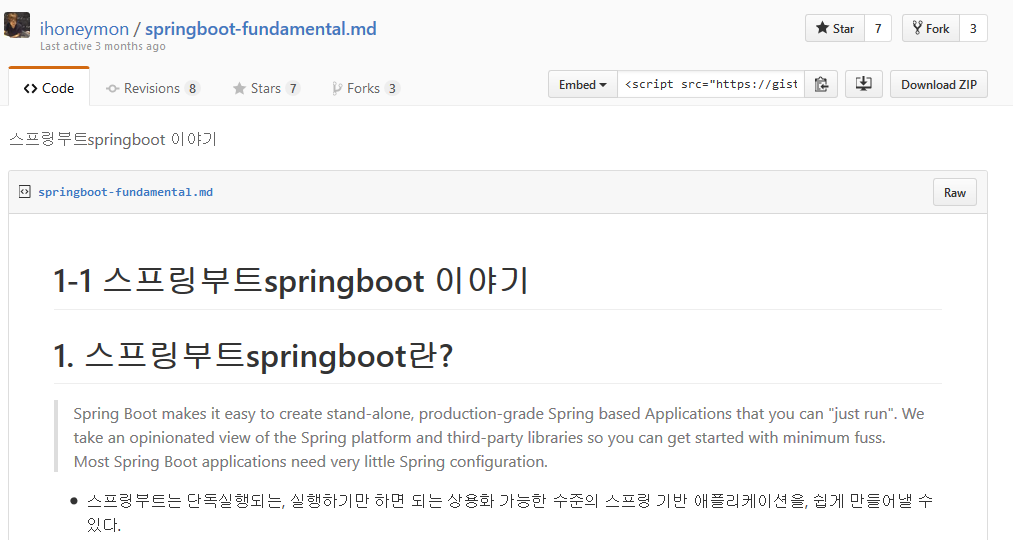
<https://gist.github.com/ihoneymon/a343e2f4a0299988206e>



<http://millky.com/@origoni/post/1100>

<https://gist.github.com/ihoneymon>

<http://java.ihoney.pe.kr/348>

spring boot reference

<http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/>

spring boot란

<https://gist.github.com/ihoneymon/a343e2f4a0299988206e>

spring appication properties

<http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>

spring boot default class

|  |
| --- |
| package com;  import org.springframework.boot.\*;  import org.springframework.boot.autoconfigure.\*;  import org.springframework.stereotype.\*;  import org.springframework.web.bind.annotation.\*;  @Controller  @EnableAutoConfiguration  public class SampleController {  @RequestMapping("/")  @ResponseBody  String home() {  return "Hello World!";  }  public static void main(String[] args) throws Exception {  SpringApplication.*run*(SampleController.class, args);  }  } |

filter 등록

|  |
| --- |
| @Bean  public FilterRegistrationBean shallowEtagHeaderFilter() {  FilterRegistrationBean registration = new FilterRegistrationBean();  // registration.setFilter(new ShallowEtagHeaderFilter());  registration.setFilter(new TestFilter());  registration.setDispatcherTypes(EnumSet.*allOf*(DispatcherType.class));  registration.addUrlPatterns("/cue-categories");  return registration;  } |

servlet등록

|  |
| --- |
| @Bean  /\*  h2 console servlet 등록할때 application.properties 에서 처리도 가능하다.  <http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>  \*/  public ServletRegistrationBean axisServletRegi3strationBean() {  ServletRegistrationBean registration = new ServletRegistrationBean(new WebServlet());  registration.addUrlMappings("/console/\*");  return registration;  }  @Bean  public ServletRegistrationBean axisServletRegistrationBean() {  ServletRegistrationBean registration = new ServletRegistrationBean(new TestServlet(), "/services/\*");  registration.addUrlMappings("\*.jws");  return registration;  } |

Message char set 설정

|  |
| --- |
| @Bean  public HttpMessageConverter<String> responseBodyConverter() {  return new StringHttpMessageConverter(Charset.*forName*("UTF-8"));  } |

Login

<http://docs.spring.io/spring-boot/docs/current/reference/html/howto-logging.html>

<http://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-logging.html>

|  |
| --- |
| **logging:**  **level:**  **root:** DEBUG  **org.springframework.web:** DEBUG  **org.hibernate:** DEBUG  application.yml |
| <?xml version="1.0" encoding="UTF-8"?>  <configuration>  <include resource="org/springframework/boot/logging/logback/base.xml"/>  <include resource="org/springframework/boot/logging/logback/defaults.xml" />  <include resource="org/springframework/boot/logging/logback/defaults.xml" />  <property name="LOG\_FILE" value="${LOG\_FILE:-${LOG\_PATH:-${LOG\_TEMP:-${java.io.tmpdir:-/tmp}}/}spring.log}"/>  <include resource="org/springframework/boot/logging/logback/file-appender.xml" />  <!-- log4jdbc-log4j2 -->  <logger name="jdbc.sqlonly" level="DEBUG"/>  <logger name="jdbc.sqltiming" level="DEBUG"/>  <logger name="jdbc.audit" level="DEBUG"/>  <logger name="jdbc.resultset" level="DEBUG"/>  <logger name="jdbc.resultsettable" level="DEBUG"/>  <logger name="jdbc.connection" level="DEBUG"/>  <logger name="com.khh.project" level="DEBUG"/>  <!--<root level="INFO">-->  <!--<appender-ref ref="FILE" />-->  <!--</root>-->  </configuration>  resource/logback.xml |

tamplates 엔진 템플릿 엔진

<http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-mvc-template-engines>

[FreeMarker](http://freemarker.org/docs/) , [Groovy](http://docs.groovy-lang.org/docs/next/html/documentation/template-engines.html#_blank), [Thymeleaf](http://www.thymeleaf.org/), [Velocity](http://velocity.apache.org/) (deprecated in 1.4), [Mustache](http://mustache.github.io/)

## 템플릿 엔진

* 참고: [Template engines](http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-mvc-template-engines)
* JSP 보다는 템플릿 엔진의 사용을 권장한다.
* 기본 디렉토리는 src/main/resources/templates 사용을 권장한다.

groovy teamplate

<http://docs.groovy-lang.org/docs/next/html/documentation/template-engines.html#_the_markuptemplateengine>

<https://github.com/spring-projects/spring-boot/tree/master/spring-boot-samples/spring-boot-sample-web-groovy-templates>

<https://github.com/spring-projects/spring-boot/blob/master/spring-boot-samples/spring-boot-sample-web-groovy-templates/src/main/resources/templates/messages/form.tpl>

<https://spring.io/blog/2014/05/28/using-the-innovative-groovy-template-engine-in-spring-boot>

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

<http://mrhaki.blogspot.kr/2014/08/groovy-goodness-using-layouts-with.html>

## 정적 자원

## org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration 중에서

private static final String[] CLASSPATH\_RESOURCE\_LOCATIONS = {

"classpath:/META-INF/resources/", "classpath:/resources/",

"classpath:/static/", "classpath:/public/" };

위에서 보는 것과 같이 클래스패스classpath 상에서 /META-INF/resources/, /resources, /static, /public 경로를 기본탐색한다.

/WEB-INF/resources 의 경우, jar 파일로 배포할 경우에는 인식하지 않기때문에 사용하지 않도록 주의한다.

정적 자원들은 /static 디렉토리 사용을 권장한다.

별도의 루트('/') 경로에 대한 설정이 되어 있지 않은 경우에는 [Spring MVC auto-configuration](http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-mvc-auto-configuration) 와 관련된 자동설정에 따라서 static index.html을 지원하게 된다.

security

<http://cusonar.tistory.com/8>

|  |
| --- |
| compile("org.springframework.boot:spring-boot-starter-security:${springBootVersion}") |

auth object

|  |
| --- |
| @Slf4j @Entity @Table(name = "USER") @Data public class LoginUser implements UserDetails {   @Id  private String id;  private String pwd;  private String name;  private String addr;   private boolean auth\_accountNonExpired;  private boolean auth\_accountNonLocked;  private boolean auth\_credentialsNonExpired;  private boolean auth\_enabled;   @Transient  private Collection<? extends GrantedAuthority> authorities;   @Override  public Collection<? extends GrantedAuthority> getAuthorities() {  return null;  }   @Override  public String getPassword() {  return getPwd();  }   @Override  public String getUsername() {  return getName();  }   @Override  public boolean isAccountNonExpired() {  return this.isAuth\_accountNonExpired();  }   @Override  public boolean isAccountNonLocked() {  return this.isAuth\_accountNonLocked();  }   @Override  public boolean isCredentialsNonExpired() {  return this.isCredentialsNonExpired();  }   @Override  public boolean isEnabled() {  return this.isEnabled();  } } |
| interface LoginUserRepository extends JpaRepository<LoginUser, String> { } |

security setting adapter

WebSecurityConfigurerAdapter

|  |
| --- |
| @Slf4j @Configuration @EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true) @EnableWebSecurity public class WebSecurityConfigurerAdapter extends org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter {  @Override protected void configure(HttpSecurity http) throws Exception {  *log*.debug("-----security HttpSecurity-----"+http);  http  .authorizeRequests()  .antMatchers("/", "/home").permitAll()  .anyRequest().authenticated()  .and()  .formLogin()  .loginProcessingUrl("/login") //login-processing-url 로그인 페이지 form action에 입력할 주소 지정  .loginPage("/login").usernameParameter("id").passwordParameter("pwd").defaultSuccessUrl("/")  .permitAll()  .and()  .logout().deleteCookies("remember-me").logoutUrl("/logout").logoutSuccessUrl("/")  .permitAll();  }  }  아래는 기존 xml  <!--   login-page 로그인 페이지 주소를 지정  username-parameter 로그인 페이지 form에 있는 username을 저장한 변수이름 지정  password-parameter 로그인 페이지 form에 있는 password을 저장한 변수이름 지정   login-processing-url 로그인 페이지 form action에 입력할 주소 지정  default-target-url 로그인 성공인 경우 호출할 주소 지정  authentication-failure-url 로그인 실패인 경우 호출할 주소 지정  always-use-default-target --> <sec:form-login login-page="/"   username-parameter="username"   password-parameter="password"   login-processing-url="/sign/login"  default-target-url="/"   authentication-failure-url="/"  authentication-success-handler-ref="customAuthenticationSuccessHandler"  />  <sec:logout logout-url="/sign/logout" logout-success-url="/?msg=logout" delete-cookies="JSESSIONID" /><!-- success-handler-ref="customLogoutSuccessHandler" --> |

Spring

# Example of CachingConfigurer in Spring

http://www.concretepage.com/spring/example\_cachingconfigurer\_spring

application.yml

|  |
| --- |
| *#http://docs.spring.io/spring-boot/docs/current/reference/html/howto-properties-and-configuration.html* **application:  version:** 1.0  **title:** boardApplication  **spring:  application:  name:** springboot  **main:  banner-mode:** off  **h2:  console:  enabled:** true *# Enable the console.* **path:** /h2-console *# Path at which the console will be available.* **settings:  trace:** false *# Enable trace output.* **web-allow-others:** false *# Enable remote access.* **datasource:  name:** testdb  **initialize:** true *# Populate the database using 'data.sql'.* **platform:** all *# Platform to use in the schema resource (schema-${platform}.sql).  #url: jdbc:h2:mem:testdb;DB\_CLOSE\_DELAY=- 1;DB\_CLOSE\_ON\_EXIT=FALSE* **driverClassName:** org.h2.Driver  **username:** sa  **password:  continue-on-error:** true  **hibernate:  hbm2ddl:  auto:** update  **show\_sql:** true  **jpa:  database-platform:** org.hibernate.dialect.H2Dialect  **hibernate:  generate-ddl:** true  **ddl-auto:** create-drop  **show-sql:** true  **data:  repositories:  enabled:** true  **http:  multipart:  enabled:** true *# Enable support of multi-part uploads.  # SECURITY (SecurityProperties)* **security:  basic:  authorize-mode:** role *# Security authorize mode to apply.* **enabled:** true *# Enable basic authentication.* **path:** /\*\* *# Comma-separated list of paths to secure.* **realm:** Spring *# HTTP basic realm name.* **enable-csrf:** true *# Enable Cross Site Request Forgery support.* **filter-order:** 0 *# Security filter chain order.* **filter-dispatcher-types:** ASYNC, FORWARD, INCLUDE, REQUEST *# Security filter chain dispatcher types.* **headers:  cache:** true *# Enable cache control HTTP headers.* **content-type:** true *# Enable "X-Content-Type-Options" header.* **frame:** true *# Enable "X-Frame-Options" header.  #hsts= # HTTP Strict Transport Security (HSTS) mode (none, domain, all).* **xss:** true *# Enable cross site scripting (XSS) protection.  #ignored= # Comma-separated list of paths to exclude from the default secured paths.* **require-ssl:** false *# Enable secure channel for all requests.* **sessions:** stateless *# Session creation policy (always, never, if\_required, stateless).* **user:  name:** user *# Default user name.* **password:** user *# Password for the default user name. A random password is logged on startup by default.* **role:** USER *# Granted roles for the default user name. # view: # prefix: # suffix: :.vm* **groovy:  template:  cache:** false **logging:  level:  root:** INFO  **org.springframework.web:** INFO  **org.hibernate:** INFO  **com.khh:** DEBUG   ---  **spring:  profiles:** development  **logging:  level:  root:** INFO  **org.springframework.web:** DEBUG  **org.hibernate:** DEBUG  **com.khh:** DEBUG  ---  **spring:  profiles:** production  실행시 profiles를 지정하면 그쪽의 설정값이 먹힌다.  -Dspring.profiles.active=development |

