viewResolver = new ThymeleafViewResolv
er();
    viewResolver.setTemplateEngine(templateEngine);
    return viewResolver;
}

@Bean
public SpringTemplateEngine templateEngine(TemplateResolver te
mplateResolver) {
    SpringTemplateEngine templateEngine = new SpringTemplateEngi
ne();
    templateEngine.setTemplateResolver(templateResolver);
    return templateEngine;
}

@Bean
public TemplateResolver templateResolver() {
    TemplateResolver templateResolver = new ServletContextTempla
teResolver();
    templateResolver.setPrefix("/WEB-INF/views/");
    templateResolver.setSuffix(".html");
    templateResolver.setTemplateMode("HTML5");
    return templateResolver;
}
```

만약 spring boot 라면 spring-boot-starter-thymeleaf 하나 추가 해 주면 된다.

```
compile('org.springframework.boot:spring-boot-starter-thymeleaf')
)
```

spring boot에서는 별도 설정을 안하면 기본 자동설정으로 resources/templates 에 템플릿 파일을 넣으면 인식한다.

thymeleaf 에서의 layout은?

<http://www.thymeleaf.org/doc/articles/layouts.html> 그냥 include 해서 쓰거나

- Apache Tiles 2 Dialect <https://github.com/thymeleaf/thymeleaf-extras-tiles2>
- Thymeleaf Layout Dialect <https://github.com/ultraq/thymeleaf-layout-dialect>

를 쓰면 될 듯 하다.

Spring Initializr 에서 선택 가능한 템플릿 엔진

Spring Initializr 에서 선택 가능한 템플릿 엔진으로 Freemarker, Velocity, Groovy Templates, Thymeleaf, Mustache 가 있다.

Template Engines

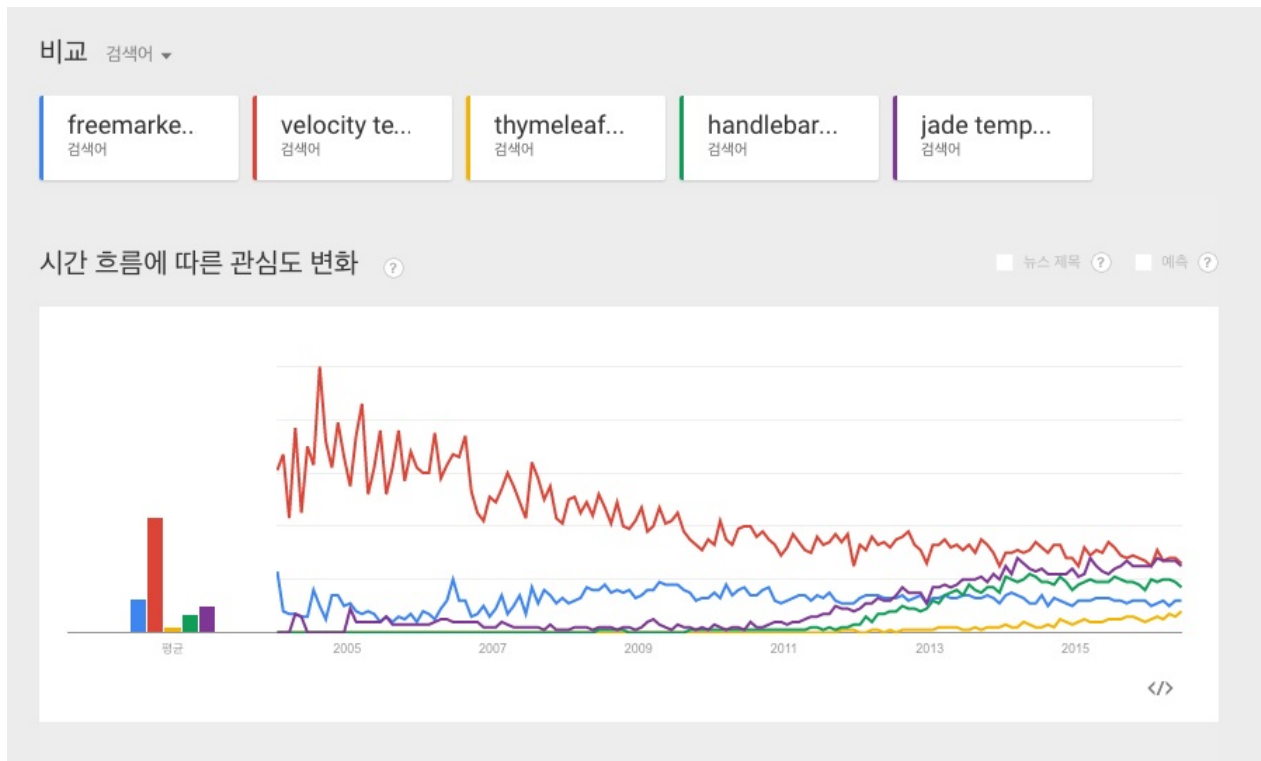
- ☒ Freemarker
FreeMarker templating engine
- ☒ Velocity
Velocity templating engine
- ☒ Groovy Templates
Groovy templating engine
- ☒ Thymeleaf
Thymeleaf templating engine, including integration with Spring
- ☒ Mustache
Mustache templating engine

출처:

<http://start.spring.io/>

기타 템플릿 엔진들

다른 템플릿 엔진들 맛보기



- <https://www.google.com/trends/explore#q=freemarker%20template%2C%20velocity%20template%2C%20thymeleaf%20template%2C%20handlebars%20template%2C%20jade%20template&cmpt=q&tz=Etc%2FGMT-9>
- 2016.06.14 캡처
- velocity 가 여전히 강세이나 점점 사그라 들고 있음
- jade 가 치고 올라옴
- handlebars 도 만만치 않음
- freemarker 는 꾸준함
- thymeleaf 도 조금씩 올라오고 있지만 미약함

Velocity

- <http://velocity.apache.org/>
- <http://velocity.apache.org/engine/index.html>
- <http://velocity.apache.org/engine/1.7/user-guide.html>


```

<html>
  <body>
    Hello $customer.Name!
    <table>
      #foreach( $mud in $mudsOnSpecial )
        #if ( $customer.hasPurchased($mud) )
          <tr>
            <td>
              $flogger.getPromo( $mud )
            </td>
          </tr>
        #end
      #end
    </table>
  </body>
</html>

```

인기 좋았던 템플릿 엔진. 그러나 Spring 4.3 에서 Deprecated 됨



SRP-13235 Deprecate Velocity support <https://jira.spring.io/browse/SPR-13235>


Spring Framework / SPR-13235

Deprecate Velocity support

Agile Board







Details

Type:	 Task	Status:	CLOSED
Priority:	 Major	Resolution:	Complete
Affects Version/s:	None	Fix Version/s:	4.3 RC1
Component/s:	None		
Labels:	None		
Last commented by a User:	true		

Description

Velocity 1.7 dates back to 2010. After more than five years of no maintenance in the original project, it's about time to deprecate Velocity support on Spring's side.

Issue Links

is related to	 SPR-13795 Remove Velocity support	 OPEN
relates to	 SPR-13062 Require Jackson 2.6+, FreeMarker 2.3.21+, XStrea...	 CLOSED
	 SPR-13230 Deprecate Hibernate 3 support	 CLOSED

이유는 Velocity 1.7 이후로 5년 넘게 유지보수 되고 있지 않아서 deprecate 함

Freemarker

<http://freemarker.org/> <http://freemarker.org/docs/>

템플릿엔진이다. html, email, 설정파일, 소스코드 생성에 사용

```
<html>
<head>
  <title>Welcome!</title>
</head>
<body>
  <h1>Welcome ${user}!</h1>
  <p>Our latest product:
  <a href="${latestProduct.url}">${latestProduct.name}</a>
</body>
</html>
```

조건문

```
<h1>
  Welcome ${user}<#if user == "Big Joe">, our beloved leader</#if>!
</h1>
```

```
<#if animals.python.price == 0>
  Pythons are free today!
</#if>
```

Template Toolkit

<http://template-toolkit.org/> perl 에서 주로 사용. python 도 사용 가능.

<http://template-toolkit.org/about.html>

layout

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Strict//EN">
<html>
  <head>
    <title>[% title %]</title>
  </head>
  <body>
    <div id="header">
      <a href="/index.html" class="logo" alt="Home Page"></a>
      <h1 class="headline">[% title %]</h1>
    </div>

    [% content %]

    <div id="footer">
      <div id="copyright">
        &copy; [% copyright %]
      </div>
    </div>
  </body>
</html>
```

content

```
[% WRAPPER layout
  title      = "My First Example"
  copyright  = "2007 Arthur Dent"
%]
<p>
  Hello World!
</p>
[% END %]
```

Groovy Templates

<http://docs.groovy-lang.org/latest/html/documentation/template-engines.html>

ex) xml

```
xmlDeclaration()  
cars {  
  cars.each {  
    car(make: it.make, model: it.model)  
  }  
}
```

ex) html

```
yieldUnescaped '<!DOCTYPE html>'  
html(lang:'en') {  
  head {  
    meta('http-equiv': '"Content-Type" content="text/html; ch  
arset=utf-8"')  
    title('My page')  
  }  
  body {  
    p('This is an example of HTML contents')  
  }  
}
```

layout 구성. layout-main.tpl

```
html {  
  head {  
    title(title)  
  }  
  body {  
    bodyContents()  
  }  
}
```

content

```
layout 'layout-main.tpl',  
  title: 'Layout example',  
  bodyContents: contents { p('This is the body') }
```

Mustache 와 handlebars.js

<https://mustache.github.io/> Logic-less templates.

<http://mustache.github.io/mustache.5.html>

지원언어: Ruby, JavaScript, Python, Erlang, node.js, PHP, Perl, Perl6, Objective-C, Java, C#/.NET, Android, C++, Go, Lua, ooc, ActionScript, ColdFusion, Scala, Clojure, Fantom, CoffeeScript, D, Haskell, XQuery, ASP, Io, Dart, Haxe, Delphi, Racket, Rust, OCaml, Swift, Bash, Julia, R, Crystal, Common Lisp, and for Nim

템플릿

```
<h1>{{header}}</h1>
{{#bug}}
{{/bug}}

{{#items}}
  {{#first}}
    <li><strong>{{name}}</strong></li>
  {{/first}}
  {{#link}}
    <li><a href="{{url}}">{{name}}</a></li>
  {{/link}}
{{/items}}

{{#empty}}
  <p>The list is empty.</p>
{{/empty}}
```

값

```
{
  "header": "Colors",
  "items": [
    {"name": "red", "first": true, "url": "#Red"},
    {"name": "green", "link": true, "url": "#Green"},
    {"name": "blue", "link": true, "url": "#Blue"}
  ],
  "empty": false
}
```

결과

```
<h1>Colors</h1>

<li><strong>red</strong></li>

<li><a href="#Green">green</a></li>
<li><a href="#Blue">blue</a></li>

empty 를 true 로 바꾸면 아래에 한줄 더 나온다.

<p>The list is empty.</p>
```

handlebars 는 mustache 의 확장판

<http://handlebarsjs.com/>

Handlebars.js is an extension to the Mustache templating language created by Chris Wanstrath. Handlebars.js and Mustache are both logicless templating languages that keep the view and the code separated like we all know they should be.

