Configure another port number(eg:8181) in /src/main/resources/application.properties

server.port=8181

<http://javasampleapproach.com/spring-framework/spring-batch/spring-batch-job-parallel-steps>

<http://www.devglan.com/spring-security/spring-boot-security-oauth2-example>

<http://www.devglan.com/spring-security/securing-spring-boot-actuator-endpoints-with-spring-security>

Below are the important view resolvers provided by spring framework:

1. **AbstractCachingViewResolver :** Abstract view resolver that caches views. Often views need preparation before they can be used; extending this view resolver provides caching.
2. **XmlViewResolver :**Implementation of ViewResolver that accepts a configuration file written in XML with the same DTD as Spring’s XML bean factories. The default configuration file is /WEB-INF/views.xml.
3. **ResourceBundleViewResolver :**Implementation of ViewResolver that uses bean definitions in a ResourceBundle, specified by the bundle base name. Typically you define the bundle in a properties file, located in the classpath. The default file name is views.properties.
4. **UrlBasedViewResolver :**Simple implementation of the ViewResolver interface that effects the direct resolution of logical view names to URLs, without an explicit mapping definition. This is appropriate if your logical names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.
5. **InternalResourceViewResolver :**Convenient subclass of UrlBasedViewResolver that supports InternalResourceView (in effect, Servlets and JSPs) and subclasses such as JstlView and TilesView. You can specify the view class for all views generated by this resolver by using setViewClass(..).
6. **VelocityViewResolver/FreeMarkerViewResolver :**Convenient subclass of UrlBasedViewResolver that supports VelocityView (in effect, Velocity templates) or FreeMarkerView ,respectively, and custom subclasses of them.
7. **ContentNegotiatingViewResolver :**Implementation of the ViewResolver interface that resolves a view based on the request file name or Accept header.

**Spring Boot**

@SpringBootApplication enables many defaults. It also enables @EnableWebMvc that activates web endpoints.

Spring Boot is the best Java framework for microservices. We recommend you become an expert at Spring Boot!

##### Spring Boot

The problem with Spring and Spring MVC is the amount of configuration that is needed.

<bean

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix">

<value>/WEB-INF/views/</value>

</property>

<property name="suffix">

<value>.jsp</value>

</property>

</bean>

<mvc:resources mapping="/webjars/\*\*" location="/webjars/"/>

Spring Boot solves this problem through a combination of Auto Configuration and Starter Projects. Spring Boot also provide a few nonfunctional features to make building production ready applications faster.

### Q : Can you explain more about Starters with an example?

Let’s consider an example starter - Spring Boot Starter Web.

If you want to develop a web application or an application to expose restful services, Spring Boot Start Web is the starter to pick. Lets create a quick project with Spring Boot Starter Web using Spring Initializr.

Dependency for Spring Boot Starter Web

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

### Q : What are the other Starter Project Options that Spring Boot provides?

Spring Boot also provides other starter projects including the typical dependencies to develop specific type of applications

* spring-boot-starter-web-services - SOAP Web Services
* spring-boot-starter-web - Web & RESTful applications
* spring-boot-starter-test - Unit testing and Integration Testing
* spring-boot-starter-jdbc - Traditional JDBC
* spring-boot-starter-hateoas - Add HATEOAS features to your services
* spring-boot-starter-security - Authentication and Authorization using Spring Security
* spring-boot-starter-data-jpa - Spring Data JPA with Hibernate
* spring-boot-starter-data-rest - Expose Simple REST Services using Spring Data REST

### Q : How does Spring enable creating production ready applications in quick time?

Spring Boot aims to enable production ready applications in quick time. Spring Boot provides a few non functional features out of the box like caching, logging, monitoring and embedded servers.

* spring-boot-starter-actuator - To use advanced features like monitoring & tracing to your application out of the box
* spring-boot-starter-undertow, spring-boot-starter-jetty, spring-boot-starter-tomcat - To pick your specific choice of Embedded Servlet Container
* spring-boot-starter-logging - For Logging using logback
* spring-boot-starter-cache - Enabling Spring Framework’s caching support

### What is the minimum baseline Java Version for Spring Boot 2 and Spring 5?

Spring Boot 2.0 requires Java 8 or later. Java 6 and 7 are no longer supported.

Recommended Reading

* https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.0.0-M1-Release-Notes

### Q : What is the easiest approach to create a Spring Boot Project?

Spring Initializr <http://start.spring.io/> is great tool to bootstrap your Spring Boot projects.