<https://justinrodenbostel.com/2014/05/13/part-4-internationalization-in-spring-boot/>

# Part 4: Internationalization in Spring Boot

<http://stackoverflow.com/questions/28750292/custom-spring-boot-starter-how-do-you-contribute-i18n-messages-to-the-messageso>

ReloadableResourceBundleMessage

# Spring에서 YAML 파일 데이터 객체에 매핑하여 로드하기

<http://blog.saltfactory.net/java/load-yaml-file-in-spring.html>

csrf 추가

|  |
| --- |
| @Override public void addInterceptors(InterceptorRegistry registry) {  registry.addInterceptor(csrfTokenAddingInterceptor()).addPathPatterns("/\*\*"); }  @Bean public HandlerInterceptor csrfTokenAddingInterceptor() {  return new HandlerInterceptorAdapter() {  @Override  public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView view) {  CsrfToken token = (CsrfToken) request.getAttribute(CsrfToken.class.getName());  if (token != null) {  view.addObject(token.getParameterName(), token);  }  }  }; } |
| layout 'layout.tpl', title: 'login',  content: contents {  div{  form(action:"${WebSecurityConfigurerAdapter.*LOGIN\_PROCESSING\_URL*}", method:'POST'){  input(type:'text', name:'username', value:'')  input(type:'text', name:'password', value:'')  input(type:'hidden', name:"${\_csrf.parameterName}", value:"${\_csrf.token}")  input(type:'submit', value:'submit')  }  }  } |

다국어 추가

|  |
| --- |
| //다국어 https://justinrodenbostel.com/2014/05/13/part-4-internationalization-in-spring-boot/ @Bean public LocaleResolver localeResolver() {  SessionLocaleResolver slr = new SessionLocaleResolver();  slr.setDefaultLocale(locale);  return slr; } @Bean public LocaleChangeInterceptor localeChangeInterceptor() {  LocaleChangeInterceptor lci = new LocaleChangeInterceptor();  lci.setParamName("lang");  return lci; }  @Override public void addInterceptors(InterceptorRegistry registry) {   registry.addInterceptor(localeChangeInterceptor()).addPathPatterns("/\*\*"); } |

구르비 템플릿 groovy template

requestContext사용

|  |
| --- |
| import org.springframework.security.core.Authentication import org.springframework.security.core.context.SecurityContextHolder  //http://www.w3ii.com/ko/groovy/groovy\_template\_engines.html layout 'layout.tpl', title: '메인화면입니다.',  content: contents {  div{  yield "방갑습니다. ${springMacroRequestContext} >>> ${spring.request} >>>${spring.response} >>> ${spring.getModelObject('\_csrf').token} "  }  Authentication authentication = SecurityContextHolder.*getContext*().getAuthentication();  div{  yield authentication  }  } |

Classpath antpattern 사용

|  |
| --- |
| //log.debug(System.getProperty("java.class.path")); // ClassPathResource resource = new ClassPathResource("resource/hibernate/mapper.hbm.xml"); // BufferedReader reader = new BufferedReader(new InputStreamReader(resource.getInputStream())); // reader.lines().forEach(System.out::println); // Resource resource = new ClassPathResource(hibernateProperties.getMappingLocations()); // ResourceLoader loader = new DefaultResourceLoader(); // Resource resource = new FileSystemResource(hibernateProperties.getMappingLocations()); // factory.setMappingLocations(classPathResource);  ClassLoader cl = this.getClass().getClassLoader();  ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver(cl);  Resource[] resources = resolver.getResources(hibernateProperties.getMappingLocations()) ; // factory.setMappingLocations(resources);  for (Resource resource: resources){  factory.setMappingLocations(resources);  //log.info(resource.getFilename());  } |

config쪽 불러오기

|  |
| --- |
| private static final String ENV\_OAUTH = "authentication.oauth.";  private static final String PROP\_CLIENTID = "clientid";  private static final String PROP\_SECRET = "secret";  private static final String PROP\_TOKEN\_VALIDITY\_SECONDS = "tokenValidityInSeconds";  private RelaxedPropertyResolver propertyResolver;  @Override  public void configure(ClientDetailsServiceConfigurer clients) throws Exception {  clients  .inMemory()  .withClient(propertyResolver.getProperty(PROP\_CLIENTID))  .scopes("read", "write")  .authorities(Authorities.ROLE\_ADMIN.name(), Authorities.ROLE\_USER.name())  .authorizedGrantTypes("password", "refresh\_token")  .secret(propertyResolver.getProperty(PROP\_SECRET))  .accessTokenValiditySeconds(propertyResolver.getProperty(PROP\_TOKEN\_VALIDITY\_SECONDS, Integer.class, 1800));  }  @Override  public void setEnvironment(Environment environment) {  this.propertyResolver = new RelaxedPropertyResolver(environment, ENV\_OAUTH);  } |

그래서 역시나 찾아보니 Spring boot용 동일한 기능을 하는 새로운 인터페이스를 제공한다.

org.springframework.boot.context.embedded.ServletContextInitializer

이 인터페이스를 상속 받아야 원하는 기능이 수행하게 된다.

그리고 따로 인터페이스를 상속 받지 않고 Bean등록만으로도 Servlet과 Filter를 등록할 수 있는 클래스도 따로 제공한다.

- Filter 등록 클래스

org.springframework.boot.context.embedded.FilterRegistrationBean

- Servlet 등록 클래스

org.springframework.boot.context.embedded.ServletRegistrationBean

간단한 사용방법 예제

@Bean

public ServletRegistrationBean h2servletRegistration(){

    ServletRegistrationBean registrationBean = new ServletRegistrationBean(new WebServlet());

    registrationBean.addUrlMappings("/console/\*");

    return registrationBean;

}

이런 형태로 등록해서 바로 사용이 가능하다.

자세한 부분은 역시나 [Spring boot](http://docs.spring.io/spring-boot/docs/1.2.x/reference/htmlsingle/#boot-features-embedded-container-servlets-and-filters) 레퍼런스 문서에 나와 있다.

## [[Spring Boot] 간단한 RestAPI 및 JPA 예제.](http://mudchobo.tistory.com/549)

[자바(Java)](http://mudchobo.tistory.com/category/자바(Java)) 2015.01.11 19:43 http://mudchobo.tistory.com/549

|  |  |  |
| --- | --- | --- |
|  |  |  |

스프링 오랜만에 다시 보내 새롭네요. 우연히 접할 기회가 있어서 스프링 부트를 접하게 되었습니다. 기존에 주로 회사에서는 php, ruby on rails, javascript 등 스크립트 언어를 해오다 보니 꽤나 쉽지 않네요. 뭐 이걸로 프로젝트를 한 번 하면 금방 배우겠죠. 루비 같은 경우도 작년까지 전혀 몰랐는데, 프로젝트를 하다보니 점점 알게 되더라구요.

**1. 셋팅**

이걸 하면서 gradle이라는 것을 알게 되었는데, 보니까 maven 같은 것인데, 확실히 maven보단 설정 문법이 더욱 깔끔하고 좋네요. maven에서는 xml 지옥이라 알아보기 힘들었는데, gradle은 그냥 스크립트 형식으로 되어있어서 더 알아보기 쉽게 되어있습니다.

**build.gradle**

buildscript {

repositories {

mavenCentral()

}

dependencies {

classpath("org.springframework.boot:spring-boot-gradle-plugin:1.1.10.RELEASE")

}

}

apply plugin: 'java'

apply plugin: 'spring-boot'

sourceCompatibility = 1.5

version = '1.0'

repositories {

mavenCentral()

}

dependencies {

testCompile group: 'junit', name: 'junit', version: '4.11'

compile("org.springframework.boot:spring-boot-starter-web:1.2.0.RELEASE")

compile("org.springframework.boot:spring-boot-starter-data-jpa")

compile("mysql:mysql-connector-java:5.1.34")

}

맨 위에 buildscript부분은 jar파일 만들기 위해서 필요한 것 같아요. 그 외에 plugin이 spring-boot가 추가되었는데, bootRun 이런 task 등이 추가 되어있어요.

dependencies는 스프링 부트를 쓰려면 spring-boot-starter-web이 있어야 하고, 데이터연동하는 것을 하기 위해서는 spring-boot-starter-data-jpa가 있어야 하고, mysql을 쓰려면 mysql-connector-java를 추가해야 해요.  
jpa에는 하이버네이트가 내장되어 있어요.

**2. 메인 클래스**

스프링 부트는 예전 개발방식이랑 틀리게 톰캣 받고, web.xml을 셋팅하고, 웹개발을 위한 셋팅이 필요없이 기존 java 실행 방식으로 실행하며 이 실행과 동시에 내장된 톰캣이 작동하여 서버를 만들어버립니다. 아래와 같이 메인클래스를 만들어버리면 그냥 웹서버애플리케이션이 뜹니다.

**Application.java**

@SpringBootApplication

**public** **class** **Application** {

**public** **static** void main(String[] args) {

ConfigurableApplicationContext context = SpringApplication.run(Application.class, args);

}

}

보면 예전에 자바처음 배울 때 쓰는 public static void main클래스로 실행만 합니다. @SpringBootApplication어노테이션을 붙이면 최초 기본셋팅으로 톰캣을 띄워서 8080포트로 서버를 만들어 줍니다.

**3. 모델 생성**

그 전에 접속할 db정보를 입력해야 합니다.   
**application.properties**

spring.datasource.url=jdbc:mysql://127.0.0.1/sosi?autoReconnect=true&useUnicode=true&characterEncoding=utf8

spring.datasource.username=root

spring.datasource.password=

spring.jpa.hibernate.ddl-auto=create-drop

spring.jpa.database=mysql

spring.jpa.show-sql=true

대략 설정값은 저런데 ddl-auto부분이 create-drop은 서버 재시작 때마다 테이블을 날려버리는 옵션입니다. 처음에 개발할 때에는 매우 편리합니다. 값을 매번 지우지 않고도 서버만 재시작해도 처음부터 다시 개발 해놓은 것을 테스트해볼 수 있으니깐염.

**Sosi.java**

@Entity

**public** **class** **Sosi** {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

**private** long id;

@Column(nullable = **false**)

**private** String name;

@OneToMany

@JoinColumn(name="sosi\_id", referencedColumnName="id")

**private** List<Schedule> scheduleList;

**public** Sosi() {

}

**public** Sosi(String name) {

**this**.name = name;

}

**public** long getId() {

**return** id;

}

**public** void setId(long id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** void setName(String name) {

**this**.name = name;

}

**public** List<Schedule> getScheduleList() {

**return** scheduleList;

}

**public** void setScheduleList(List<Schedule> scheduleList) {

**this**.scheduleList = scheduleList;

}

}

**Schedule.java**

@Entity

**public** **class** **Schedule** {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

**private** Long id;

@ManyToOne

@JsonBackReference

**private** Sosi sosi;

@Column

**private** String program;

**public** Schedule() {

}

**public** Schedule(Sosi sosi, String program) {

**this**.sosi = sosi;

**this**.program = program;

}

**public** Long getId() {

**return** id;

}

**public** void setId(Long id) {

**this**.id = id;

}

**public** Sosi getSosi() {

**return** sosi;

}

**public** void setSosi(Sosi sosi) {

**this**.sosi = sosi;

}

**public** String getProgram() {

**return** program;

}

**public** void setProgram(String program) {

**this**.program = program;

}

}

두 개의 모델이 1:N관계 입니다. Sosi가 스케줄을 여러개 가질 수 있는 구조입니다. 여기서 Schedule에는 ManyToOne을 걸었는데, @JsonBackReference도 같이 넣었습니다. 이걸 안 넣으면 서로 계속 참조해서 JSON출력할 때 무한루프에 빠지더라구요-\_- 이제 Sosi참조는 json에서 안쓰는 그런 옵션 같습니다.