Jinja2

python 의 템플릿 엔진. ansible 에서도 사용함. <http://jinja.pocoo.org/>

```
{% extends "layout.html" %}
{% block body %}
  <ul>
    {% for user in users %}
      <li><a href="{{ user.url }}">{{ user.username }}</a></li>
    {% endfor %}
  </ul>
{% endblock %}
```

Jade

<http://jade-lang.com/> nodejs 에서 사용되는 템플릿엔진. HTML 과 달리 닫는태그가 없고 상속을 지원함.

```
doctype html
html(lang="en")
  head
    title= pageTitle
    script(type='text/javascript').
      if (foo) {
        bar(1 + 5)
      }
  body
    h1 Jade - node template engine
    #container.col
      if youAreUsingJade
        p You are amazing
      else
        p Get on it!
      p.
        Jade is a terse and simple
        templating language with a
        strong focus on performance
        and powerful features.
```

아래처럼 변환된다.


```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Jade</title>
    <script type="text/javascript">
      if (foo) {
        bar(1 + 5)
      }
    </script>
  </head>
  <body>
    <h1>Jade - node template engine</h1>
    <div id="container" class="col">
      <p>You are amazing</p>
      <p>
        Jade is a terse and simple
        templating language with a
        strong focus on performance
        and powerful features.
      </p>
    </div>
  </body>
</html>
```

레이아웃 구성하기

layout.jade

```
doctype html
html
  head
    title hello world
    link(rel='stylesheet', href='/csslib/bootstrap.min.css')
    script(src='/jslib/jquery-2.1.3.min.js')

  body
    block content
```

block content 에 본문이 들어가게 된다.

home.jade

```
extends layout
block content
  #container
    include nav
    .boxed
      #content-container
      ...
```

스프링에서 사용하도록 만든 `spring jade4j` 도 있다.

<https://github.com/neuland/spring-jade4j>

템플릿과 상속

- Jade, Jinja2 등 의 템플릿 엔진은 템플릿 상속(template inheritance) 을 통해 레이아웃을 쉽게 관리한다.
- 상속기능을 제공하지 않는 템플릿 엔진들을 레이아웃 엔진의 도움을 받게 된다.
- JSP 나 Freemarker 도 상속기능을 구현해서 사용하기도 한다.
 - ex)
 - <https://github.com/kwon37xi/jsp-template-inheritance>
 - <https://github.com/kwon37xi/freemarker-template-inheritance>

주저리

예전에 비해 서버사이드 템플릿 엔진의 필요성이 많이 떨어졌다. 더 이상 많은 페이지를 작성하고 있지 않다. 적은 수의 페이지에서 화면을 다양하게 그린다. 데이터는 따로 조회해 와서 화면에 그린다. angular, backbone, react 등 이 뜬다.

참고자료

- https://en.wikipedia.org/wiki/Comparison_of_web_template_engines
- Spring과 Template Engine, 현업에서는 어떻게 사용하나요? 2014.10.01
<https://slipp.net/questions/294>

- Thymeleaf 사용 소감 2013/08/12 <https://blog.outsider.ne.kr/969>
 - 레이아웃 템플릿 엔진 Tiles, Sitemesh
 - 텍스트 템플릿 엔진 Freemarker, JSP, thymeleaf
- <https://spring.io/blog/2012/10/30/spring-mvc-from-jsp-and-tiles-to-thymeleaf>
- <http://coding-slave.blogspot.kr/2016/01/web-spring-spring-boot.html>
 - spring-boot-starter-web 에 포함된 tomcat 은 jsp 엔진을 포함하지 않는다.
 - tomcat-embed-jasper, jstl 을 의존성에 포함시켜줘야 JSP 파일 구동이 가능
 - JSP 는 spring boot 기본 templates 폴더 안에서 작동하지 않는다.
 - application.properties 에 다음과 같은 설정을 추가해 주어야 한다.

```
spring.mvc.view.prefix=/WEB-INF/jsp/  
spring.mvc.view.suffix=.jsp
```
- <http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>
- JSP/Freemarker 템플릿 상속을 통한 레이아웃 관리 2013/09/12
<http://kwon37xi.egloos.com/4827957>
- <https://github.com/mbosecke/template-benchmark>
- <http://stackoverflow.com/questions/17163275/is-there-any-java-template-engine-benchmarks/35553058#35553058>

Chapter10. 스프링과 JDBC를 사용하여 데이터베이스 사용하기

데이터 액세스

- 데이터 액세스 프레임워크 초기화
- 커넥션 오픈
- 다양한 예외처리
- 연결 종료

스프링 데이터 액세스

- 다양한 기술들로 결합된 데이터 액세스 프레임워크 제공
ex) JDBC, 하이버네이트, JPA(Java Persistence API), 퍼시스턴스 프레임워크, NoSQL 데이터베이스 등
- 퍼시스턴스 코드에서 데이터 액세스를 제거
- 하위 레벨(low-level) 데이터 액세스 작업시 스프링을 사용하여 데이터 관리를 수행

이번 장에서는 JDBC를 위한 스프링 지원 사항에 대해서 중점적으로 살펴본다.

10-1 스프링의 데이터 액세스 철학

저장소 (DAO, Data-Acess Object)

- 퍼시스턴스 로직이 산란(scattering)되는 것을 피하기 위해서 데이터 액세스는 해당 태스크에 집중된 하나 이상의 컴포넌트로 이루어진다.
- 애플리케이션이 저장소 내 특정 데이터 액세스 전략에 커플링되는 것을 막기 위해 인터페이스를 사용하여 기능을 외부로 제공한다.
- [그림10.1] 서비스 객체는 자신의 데이터 액세스를 처리하지 않는다. 대신 데이터 액세스를 저장소에 위임한다. 저장소의 인터페이스는 서비스 객체에 느슨하게 커플링한다. (p341)

인터페이스와 스프링

- 서비스 객체는 인터페이스를 통해서 저장소에 액세스 한다. (테스트가 용이해 짐)
- 데이터 액세스 계층은 퍼시스턴스 기술에 상관없이 액세스 한다.
- 데이터 액세스 메소드만 인터페이스를 통해서 노출시킨다. (애플리케이션 나머지 부분에 최소한의 영향)
- 스프링은 데이터 액세스 계층을 모든 퍼시스턴스 옵션을 가지는 일관적인 예외 계층 구조를 사용하여 애플리케이션의 나머지와 격리한다.

10-1-1 스프링의 데이터 액세스 예외 계층 구조

SQLException은 무쓸모

- 데이터베이스 액세스 동안 무엇인가 문제가 있다는 사실을 의미한다.
- 하지만 무엇이 잘못되었고 어떻게 처리해야 하는지에 대한 설명은 거의 없다.
- 대부분 catch 블록에서 복구가 불가능한 치명적인 상황이다.
- 예외 발생 근본 원인에 대한 상세 정보를 획득하기 위해 예외가 가진 모든 프로퍼티를 낱알이 살펴봐야 한다.
 - 하이버네이트는 다양한 예외를 제공하지만 하이버네이트에 특화되어 있다. (플랫폼 독립적인 예외로 다시 던져야 함)
 - JDBC의 예외 계층 구조는 너무 포괄적이다.

스프링의 독립적인 예외

- 예외가 발생한 문제상황을 잘 설명하는 다양한 데이터 액세스 예외들 제공
ex) [표10.1] JDBC 예외 vs 스프링 데이터 액세스 예외 (p344)
- 퍼시스턴스 솔루션에 독립적인 일관성 있는 예외 (퍼시스턴스 기술을 데이터 액세스 계층 내에 캡슐화)

스프링의 비검사형(unchecked) 예외

- 스프링 데이터 액세스 예외는 `DataAccessException`을 확장한 것이다.
- `DataAccessException`은 비검사형 예외 이다.
- 스프링이 던지는 모든 데이터 액세스 예외는 반드시 잡아서 처리할 필요가 없다. (예외를 잡을지 말지는 개발자가 결정)
- 이러한 이점을 취하기 위해서는 반드시 스프링이 제공하는 데이터 액세스 템플릿을 이용해야 한다.

참고: 역자 한마디 "스프링의 핵심 장점 중 하나는 검사형 예외를 비검사형 예외로 변환해 주는 것" (p345)

- **검사형(checked) 예외**
 - throws 된다고 선언된 메소드는 호출시 반드시 try-catch 블록으로 감싸야만 컴파일 가능
 - ex) `java.sql.SQLException`, `java.io.IOException` 등
- **비검사형(unchecked) 예외**
 - try-catch 블록을 사용하지 않아도 컴파일이 되는 예외 (런타임 예외, 실행 시간 예외)
 - ex) `java.lang.RuntimeException`, `java.lang.Error`를 확장한 `NullPointerException`, `IllegalArgumentException` 등
- **스프링은 비검사형 예외를 강력히 지지 한다.**
 - 역자 또한 자바 API를 포함한 많은 라이브러리가 검사형 예외를 남용한다고 생각한다.
 - 스프링은 많은 예외가 catch 블록 내에서 해결할 수 없는 문제 때문에 발생한다고 보고 있다.
 - catch 블록의 작성을 강제하는 대신 비검사형 예외를 사용하도록 촉진한다.
 - 여타 프레임워크가 가진 검사형 예외를 비검사형 예외로 변환해주는 프레임워크는 스프링이 거의 유일하다.

10-1-2 데이터 액세스 템플릿화

ex) 비행기 수하물 이동

템플릿 메소드(Template Method) 패턴

- 어떤 절차의 골격을 정의
- 어떤 절차의 구현 종속적인 부분을 특정 인터페이스에 위임
- 인터페이스를 다르게 구현함으로써 위임된 부분에 특화된 부분을 정의
- 스프링이 데이터 액세스에 적용한 패턴

스프링 데이터 액세스 절차

- 고정된 단계와 가변적인 단계를 템플릿(template)과 콜백(callback)이라는 두 가지의 별도의 클래스로 분리
 - **템플릿 클래스**: 트랜잭션 제어, 자원 관리 및 예외 처리와 같은 데이터 액세스의 고정된 부분 담당
 - **콜백(callback) 클래스**: 질의객체(statement) 생성, 파라미터 바인딩, 질의 결과 추출과 변환 등
- 스프링의 데이터 액세스 템플릿 클래스는 공통적인 데이터 액세스 의무에 대한 책임을 진다.
- 애플리케이션에 특화된 작업을 처리하기 위해서 템플릿 클래스는 맞춤형 콜백 객체를 호출한다.
- 데이터 액세스 로직에만 집중할 수 있다.

[표10.2] 스프링 데이터 액세스 템플릿 종류 (p348)

스프링은 퍼시스턴스 플랫폼에 따라 선택할 수 있는 다양한 템플릿을 제공한다.

