<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/>

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 레퍼런스 문서](http://docs.spring.io/spring-boot/docs/1.2.x/reference/htmlsingle/#boot-features-embedded-container-servlets-and-filters)에 나와 있다.

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

[자바(Java)](http://mudchobo.tistory.com/category/%EC%9E%90%EB%B0%94%28Java%29) 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**

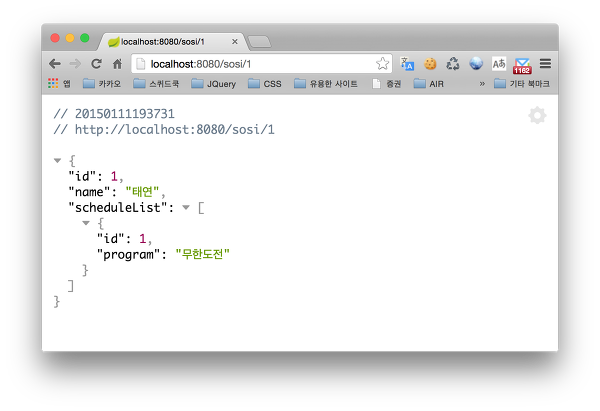
소녀시대 정보 가져오기  
http://localhost:8080/sosi/1



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