그리고 예전에는 Dao만들어서 뭔가 하이버네이트세션 가져와서 그걸 통해서 했던 것 같은데, Spring Data JPA에서는 JpaRepository라는 것을 제공하는데, 이것을 통하면 해당 모델에 대해서 CRUD를 제공합니다.

**SosiRepository.java**

**public** **interface** **SosiRepository** **extends** JpaRepository<Sosi, Long>{

}

**ScheduleRepository.java**

**public** **interface** **ScheduleRepository** **extends** JpaRepository<Schedule, Long> {

}

**4. 컨트롤러 생성**

간단하게 소녀시대 정보를 가져오는 컨트롤러와 스케줄 추가 및 가져오는 컨트롤러를 생성합니다.

**SosiController.java**

@RestController

@RequestMapping("/sosi")

**public** **class** **SosiController** {

@Autowired

**private** SosiRepository sosiRepository;

@RequestMapping("{sosiId}")

**public** Sosi getSosi(@PathVariable Long sosiId) {

Sosi sosi = sosiRepository.findOne(sosiId);

**return** sosi;

}

}

**ScheduleController.java**

@RestController

@RequestMapping("/schedule")

**public** **class** **ScheduleController** {

@Autowired

**private** ScheduleRepository scheduleRepository;

@Autowired

**private** SosiRepository sosiRepository;

@RequestMapping("{scheduleId}")

**public** Schedule getSchedule(@PathVariable Long scheduleId) {

Schedule schedule = scheduleRepository.findOne(scheduleId);

Sosi sosi = schedule.getSosi();

**return** schedule;

}

@RequestMapping("add/{sosiId}")

**public** Schedule addSchedule(@PathVariable Long sosiId, @RequestParam(value="program") String program) {

Sosi sosi = sosiRepository.findOne(sosiId);

Schedule schedule = scheduleRepository.save(**new** Schedule(sosi, program));

**return** schedule;

}

}

소스 내용을 보면 Repository클래스를 통해 Autowired하면 기본 인터페이스를 해당 모델기반 구현체를 만듭니다. 그 구현체에서 save, findOne 등 함수를 통해 데이터 삽입 및 가져올 수 있습니다.

**5. 기본 값 삽입**

기본적으로 소녀시대 멤버를 삽입하고 시작할 수 있습니다. 기존 application.properties에 하이버네이트 설정을 재시작하면 꺼지게 해놨기 때문에 기본적으로 앱을 시작할 때 값을 삽입하고 하면 편하게 테스트할 수 있습니다.

**Application.java**

@SpringBootApplication

**public** **class** **Application** {

**public** **static** void main(String[] args) {

ConfigurableApplicationContext context = SpringApplication.run(Application.class, args);

SosiRepository sosiRepository = context.getBean(SosiRepository.class);

sosiRepository.save(**new** Sosi("태연"));

sosiRepository.save(**new** Sosi("윤아"));

sosiRepository.save(**new** Sosi("수영"));

sosiRepository.save(**new** Sosi("효연"));

sosiRepository.save(**new** Sosi("유리"));

sosiRepository.save(**new** Sosi("티파니"));

sosiRepository.save(**new** Sosi("써니"));

sosiRepository.save(**new** Sosi("서현"));

}

}

**6. 실행**

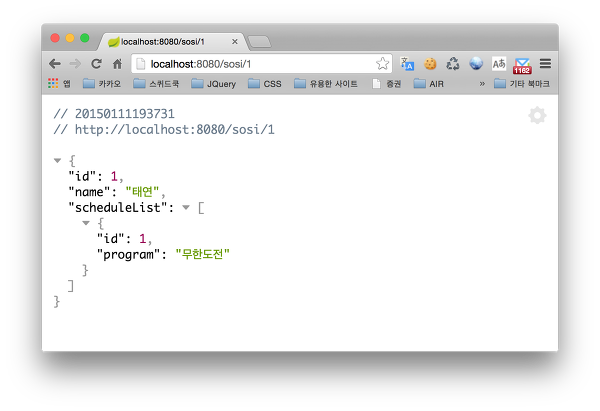
소녀시대 정보 가져오기  
http://localhost:8080/sosi/1



스케줄 추가하기  
http://localhost:8080/schedule/add/1?program=무한도전



다시 소녀시대정보 가져오기  
http://localhost:8080/sosi/1



예제 소스는 깃헙에...  
<https://github.com/mudchobo/sosi-schedule-sb>

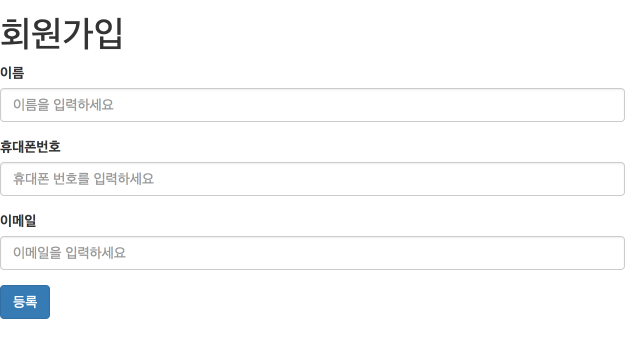
끗

### [Spring Validation 공통모듈 만들기](http://jojoldu.tistory.com/129)

창천향로 창천향로 2017.03.06 00:04

# Validation 공통모듈 만들기

안녕하세요? 이번 시간엔 유효성 체크(이하 validation) 공통 모듈 생성하는 예제를 진행해보려고 합니다. 모든 코드는 [Github](https://github.com/jojoldu/blog-code/tree/master/spring-validation)에 있기 때문에 함께 보시면 더 이해하기 쉬우실 것 같습니다.   
(공부한 내용을 정리하는 [Github](https://github.com/jojoldu/blog-code)와 세미나+책 후기를 정리하는 [Github](https://github.com/jojoldu/review), 이 모든 내용을 담고 있는 [블로그](http://jojoldu.tistory.com/)가 있습니다. )

예를 들어 아래와 같은 회원 가입 양식이 있다고 가정해보겠습니다.   


대부분의 입력 양식처럼 위 양식도 등록시 몇가지 조건들이 있습니다.

* 이름/휴대폰번호/이메일은 **필수값**입니다.
* 휴대폰번호는 **10자리 혹은 11자리의 숫자**로 이루어져야 합니다.
* 이메일은 이메일 양식을 지켜야한다 (xxx@xxx)
* 이메일은 **중복 등록 할 수 없다**.

이 조건들을 체크하기 위해서 많은 분들이 하는 방법중 하나가 jquery validate 등을 이용한 **클라이언트 사이드에서 체크**를 하는 것입니다.   
하지만 클라이언트 사이드에서 validation을 하게 될 경우 얼마든지 브라우저에서 조작이 가능하기 때문에 validation을 무효화해서 잘못된 데이터를 전송할 수 있습니다.   
그래서 이런 체크는 백엔드에서 필수적으로 하게 되는데, 이때 유용하게 사용되는 것이 **Validation 어노테이션**들입니다.   
위 조건을 만족하도록 간단하게 프로젝트를 구성해보겠습니다.

### 코드

예제코드에 앞서서 좀 더 실제 프로젝트와 유사하게 맞추기 위해 조건을 추가하겠습니다.

* 휴대폰 번호는 입력 받을때는 숫자만 입력 받는다.
* 휴대폰 번호를 DB 저장시에는 조건 검색이 가능하도록 3개 컬럼에 나눠서 저장한다. (ex: 컬럼1: 010, 컬럼2: 1234, 컬럼3: 5678)
* 휴대폰 번호 출력시에는 - 문자가 포함되야 한다 (ex: 010-1234-5678)

그럼 이제 추가 조건까지 포함해서 코드를 작성해보겠습니다.   
먼저 프로젝트 의존성 관리를 위해 build.gradle을 아래와 같이 추가하였습니다.   
  
  
**build.gradle**

buildscript {

ext {

springBootVersion = '1.5.1.RELEASE'

}

repositories {

mavenCentral()

}

dependencies {

classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")

}

}

apply plugin: 'java'

apply plugin: 'eclipse'

apply plugin: 'org.springframework.boot'

jar {

baseName = 'spring-validation'

version = '0.0.1-SNAPSHOT'

}

sourceCompatibility = 1.8

repositories {

mavenCentral()

}

dependencies {

compile('org.springframework.boot:spring-boot-starter-data-jpa')

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

compile('org.springframework.boot:spring-boot-devtools')

runtime('com.h2database:h2')

testCompile('org.springframework.boot:spring-boot-starter-test')

}

이외 추가 설정은 없기 때문에 바로 자바 코드로 넘어가겠습니다.   
  
  
**Member**

@Entity

public class Member {

@Id

@GeneratedValue

private Long id;

@Column(nullable = false)

private String name;

@Column(nullable = false)

private String phone1;

@Column(nullable = false)

private String phone2;

@Column(nullable = false)

private String phone3;

@Column(nullable = false)

private String email;

public Member() {}

public Member(String name, String phone1, String phone2, String phone3, String email) {

this.name = name;

this.phone1 = phone1;

this.phone2 = phone2;

this.phone3 = phone3;

this.email = email;

}

public Long getId() {

return id;

}

public String getName() {

return name;

}

public String getPhone1() {

return phone1;

}

public String getPhone2() {

return phone2;

}

public String getPhone3() {

return phone3;

}

public String getEmail() {

return email;

}

}

**MemberRequestDto**

public class MemberRequestDto {

private Long id;

@NotBlank(message = "이름을 작성해주세요.")

private String name;

@NotBlank(message = "전화번호를 작성해주세요.")

@Pattern(regexp = "[0-9]{10,11}", message = "10~11자리의 숫자만 입력가능합니다")

private String phoneNumber;

@NotBlank(message = "메일을 작성해주세요.")

@Email(message = "메일의 양식을 지켜주세요.")

private String email;

public MemberRequestDto() {}

public Member toEntity(){

String[] phones = parsePhone();

return new Member(name, phones[0], phones[1], phones[2], email);

}

private String[] parsePhone(){

String[] phones = new String[3];

int mid = phoneNumber.length() == 10? 7:8;

phones[0] = phoneNumber.substring(0,3);

phones[1] = phoneNumber.substring(4,mid);

phones[2] = phoneNumber.substring(mid,phoneNumber.length()-1);

return phones;

}

public Long getId() {

return id;

}

public String getName() {

return name;

}

public String getPhoneNumber() {

return phoneNumber;

}

public String getEmail() {

return email;

}

}

**MemberResponseDto**

public class MemberResponseDto {

private Long id;

private String name;

private String phoneNumber;

private String email;

public MemberResponseDto() {}

public MemberResponseDto(Member member) {

id = member.getId();

name = member.getName();

phoneNumber = toStringPhone(member.getPhone1(), member.getPhone2(), member.getPhone3());

email = member.getEmail();

}

private String toStringPhone(String phone1, String phone2, String phone3){

return phone1+"-"+phone2+"-"+phone3;

}

public Long getId() {

return id;

}

public String getName() {

return name;

}

public String getPhoneNumber() {

return phoneNumber;

}

public String getEmail() {

return email;

}

}

Member, MemberRequestDto, MemberResponseDto 3개의 클래스를 생성하였습니다.   
회원 정보를 나타내기 위해 Member클래스만 사용하지 않은 이유는, Entity 클래스를 파라미터 혹은 View 데이터로 사용하게 되면 **변화에 대응하기가 힘들기 때문**입니다.   
휴대폰 번호 같은 경우 테이블에 저장되는 형태는 3개의 컬럼으로 저장되는데, 화면에 입력 받는 형태는 하나의 문자열입니다. 이걸 Entity에서 구현하려면 Entity 클래스는 테이블의 역할을 벗어난 **많은 책임을 담당**하게 되고, **파라미터가 변경될 때마다 메인이 되는 Entity 클래스의 구조가 계속해서 변경**되게 됩니다.   
반면에 MemberRequestDto가 화면에서 입력 받는 파라미터 타입의 역할을 하게 될 경우 파라미터 형태가 변경되어도 MemberRequestDto만 변경하면 되고 Entity 클래스는 변경되지 않아도 되기에 테이블 구조가 변경될 일은 없습니다.   
추가적으로 **Entity 클래스에 validation 어노테이션까지 포함**되어있으면 코드 자체가 너무 지저분해지는데, 분리하게 될 경우 이역시도 깔끔하게 코드가 작성 될 수 있습니다.   
동일한 이유로 View에 출력되는 타입 역시 Entity가 아닌 Response 클래스를 별도로 두어 어떤 출력 형태라도 큰 변경 없이 대응할 수 있도록 하였습니다.   
  