템플릿 클래스 (<code>org.springframework.*</code>)	용도
<code>jca.cci.core.CciTemplate</code>	JCA CCI 연결
<code>jdbc.core.JdbcTemplate</code>	JDBC 연결
<code>jdbc.core.namedparam.NamedParameterJdbcTemplate</code>	명명된 파라미터 (named parameter)가 지원되는 JDBC 연결
<code>jdbc.core.simple.SimpleJdbcTemplate</code>	자바5를 활용해서 더 쉬워진 JDBC 연결 (스프링3.1에선 없어짐)
<code>orm.hibernate3.HibernateTemplate</code>	Hibernate 3.x 세션
<code>orm.ibatis.SqlMapClientTemplate</code>	iBATIS SqlMap 클라이언트
<code>orm.jdo.JdoTemplate</code>	JDO(Java Data Object) 구현체
<code>orm.jpa.JpaTemplate</code>	JPA(Java Persistence API) 엔티티 관리자

- 10장 JDBC: 가장 기본적인 방법
- 11장 하이버네이트, JPA: POJO 기반 ORM(Object-Relational Mapping) 솔루션
- 12장 NoSQL: 비스키마(schemaless) 데이터

10.2 데이터 소스 설정

사용자가 사용하는 스프링 지원 데이터 액세스가 어떤 형태라도 데이터 소스에 대한 레퍼런스 설정이 필요함

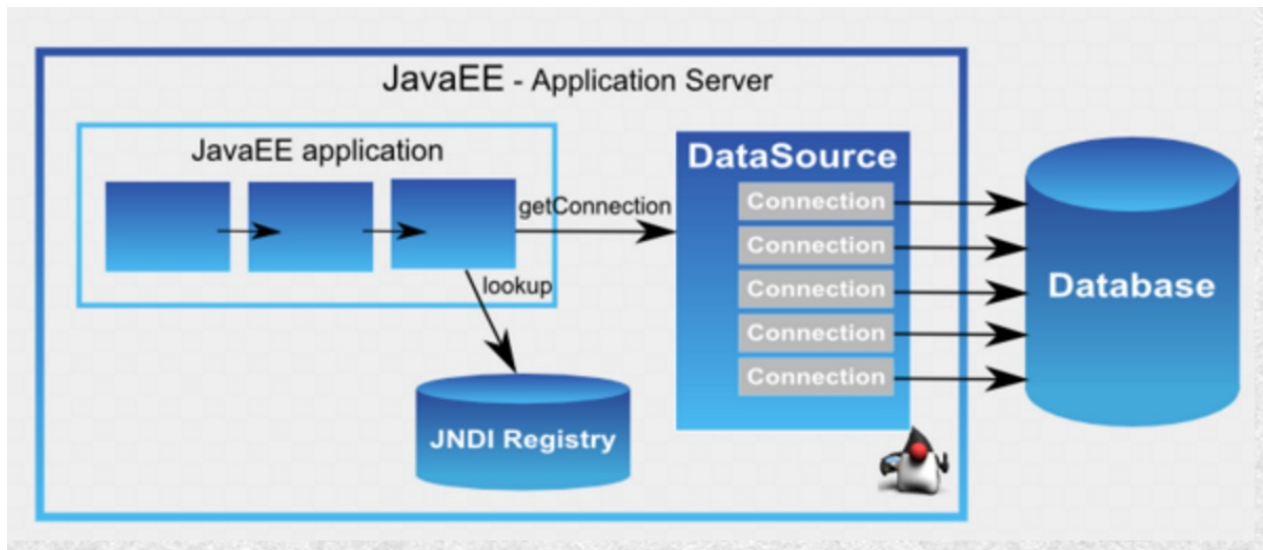
- JDBC 드라이버를 통해 선언된 데이터 소스
- JNDI에 등록된 데이터 소스
- 커넥션 풀링(pooling)하는 데이터 소스
 - 상용 application에서는 커넥션 풀에서 커넥션을 획득하는 데이터 소스를 권장

10.2.1 JNDI 데이터 소스 이용

- 스프링 애플리케이션은 대부분 JEE 애플리케이션 서버에 배포 되어서 사용됨.

- 이런 서버들은 WebSphere나 JBoss 또는 Tomcat 과 같은 웹컨테이너 역할을 함

JNDI 설명 (Java Naming and Directory Interface)



- server.xml 등 서버에 DataSource 에 대한 몇몇 설정이 필요함
- 장점
 - 데이터 소스가 애플리케이션 외부에서 관리
 - 애플리케이션에서는 데이터소스에 접근할 시점에 단순히 데이터소스를 요청하기만 하면 됨
 - 서버에 의해 관리되는 데이터소스는 향상된 성능을 위해 풀링 기능을 갖기도 하고 시스템 관리자가 운영시 동적으로 교체 가능함

애플리케이션 설정 방식

```
<jee:jndi-lookup id="datasource" jndi-name="/jdbc/SplitterDS"
resource-ref="true" />
```

```

@Bean
public JndiObjectFactoryBean dataSource() {
    JndiObjectFactoryBean jndiObjectFB = new JndiObjectFactoryBean();
    jndiObjectFB.setJndiName("jdbc/SplitterDS");
    jndiObjectFB.setResourceRef(true);
    jndiObjectFB.setProxyInterface(javax.sql.DataSource.class);

    return jndiObjectFB;
}

```

- 자바 설정을 사용한다면 JNDI에서 DataSource를 검색하기 위해서 JndiObjectFactoryBean을 사용함

10.2.2 풀링 기능이 있는 데이터 소스 사용하기

JNDI 를 사용하지 않을 경우, 풀링 기능이 있는 데이터 소스를 스프링에 직접 설정 가능함

- 스프링 자체가 풀링 기능을 제공하지는 않음
- 오픈 소스를 사용해서 풀링 가능한 데이터 소스 활용을 할 수 있음
 - Apache 공통 DBCP (<http://jakarta.apache.org/commons/dbcp>)
 - c3p0 (<http://sourceforge.net/c3p0>)
 - BoneCP (<http://jolbox.com>)

```

<bean id="datasource" class="org.apache.commons.dbcp.BasicDataSource"
    p:driverClassName="org.h2.Driver"
    p:url="jdbc:h2:tcp://localhost/~spitter"
    p:username="sa"
    p:password=""
    p:initailSize="5"
    p:maxActive="10" />

```

```

@Bean
public BasicDataSource dataSource() {
    BasicDataSource ds = new BasicDataSource();
    ds.setDriverClassName("org.h2.Driver");
    ds.setUrl("jdbc:h2:tcp://localhost/~spitter");
    ds.setUsername("sa");
    ds.setPassword("");
    ds.setInitialSize(5);
    ds.setMaxActive(10);
    return ds;
}

```

BasicDataSource 의 풀 프로퍼티

풀 설정 프로퍼티	설명
initialSize	해당 풀이 시작될 때 생성할 커넥션 수
maxActive	해당 풀에서 동시에 제공할 수 있는 최대 커넥션 수. 0은 무제한을 나타낸다
maxIdle	해당 풀에서 동시에 휴면 상태로 유지될 수 있는 최대 커넥션 수. 0은 무제한을 나타낸다
maxOpenPreparedStatements	질의 객체(statement) 풀에서 동시에 제공할 수 있는 최대 PreparedStatement 의 수. 0은 무제한을 나타낸다
maxWait	해당 풀에 커넥션을 요청했을 때(제공 받을 수 있는 커넥션이 없어서) 대기 가능한 최대 시간. 이 시간이 지나면 예외가 발생한다. 1은 무한히 대기함을 의미한다.
minEvictableIdleTimeMillis	해당 풀에서 커넥션을 제거하기 전에 휴면 상태로 남아 있을 수 있는 시간
minIdle	해당 풀에서 커넥션이 휴면 상태로 유지될 수 있는 최소 커넥션 수
poolPreparedStatements	PreparedStatement 의 풀링(pooling) 여부를 나타내는 boolean 값

[참고] Statement vs PreparedStatement

Statement

executeQuery() 나 executeUpdate() 를 실행하는 시점에 파라미터로 SQL문을 전달하는데, 이 때 전달되는 SQL 문은 완성된 형태로 한눈에 무슨 SQL 문인지 파악하기 쉽다. 하지만, 이 녀석은 SQL문을 수행하는 과정에서 매번 컴파일을 하기 때문에 성능상 이슈가 있다. (이 컴파일을 Parsing 한다고도 표현한다.)

```
String sql = "select * from users where _id=1";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery( sql );
```

PreparedStatement

미리 컴파일 되기 때문에 Statement 에 비해 성능상 이점이 있다.

```
String sql = "select * from users where _id=?";
PreparedStatement pstmt = conn.prepareStatement( sql );
pstmt.setInt( 1, 1 );
ResultSet rs = pstmt.executeQuery();
```

10.2.3 JDBC 드라이버 기반 데이터 소스

스프링에 설정할 수 있는 가장 단순한 데이터 소스는 JDBC 드라이버를 통해 정의 (스프링 내 org.springframework.jdbc.datasource 패키지 안에 존재)

JDBC Driver	설명	Pooling 기능 제 공
DriverManagerDataSource	애플리케이션이 커넥션 요청 시마다 새로운 커넥션을 반환	풀링 지 원 안함
SimpleDriverDataSource	OSGi 컨테이너와 같이 특정 환경에서 발생할 수 있는 클래스 로딩 문제를 극복하기 위해 사용하는 것을 제외하면 DriverManagerDataSource와 동일	풀링 지 원 안함
SingleConnectionDataSource	항상 동일 커넥션을 반환	오직 한 커넥션 만을 풀 링

```
<bean id="datasource" class="org.springframework.jdbc.datasource.
DriverManagerDataSource"
    p:driverClassName="org.h2.Driver"
    p:url="jdbc:h2:tcp://localhost/~spitter"
    p:username="sa"
    p:password="" />
```

- 풀링 설정 프로퍼티가 존재 안함

```
@Bean
public BasicDataSource dataSource() {
    DriverManagerDataSource ds = new DriverManagerDataSource();
    ds.setDriverClassName("org.h2.Driver");
    ds.setUrl("jdbc:h2:tcp://localhost/~spitter");
    ds.setUsername("sa");
    ds.setPassword("");
    return ds;
}
```

Guide

- DriverManagerDataSource, SingleConnectionDataSource : 작은 규모의 애플리케이션과 개발 단계에서는 데이터 소스가 좋은 선택이 될 수 있지만,
- 상용 애플리케이션에서는 심각한 성능 저하를 고려하여 커넥션 풀링이 있는 데이터 소스를 사용할 것을 강력하게 권장함

10.2.4 임베디드 데이터 소스 사용하기

- 사용자가 사용할 추가 데이터 소스로 임베디드 데이터베이스가 있음.
- 애플리케이션이 연결하는 독립 데이터베이스 서버 대신에 임베디드 데이터 베이스가 애플리케이션 중 하나로 동작

<jdbc:embedded-database> type 프로퍼티를 사용함

10.2.5 데이터 소스 선택을 위한 프로파일링하기

- 개발, 사용 단계에서의 데이터 소스 선택이 필요한 상황이 있음
- 런타임 시 스프링의 데이터 소스 선택을 위한 프로파일링이 필요

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:jdbc="http://www.springframework.org/schema/jdbc"
        xmlns:jee="http://www.springframework.org/schema/jee"
        xsi:schemaLocation="...">

    <bean id="accountRepository" class="com.bank.repository.internal.JdbcAccountRepository">
        <constructor-arg ref="dataSource"/>
    </bean>

    <beans profile="dev">
        <jdbc:embedded-database id="dataSource">
            <jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>
            <jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>
        </jdbc:embedded-database>
    </beans>
    <beans profile="production">
        <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
    </beans>
</beans>
```

```

@Configuration
public class DataSourceConfiguration {

    @Profile("dev")
    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql"
        )
            .addScript("classpath:com/bank/config/sql/test-data.
sql")
            .build();
    }

    @Profile("production")
    @Bean
    public DataSource dataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datas
ource");
    }
}

```

10.3 스프링과 JDBC

10.3.1 지저분한 JDBC 코드 해결

- cf. jdbc example

```

//STEP 1. Import required packages
import java.sql.*;

public class JDBCExample {
    // JDBC driver name and database URL
    static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
    static final String DB_URL = "jdbc:mysql://localhost/STUDENTS"

```



```
;

// Database credentials
static final String USER = "username";
static final String PASS = "password";

public static void main(String[] args) {
    Connection conn = null;
    Statement stmt = null;
    try{
        //STEP 2: Register JDBC driver
        Class.forName("com.mysql.jdbc.Driver");

        //STEP 3: Open a connection
        System.out.println("Connecting to a selected database...");
;
        conn = DriverManager.getConnection(DB_URL, USER, PASS);
        System.out.println("Connected database successfully...");

        //STEP 4: Execute a query
        System.out.println("Inserting records into the table...");
        stmt = conn.createStatement();

