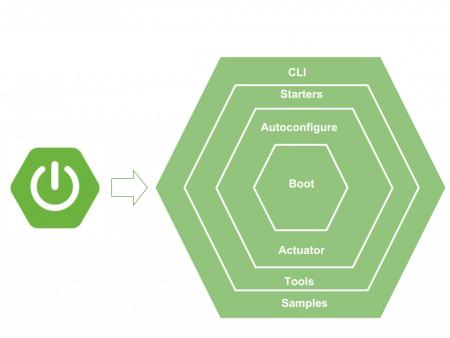
**Spring Boot Tutorial**

**Spring Boot** is a Spring framework module which provides RAD (**Rapid Application Development**) feature to the Spring framework.



**Spring boot modules**

## **What is starter template?**

Spring Boot starters are templates that contain a **collection of all the relevant transitive dependencies** that are needed to start a particular functionality. For example, If you want to create a Spring WebMVC application then in a traditional setup, you would have included all required dependencies yourself. It leaves the chances of **version conflict** which ultimately result in more **runtime exceptions**.

With String boot, to create MVC application all you need to import is spring-boot-starter-web dependency.

|  |
| --- |
| **pom.xml** |
| <!-- Parent pom is mandatory to control versions of child dependencies -->  <parent>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-starter-parent</artifactId>      <version>2.0.4.RELEASE</version>      <relativePath />  </parent>    <!-- Spring web brings all required dependencies to build web application. -->  <dependency>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-starter-web</artifactId>  </dependency> |

Above spring-boot-starter-web dependency, internally imports all given dependencies and add to your project.

Notice how some dependencies are direct, and some dependencies further refer to other starter templates which transitively downloads more dependencies.

Also, notice that **you do not need to provide version information into child dependencies**. All versions are resolved in relation to version of parent starter (in our example it’s 2.0.4.RELEASE).