  
그럼 이를 사용하는 Controller/Service/Repository를 생성하겠습니다.   
  
  
**MemberRepository**

public interface MemberRepository extends JpaRepository<Member, Long>{

Optional<Member> findByEmail(String email);

}

repository는 간단하게 JpaRepository를 상속 받겠습니다.   
  
  
**MemberService**

@Service

public class MemberService {

private MemberRepository memberRepository;

public MemberService(MemberRepository memberRepository) {

this.memberRepository = memberRepository;

}

@Transactional

public Long save(MemberRequestDto memberRequestDto){

return memberRepository.save(memberRequestDto.toEntity()).getId();

}

@Transactional(readOnly = true)

public List<MemberResponseDto> findAll() {

return memberRepository

.findAll()

.stream()

.map(MemberResponseDto::new)

.collect(Collectors.toList());

}

}

스프링에 대한 의존성을 낮추고 Mock 객체 주입을 좀 더 쉽게하기 위해 생성자 Injection을 하였습니다.   
보시면 save할때는 requestDto를 entity로, find를 할 때는 entity를 responseDto로 전환하는 과정을 거치게 됩니다.   
만약 리턴되는 데이터가 많을 경우에 모든 데이터를 map으로 전환하기가 부담되신다면, **querydsl을 사용하여 조회 결과를 responseDto로** 받을 수 도 있습니다.

**MemberController**

@RestController

public class MemberController {

private MemberService memberService;

public MemberController(MemberService memberService) {

this.memberService = memberService;

}

@PostMapping("/member")

public Long saveMember(@RequestBody @Valid MemberRequestDto memberRequestDto) {

return memberService.save(memberRequestDto);

}

@GetMapping("/members")

public List<MemberResponseDto> findAll(){

return memberService.findAll();

}

}

Controller는 별다를 것이 없지만, JSON 데이터를 인스턴스에 매핑하기 위해 @RequestBody를, validation 실행을 위해 @Valid를 사용하였습니다.   
  
  
백엔드구성은 마쳤습니다. 다음은 화면단 코드를 작성하겠습니다.   
Ajax로 데이터를 주고 받을 예정이기에 별도의 템플릿 엔진(freemarker, thymleaf 등)을 사용하지 않겠습니다.   
단, SpringBoot의 경우 / url에 매핑된 메소드가 없고, src/main/resources/static/index.html을 생성하면 / 접속시 **자동으로 index.html을 호출**하게 됩니다. 이를 이용하여 index.html을 작성하겠습니다.   
  
  
**index.html**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Validation 서버에 집중하기</title>

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css">

</head>

<body>

<div class="row">

<div class="col-md-6">

<h1> 회원가입 </h1>

<form>

<div class="form-group">

<label for="name">이름</label>

<input type="text" class="form-control" id="name" placeholder="이름을 입력하세요">

</div>

<div class="form-group">

<label for="phoneNumber"> 휴대폰번호 </label>

<input type="text" class="form-control" id="phoneNumber" placeholder="휴대폰 번호를 입력하세요">

</div>

<div class="form-group">

<label for="email">이메일</label>

<input type="text" class="form-control" id="email" placeholder="이메일을 입력하세요">

</div>

<button type="button" class="btn btn-primary" id="btnSave">등록</button>

</form>

</div>

<div class="col-md-6">

<h1> 회원 리스트 </h1>

<table class="table table-horizontal table-bordered">

<thead class="thead-strong">

<tr>

<th>ID</th>

<th>이름</th>

<th>휴대폰번호</th>

<th>Email</th>

</tr>

</thead>

<tbody id="tbody">

</tbody>

</table>

</div>

</div>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.1.1/jquery.min.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/4.0.6/handlebars.min.js"></script>

<script id="table-template" type="text/x-handlebars-template">

{{#each members}}

<tr>

<td>{{id}}</td>

<td>{{name}}</td>

<td>{{phoneNumber}}</td>

<td>{{email}}</td>

</tr>

{{/each}}

</script>

<script type="application/javascript">

var $tbody = $('#tbody');

var findAll = function () {

$.ajax({

url:'/members',

method: 'GET',

success: function (response) {

var source = $("#table-template").html();

var template = Handlebars.compile(source);

var html = template({members:response});

$tbody.html('');

$tbody.html(html);

}

});

};

$('#btnSave').click(function () {

var member = {

name: $('#name').val(),

phoneNumber: $('#phoneNumber').val(),

email: $('#email').val()

};

$.ajax({

url:'/member',

method: 'POST',

data: JSON.stringify(member),

contentType: "application/json; charset=utf-8",

success: function (response) {

findAll();

}

});

});

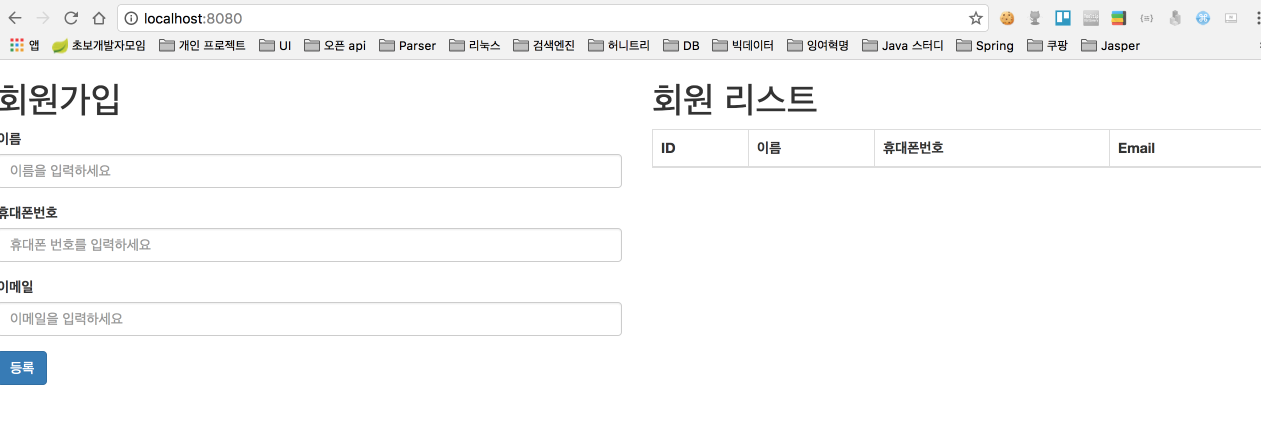
findAll();

</script>

</body>

</html>

bootstrap, jquery, handlebars를 사용한 간단한 회원가입, 회원리스트 기능의 게시판입니다.   
(혹시나 handlebars에 대해 잘 모르시는 분들은 이전에 제가 작성한 [handlebars 글](http://jojoldu.tistory.com/23)을 읽어보시길 추천드립니다.)   
프로젝트를 구동시켜 http://localhost:8080/ 에 접속하여 아래와 같은 화면이 출력되는지 확인하겠습니다.



자 그럼 여기서 개발자 도구를 열고, 일부러 조건에 맞지 않는 값들을 입력하여 등록 버튼을 클릭해보겠습니다.



그럼 위 처럼 400 에러와 함께 여러 에러 결과를 확인이 가능한데, 이를 [jsoneditoronline](http://www.jsoneditoronline.org/) 에서 정리해서 확인해보겠습니다.



이 많은 결과값에서 저희가 집중해야할 것은 **defaultMessage**와 **field**입니다.   
전체 결과를 보시면 아시겠지만, 모든 validation 에러들은 errors 배열에 담겨오며 defaultMessage와 field값을 가지고 있습니다.   
defaultMessage는 validation 어노테이션에 지정한 메세지를 얘기하며, field는 해당 클래스 인스턴스의 필드명을 얘기합니다.   
그렇다면 field 값과 defaultMessage를 활용하면 **validation 에러가 발생하면 자동으로 해당 필드에 메세지를 출력**할 수 있지 않을까요?   
validation을 어노테이션을 통해 간단하고 일관되게 처리후 그 결과는 자동으로 해당 필드에 표현 될 수 있도록 jquery 코드를 추가하겠습니다.   
이 기능을 하는 함수명은 markingErrorField라 하겠습니다.   
  
  
**index.html**

var markingErrorField = function (response) {

const errorFields = response.responseJSON.errors;

if(!errorFields){

alert(response.response.message);

return;

}

var $field, error;

for(var i=0, length = errorFields.length; i<length;i++){

error = errorFields[i];

$field = $('#'+error['field']);

if($field && $field.length > 0){

$field.siblings('.error-message').remove();

$field.after('<span class="error-message text-muted taxt-small text-danger">'+error.defaultMessage+'</span>');

}

}

};

코드 자체는 특별한 것이 없습니다.   
error 배열에 있는 값들을 하나씩 호출하여 해당 필드명을 ID로 가진 Dom Element를 찾아 defaultMessage를 붙이는 것($field.after('<span class="error-message text-muted taxt-small text-danger">'+error.defaultMessage+'</span>')입니다.   
그 과정에서 이전에 출력시킨 에러메세지들은 전부 삭제($field.siblings('.error-message').remove())하여 중복으로 남기지 않도록 하였습니다.   
그리고 기존의 save 코드에 error 발생시 markingErrorField를 실행시키도록 수정하겠습니다.

$('#btnSave').click(function () {

var member = {

name: $('#name').val(),

phoneNumber: $('#phoneNumber').val(),

email: $('#email').val()

};

$.ajax({

url:'/member',

method: 'POST',

data: JSON.stringify(member),

contentType: "application/json; charset=utf-8",

success: function (response) {

findAll();

},

error: function (response) {

markingErrorField(response);

}

});

});

자 그럼 실제로 작동되는 모습을 확인해보겠습니다.



짠! 잘되는 것을 확인할 수 있습니다.   
validation 공통 모듈은 여기서 끝일까요?   
위에 있는 조건중 만족시키지 못한 것이 무엇일까요?  
validation 어노테이션으로 대부분의 처리는 가능하나, **중복체크와 같이 다른 서비스 Layer가 필요한 경우**는 불가능합니다.   
그래서 이 문제를 email 중복체크를 해결하면서 진행해 보겠습니다.

### ValidCustomException

validation 어노테이션들로 해결할 수 없는 문제들이 발생할 경우에도 markingErrorField가 처리할 수 있도록 defaultMessage와 field를 가진 값들을 errors 배열 안에 담은 Exception 클래스를 생성하겠습니다.   
  
  
**ValidCustomException**

@ResponseStatus(HttpStatus.BAD\_REQUEST)

public class ValidCustomException extends RuntimeException{

private Error[] errors;

public ValidCustomException(String defaultMessage, String field){

this.errors = new Error[]{new Error(defaultMessage, field)};

}

public ValidCustomException(Error[] errors) {

this.errors = errors;

}

public Error[] getErrors() {

return errors;

}

public static class Error {

private String defaultMessage;

private String field;

public Error(String defaultMessage, String field) {

this.defaultMessage = defaultMessage;

this.field = field;

}

public String getDefaultMessage() {

return defaultMessage;

}

public String getField() {

return field;

}

}

}

validation 에러가 발생했을때처럼 동일하게 400 status code를 지정하였습니다. (@ResponseStatus(HttpStatus.BAD\_REQUEST))   
**ValidCustomException에서만 사용**하는 Error 이너클래스를 생성하여 defaultMessage와 field를 추가하였습니다.   
이것만 가지고 바로 사용할수는 없습니다.   
Spring에서는 직접 생성한 Exception에 새로운 field(여기선 Error[] errors)를 포함해서 view에 전달하기 위해선 **DefaultErrorAttributes에서 필드를 직접 추가**해야만 합니다.