        String sql = "INSERT INTO Registration " +
            "VALUES (100, 'Zara', 'Ali', 18)";
        stmt.executeUpdate(sql);
        sql = "INSERT INTO Registration " +
            "VALUES (101, 'Mahnaz', 'Fatma', 25)";
        stmt.executeUpdate(sql);
        sql = "INSERT INTO Registration " +
            "VALUES (102, 'Zaid', 'Khan', 30)";
        stmt.executeUpdate(sql);
        sql = "INSERT INTO Registration " +
            "VALUES(103, 'Sumit', 'Mittal', 28)";
        stmt.executeUpdate(sql);
        System.out.println("Inserted records into the table...");

    }catch(SQLException se){
        //Handle errors for JDBC
        se.printStackTrace();
    }
}
```

```
}catch(Exception e){
    //Handle errors for Class.forName
    e.printStackTrace();
}finally{
    //finally block used to close resources
    try{
        if(stmt!=null)
            conn.close();
    }catch(SQLException se){
    }// do nothing
    try{
        if(conn!=null)
            conn.close();
    }catch(SQLException se){
        se.printStackTrace();
    }//end finally try
    }//end try
    System.out.println("Goodbye!");
}//end main
}//end JDBCExample
```

```
package com.vaannila.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.sql.DataSource;

import com.vaannila.domain.Forum;

public class JDBCForumDAOImpl implements ForumDAO {

    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

```
}

public void insertForum(Forum forum) {
    /**
     * Specify the statement
     */
    String query = "INSERT INTO FORUMS (FORUM_ID, FORUM_NAME
, FORUM_DESC) VALUES (?, ?, ?)";
    /**
     * Define the connection and preparedStatement parameter
S
     */
    Connection connection = null;
    PreparedStatement preparedStatement = null;
    try {
        /**
         * Open the connection
         */
        connection = dataSource.getConnection();
        /**
         * Prepare the statement
         */
        preparedStatement = connection.prepareStatement(quer
y);
        /**
         * Bind the parameters to the PreparedStatement
         */
        preparedStatement.setInt(1, forum.getForumId());
        preparedStatement.setString(2, forum.getForumName())
;
        preparedStatement.setString(3, forum.getForumDesc())
;

        /**
         * Execute the statement
         */
        preparedStatement.execute();
    } catch (SQLException e) {
        /**
         * Handle any exception
         */
    }
}
```

```
        e.printStackTrace();
    } finally {
        try {
            /**
             * Close the preparedStatement
             */
            if (preparedStatement != null) {
                preparedStatement.close();
            }
            /**
             * Close the connection
             */
            if (connection != null) {
                connection.close();
            }
        } catch (SQLException e) {
            /**
             * Handle any exception
             */
            e.printStackTrace();
        }
    }
}

}
```

- 간단한 테이블 입력 하나에도 너무나 많은 Exception 코드가 들어 있다.
- 실질적인 작업 수행은 단순한 몇개의 줄로만 이루어진다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd"
>

    <bean id="dataSource" destroy-method="close" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
        <property name="url" value="jdbc:hsqldb:hsqldb://localhost" />
        <property name="username" value="sa"/>
        <property name="password" value=""/>
    </bean>

    <bean id="forumDAO" class="com.vaannila.dao.ForumDAOImpl">
        <property name="dataSource" ref="dataSource"/>
    </bean>

</beans>
```

10.3.2 JDBC 템플릿과 놀아보자

- 스프링의 JDBC 프레임워크는 자원 관리와 예외 처리를 처리함으로서 개발자는 오직 데이터 처리 부분 구현에 집중 가능
- 세개의 선택 가능한 JDBC 템플릿 클래스를 제공
 - JdbcTemplate : 스프링의 가장 기본적인 JdbcTemplate 으로, 색인된 파라미터(**indexed parameter**) 기반 의 쿼리를 통해서 데이터베이스에 쉽게 액세스 하는 기능을 제공
 - NamedParameterJdbcTemplate : 명명된 파라미터(**named parameter**)로 바인딩하여 쿼리 실행
 - SimpleJdbcTemplate : 스프링 버전 3.1에서 삭제

```
// JDBC를 사용하는 DAO 클래스...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfActors(Actor exampleActor) {

    // 이름있는 파라미터들이 어떻게 위의 'Actor' 클래스의 프로퍼티에서 일치되는
    // 것을 찾는지 주의깊게 보라.
    String sql = "select count(*) from T_ACTOR where first_name = :firstName and last_name = :lastName";

    SqlParameterSource namedParameters = new BeanPropertySqlParameterSource(exampleActor);

    return this.namedParameterJdbcTemplate.queryForInt(sql, namedParameters);
}
```

```
package com.vaannila.dao;

import java.sql.ResultSet;
import java.sql.SQLException;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;

import com.vaannila.domain.Forum;

public class ForumDAOImpl implements ForumDAO {

    private JdbcTemplate jdbcTemplate;
```

```

public void setDataSource(DataSource dataSource) {
    this.jdbcTemplate = new JdbcTemplate(dataSource);
}

public Forum selectForum(int forumId) {
    /**
     * Specify the statement
     */
    String query = "SELECT * FROM FORUMS WHERE FORUM_ID=?";
    /**
     * Implement the RowMapper callback interface
     */
    return (Forum) jdbcTemplate.queryForObject(query, new Object[] { Integer.valueOf(forumId) },
        new RowMapper() {
            public Object mapRow(ResultSet resultSet, int rowNum) throws SQLException {
                return new Forum(resultSet.getInt("FORUM_ID"), resultSet.getString("FORUM_NAME"),
                    resultSet.getString("FORUM_DESC"));
            }
        });
}

```

- RowMapper를 통해 재사용성을 높일 수 있음
 - org.springframework.jdbc.core.RowMapper

```

        (Forum) jdbcTemplate.queryForObject(query, new Object[] { Integer.valueOf(forumId) },
            new ForumRowMapper());

```

```
package com.javarticles.spring.integration.jdbc;

import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class ForumRowMapper implements RowMapper<Forum> {

    public Article mapRow(ResultSet rs, int rowNum) throws SQLException {
        return new Forum(resultSet.getInt("FORUM_ID"), resultSet
            .getString("FORUM_NAME"),
                           resultSet.getString("FORUM_DESC"
        ));
    }
}
```

- main method


```
package com.vaannila.dao;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplication
onContext;

import com.vaannila.domain.Forum;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplication
Context("beans.xml");
        ForumDAO forumDAO = (ForumDAO) context.getBean("forumDAO"
);
        Forum springForum = new Forum(1, "Spring Forum", "Discuss
everything related to Spring");
        forumDAO.insertForum(springForum);
        System.out.println(forumDAO.selectForum(1));

    }

}
```

Chapter11. 객체 관계형 매핑을 통한 데이터

- 객체 관계형 매핑 - ORM(Object-Relational Mapping)
 - 관계 모델과 객체 모델을 연결
 - 장점
 - 코드 양과 개발 시간을 단축할 수 있다.
 - 오류가 발생하기 쉬운 SQL 코드 작성을 하지 않아도 된다.
 - 특징
 - 지연 로딩(lazy loading)
 - 객체의 프로퍼티 중 실제로 필요한 프로퍼티만 가져올 수 있다.
 - 조기 인출(eager fetching)
 - 객체의 프로퍼티를 단 한번의 연산으로 모두 가져올 수 있다.
 - 캐스케이딩(cascading)
 - 테이블의 변경에 따라 영향을 받는 여러 테이블을 한꺼번에 변경할 수 있다.
 - 프레임워크들
 - 하이버네이트, iBATIS, Apache OJB, JDO, Oracle TopLink, JPA 등
- 스프링에서 JDBC는 잘 작동하지만, 복잡한 어플리케이션의 객체지향 요구 사항을 매끄럽게 따르기 위해 ORM을 사용한다.

11.1 스프링과 하이버네이트 통합

11.1.1 하이버네이트 세션 팩토리 선언

- Session과 SessionFactory
 - **Session**: 저장, 업데이트, 삭제 등의 기본적인 데이터 접근 기능을 제공한다.

```

Session session = sessionFactory.openSession();
session.beginTransaction();
session.save( new Event( "Our very first event!", new Date() ) );
session.save( new Event( "A follow up event", new Date() ) );
session.getTransaction().commit();
session.close();

```

- **SessionFactory**: Session 객체들을 관리한다.

```

StandardServiceRegistry standardRegistry = new StandardServiceRegistryBuilder()
    .configure( "org/hibernate/example/hibernate.cfg.xml" )
    .build();

Metadata metadata = new MetadataSources( standardRegistry )
    .addAnnotatedClass( MyEntity.class )
    .addAnnotatedClassName( "org.hibernate.example.Customer" )
    .addResource( "org/hibernate/example/Order.hbm.xml" )
    .addResource( "org/hibernate/example/Product.orm.xml" )
    .getMetadataBuilder()
    .applyImplicitNamingStrategy( ImplicitNamingStrategyJpaCompliantImpl.INSTANCE )
    .build();

SessionFactory sessionFactory = metadata.getSessionFactoryBuilder()
    .applyBeanManager( getBeanManager() )
    .build();

```

- 참고: Hibernate ORM User Guide

(http://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#bootstrap-jpa)

- 스프링에서 SessionFactory 가져오기
 - XML : **LocalSessionFactoryBean**을 사용한다.

```
@Bean
public LocalSessionFactoryBean sessionFactory(dataSource
dataSource) {
    LocalSessionFactoryBean sfb = new LocalSessionFactory
    Bean();

    sfb.setDataSource(dataSource);
    sfb.setMappingResources(new String[] { "Spitter.hbm.x
    ml" });
    // 버전 4이상 annotation: sfb.setPackagesToScan(new St
    ring[] { "com.habuma.spittr.domain" });

    ...

    return sfb;
}
```

- hbm.xml 예제

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.
dtd">

<hibernate-mapping>
    <class name="Employee" table="EMPLOYEE">
        <meta attribute="class-description">
            This class contains the employee detail.
        </meta>
        <id name="id" type="int" column="id">
            <generator class="native"/>
        </id>
        <property name="firstName" column="first_name"
type="string"/>
        <property name="lastName" column="last_name" ty
pe="string"/>
        <property name="salary" column="salary" type="i
nt"/>
    </class>
</hibernate-mapping>
```

- Annotation

- 버전 3: **AnnotationSessionFactoryBean**을 사용한다.
- 버전 4: **LocalSessionFactoryBean**을 사용한다.

```

@Bean
public AnnotationSessionFactoryBean sessionFactory(DataSource dataSource) {
    AnnotationSessionFactoryBean sfb = new AnnotationSessionFactoryBean();

    sfb.setDataSource(dataSource);
    sfb.setPackagesToScan(new String[] { "com.habuma.spitter.domain" });

    ...

    return sfb;
}

```

- setPackagesToScan에 준 패키지안에서 @Entity나 @MappedSuperclass 등의 annotation을 찾는다.