|  |
| --- |
| **Dependencies brought in by webmvc starter template:** |
| <dependencies>      <dependency>          <groupId>org.springframework.boot</groupId>          <artifactId>spring-boot-starter</artifactId>      </dependency>      <dependency>          <groupId>org.springframework.boot</groupId>          <artifactId>spring-boot-starter-json</artifactId>      </dependency>      <dependency>          <groupId>org.springframework.boot</groupId>          <artifactId>spring-boot-starter-tomcat</artifactId>      </dependency>      <dependency>          <groupId>org.hibernate.validator</groupId>          <artifactId>hibernate-validator</artifactId>      </dependency>      <dependency>          <groupId>org.springframework</groupId>          <artifactId>spring-web</artifactId>      </dependency>      <dependency>          <groupId>org.springframework</groupId>          <artifactId>spring-webmvc</artifactId>      </dependency>  </dependencies> **2. Spring boot autoconfiguration:** Autoconfiguration is enabled with @EnableAutoConfiguration annotation.  Spring boot auto configuration scans the classpath, finds the libraries in the classpath and then attempt to guess the best configuration for them, and finally configure all such beans.  Auto-configuration is always applied after user-defined beans have been registered.  Spring boot auto-configuration logic is implemented in **spring-boot-autoconfigure.jar** **3. Embedded server:** Spring boot applications always include **tomcat** as **embedded server** dependency.  You can exclude tomcat and include any other embedded server if you want. Or you can make exclude server environment altogether. It’s all configuration based.  For example, below configuration **exclude tomcat** and **include jetty** as embedded server.   |  | | --- | | **pom.xml:** | | <dependency>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-starter-web</artifactId>      <exclusions>          <exclusion>              <groupId>org.springframework.boot</groupId>              <artifactId>spring-boot-starter-tomcat</artifactId>          </exclusion>      </exclusions>  </dependency>    <dependency>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-starter-jetty</artifactId>  </dependency> **4. Bootstrap the application:** To **run the application**, we need to use @SpringBootApplication annotation. Behind the scenes, that’s equivalent to @Configuration, @EnableAutoConfiguration, and @ComponentScan together.  It enables the scanning of config classes, files and load them into **spring context**. In below example, execution start with main() method. It start loading all the config files, configure them and bootstarp  the application based on [application properties](https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html) in **application.properties** file in /resources folder.   |  | | --- | | **MyApplication.java:** | | @SpringBootApplication  public class MyApplication  {      public static void main(String[] args)      {          SpringApplication.run(Application.class, args);      }  } | | **application.properties:** | | ### Server port #########  server.port=8080    ### Context root ########  server.contextPath=/home |   To execute the application, you can run the **main() method** from IDE such **eclipse**, or you can build the jar file and execute from command prompt.   |  | | --- | | **Console** | | $ java -jar spring-boot-demo.jar **5. Advantages of Spring boot**  * Spring boot helps in **resolving dependency conflict**. It identifies required dependencies   and import them for you.   * It has information of **compitable version** for all dependencies. It minimizes the runtime   **Classloader** issues.   * It’s “**opinionated defaults configuration**” approach helps you in configuring most   important pieces behind the scene. Override them only when you need.  Otherwise everything just works, perfectly.  It helps in avoiding **boilerplate code**, annotations and XML configurations.   * It provides embedded HTTP server Tomcat so that you can develop and test quickly. * It has excellent integration with IDEs like eclipse and **intelliJ idea**. | |   **Create launch application:**   |  | | --- | | package com.howtodoinjava.demo;    import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.context.ApplicationContext;    @SpringBootApplication  public class App  {      Public  static  void  main(String[] args)      {          ApplicationContext ctx = SpringApplication.run(App.class, args);      }  } |  **What this launch class does?** Above class is called spring boot application launch class. It used to Bootstrap and launch a Spring application from a Java main() method. It typically does following things:-   * Create an instance of Spring’s ApplicationContext. * Enable the functionality to accept command-line arguments and expose them as Spring properties. Eg: to set profile active: **spring.profile.active=dev** * Load all the Spring beans as per the configuration. You can do other operations as well as per project need arises.  **@SpringBootApplication Annotation:** This annotation is a shortcut of applying 3 annotations in one statement- 1. @SpringBootConfiguration:  [@SpringBootConfiguration](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/SpringBootConfiguration.html) is new annotation in Spring boot 2. Previously, we have been using [@Configuration](https://docs.spring.io/spring-framework/docs/5.0.4.RELEASE/javadoc-api/org/springframework/context/annotation/Configuration.html) annotation. You can use @Configuration in place of this. Both are same thing.  * It simply means that annotated class is a configuration class and shall be scanned for further configurations and bean definitions.   2. @EnableAutoConfiguration : You can always manually exclude any configuration that you never want to apply using two methods –  i) Use excludeName() ii) Using the spring.autoconfigure.exclude property in properties file. e.g.   |  | | --- | | @EnableAutoConfiguration(excludeName = {"multipartResolver","mbeanServer"}) |   Auto-configuration is always applied after user-defined beans have been registered.  3.@ComponentScan :  This annotation provides support parallel with Spring XML’s context:component-scan element.  Either basePackageClasses() or basePackages() may be specified to define specific packages to scan.  If specific packages are not defined, scanning will occur from the package of the class that declares this annotation.  **Note : For getting insight of registered beans, I have added modified the launch application as below.** |
| @SpringBootApplication  public class App  {      public static void main(String[] args)      {          ApplicationContext ctx = SpringApplication.run(App.class, args);            String[] beanNames = ctx.getBeanDefinitionNames();            Arrays.sort(beanNames);            for (String beanName : beanNames) {              System.out.println(beanName);          }      }  }  **O/P:**  Tomcat started on port(s): 8080 (http) with context path ''  2018-04-02 13:09:45.076  INFO 11452 --- [           main] com.howtodoinjava.demo.App               : Started App in 4.609 seconds (JVM running for 5.263)  app  basicErrorController  beanNameHandlerMapping  beanNameViewResolver  characterEncodingFilter  conventionErrorViewResolver  defaultServletHandlerMapping  defaultValidator  defaultViewResolver  dispatcherServlet  dispatcherServletRegistration  error  errorAttributes  errorPageCustomizer  errorPageRegistrarBeanPostProcessor  faviconHandlerMapping  faviconRequestHandler  handlerExceptionResolver  hiddenHttpMethodFilter  httpPutFormContentFilter  httpRequestHandlerAdapter  jacksonCodecCustomizer  jacksonObjectMapper  jacksonObjectMapperBuilder  jsonComponentModule  localeCharsetMappingsCustomizer  mappingJackson2HttpMessageConverter  mbeanExporter  mbeanServer  messageConverters  methodValidationPostProcessor  multipartConfigElement  multipartResolver  mvcContentNegotiationManager  mvcConversionService  mvcHandlerMappingIntrospector  mvcPathMatcher  mvcResourceUrlProvider  mvcUriComponentsContributor  mvcUrlPathHelper  mvcValidator  mvcViewResolver  objectNamingStrategy  org.springframework.boot.autoconfigure.AutoConfigurationPackages  org.springframework.boot.autoconfigure.condition.BeanTypeRegistry  org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration  org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration  org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration  You see how many beans got registered automatically. That’s beauty of spring boot. |