**Application**

@Bean

public ErrorAttributes errorAttributes() {

return new DefaultErrorAttributes() {

@Override

public Map<String, Object> getErrorAttributes(

RequestAttributes requestAttributes,

boolean includeStackTrace) {

Map<String, Object> errorAttributes = super.getErrorAttributes(requestAttributes, includeStackTrace);

Throwable error = getError(requestAttributes);

if (error instanceof ValidCustomException) {

errorAttributes.put("errors", ((ValidCustomException)error).getErrors());

}

return errorAttributes;

}

};

}

Exception의 타입을 체크하여 ValidCustomException일 경우에는 errors 필드를 추가하는 코드입니다.   
자 드디어 ValidCustomException을 사용할 수 있게 되었습니다!!   
그럼 email 체크하는 코드를 추가하여 테스트를 진행해보겠습니다.

**MemberService**

@Transactional

public Long save(MemberRequestDto memberRequestDto){

verifyDuplicateEmail(memberRequestDto.getEmail());

return memberRepository.save(memberRequestDto.toEntity()).getId();

}

private void verifyDuplicateEmail(String email){

if(memberRepository.findByEmail(email).isPresent()){

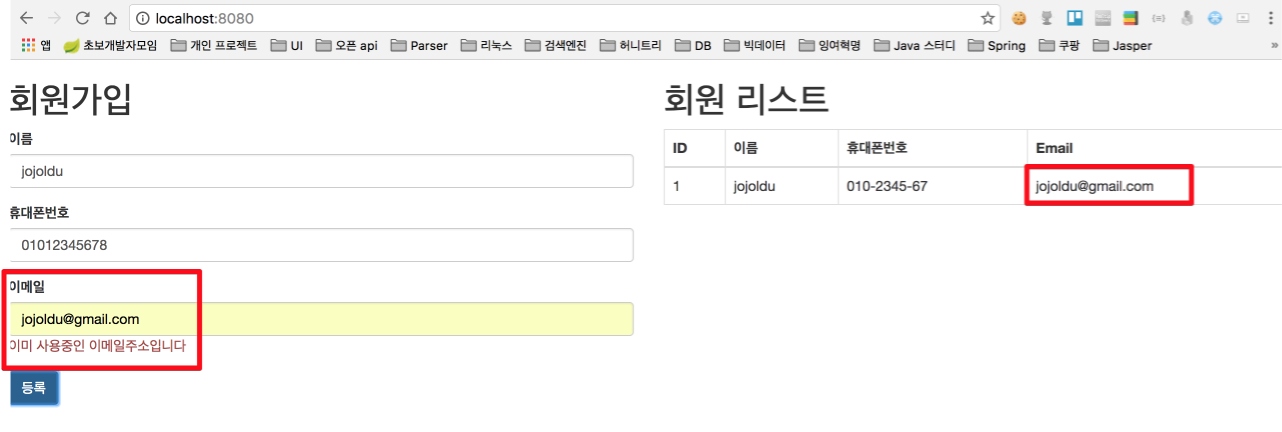
throw new ValidCustomException("이미 사용중인 이메일주소입니다", "email");

}

}

save 하기전, verifyDuplicateEmail 메소드를 통해 email 체크를 진행하고, 동일한 email이 이미 등록되어 있을 경우 **email 필드를 가진** ValidCustomException 을 발생시키도록 하였습니다.

기능이 잘 작동되는지 확인하기 위해 동일한 이메일을 2번 입력해보겠습니다.



처음 jojoldu@gmail.com으로 등록 후, 다시 한번 jojoldu@gmail.com 입력 시 이메일 항목 아래에 메세지가 출력되는 것을 확인할 수 있습니다.   
이렇게 만들 경우 비슷한 많은 체크성 코드들을 전부 제거할 수 있습니다.   
어떠셨나요?   
막상 시작하기전에는 짧게 끝나겠다고 생각하였는데 하다보니 내용이 너무 길어져 읽으시는데 지겹진 않으셨을까 걱정이 되기도 하였습니다.   
그래도 저와 같은 초보자 분들에겐 길더라도 상세하게 작성되는 것이 이해에 도움이 되실것 같아 줄이지 않았습니다.   
여튼 이 긴글을 끝까지 읽어주셔서 다시 한번 감사드립니다!

출처: <http://jojoldu.tistory.com/129> [기억보단 기록을]

# Spring에서 YAML 파일 데이터 객체에 매핑하여 로드하기

# http://blog.saltfactory.net/java/load-yaml-file-in-spring.html

## 서론

Spring 프로젝트를 진행할 때 외부에서 데이터를 로드할 경우가 종종 있다. 가장 쉽게는 Spring Boot에서 사용하는 Configuration Porperty를 로드하는 것이다. Spring Boot는 기본적으로 application.properties 파일을 추가하면 자동으로 [Common application properteis](http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html) 로드하여 프로퍼티 값을 적용할 수 있다. 하지만 자바의 [Properties](https://docs.oracle.com/javase/tutorial/essential/environment/properties.html) 의 파일의 사용에는 표현의 한계가 있기 때문에 최근에는 Properties를 [YAML](http://yaml.org)을 많이 사용하고 있다. Spring Boot에서는 [SnakeYAML](https://bitbucket.org/asomov/snakeyaml)을 포함하고 있어서 쉽게 외부 파일을 YAML으로 작성하여 쉽게 로드하여 객체로 매핑할 수 있다. 이번 포스팅에서는 Spring Boot에서 YAML로 작성한 파일을 객체로 매핑하여 사용하는 방법을 소개한다.

## 테스트를 위한 데이터 YAML 파일로 만들기

테스트를 위해 간단한 데이터가 필요하다. 이 때 처음부터 JPA와 같이 ORM을 가지고 데이터를 만들어서 사용하려면 꽤 여러가지 일을 해야한다. 우리는 테스트를 위한 파일을 쉽게 가져올 수 있게하기 위해서 YAML 파일을 사용하기로 한다. Spring Boot는 [application.properties 이나 application.yml](http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html) 파일에 필요한 설정을 정의하면 어플리케이션에서 자동으로 읽어들일 수 있다. Spring Boot 프로젝트를 생성하면 **src/main/resources/application.properties** 라는 어플리케이션 프로퍼티 파일이 만들어 진다. Spring에서 복잡한 XML 설정을 Spring Boot에서는 이 파일 안에서 간단하게 설정하여 어플리케이션에 적용할 수 있다. http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html 자세한 사항은 링크를 참조하면 된다.

Spring Boot에서는 [Snake YAML](https://bitbucket.org/asomov/snakeyaml) 라이브러리를 내장하고 있다. 이런 이유로 Spring Boot에서는 [YAML](http://yaml.org/) 파일을 로드하여 사용할 수 있다. http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#boot-features-external-config-yaml (링크를 참조). 뿐만아니라 **src/main/resources/application.yml** 파일에 설정을 정의하면 application.properties와 같이 Spring Boot에서 자동으로 읽어와 필요한 곳에 매핑된다.

[YAML](http://yaml.org/) 파일은 .properties 파일과 달리 계층과 배열 구조의 데이터를 쉽게 만들 수 있고 이것을 Map, List 또는 Bean에 쉽게 편리하게 매핑할 수 있다. Ruby on Rails에서는 이미 오래전부터 YAML 파일을 사용하여 데이터베이스나 시스템 설정 파일로 사용해 왔다. YAML은 JSON과 비슷하지만 표현법이 더 간단하여 인기있기 사용되고 있다.

우리는 테스트를 위한 데모 데이터세트를 Spring Boot가 환경 설정을 읽어오는 것과 유사하게 로드하게 할 것이다. 먼저 데모 데이터를 **src/main/resources/fixtures.yml** 에 저장한다.

fixtures:

articles:

-

id: 1

title: title1

content: content1

-

id: 2

title: title2

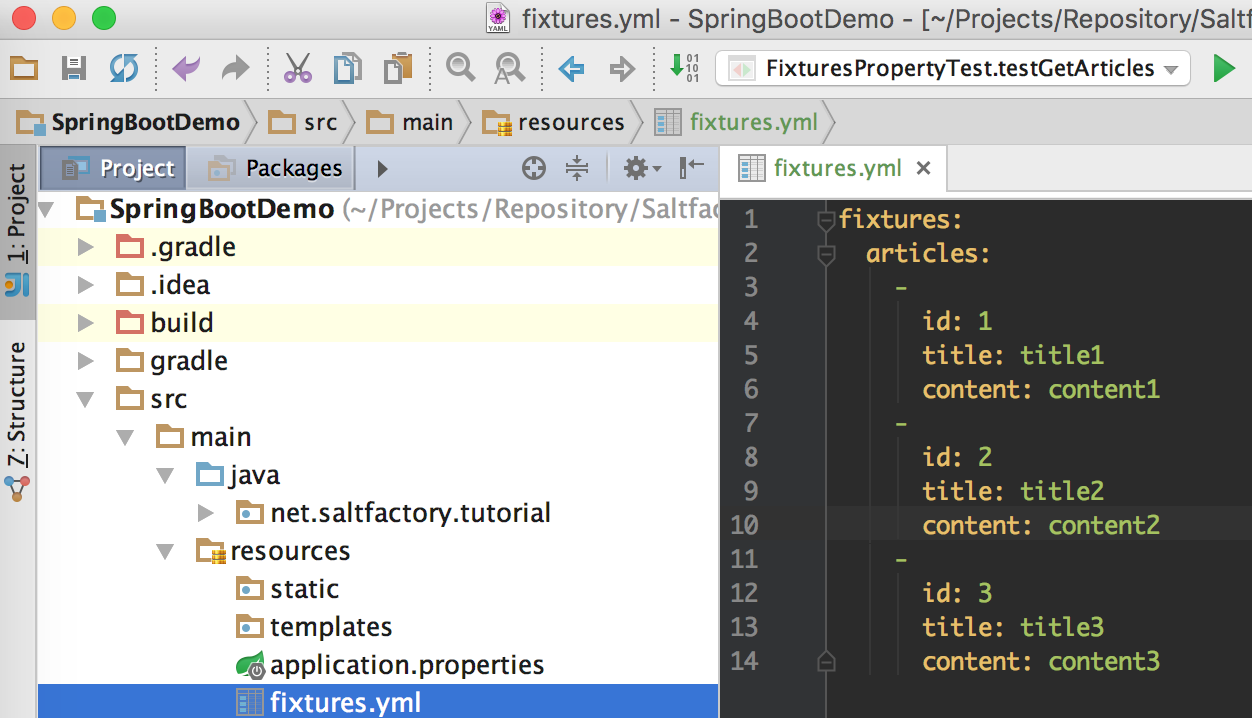
content: content2

-

id: 3

title: title3

content: content3



## YAML 파일을 Map으로 매핑하기

먼저 YAML 파일을 로드하기 위해서는 외부에서 Configuration을 로드 하기 위해 [Externalized Configuration](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html) (글을 참조) 설정을 해야한다. 간단하게 다시 말하자면 YAML 파일을 Spring 어플리케이션어 로드하기 위한 컴포넌트가 필요하다. 우리는 fixtures.yml 파일을 Configuration Property 파일 형태로 로드하기 위해서 **src/main/java/{패캐지명}/FixturesProperty.java** 파일을 만든다. 여기서 주의할 점은 앞에서 만든 **fixtures.yml** 파일을 로드하기 위해서 classpath 위치를 지정하는데 SpringBoot는 기본적으로 **src/main/resources/** 디렉토리를 클래스 패스로 가지고 있기 때문에 아래와 같이 지정한다. 그리고 fixtures.yml 파일 안에 articles는 **fixtures** 라는 Key의 List value로 만들어 놓았기 때문에 YAML 파일을 읽어들일 때 **prefix**로 **fixtures**를 정의하여 이 키 값 안의 데이터를 로드하게 한다. 그리고 우리가 만든 fixtures.yml 파일은 article 내용이 YAML의 배열로 작성했기 때문에 나중에 **List<Map>**로 매핑될 것이다. 외부 프로퍼티 파일(.propertis 나 .yml)을 Injection 로드하기 위해서는 Spring의 컴포넌트가 되어야하기 때문에 **@Component** 어노테이션을 추가하고 프로퍼티 파일을 로드하기 위한 클래스라는 것을 정의하기 위해서 **@ConfigurationProperties** 어노테이션을 추가한다.

package net.saltfactory.tutorial;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.boot.context.properties.NestedConfigurationProperty;

import org.springframework.stereotype.Component;

import java.util.ArrayList;

import java.util.List;

import java.util.Map;

/\*\*

\* Filename : FixturesProperty.java

\* Author : saltfactory<saltfactory@gmail.com>

\* Created : 11/23/15.