- Entity 예제

```

@Entity
public class Employee {
    @Id
    private int id;
    private String firstName;
    private String lastName;
    private int salary;
}

```

- 클래스가 적을 때는 setAnnotatedClasses를 이용해 도메인 클래스를 나열할 수 있다.

```

sfb.setAnnotatedClasses(new Class<?>[] { Spitter.class, Spittle.class });

```

11.1.2 스프링으로부터 해방된 하이버네이트 구성

- 예전에는 스프링의 `HibernateTemplate`를 사용하도록 저장소 클래스를 만들었다.
 - 예제

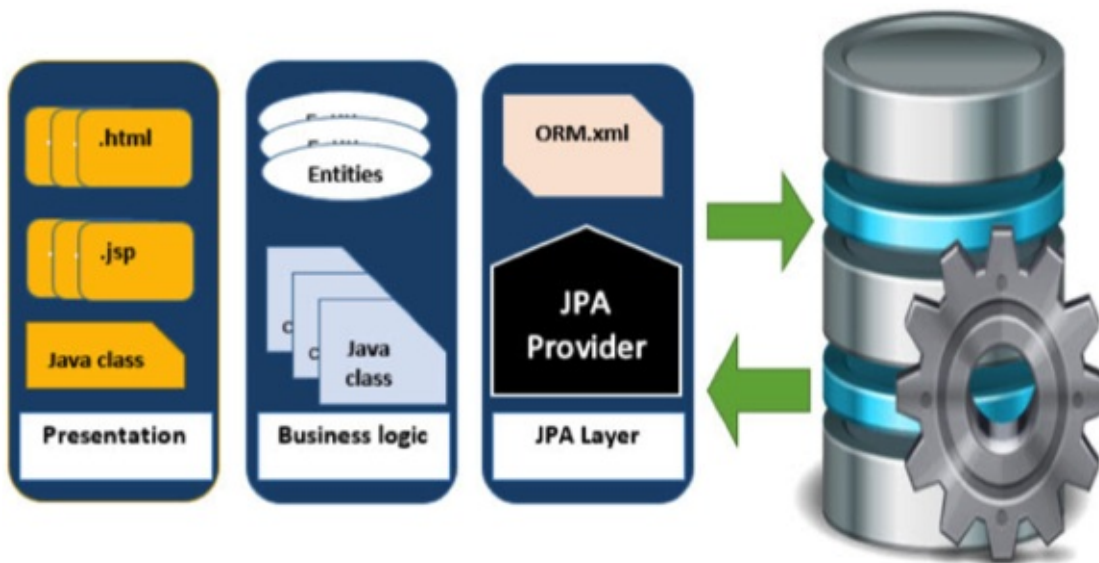
```
public class ProductDaoImpl implements ProductDao {
    private HibernateTemplate hibernateTemplate;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.hibernateTemplate = new HibernateTemplate(sessionFactory);
    }

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        return this.hibernateTemplate.execute(new HibernateCallback() {
            public Object doInHibernate(Session session) {
                Criteria criteria = session.createCriteria(Product.class);
                criteria.add(Expression.eq("category", category));
                criteria.setMaxResults(6);
                return criteria.list();
            }
        });
    }
}
```

- 요즘은 `HibernateTemplate`을 쓰지 않고 직접 저장소 클래스 안에서 `SessionFactory`를 와이어링한다.
 - 스프링과 분리된 하이버네이트 저장소를 구현할 수 있다.
- 하이버네이트에 특화된 예외를 스프링의 예외로 변환하는 방법
 - `PersistenceExceptionTranslationPostProcessor` 빈을 만든다.
 - 이 빈은 `@Repository`가 적용된 모든 빈의 플랫폼에 특화된 예외를 스프링의 예외로 변환한다.

11.2 스프링과 자바 퍼시스턴스 API



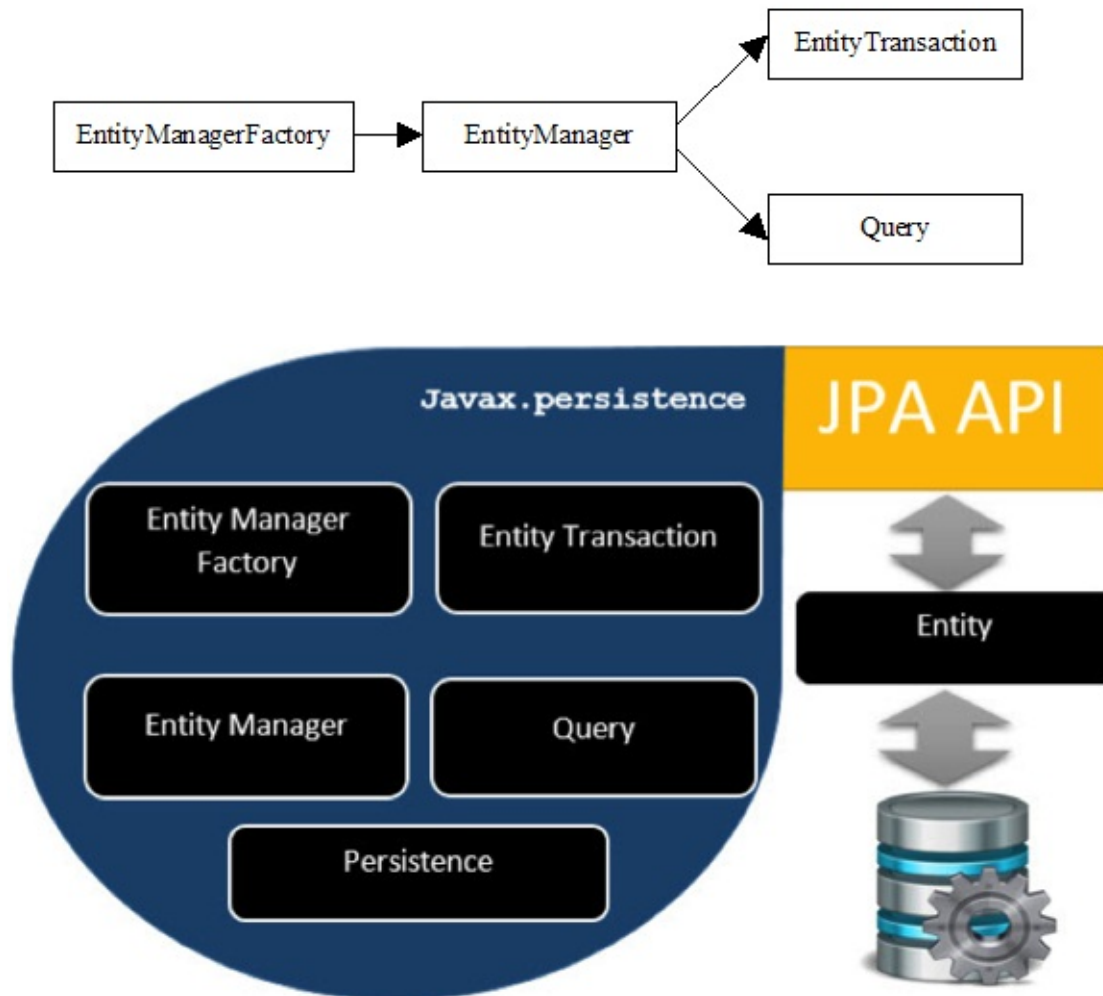
- JPA
 - 차세대 자바 퍼시스턴스 표준!
 - POJO 기반의 퍼시스턴스 메카니즘

The Java Persistence API (JPA) is a Java application programming interface specification that describes the management of relational data in applications using Java Platform, Standard Edition and Java Platform, Enterprise Edition.

Motivation

Prior to the introduction of EJB 3.0 specification, many enterprise Java developers used lightweight persistent objects, provided by either persistence frameworks (for example Hibernate) or data access objects instead of entity beans. This is because entity beans, in previous EJB specifications, called for too much complicated code and heavy resource footprint, and they could be used only in Java EE application servers because of interconnections and dependencies in the source code between beans and DAO objects or persistence framework. Thus, many of the features originally presented in third-party persistence frameworks were incorporated into the Java Persistence API, and, as of 2006, projects like Hibernate (version 3.2) and TopLink Essentials have become themselves implementations of the Java Persistence API specification.

11.2.1 엔티티 관리자 팩토리 설정



- EntityManager의 두가지 유형
 - Application-managed
 - 어플리케이션이 직접 엔티티 관리자를 요청함으로써 엔티티 관리자가 생성되는 유형
 - 직접 엔티티관리자를 다뤄야하며 트랜잭션처리까지 직접 책임
 - Container-managed
 - 자바 ee 컨테이너에 의해 생성되고 관리(책임)되는 엔티티 관리자
 - 어플리케이션이 직접 엔티티관리자와 상호작용하지 않고 객체주입같은 방식으로 획득하는 방식

즉, EntityManager가 생성되고 관리되는 방식 차이 하지만 어차피 Spring이 EntityManager를 관리하고 책임지어 준다. 스프링은 위 유형에 대응하는 Factory빈을 갖추고 있어서 적절하게 엔티티관리자 팩토리를 생성한다.

Entity Factory Bean

- LocalEntityManagerFactoryBean - 어플리케이션 관리형
- LocalContainerEntityManagerFactoryBean - 컨테이너 관리형

스프링 입장에서는 두 방식의 차이점이라면 **스프링 어플리케이션 컨텍스트에서 설정하는 방법**뿐

어플리케이션 관리형 JPA 구성하기

- persistence.xml
- 퍼시스턴스 클래스와 데이터 소스 정보, 매핑 파일 이름과 같은 기본적인 메타 데이터 설정 정보를 기술

persistence.xml

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">

    <persistence-unit name="org.hibernate.tutorial.jpa" transaction-type="RESOURCE_LOCAL">
        <description>
            Persistence unit for the JPA tutorial of the Hibernate Getting Started Guide
        </description>
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <class>org.halyph.sessiondemo.Event</class>

        <properties>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/jpatestdb" />
            <property name="javax.persistence.jdbc.user" value="root" />
            <property name="javax.persistence.jdbc.password" value="root" />

            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.hbm2ddl.auto" value="create" />
        </properties>

    </persistence-unit>

</persistence>
```

Bean 설정

```
@Bean
public LocalEntityManagerFactoryBean entityManagerFactoryBean()
{
    LocalEntityManagerFactoryBean emfb = new LocalEntityManagerFac
toryBean();
    emfb.setPersistenceUnitName("org.hibernate.tutorial.jpa");
    return emfb;
}
```

- persistece.xml로 빼놓음으로써 스프링과의 연관관계, 의존성을 최소화할 수 있다.
- 하지만, 동시에 단점이 되며 단점의 요소가 반복적으로 불편하게 만듦.
 - jpaTemplate를 통해서 데이터에 접근해야한다.(반복적 코드 상승)
 - 스프링의 bean을 이용할 수 없다.

컨테이너 관리형 JPA 구성하기

- 스프링 컨텍스트

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerF
actory(DataSource ds, JpaVenderAdapter jpaVenderAdapter) {
    LocalContainerEntityManagerFactoryBean emfb = new LocalCon
tainerEntityManagerFactoryBean();
    emfb.setDataSource(ds);
    emfb.setJpaVendorAdapter(jpaVenderAdapter);
    return emfb;
}
```

- jpaVenderAdapter 프로퍼티는 사용하는 특정 jpa 구현체에 특화된 정보 제공
 - EclipseLinkJpaVendorAdapter
 - HibernateJpaVendorAdapter
 - OpenJpaVendorAdapter
 - TopLinkJpaVendorAdapter

HibernateJpaVendorAdapter

```

@Bean
public JpaVendorAdapter jpaVendorAdapter() {
    HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
    adapter.setDatabase("HSQL");
    adapter.setShowSql(true);
    adapter.setGeneralDdl(false);
    adapter.setDatabasePlatform("org.hibernate.dialect.HSQLDialect");
    return adapter;
}

```

- 컨테이너 관리형 jpa를 구성하면 런타임의 동적인 처리를 퍼시스턴스 객체의 클래스에 조작해야한다.
 - Lazy Loading, Eager Loading, 관계등..
- packageToScan으로 엔티티 클래스를 간편하게 구별할 수 있다.
 - @Entity 어노테이션된 클래스를 지정된 패키지에서 스캔한다.
 - persistence.xml에 명시적으로 상세히 선언할 필요가 없다.