=======Spring Batch to Run Multiple Instances:=====

|  |
| --- |
| @SpringBootApplication  @EnableBatchProcessing  @ImportResource("classpath:batchjob.xml")  **public** **class** SpringBatchParallelStepApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(SpringBatchParallelStepApplication.**class**, args);  }  } |

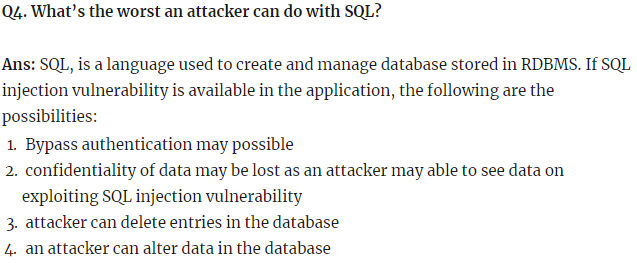
batchjob.xml

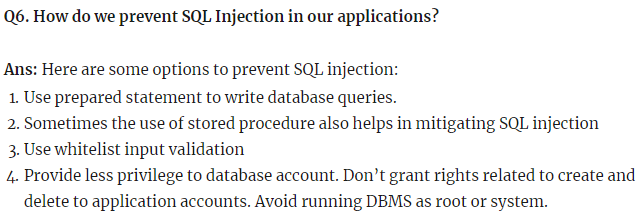
|  |
| --- |
| <job id=*"job"*>  <split id=*"split\_1"* task-executor=*"taskExecutor"* next=*"step\_4"*>  <flow>  <step id=*"step\_1"*>  <tasklet ref=*"taskletStep\_1"* />  </step>  </flow>  <flow>  <step id=*"step\_2"*>  <tasklet ref=*"taskletStep\_2"* />  </step>  </flow>    <flow>  <step id=*"step\_3"*>  <tasklet ref=*"taskletStep\_3"* />  </step>  </flow>  </split>  <step id=*"step\_4"*>  <tasklet ref=*"taskletStep\_3"* />  </step>  </job>  <beans:bean id=*"taskletStep\_1"*  class=*"com.javasampleapproach.batch.parallelstep.step.SimpleStep"* />  <beans:bean id=*"taskletStep\_2"*  class=*"com.javasampleapproach.batch.parallelstep.step.SimpleStep"* />  <beans:bean id=*"taskletStep\_3"*  class=*"com.javasampleapproach.batch.parallelstep.step.SimpleStep"* />  <beans:bean id=*"taskletStep\_4"*  class=*"com.javasampleapproach.batch.parallelstep.step.SimpleStep"* />  <beans:bean id=*"taskExecutor"*  class=*"org.springframework.core.task.SimpleAsyncTaskExecutor"* />  </beans:beans> |

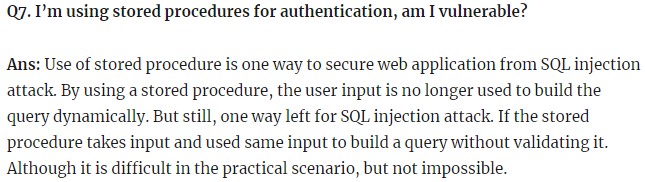
|  |
| --- |
| JobLauncherController.java |
| @RestController  **public** **class** JobLauncherController {  @Autowired  JobLauncher jobLauncher;  @Autowired  Job job;  @RequestMapping("/launchjob")  **public** String handle() **throws** Exception {  Logger logger = LoggerFactory.*getLogger*(**this**.getClass());  **try** {  JobParameters jobParameters = **new** JobParametersBuilder().addLong("time", System.*currentTimeMillis*())  .toJobParameters();  jobLauncher.run(job, jobParameters);  } **catch** (Exception e) {  logger.info(e.getMessage());  }  **return** "Done";  }  } |

|  |
| --- |
| SimpleStep.java |
| **public** **class** SimpleStep **implements** Tasklet {  @Override  **public** RepeatStatus execute(StepContribution contribution,  ChunkContext chunkContext) **throws** Exception {  workload();  System.***out***.println("Done SimpleStep");  **return** RepeatStatus.***FINISHED***;  }    **private** **void** workload() **throws** Exception {  Thread.*sleep*(5000);  }  } |

SQL Injections:







## Difference between [RequestParam](https://www.dineshonjava.com/requestparam-annotation-in-spring-mvc-with-example/) & PathVariable

|  |
| --- |
| [RequestParam](https://www.dineshonjava.com/requestparam-annotation-in-spring-mvc-with-example/) |
| **Look at the following request URL:**  ***http://localhost:8080/tutorials/bookmark/?site=dineshonjava&id=200*** |
| @RequestMapping(value = "/tutorials/bookmark")  public String bookmark(  @RequestParam(value="site", required=true) String site,  @RequestParam(value="id", required=false) String id){  ...  } |

|  |
| --- |
| PathVariable |
| Look at the following request URL:  *http://localhost:8080/tutorials/bookmark/100?site=dineshonjava&id=200* |
| @RequestMapping(value = "/tutorials/bookmark/{siteId}")  public String bookmark(  @PathVariable(value="siteId") String siteId  @RequestParam(value="site", required=true) String site,  @RequestParam(value="id", required=false) String id){  ...  } |

Spring Boot Actuator is available since April 2014,

<https://www.devglan.com/spring-boot/spring-boot-actuator-rest-endpoints-example>

This guide is split into 3 main sections:

* [**What is an Actuator?**](http://www.baeldung.com/spring-boot-actuators#understanding-actuator)

## ****What is an Actuator?****

## <http://www.baeldung.com/spring-boot-actuators>

In essence, Actuator brings production-ready features to our application.