\*/

@Component

@ConfigurationProperties(locations = {"fixtures.yml"}, prefix = "fixtures")

public class FixturesProperty {

@NestedConfigurationProperty

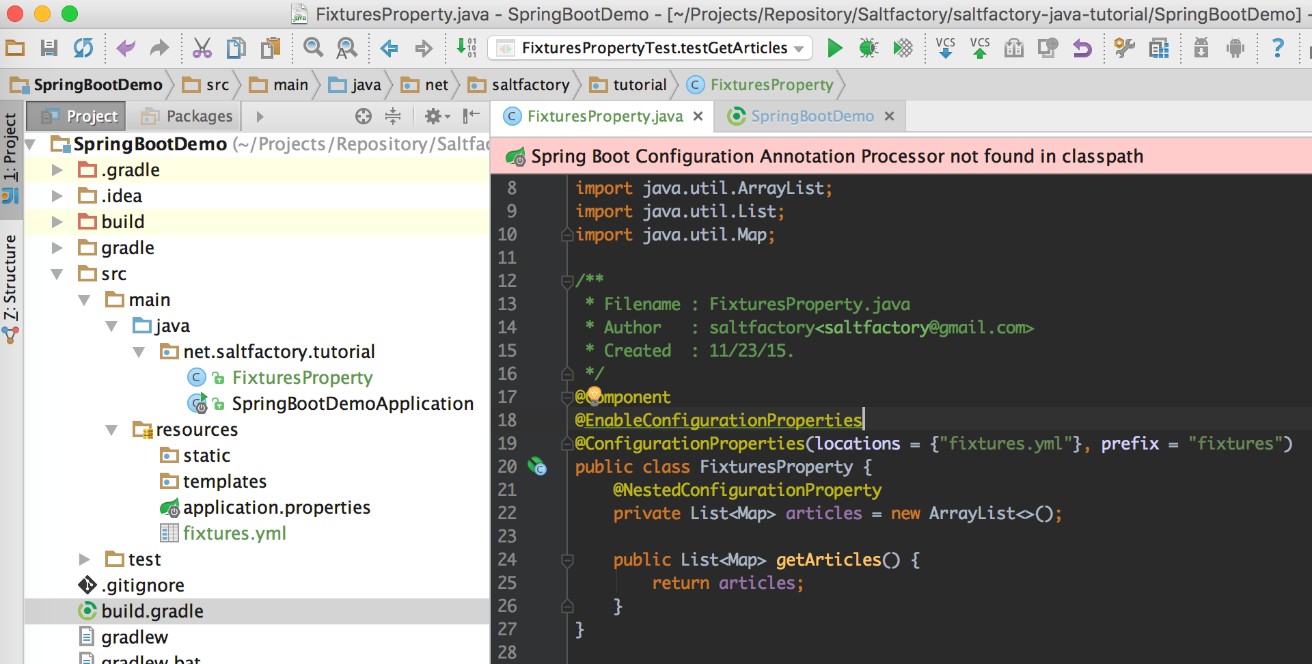
private List<Map> articles = new ArrayList<>();

public List<Map> getArticles() {

return articles;

}

}



스크린샷과 같이 Spring Boot Configuration Annoation Processor를 찾을 수 없다는 에러가 나타나면 다음과 build.gradle 파일을 열어서 다음 내용을 추가한다. 원래 Spring Boot에서 외부 프로퍼티 파일을 로드하기 위해서는 메타 정보를 파일로 만들어서 추가해야는데 [propdes-plugin](https://github.com/spring-projects/gradle-plugins/tree/master/propdeps-plugin)을 사용하면 메타 파일을 추가하지 않고 자동으로 적용할 수 있다. http://docs.spring.io/spring-boot/docs/1.3.0.RELEASE/reference/html/configuration-metadata.html#configuration-metadata-annotation-processor (글을 참조)

프로젝트 안의 **build.gradle** 파일을 열어서 다음과 같이 수정하고 gradle.properties 파일을 적용한다.(IntelliJ에서는 이 파일을 수정하고 Gradle projects 패널에서 새로고침 버튼을 누르면 의존성 있는 라이브러리를 자동으로 다운 받게 된다)

buildscript {

ext {

springBootVersion = '1.3.0.RELEASE'

}

repositories {

mavenCentral()

maven { url 'http://repo.spring.io/plugins-release' }

}

dependencies {

classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")

classpath("org.springframework.build.gradle:propdeps-plugin:0.0.7")

}

}

apply plugin: 'java'

apply plugin: 'eclipse'

apply plugin: 'idea'

apply plugin: 'propdeps'

apply plugin: 'spring-boot'

apply plugin: 'propdeps-idea'

jar {

baseName = 'spring-boot-demo'

version = '0.0.1-SNAPSHOT'

}

sourceCompatibility = 1.8

targetCompatibility = 1.8

repositories {

mavenCentral()

}

dependencies {

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

optional("org.springframework.boot:spring-boot-configuration-processor")

testCompile('org.springframework.boot:spring-boot-starter-test')

}

compileJava.dependsOn(processResources)

eclipse {

classpath {

containers.remove('org.eclipse.jdt.launching.JRE\_CONTAINER')

containers 'org.eclipse.jdt.launching.JRE\_CONTAINER/org.eclipse.jdt.internal.debug.ui.launcher.StandardVMType/JavaSE-1.8'

}

}

task wrapper(type: Wrapper) {

gradleVersion = '2.9'

}

이제 외부 프로퍼티 파일을 로드하기 위한 설정을 모두 마쳤다. 우리가 만든 fixtures.yml 파일을 FixturesProperty 클래스가 잘 로드하는지 확인하기 위해서 Test 파일을 만들어보자. **src/test/java/{패키지경로}/FixturesPropertyTest.java**. 앞에서 FixtureProperty는 Spring의 **@Component**로 만들었기 때문에 스캐닝되고 **@ConfigurationProperty** 때문에 YAML 파일을 오브젝트에 매핑되어 반환하게 될 것이다. 우리는 fixtures.yml에 3개의 아이템을 리스트로 만들었기 때문에 테스트에서 리스트의 사이즈를 3이 맞는지 테스트를 진행하였다.

package net.saltfactory.tutorial;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.SpringApplicationConfiguration;

import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import java.util.List;

import java.util.Map;

import static org.hamcrest.MatcherAssert.assertThat;

import static org.hamcrest.core.Is.is;

/\*\*

\* filename : FixturespropertyTest.java

\* author : saltfactory<saltfactory@gmail.com>

\* created : 11/23/15

\*/

@RunWith(SpringJUnit4ClassRunner.class)

@SpringApplicationConfiguration(classes = SpringBootDemoApplication.class)

public class FixturesPropertyTest {

@Autowired

private FixturesProperty fixturesProperty;

@Test

public void testGetArticles() {

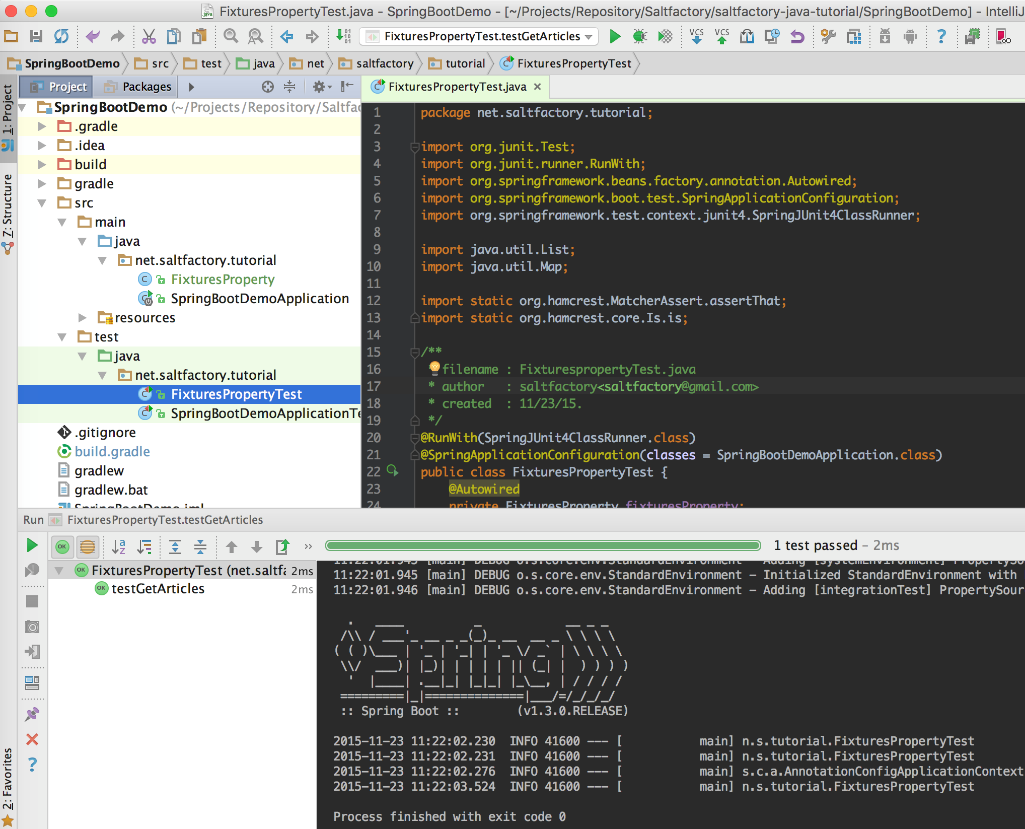
List<Map> articles = fixturesProperty.getArticles();

assertThat(articles.size(), is(3));

}

}

테스트 파일을 만들었으면 단위 테스트를 실행해보자.



## YAML 파일을 POJO로 매핑하기

위에 yaml로 만든 외부 파일을 Map으로 매핑을 할 수 있는 것을 살펴보았다. Spring 프로젝트에서는 데이터를 저장하는 객체를 POJO로 만들어서 setter/getter를 한다. YAML의 데이터를 POJO 객체로 바로 매핑하는 방법을 살펴보자. 우선 Article 객체를 fixtures.yml 파일에 정의한 key 값과 동일한 이름으로 field와 setter/getter를 만든다.

package net.saltfactory.tutorial;

import java.io.Serializable;

/\*\*

\* filename : Article.java

\* author : saltfactory<saltfactory@gmail.com>

\* created : 11/23/15

\*/

public class Article implements Serializable {

private long id;

private String title;

private String content;

public long getId() {

return id;

}

public void setId(long id) {

this.id = id;

}

public String getTitle() {

return title;

}

public void setTitle(String title) {

this.title = title;

}

public String getContent() {

return content;

}

public void setContent(String content) {

this.content = content;

}

}

앞에서 외부 Configuration Property를 로드하기 위한 **FixturesProperty**를 열어서 로드할 타입을 **Map**에서 **Article**로 변경한다.

package net.saltfactory.tutorial;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.boot.context.properties.NestedConfigurationProperty;

import org.springframework.stereotype.Component;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* Filename : FixturesProperty.java

\* Author : saltfactory<saltfactory@gmail.com>

\* Created : 11/23/15.

\*/

@Component

@ConfigurationProperties(locations = {"fixtures.yml"}, prefix = "fixtures")

public class FixturesProperty {

private List<Article> articles = new ArrayList<>();

public List<Article> getArticles() {

return articles;

}

}

테스트를 위한 **FixturesPropertyTest** 파일을 수정하고 브레이크포인트를 걸어서 확인해보자.

package net.saltfactory.tutorial;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.SpringApplicationConfiguration;

import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import java.util.List;

import static org.hamcrest.MatcherAssert.assertThat;

import static org.hamcrest.core.Is.is;

/\*\*

\* filename : FixturespropertyTest.java

\* author : saltfactory<saltfactory@gmail.com>

\* created : 11/23/15.