11.2.2 JPA 기반 저장소 작성

- 템플릿 기반의 JPA보다 순수 JPA 선호

```

@Repository
@Transactional
public class JpaSpittleRepository implements SpittleRepository{

    @PersistenceContext
    private EntityManager em; //EntityManager 주입

    public void addSpitter(Spitter spitter) {
        em.persist(spitter); //EntityManager 사용
    }

}

```

- **@PersistenceContext**
 - 사용해서 직접 엔티티매니저를 생성할 필요없다.
 - EntityManager를 주입하지 않음.
- **@PersistenceContext** 와 **@PersistenceUnit**의 차이
 - 둘다 스프링 어노테이션이 아니다. JPA 스펙
 - 스프링과 엔티티매니저를 이해하고 주입하기 위해선 스프링의 PersistenceAnnotationBeanPostProcessor를 구성 필요
 - `<context:annotation-config/>` , `<context:component-scan/>` 으로 자동적으로 구성
- **@Transactional**
- **@Repository**

PersistenceExceptionTranslationPostProcessor

- JPA, 하이버네이트에 특화된 예외가 발생되도록 하려면 위의 빈 등록

```
@Bean
public BeanPostProcessor persistenceTranslation() {
    return new PersistenceExceptionTranslationPostProcessor();
}
```

11.3 스프링 데이터를 사용한 자동 JPA 저장소

기존의 DB 질의(쿼리)를 하기 위해선 EntityManager와 직접 상호연동해야 했다. 직접 연동과 부가적인 코드를 줄이기 위해 spring data jpa는 JpaRepository를 제공한다.

JpaRepository

```
public interface SpitterRepository extends JpaRepository<Spitter, Long> {}
```

- spring data jpa 의 JpaRepository를 확장
- 일반적인 퍼시스턴스 작업을 수행하기 위한 여러 방법을 상속한다.
- 스프링 컨텍스트가 생성되는 것처럼, 어플리케이션 시작 타임에 저장소 구현이 이루어진다.

스프링 데이터 JPA 설정

xml config

```
<jpa:repositories base-package="com.study.spring.db" />
```

java config

```
@Configuration
@EnableJpaRepositories(basePackage="com.study.spring.db")
public class JpaConfiguration {...}
```

11.3.1 쿼리 메소드 정의하기

새로운 질의(쿼리) 추가

```
public interface SpitterRepository extends JpaRepository<Spitter
, Long> {
    Spitter findByUsername(String username);
}
```

- 단순 메소드 시그니처로 메소드 구현체를 만들기 위한 정보를 스프링 데이터 JPA에 전달.
- 구현체를 만들 시점에, 저장소 인터페이스의 메소드를 검증하고 파싱한다.

findByUsername()

- 동사, 선택 대상, 조건으로 구성
- 동사 : find, 조건 : Username, 선택대상 : Spitter
- 대상은 대부분 무시된다.
 - JpaRepository 인터페이스의 타입 파라미터에 의해 정해진다.

Example

```
List<Spitter> readByFirstnameOrLastname(String first, String last);
```

대소문자 무시

```
List<Spitter> readByFirstnameOrLastnameIgnoresCase(String first,
String last);
```

정렬

```
List<Spitter> readByFirstnameOrLastnameOrderByLastnameAsc(String
first, String last);
```

조건 파트는 AND 또는 OR 로 분리

11.3.2 맞춤형 쿼리 선언하기

위의 쿼리메소드의 명명규칙이 지원하지않는 복잡한 쿼리를 작성할 경우는 맞춤형 쿼리 를 사용한다.

@Query

```
@Query("select s from Spitter s where s.email like '%gmail.com'")
)
List<Spitter> findAllGmailSpitters();
```

- 구현체가 따로 없어도 @Query 메소드에 있는 쿼리를 실행.

The Java Persistence Query Language (JPQL) is a platform-independent object-oriented query language defined as part of the Java Persistence API (JPA) specification.

JPQL is based on the Hibernate Query Language (HQL), an earlier non-standard query language included in the Hibernate object-relational mapping library. Hibernate and the HQL were created before the JPA specification. As of Hibernate 3 JPQL is a subset of HQL.

11.3.3 맞춤형 기능 혼합

쿼리메소드와 맞춤형 쿼리도 한계가 있다. 그럴때는 EntityManager 로 직접 접근하여 질의(쿼리)를 작성할 경우가 있다.


```
public interface SpitterSweeper {  
  
    int eliteSweep();  
  
}
```

```
public interface SpitterRepository extends JpaRepository<Spitter  
, Long>, SpitterSweeper {  
  
    Spitter findByUsername(String username);  
  
    List<Spitter> findByUsernameOrFullNameLike(String username,  
String fullName);  
  
}
```

```
public class SpitterRepositoryImpl implements SpitterSweeper {  
  
    @PersistenceContext  
    private EntityManager em;  
  
    public int eliteSweep() {  
        String update =  
            "UPDATE Spitter spitter " +  
            "SET spitter.status = 'Elite' " +  
            "WHERE spitter.status = 'Newbie' " +  
            "AND spitter.id IN (" +  
            "SELECT s FROM Spitter s WHERE (" +  
            "    SELECT COUNT(spittles) FROM s.spittles spittle  
s) > 10000" +  
            ")";  
        return em.createQuery(update).executeUpdate();  
    }  
  
}
```

```
@Configuration
@EnableJpaRepositories("com.habuma.spitter.db", repositoryImplementationPostfix="Impl")
public class SpringDataJpaConfig { }
```

Chapter16. 스프링 MVC로 REST API 사용하기

16 스프링 MVC로 REST API 사용하기

데이터는 왕

- 소프트웨어는 대체되어도 수년간 쌓인 데이터는 대체할 수 없다
- SOAP 동작과 프로세싱에 집중
- REST 관심은 데이터 처리

스프링 3.0 부터 REST 작업에 대한 최고 수준의 지원을 제공 3.1, 3.2 4.0 에도 계속 됨

16.1 휴식(REST)을 취하다 (Getting REST)

- SOAP - 많은 애플리케이션에게 부담
- REST - 더 간편한 대안 제공
- 많은 사람들이 REST가 무엇인지 확실하게 이해하지 못한다.

16.1.1 REST의 기본 개념

- REST - SOAP과 같이 또 다른 RPC(Remote Procedure Call) 메커니즘? 아니다
- REST - 평범한 HTTP URL 을 통해 호출
- SOAP - 수많은 XML 네임스페이스를 이용
- RPC(Remote Procedure Call)와 거의 관련 없다.
- RPC - 서비스 지향적, 액션과 동사에 초점
- REST - 리소스 지향적이고 애플리케이션을 표현하는 객체와 명사를 강조
- 표현(Representational) - REST 리소스는 XML, JSON(JavaScript Object Notation), HTML 등 사실상 거의 모든 형식으로 표현

- 상태(State) - 리소스에 대해 액션보다 상태에 더 많은 관심
- 전달(Transfer) - REST는 한 애플리케이션에서 다른 애플리케이션으로 어떤 표현 형식으로 리소스 데이터 전달을 포함한다.
- URL에서 리소스 구별, 서버에 명령어를 보내지 않음

CRUD 매핑

- C 생성 - POST
- R 읽기 - GET
- U 업데이트 - PUT 또는 PATCH
- D 삭제 - DELETE



- 실제로 확인 해 보면 DELETE, GET, HEAD, OPTIONS, PATCH, POST, PUT, TRACE 가 있다.

16.1.2 스프링이 REST를 지원하는 방법

- 이전부터 가능. 스프링 3부터 개선. 4.0 기준 설명
- 컨트롤러 REST 네 가지 주요 메소드 - GET, PUT, DELETE, POST. PATCH 는 스프링 3.2 이상
- `@PathVariable` 파라미터화된 URL(경로의 일부분에 변수 입력이 있는 URL)에 대한 요청을 처리.

```
@RequestMapping("/dog/{name}")
public String dog(@PathVariable String name) {
```

- XML, JSON, Atom, RSS 등 다양한 방식으로 표현. 스프링의 뷰와 뷰 리졸버를 이용
- 클라이언트에 대한 가장 적합한 표현은 새로운 `ContentNegotiatingViewResolver` 를 이용해 선택
- `@ResponseBody` 애너테이션, 다양한 `HttpMessageConverter` 로 뷰 기반 렌더링 무시
- 마찬가지로 새로운 `@RequestBody` 애너테이션은 `HttpMethodConverter` 구현체와 함께 인바운드 HTTP 데이터를 컨트롤러의 핸들러 메소드에 전달하는 자바 객체로 변환
- `RestTemplate`은 클라이언트 측의 REST 리소스 사용을 간소화

16.2 첫 번째 REST 엔드포인트 만들기

- RESTful 컨트롤러 생성

```
@Controller
public class DemoController {
    @ResponseBody
    @RequestMapping(value="/hello2", method= RequestMethod.GET
    )
    public HashMap<String, Object> test2() {
        HashMap<String, Object> map = new HashMap<>();
        map.put("abc", "ddd");
        return map;
    }
}
```

- @RestController 이용하기. @RestController 는 @Controller 와 @ResponseBody 포함. 스프링 4.0

```
@RestController
public class DemoController {
    @RequestMapping(value="/hello3", method= RequestMethod.GET
    )
    public HashMap<String, Object> test3() {
        HashMap<String, Object> map = new HashMap<>();
        map.put("abcaa", "ddeeed");
        return map;
    }
}
```

- 최소한 json을 지원하기를 추천
- 실제로 Spring Boot 환경에서 별다른 설정 안하면 json 이 기본

스프링은 리소스의 자바 표현을 클라이언트에 전달될 표현으로 변환하는 두 가지 방법을 제공

- 콘텐츠 협상 (Content Negotiation) - 모델이 클라이언트에 제공되는 표현으로 렌더링될 수 있도록 뷰는 선택된다
- 메시지 변환 (Message Converter) - 메시지 변환기는 컨트롤러에서 반환된 객체

를 클라이언트에 제공되는 표현으로 변경

16.2.1 리소스 표현 협상

스프링의 `ContentNegotiatingViewResolver`는 클라이언트가 고려하려는 콘텐츠 타입을 선택하는 특별한 뷰 리졸버

```
@Bean
public ViewResolver cnViewResolver() {
    return new ContentNegotiatingViewResolver();
}
```

`ContentNegotiatingViewResolver`의 동작방식을 이해하려면 콘텐츠 협상의 두 단계를 알아야 한다.