**Monitoring our app, gathering metrics, understanding traffic or the state of our database becomes trivial with this dependency.**

The main benefit of this library is that we can get production grade tools without having to implement these features ourselves.

Actuator is mainly used to **expose operational information about the running application** – health, metrics, info, dump, env, etc. It uses HTTP endpoints or JMX beans to enable us to interact with it.

Once this dependency is on the classpath several endpoints are available for us out of the box. As with most Spring modules, we can easily configure or extend it in many ways.

* [**Spring Boot 1.x Actuator**](http://www.baeldung.com/spring-boot-actuators#boot-1x-actuator)
* [**Spring Boot 2.x Actuator**](http://www.baeldung.com/spring-boot-actuators#boot-2x-actuator)

## What is Spring Boot Actuator

Spring boot actuator is a sub-project of spring boot. Once spring boot actuator is integrated in your spring boot app, there are around 16 default REST endpoints you get as mentioned below to monitor and manage your application. By using these default endpoints you can monitor your application health, check the beans configured, monitor your application properties and many more. Also you can restrict the endpoints access to certain role by integrating with spring security. You can customize default endpoints and even create your own custom endpoints to monitor your application.

## Different Actuator Endpoints

Once above maven dependency is included in the pom file, you can get following 16 implicit endpoints. Some of them are sensitive and some are not. In the following list the only endpoints which are not sensitive are **health** and **info**.

1. **actuator** : It provides a hypermedia-based discovery page for the other endpoints. It is required to have Spring HATEOS on the classpath to enable it. By default it is sensitive and hence requires username/password for access or may be disabled if web security is not enabled.

2. **auditevents**: It exposes audit events information.

3. **autoconfig**: It displays an auto-configuration report showing all auto-configuration candidates.

4. **beans**: It displays complete beans configured in the app.

5. **configprops**: It displays a collated list of all @ConfigurationProperties.

6. **dump**: It performs a thread dump.

7. **env**: It exposes properties from Spring ConfigurableEnvironment.

8. **flyway**: It shows any Flyway database migrations that have been applied.

9. **health**: It shows application health information (when the application is secure, a simple ‘status’ when accessed over an unauthenticated connection or full message details when authenticated).

10. **info**: It displays arbitrary application info.

11. **loggers**: It shows and modifies the configuration of loggers in the application.

12. **liquibase**: It shows any Liquibase database migrations that have been applied.

13. **metrics**: It shows metrics information for the current application.

14. **mappings**: It displays a collated list of all @RequestMapping paths.

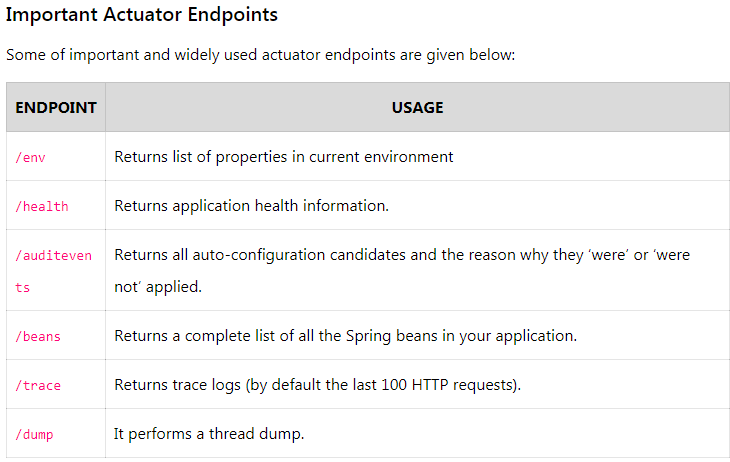
15. **shutdown**: It allows the application to be gracefully shutdown (not enabled by default).

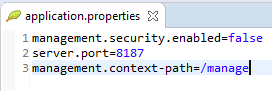
16. **trace**: It displays trace information (by default the last 100 HTTP requests).

If you are also using spring mvc then other 4 additional endpoints such as **docs**, **heapdump**, **jolokia** and **logfile** can be used

==============Examples of Actuator =======================

<https://howtodoinjava.com/spring/spring-boot/actuator-endpoints-example/#actuator-example>





Tested URL:

Normal: <http://localhost:8187/example> , <http://localhost:8187/manage/health>

Actuator\_Url: <http://localhost:8187/manage/dump>

===============MultiThreading===================

**Busy**-waiting or **spinning** is a technique in which a process repeatedly checks to see if a condition is true instead of calling wait or sleep method and releasing CPU. 1.It is mainly useful in multicore processor where condition is going to be true quite quickly i.e. in millisecond or micro second.

**When to use Busy Spin :**  
Busy spin is used in multicore processors where it is possible that a paused thread can resume on different core and this thread will use old cache and there is no need to rebuild cache as CPU is not released in busy spin.  
It is used in trading applications where waiting time between two message is very less and frequent.