\*/

@RunWith(SpringJUnit4ClassRunner.class)

@SpringApplicationConfiguration(classes = SpringBootDemoApplication.class)

public class FixturesPropertyTest {

@Autowired

private FixturesProperty fixturesProperty;

@Test

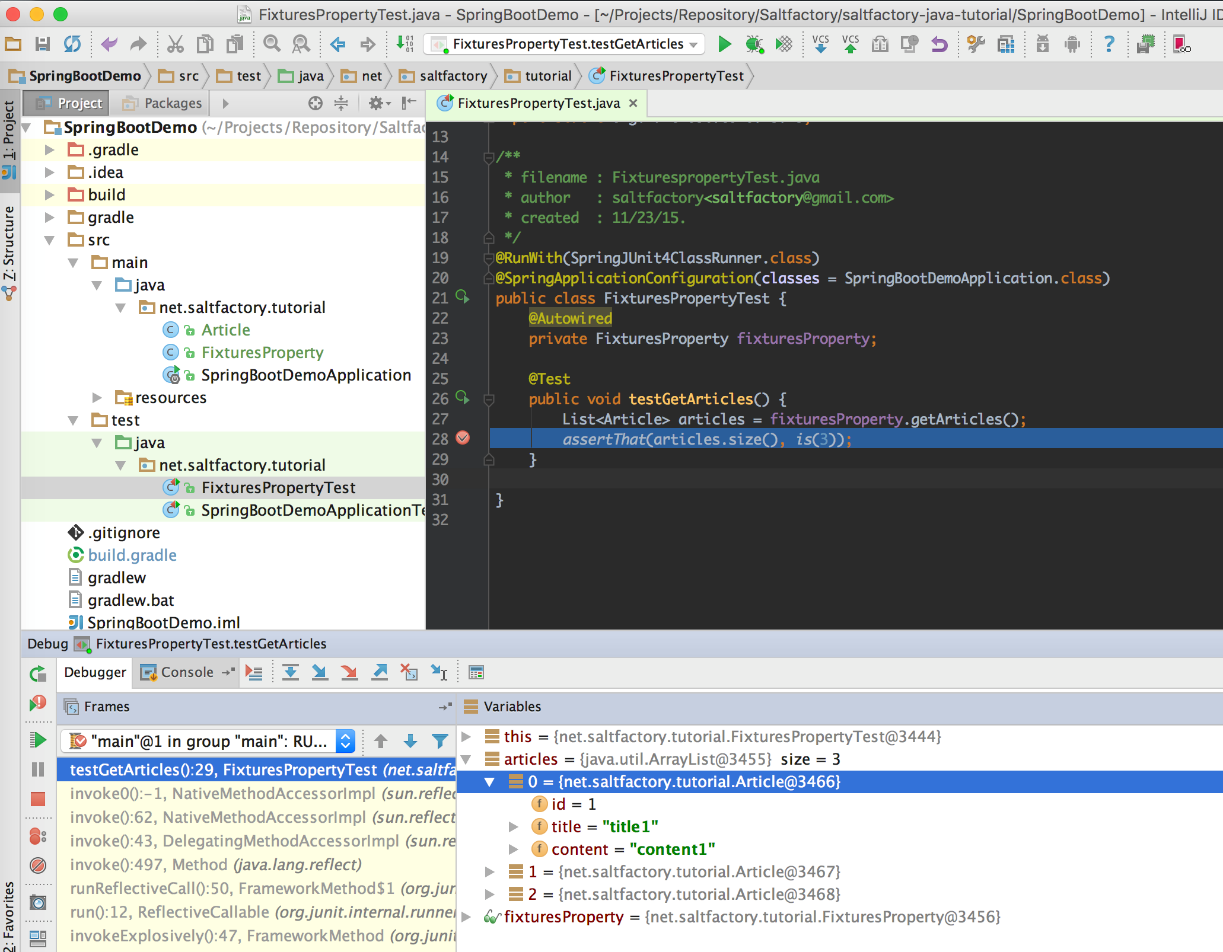
public void testGetArticles() {

List<Article> articles = fixturesProperty.getArticles();

assertThat(articles.size(), is(3));

}

}



테스트의 브레이크 포인트를 확인하면 fixtureProperty.getArticles() 에서 YAML 파일에서 로드된 데이터가 Article의 객체로 매핑되어 로드된 것을 확인할 수 있다.

## 계층 구조의 YAML 파일을 POJO로 매핑하기

YAML의 장점은 계층 구조의 데이터를 잘 표현할 수 있는 것이다. properties 파일은 계층 구조를 표현하기 위해서 좀 더 복잡한 방법을 사용해야하지만 YAML을 사용하면 들여쓰기 기준으로 계층 구조를 쉽게 표현할 수 있다. 또한 계층 구조로 된 POJO로 데이터를 로드하여 사용할 수 있다. 만약 Article 안에 Comment를 리스트로 가지고 있는 구조가 있다고 가정해보자. fixtures.yml 파일을 다음과 같이 수정한다.

fixtures:

articles:

-

id: 1

title: title1

content: content1

comments:

-

id: 10

articleId: 1

content: comment11

-

id: 11

articleId: 1

content: comment12

-

id: 2

title: title2

content: content2

comments:

- id: 20

articleId: 2

content: comment21

- id: 21

articleId: 2

content: comment22

-

id: 3

title: title3

content: content3

comments:

- id: 30

articleId: 3

content: comment31

- id: 31

articleId: 3

content: comment32

그리고 Comment가 매핑될 객체를 만든다.

package net.saltfactory.tutorial;

import java.io.Serializable;

/\*\*

\* filename : Comment.java

\* author : saltfactory<saltfactory@gmail.com>

\* created : 11/23/15

\*/

public class Comment implements Serializable {

private long id;

private String content;

private long articleId;

public long getId() {

return id;

}

public void setId(long id) {

this.id = id;

}

public String getContent() {

return content;

}

public void setContent(String content) {

this.content = content;

}

public long getArticleId() {

return articleId;

}

public void setArticleId(long articleId) {

this.articleId = articleId;

}

}

그리고 Article 안에 Comment 리스트를 추가한다.

package net.saltfactory.tutorial;

import org.springframework.stereotype.Component;

import java.io.Serializable;

import java.util.List;

/\*\*

\* filename : Article.java

\* author : saltfactory<saltfactory@gmail.com>

\* created : 11/23/15

\*/

public class Article implements Serializable {

private long id;

private String title;

private String content;

private List<Comment> comments;

public long getId() {

return id;

}

public void setId(long id) {

this.id = id;

}

public String getTitle() {

return title;

}

public void setTitle(String title) {

this.title = title;

}

public String getContent() {

return content;

}

public List<Comment> getComments() {

return comments;

}

public void setComments(List<Comment> comments) {

this.comments = comments;

}

}

단위 테스트를 다음과 같이 수정하자.

package net.saltfactory.tutorial;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.SpringApplicationConfiguration;

import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import java.util.List;

import static org.hamcrest.MatcherAssert.assertThat;

import static org.hamcrest.core.Is.is;

/\*\*

\* filename : FixturespropertyTest.java

\* author : saltfactory<saltfactory@gmail.com>

\* created : 11/23/15.

\*/

@RunWith(SpringJUnit4ClassRunner.class)

@SpringApplicationConfiguration(classes = SpringBootDemoApplication.class)

public class FixturesPropertyTest {

@Autowired

private FixturesProperty fixturesProperty;

@Test

public void testGetArticles() {

List<Article> articles = fixturesProperty.getArticles();

assertThat(articles.size(), is(3));

}

@Test

public void testGetCommentsByArticle() {

List<Article> articles = fixturesProperty.getArticles();

Article article = articles.get(0);

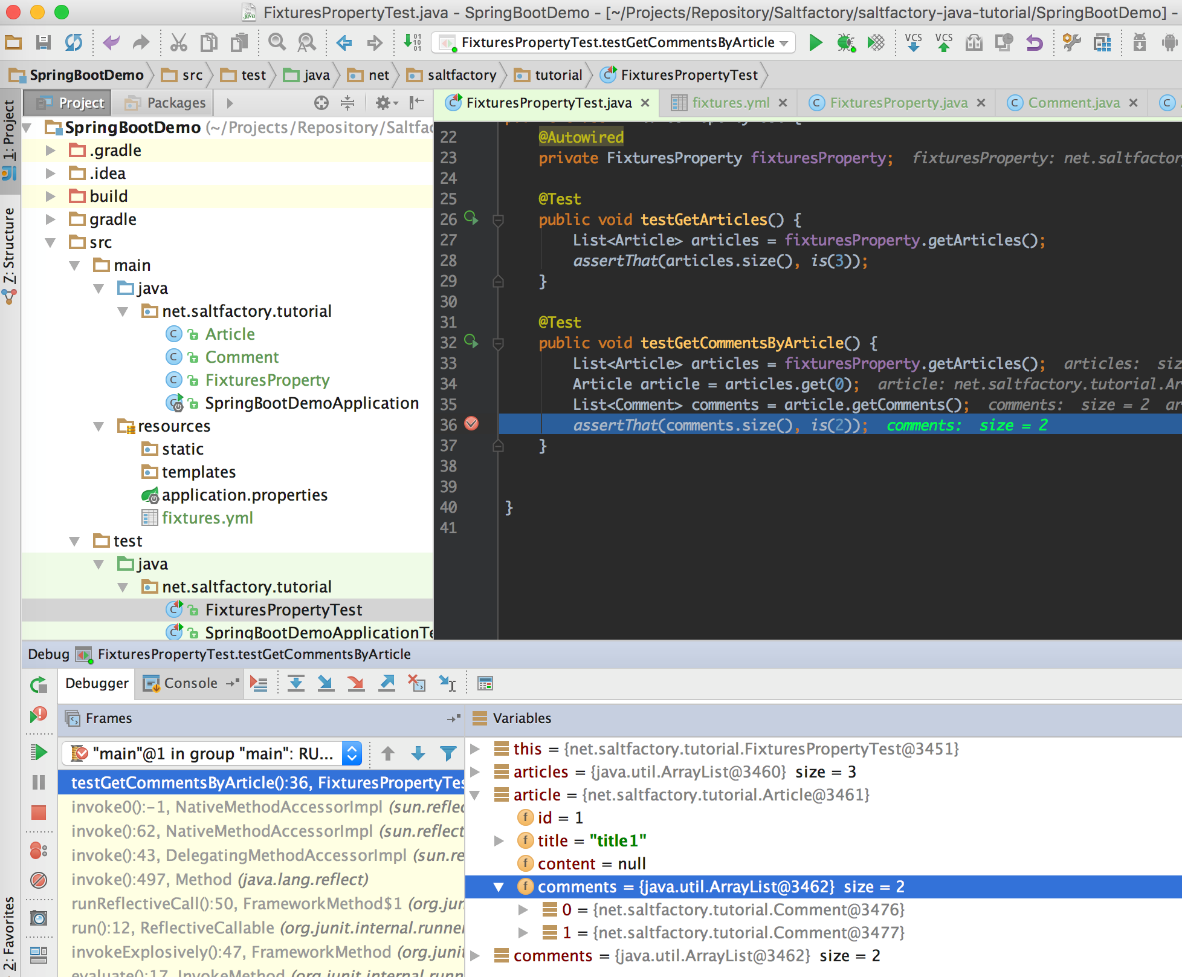
List<Comment> comments = article.getComments();

assertThat(comments.size(), is(2));

}

}

단위 테스트를 실행시켜서 Article 객체 안에 Comment가 정상적으로 로드 되었는지 확인해보자.



브레이크 포인트를 확인해보면 Article 객체 안에 Comment가 리스트로 정상적으로 로드된 것을 확인할 수 있다.