1. 요청 미디어 타입 결정
2. 요청 미디어 타입에 대해 최적의 뷰 검색

요청 미디어 타입 결정

- `ContentNegotiatingViewResolver`: URL의 파일 확장자 > Accept 헤더 > 기본 콘텐츠 타입
- 확장자 ex) <http://.../dog/123.json>
 - .json 이면 `application/json`
 - .xml 타입이면 `application/xml`
 - .html `text/html`
- 단점: `PathVariable` 사용시 "." 들어간 인자 넣으면 "." 이후 인식 못하기도 한다.
- 장점: API 만들기 유용함
- <http://1boon.kakao.com/p/popular> - 일반 웹 페이지
- <http://1boon.kakao.com/p/popular.json> - json 데이터

```
{
  status: {
    code: 200,
    name: "OK"
  },
  data: [
    {
      docId: "5779da0ee787d0000017b480f",
      title: "[7월 1주] 홍관조 군단 새 수호신을 소개합니다",
      path: "sportsvod/jukos20160701",
      imgUrl: "http://i2.media.daumcdn.net/photo-media/201607/04/mediadaum/20160704130128709.jpeg",
      totalView: 152739
    },
    ...
  ]
}
```

- <http://1boon.kakao.com/p/popular.xml> - xml 데이터

```
<commonResult>
<status>OK</status>
<data>
<linkedTreeMap>
<entry key="docId">5779da0ee787d0000017b480f</entry>
<entry key="title">[7월 1주] 홍관조 군단 새 수호신을 소개합니다</entry>
<entry key="path">sportsvod/jukos20160701</entry>
<entry key="imgUrl">
http://i2.media.daumcdn.net/photo-media/201607/04/mediadaum/
20160704130128709.jpeg
</entry>
<entry key="totalView">152739.0</entry>
</linkedTreeMap>
```

미디어 타입 선택 방식이 미치는 영향

- 내용협상 (Content Negotiation)
 - <http://httpd.apache.org/docs/current/ko/content-negotiation.html>

- HTTP 규약
- Accept-Language: kr
- Accept: text/html; q=1.0, text...
- Accept-Charset, Accept-Encoding...
- 스프링 3.2에서 추가된 ContentNegotiationManager
 - ContentNegotiatingViewResolver 의 setter 메소드는 대부분 삭제.
ContentNegotiationManager 를 통해서 설정
- ContentNegotiationManager 설정
- xml config

```
<bean id="contentNegotiationManager" class="org.springframework
ork.http.ContentNegotiationManagerFactoryBean" p:defaultCont
entType="application/json">
```

- java config

```
@Override
public void configureContentNegotiation(ContentNegotiationCo
nfigurer configurer) {
    configurer.defaultContentType(MediaType.APPLICATION_JSON);
}
```

ContentNegotiatingViewResolver의 장점과 한계

- 장점: 컨트롤러 코드를 변경하지 않고 스프링 MVC의 최상단에서 REST 리소스 표현을 제공
- 단점: 클라이언트가 무엇을 예상하는지를 나타내지는 못한다.
- 이러한 한계점 때문에 ContentNegotiatingViewResolver 사용을 더 선호하지는 않음.
- 대신, 리소스 표현을 제공하기 위한 스프링의 메시지 변환기(message converter)를 사용하는 것에 더 의존

16.2.2 HTTP 메시지 변환기 사용

- 메시지 변환은 컨트롤러에 의해서 만들어지는 데이터를 클라이언트에 제공되는 표

현으로변환시키기 위한 보다 직접적인 방법을 제공

표 16.1

- AtomFeedHttpMessageConverter - Rome Feed 객체를 Atom 피드로 (또는 반대 방향으로) 변환 (미디어 타입은 application/atom+xml) Rome 라이브러리가 클래스패스에 존재하는 경우 등록
- BufferedImageHttpMessageConverter - BufferedImages를 이미지 바이너리 데이터로 (또는 반대 방향으로) 변환
- ByteArrayHttpMessageConverter - 바이트 배열 읽기/쓰기. 모든 미디어 타입(*/*)에서 읽고 application/octet-stream으로 쓴다.
- FormHttpMessageConverter - application/x-www-form-urlencoded의 콘텐츠를 MultiValueMap으로 읽는다. 또한 MultiValueMap을 application/x-www-form-urlencoded로 쓰고 MultiValueMap를 multipartform-data로 쓴다.
- Jaxb2RootElementHttpMessageConverter - JAXB2 애너테이션이 적용된 객체로부터 XML(text/xml 또는 applicatoin/xml)을 (또는 반대 방향으로) 읽고 쓴다. JAXB v2 라이브러리가 클래스패스에 존재하는 경우 등록된다.
- MappingJacksonHttpMessageConverter - 타입이 있는 객체 또는 타입이 없는 HashMap으로부터 JSON을 (또는 반대 방향으로) 읽고 쓴다. Jackson JSON 라이브러리가 클래스패스에 존재하는 경우 등록된다.
- MappingJackson2HttpMessageConverter - Jackson2JSON 라이브러리가 클래스패스에 존재하는 경우 등록된다.
- MarshallingHttpMessageConverter - 주입된 마샬러(marshaller)와 언마샬러(unmarshaller)를 이용해 XML을 읽고 쓴다. 지원된 마샬러(언마샬러)는 Castor, JAXB2, JIBX, XMLBeans 그리고 XStream이 있다.
- ResourceHttpMessageConverter - org.springframework.core.io.Resource를 읽고 쓴다.
- RssChannelHttpMessageConverter - Rome Channel 객체로부터 RSS 피드를 (또는 반대 방향으로) 읽고 쓴다. Rome 라이브러리가 클래스패스에 존재하는 경우 등록된다.
- SourceHttpMessageConverter - javax.xml.transform.Source 객체로부터 XML을 (또는 그 반대 방향으로) 읽거나 쓴다.
- StringHttpMessageConverter - 모든 미디어 타입(*/*)을 String 으로 읽는다. text/plain에 대한 String을 쓴다.
- XmlAwareFormHttpMessageConverter - FormHttpMessageConverter를 상속 받아 SourceHttpMessageConverter를 이용해 XML 기반의 부분에 대한 지원을 추가한다.

- 예를 들어, 클라이언트가 요청의 Accept 헤더를 통해 application/json을 받을 수 있음을 나타낸다고 가정하자. Jackson JSON 라이브러리가 애플리케이션의 클래스스페이스에 있다고 가정하면, 핸들러 메소드에서 반환된 객체는 클라이언트에 반환되는 JSON 표현으로의 변환을 위해 MappingJacksonHttpMessageConverter에 할당된다. 반면에 요청 헤더가 클라이언트는 text/xml을 선호한다고 나타내면, Jaxb2RootElementHttpMessageConverter가 클라이언트에 대한 XML 응답을 생성하는 작업을 수행한다.
- 표 16.1에 있는 HTTP 메시지 변환기 중 세 개는 기본적으로 등록된다. 따라서 사용하기 위한 스프링 설정은 필요 없다. 다만 라이브러리는 추가해야 한다.

응답 보디 (Response Body) 에서 리소스 상태 반환

앞서 설명 했던 @ResponseBody 를 사용하면 일반 모델/뷰 스킵하고 메시지 변환기 사용해서 리소스를 클라이언트에 보내게 된다.

요청 보디 (Request Body) 에서 리소스 상태받기

```
@RequestMapping(method=RequestMethod.POST, consumes="application
/json")
public @ResponseBody Spittle saveSpittle(@RequestBody Spittle sp
ittle) {
    return spittleRepository.save(spittle);
}
```

consumes="application/json" @RequestBody Spittle spittle 부분이 중요

메시지 변환을 위한 기본 컨트롤러

스프링 4.0 은 @RestController 사용 가능. 이미 앞서 설명 했음. @ResponseBody 사용 할 필요 없다. @RestController 가 @Controller, @ResponseBody 를 포함하고 있다.

16.3 더 많은 리소스 사용하기

훌륭한 REST API는 클라이언트와 서버 간의 리소스 전송 이상을 제공한다.

```

@RequestMapping(value="/{id}", method=RequestMethod.GET)
public @ResponseBody Spittle spittleById(@PathVariable Long id)
{
    return spittleRepository.findOne(id);
}

```

Http status code 지정 : `@ResponseStatus` 응답에 대한 메타데이터 :
`ResponseEntity`

ResponseEntity 사용하기

`ResponseEntity`는 리소스 표현으로 반환되는 객체와 더불어 응답에 대한 메타데이터 (헤더와 상태코드와 같은)를 가지는 객체임.

예외처리

에러 핸들러 메서드는 항상 `Error`를 반환하고, HTTP 상태코드는 404(Not Found)를 사용함을 알고 있어야 하며, 유사한 정리 프로세스를 `spittleNotFound()`에 적용함.

```

@ExceptionHandler(SpittleNotFoundException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public @ResponseBody Error spittleNotFound(SpittleNotFoundException e) {
    long spittleId = e.getSpittleId();
    return new Error(4, "Spittle [" + spittleId + "] not found");
}

```

응답 시의 헤더 설정하기

`HttpHeaders` 인스턴스는 응답 헤더 값을 전달하기 위해 생성함.

```

@RequestMapping(method=RequestMethod.POST, consumes="application
/json")
@ResponseStatus(HttpStatus.CREATED)
public ResponseEntity<Spittle> saveSpittle(@RequestBody Spittle
e spittle, UriComponentsBuilder ucb) {
    Spittle saved = spittleRepository.save(spittle);

    HttpHeaders headers = new HttpHeaders();
    URI locationUri = ucb.path("/spittles/")
        .path(String.valueOf(saved.getId()))
        .build()
        .toUri();
    headers.setLocation(locationUri);

    ResponseEntity<Spittle> responseEntity = new ResponseEntity<
Spittle>(saved, headers, HttpStatus.CREATED);
    return responseEntity;
}

```

16.4 REST 리소스 사용하기

유용하지 않은 많은 코드가 리소스 사용에 보일러플레이트(boilerplate)가 많이 포함되어 있고, 이러한 포함은 공통 코드를 캡슐화하고 변화를 파라미터화함. 이 동작은 스프링의 `RestTemplate`이 수행하는 동작임.

16.4.1 RestTemplate 동작 살펴보기

`RestTemplate`은 11개의 고유 동작을 정의하고, 각각은 총 36개의 메소드에 오버로드 된다.

HTTP method	RestTemplate methods
DELETE	<code>delete(java.lang.String, java.lang.Object...)</code>
GET	<code>getForObject(java.lang.String, java.lang.Class<T>, java.lang.Object...)</code> <code>getForEntity(java.lang.String, java.lang.Class<T>, java.lang.Object...)</code>
HEAD	<code>headForHeaders(java.lang.String, java.lang.Object...)</code>
OPTIONS	<code>optionsForAllow(java.lang.String, java.lang.Object...)</code>
POST	<code>postForLocation(java.lang.String, java.lang.Object, java.lang.Object...)</code> <code>postForObject(java.lang.String, java.lang.Object, java.lang.Class<T>, java.lang.Object...)</code>
PUT	<code>put(java.lang.String, java.lang.Object, java.lang.Object...)</code>
any	<code>exchange(java.lang.String, org.springframework.http.HttpMethod, org.springframework.http.ResponseEntity<?>, java.lang.Class<T>, java.lang.Object...)</code> <code>execute(java.lang.String, org.springframework.http.HttpMethod, org.springframework.web.client.RequestCallback, org.springframework.web.client.ResponseExtractor<T>, java.lang.Object...)</code>

16.4.2 리소스 GET 하기

<code><T> ResponseEntity<T></code>	<code>getForEntity(String url, Class<T> responseType, Map<String,?> urlVariables)</code> Retrieve a representation by doing a GET on the URI template.
<code><T> ResponseEntity<T></code>	<code>getForEntity(String url, Class<T> responseType, Object... urlVariables)</code> Retrieve an entity by doing a GET on the specified URL.
<code><T> ResponseEntity<T></code>	<code>getForEntity(URI url, Class<T> responseType)</code> Retrieve a representation by doing a GET on the URL.
<code><T> T</code>	<code>getForObject(String url, Class<T> responseType, Map<String,?> urlVariables)</code> Retrieve a representation by doing a GET on the URI template.
<code><T> T</code>	<code>getForObject(String url, Class<T> responseType, Object... urlVariables)</code> Retrieve a representation by doing a GET on the specified URL.
<code><T> T</code>	<code>getForObject(URI url, Class<T> responseType)</code> Retrieve a representation by doing a GET on the URL.

16.4.3 리소스 조회

```
public Profile fetchFacebookProfile(String id) {
    RestTemplate rest = new RestTemplate();
    return rest.getForObject("http://graph.facebook.com/{spitter
}",
        Profile.class, id);
}
```

16.4.4 응답 메타데이터 추출

ResponseEntity는 HTTP 상태 코드와 응답 헤더와 같은 응답에 대한 추가적인 정보도 전달함

```
Date lastModified = new Date(response.getHeaders().getLastModifi
ed());
```

<code>List<MediaType></code>	<code>getAccept()</code> Return the list of acceptable media types , as specified by the Accept header.
<code>List<Charset></code>	<code>getAcceptCharset()</code> Return the list of acceptable charsets , as specified by the Accept-Charset header.
<code>boolean</code>	<code>getAccessControlAllowCredentials()</code> Returns the value of the Access-Control-Allow-Credentials response header.
<code>List<String></code>	<code>getAccessControlAllowHeaders()</code> Returns the value of the Access-Control-Allow-Headers response header.
<code>List<HttpMethod></code>	<code>getAccessControlAllowMethods()</code> Return the value of the Access-Control-Allow-Methods response header.
<code>String</code>	<code>getAccessControlAllowOrigin()</code> Return the value of the Access-Control-Allow-Origin response header.
<code>List<String></code>	<code>getAccessControlExposeHeaders()</code> Returns the value of the Access-Control-Expose-Headers response header.
<code>long</code>	<code>getAccessControlMaxAge()</code> Returns the value of the Access-Control-Max-Age response header.
<code>List<String></code>	<code>getAccessControlRequestHeaders()</code> Returns the value of the Access-Control-Request-Headers request header.
<code>HttpMethod</code>	<code>getAccessControlRequestMethod()</code> Return the value of the Access-Control-Request-Method request header.

16.4.5 리소스 PUT하기

```
public void updateSpittle(Spittle spittle) {
    RestTemplate rest = new RestTemplate();
    rest.put("http://localhost:8080/spittr-api/spittles/{id}",
        spittle, spittle.getId());
}
```

객체가 변환될 콘텐츠 타입은 put()에 전달되는 타입에 따라 달라짐.

- String : StringHttpmessageConverter, text/plain
- MultiMap : FormHttpMessageConverter, application/x-www-form-urlencoded
- MappingJacksonHttpMessageConverter, application/json

16.4.6 리소스 DELETE하기

```
public void updateSpittle(long id) {
    RestTemplate rest = new RestTemplate();
    rest.delete("http://localhost:8080/spittr-api/spittles/{id}"
        , id);
}
```

16.4.7 리소스 데이터 POST하기

16.4.8 POST 요청으로부터 객체 응답 수신하기

```
public void updateSpittle(Spitter spitter) {
    RestTemplate rest = new RestTemplate();
    return rest.postForObject("http://localhost:8080/spittr-api/spitters", spitter, Spitter.class);
}
```

16.4.9 POST 요청 후 리소스 위치 수신하기

```
public void updateSpittle(Spitter spitter) {
    RestTemplate rest = new RestTemplate();
    return rest.postForLocation("http://localhost:8080/spittr-api/spitters", spitter).toString()
}
```

16.4.10 리소스 교환

exchange()는 getForEntity()나 getForObject()와 달리 전송할 요청에 헤더를 설정할 수 있게 해줌.

```
MultiValueMap<String, String> headers = new LinkedMultiValueMap<>();
headers.add("Accept", "application/json");
HttpEntity<Object> requestEntity = new HttpEntity<Object>(headers);
ResponseEntity<Spitter> response = rest.exchange("http://localhost:8080/spittr-api/spitters/{spitter}", HttpMethod.GET, requestEntity, Spitter.class, spitterId);
Spitter spitter = response.getBody();
}
```

책 예제 프로젝트

[spitter-api-message-converters](#)

Spock Documentation

[spock documentation](#)

Spock Github

[spock github](#)

테스트코드 친해지기 페이스북 그룹

[테스트코드와 친해지기](#)

Chapter21. 스프링 부트를 사용한 스프링 개발 간소화

스프링 부트 스타터로 프로젝트 의존성 추가하기

빈 설정 자동화

그루비와 스프링부트 CLI

그루비 컨트롤러 작성하기

- 그루비
 - 자바가상기계(JVM) 상에서 동작하는 동적인 스크립팅 언어
 - 그루비의 문법 체계는 자바의 것을 계승 발전시켰다.
 - 자바에는 없는 간편 표기법(syntactic sugar)을 지원하는 외에 리스트, 맵, 정규식을 위한 구문을 제공
 - 프로그래밍을 쉽고 간결하게 해줌
- 그루비 코드의 작성 (아래 내용 없이 그루비 코드 작성이 가능)
 - 세미콜론
 - public 이나 private 같은 제한자
 - 프로퍼티에 대한 접근자(setter/getter) 메소드
 - 메소드 값을 반환하기 위한 return 예약어
 - 스타일 가이드 : <http://groovy-lang.org/style-guide.html>

ContactController 클래스

```
@Grab("thymeleaf-spring4")

@Controller
@RequestMapping("/")
class ContactController {

    @Autowired
    ContactRepository contactRepo

    @RequestMapping(method=RequestMethod.GET)
    String home(Map<String, Object> model) {
        List<Contact> contacts = contactRepo.findAll()
        model.putAll([contacts: contacts])
        "home"
    }

    @RequestMapping(method=RequestMethod.POST)
    String submit(Contact contact) {
        contactRepo.save(contact)
        "redirect:/"
    }
}
```

- import 구문에서 기본적인 몇가지 패키지와 클래스를 들여와서 작성됨
 - java.io.*
 - java.lang.*
 - java.math.BigDecimal
 - java.math.BigInteger
 - java.net.*
 - java.util.*
 - groovy.lang.*
 - groovy.util.*
- @Controller, @RequestMapping, @Autowired, @RequestMethod 와 같은 스프링 타입?
 - 애플리케이션이 실행될 때 스프링 부트 CLI는 그루비 컴파일러를 사용하여 그루비 클래스를 컴파일

컴파일 실패

1. 자동 설정 CLI 추가 사용 : 스프링부트 스타터 의존성을 가져와서 클래스 패스에 포함해 의존성 추가함
2. 그루비 컴파일에 목록에 필요한 패키지를 추가하여 코드를 다시 컴파일함

Contact 클래스

```
class Contact {  
    long id  
    String firstName  
    String lastName  
    String phoneNumber  
    String emailAddress  
}
```

- 세미콜론, 접근자 메소드, 제한자도 없이 작성

ContactRepository 클래스

```

@Grab("h2")

import java.sql.ResultSet

class ContactRepository {
    @Autowired
    JdbcTemplate jdbc

    List<Contact> findAll() {
        jdbc.query(
            "select id, firstName, lastName, phoneNumber, emailAddress
" +
            "from contacts order by lastName",
            new ReoMapper<Contact>() {
                Contact mapRow(ResultSet rs, int rowNum) {
                    new Contact(id: rs.getLong(1), firstName: rs.getString
(2),
                        lastName: rs.getString(3), phoneNumber: rs.getString(4
),
                        emailAddress: rs.getString(5))
                }
            }
        )
    }

    void save(Contact contact) {
        jdbc.update(
            jdbc.update(
                "insert into contacts " +
                "(firstName, lastName, phoneNumber, emailAddress) " +
                "values (?, ?, ?, ?),
                contact.firstName, contact.lastName, contact.phoneNumber
, contact.emailAddress)
            )
    }
}

```

- 스프링부트 CLI가 자동으로 JdbcTemplate, RowMapper를 들여옴
- 자동-들여오기와 자동-해결하기로 처리 못하는 ResultSet은 명시적으로 처리

- 사용할 데이터베이스도 예상할 수 없으므로 반드시 H2데이터베이스 사용을 요청하기 위한 `@Grab` 애너테이션을 활용

스프링 부트 CLI 실행

자바 애플리케이션 컴파일이 끝나고 실행시킬 수 있는 방법엔 두가지가 있다.

1. 커맨드라인을 이용하여 실행 가능한 JAR나 WAR 파일을 실행
2. 서블릿 컨테이너로 WAR 파일을 배포
3. (+) 스프링부트 CLI로 애플리케이션을 실행 (커맨드 라인 실행)
 - 애플리케이션을 먼저 JAR나 WAR 파일로 먼저 빌드할 필요가 없음
 - 애플리케이션 실행을 위해서 직접 그루비 소스코드를 CLI에 전달

CLI설치

Homebrew 를 사용한 수동 설치

```
$ brew tap pivotal/tab
$ brew install springboot
```

CLI로 연락처 애플리케이션 실행하기

스프링 부트 CLI로 애플리케이션을 실행하려면, `spring run`에 CLI를 통해 실행할 그루비 파일들을 붙여서 커맨드라인에 입력한다.

```
$ spring run Hello.groovy
$ spring run *.groovy

// 그루비 클래스 파일들이 여러 디렉토리에 있다면, Ant 스타일 와일드카드로 그루비 클래스를 재귀적으로 탐색함
$ spring run **/*.groovy
```

스프링 부트 액추에이터

액추에이터로 애플리케이션 파악하기

- 스프링 부트 액추에이터 역할
 - 스프링 부트 기반의 애플리케이션에 유용한 엔드포인트 관리 기능 추가

```
## MAVEN
```

```
<dependency>
```

```
  <groupId>org.springframework.boot</groupId>
```

```
  <artifactId>spring-boot-starter-actuator</artifactId>
```

```
</dependency>
```

```
## GRADLE
```

```
compile group: 'org.springframework.boot', name: 'spring-boot-starter-actuator'
```

```
## 스프링부트 CLI
```

```
@Grab("spring-boot-starter-actuator")
```

Endpoint

- GET /autoconfig
 - 자동 설정이 적용될 때 스프링 부트에 의한 결정을 설명
- GET /beans
 - 애플리케이션을 실행하기 위해 설정되는 빈의 카탈로그
- GET /configprops
 - 현재 값으로 애플리케이션의 빈을 설정하기 위한 모든 프로퍼티 목록
- GET /env
 - 애플리케이션 컨텍스트에서 사용가능한 모든 환경 변수와 시스템 프로퍼티 값
- GET /env/{name}
 - 특정 환경 변수나 프로퍼티 값을 표시
- GET /health
 - 현재 애플리케이션의 상태 표시
- GET /info
 - 애플리케이션 상세 정보
- GET /metrics
 - 특정 엔드포인트에 대한 요청 개수를 포함한 애플리케이션의 메트릭 목록
- POST /shutdown
 - 강제로 애플리케이션을 종료
- GET /trace

- 요청과 응답을 포함한 애플리케이션을 통해 서비스된 최근 요청에 관한 메타 데이터 목록

/health Endpoint

요청

```
$ curl http://localhost:8080/health
```

응답 결과

```
{
  "status": "UP"
}
{
  "status": "DOWN"
}
```

/info Endpoint

application.properties

```
info.app.name=Spring Sample Application
info.app.description=This is my first spring boot application
info.app.version=1.0.0
```

응답 결과

```
{
  "app" : {
    "version" : "1.0.0",
    "description" : "This is my first spring boot application",
    "name" : "Spring Sample Application"
  }
}
```

/metrics Endpoint

publishes information about OS, JVM and Application level metrics

```
{
  "mem" : 193024,
  "mem.free" : 87693,
  "processors" : 4,
  "instance.uptime" : 305027,
  "uptime" : 307077,
  "systemload.average" : 0.11,
  "heap.committed" : 193024,
  "heap.init" : 124928,
  "heap.used" : 105330,
  "heap" : 1764352,
  "threads.peak" : 22,
  "threads.daemon" : 19,
  "threads" : 22,
  "classes" : 5819,
  "classes.loaded" : 5819,
  "classes.unloaded" : 0,
  "gc.ps_scavenge.count" : 7,
  "gc.ps_scavenge.time" : 54,
  "gc.ps_marksweep.count" : 1,
  "gc.ps_marksweep.time" : 44,
  "httpsessions.max" : -1,
  "httpsessions.active" : 0,
  "counter.status.200.root" : 1,
  "gauge.response.root" : 37.0
}
```

Custom HealthIndicator example


```
@Component
public class HealthCheck implements HealthIndicator {
    @Override
    public Health health() {
        int errorCode = check(); // perform some specific health
        check
        if (errorCode != 0) {
            return Health.down().withDetail("Error Code", errorC
            ode).build();
        }
        return Health.up().build();
    }

    public int check() {
        // Your logic to check health
        return 0;
    }
}
```