## 결론

최신 Spring은 좀더 계층구조를 표현하기 쉽고 사람이 읽기 쉬운 YAML 파일을 로드할 수 있는 기능을 포함하였다. Spring Boot에서는 Spring 어플리케이션의 설정을 **src/main/resources/application.properties**에서 정의하면 자동으로 어플리케이션에 적용이되는데 YAML 파일을 사용하여 **application.yml** 파일을 만들어도 자동으로 적용을 할 수 있다. **SnakeYAML** 라이브러리를 포함하고 있는 Spring Boot에서는 YAML 파일을 읽어들어 Map이나 POJO에 바로 매핑하여 데이터를 로드할 수 있다. 더구나 **@ConfigurationProperties**를 사용하면 Spring 어플리케이션에서 YAML 파일을 Configuration Property 파일로 인식하여 특별한 자바 코드 없이도 Spring annotation 만으로도 외부의 YAML 파일을 로드할 수 있다. 만약 데이터베이스가 없는 데모 어플리케이션을 만들거나 테스트를 위한 간단한 데이터를 외부 파일에서 조작하기 위해서 YAML 파일을 사용하여 데이터를 정의하여 사용하면 매우 간단하게 처리할 수 있다.

## 소스코드

* https://github.com/saltfactory/saltfactory-java-tutorial/tree/spring-boot-yaml/SpringBootDemo

## 참조

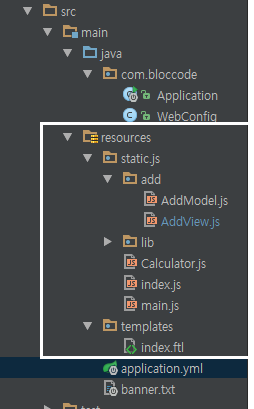
1. https://bitbucket.org/asomov/snakeyaml
2. http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html
3. https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html

### [SpringBoot에서 template, static resource 실시간 반영하기 (spring-boot-devtools)](http://jojoldu.tistory.com/48)

Intellij + SpringBoot + Template Engine 조합으로 웹 개발을 하다보면 정말 노가다라고 느껴지는게 **정적 리소스 갱신을 위한 서버 restart**이다.

무슨 말이냐하면 SpringBoot로 프로젝트를 생성하면 이전처럼 webapp 폴더에 html/css/js (정적 리소스)들을 담당하는게 아니라 resources 폴더 아래에서 관리하다보니 정적 리소스들 갱신을 하기 위해선 tomcat을 restart 해야만 한다.

아마 루비와 같은 스크립트 언어 개발하시는 분들은 Java & Spring 으로 이런 일이 벌어지는거에 너무 놀래실것 같긴 하다..



(Spring Boot의 일반적인 리소스 Path)

여튼 Javascript를 중점적으로 개발/수정할 때에는 이게 생각보다 너무 귀찮아서 새로고침만 하면 바로 반영되게 하고 싶었다.

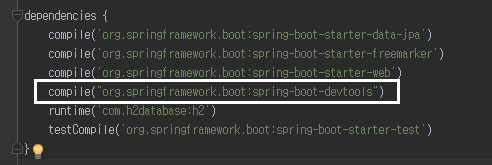
그래서 몇가지 방법들을 찾아보았는데, SpringBoot의 정적 리소스 Path를 예전처럼 webapp으로 지정해서 하는 방식을 기록해주신분이 계셨지만, 이건 패키징할때나 다른 Path을 잡을때나 귀찮은 일이 또 생기게 되서 기존 Path를 유지한채로 Hot reload가 되도록 하는 방법을 찾다가 SpringBoot Devtools에 대해 알게 되었다.

(SpringBoot 1.3 버전부터 지원하기 때문에 1.3 미만의 버전에서는 Spring Loaded를 검색해보면 될 것 같다.)

바로 설정을 시작해보자. 참고로 IDE의 지원이 함께 필요한데 여기서는 IntelliJ를 기준으로 진행한다.

**1. Gradle에 의존성 추가**

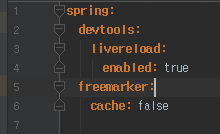
Maven Repository에서 spring-boot-devtools로 검색하여 나온 내용을 본인의 빌드관리 툴 (Mave or Gradle)에 추가하여 받도록 하자



**2. application.properties 혹은 application.yml에 reload 설정 추가**

SpringBoot 프로젝트의 경우 application.properties 혹은 application.yml 으로 프로젝트 설정을 관리한다.

본인의 설정 양식에 맞춰 아래 설정을 추가하자 (난 application.yml로 관리한다)



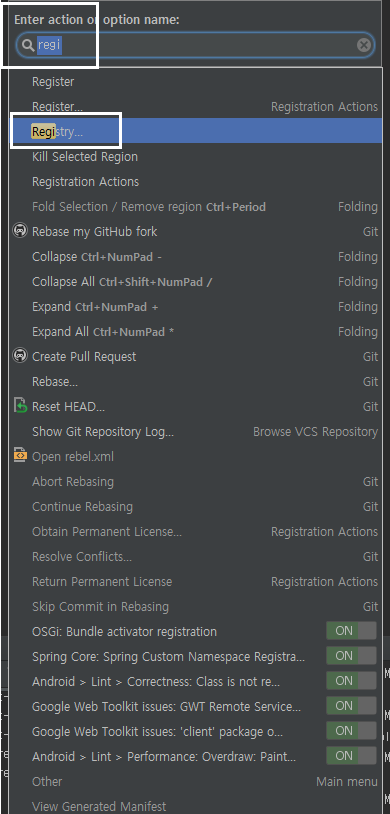
설정 내용은 어려운게 없다. livereload를 enable 하며, freemarker cache 옵션을 false로 변경하여 compile을 다시하지 않아도 반영되도록 하였다. (물론 실서버 배포시에는 이 옵션을 개발서버와 실서버용으로 분리시켜야 한다. 템플릿을 캐시하지 않으면 서버 터지는걸 볼 수 있다)

**3. IntelliJ 옵션 수정**

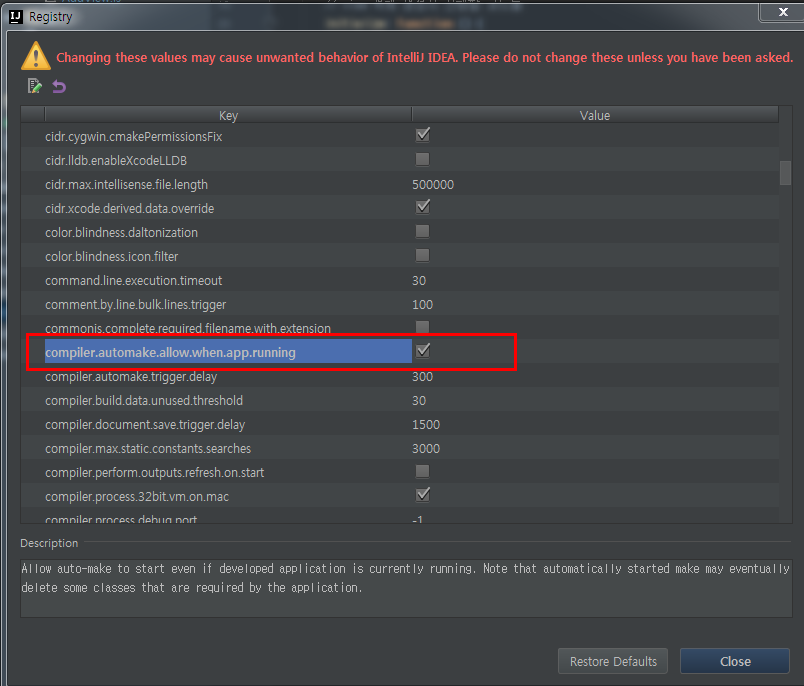
보통 블로그에서는 위 2개만 설정하면 된다고 나오는데, IntelliJ에서는 저것만 가지고는 절대 되지 않았다. 그래서 추가적으로 찾아보니 IntelliJ에서 몇가지 설정을 추가해야만 한다.

1) registry 설정

ctrl + shift + a 를 누르면 IntelliJ에서 검색창이 나온다.

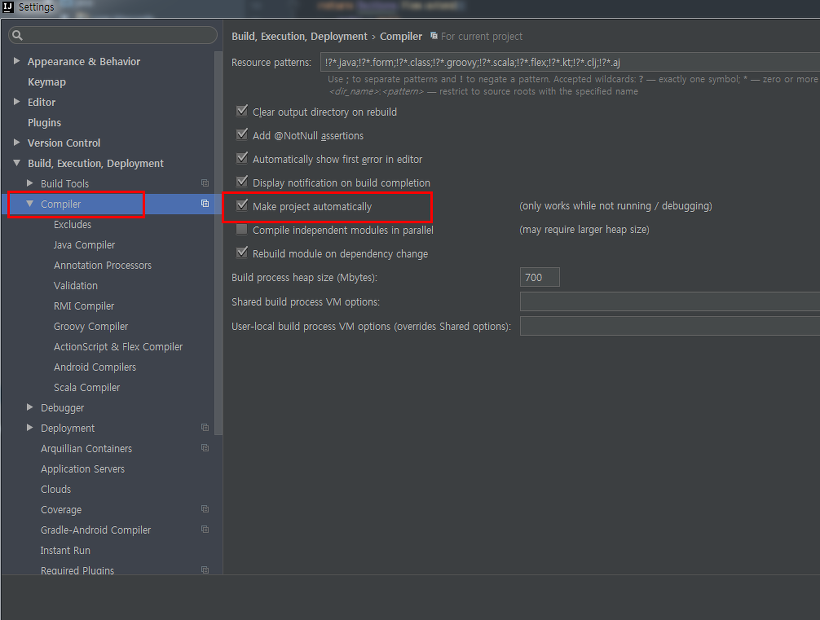


여기에서 Registry... 로 들어가면 여러 옵션이 나오는데 그중 아래와 같이 체크박스에 체크를 하자.



2) compiler 설정

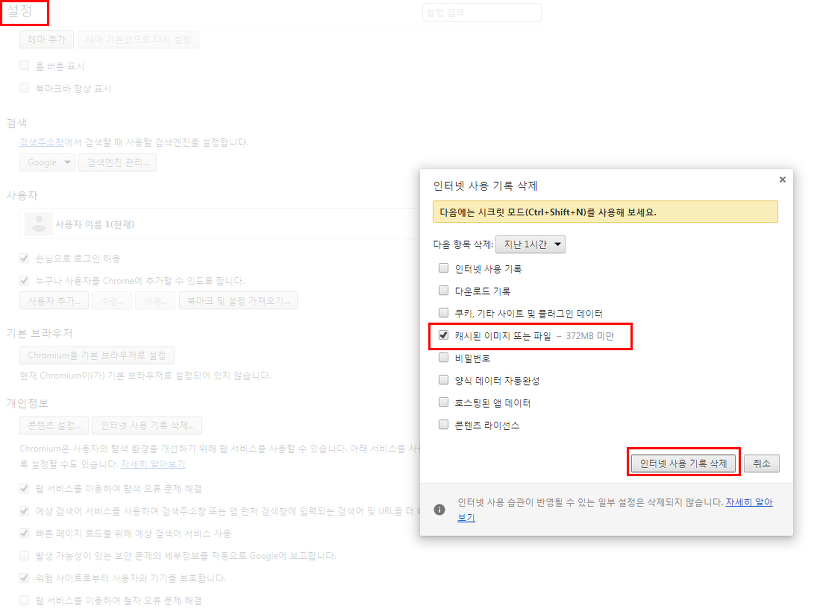
File -> Settings 옵션에 들어가서 아래와 같이 Make Project automatically 옵션을 체크하자.



프로젝트를 다시 시작해보자. Tomcat을 재시작하거나 Applicaton.java를 재시작하거나 설정을 적용할 수 있도록 서버재시작을 하면 된다.

**4. 브라우저 캐시 삭제**

3번까지 진행해도 아마 안될수 있다. 그건 별다른 이유가 아니라 브라우저에서 동일 url로 호출하는 정적 리소스들은 캐시하고 있기 때문인데, 수정한 리소스들을 반영하기 위해 브라우저 캐시를 **한번만** 삭제하자 (이후로는 이럴 필요 없다)



(크롬 기준의 브라우저 캐시 삭제)

여기까지 하게 되면 js, css, freemarker를 수정할때마다 서버 재시작을 할 필요가 없게 된다!!

불필요한 시간 낭비를 줄일수 있게되서!

귀찮은 행동을 안해도 되게 되어서!

너무 좋다!!!

<http://jinhokwon.tistory.com/category/SpringBoot> 깔끔하게 정리된 spring boot