## What is spring-boot-starter-parent dependency?

The spring-boot-starter-parent dependency is the parent POM providing dependency and plugin management for Spring Boot-based applications. It contains the default versions of Java to use, the default versions of dependencies that Spring Boot uses, and the default configuration of the Maven plugins.

# Spring Boot Devtools Tutorial:

If you have worked on latest UI development frameworks e.g. Node, [angular](https://howtodoinjava.com/angularjs/angularjs-tutorial-helloworld-example/), gulp etc. then you must have appreciated the auto-reload of UI in browser whenever there is change in some code. It’s pretty useful and saves a lot of time.

Well, same feature can be utilized in spring boot applications using spring-boot-devtools dependency provided features. Let’s learn about enabling these features and using them.

## Enabling Dev Tools Module:

add the spring-boot-devtools dependency in your build file.

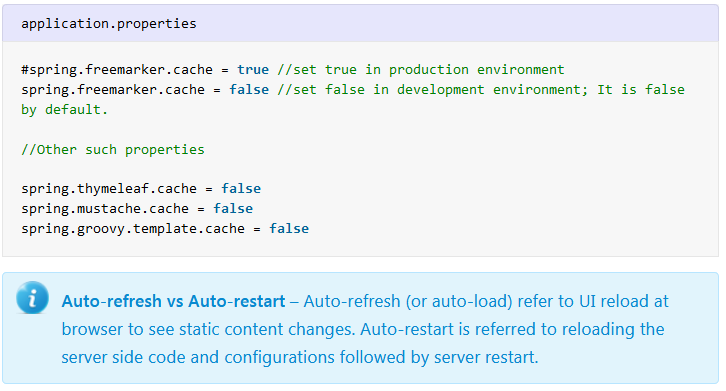


## Static Resource Caching:

To improve the performance, devtools cache the static content/template files to serve them faster to browser/client. This is very good feature in production where every milli-second [performance improvement](https://howtodoinjava.com/best-practices/improving-web-application-performance/) matters. But in development environment, it can be a problem and cause stale cache problem and you may not see your changes immediately in browser. Devtools module provide this capability by setting few properties.

By default, this feature is disabled. You can enable it to use in production environment by setting a property.

There are many such UI template libraries that support this feature. e.g. thymeleaf, freemarker, groovy, mustache etc.



## Automatic UI refresh

The spring-boot-devtools module includes an embedded LiveReload server that can be used to trigger a browser refresh when a resource is changed. Precondition is that your browser should have supported extension for it. You can find such browser extensions in this [link](https://livereload.com/extensions/).

By default, live reload is enabled. If you wish to disable this feature for some reason, then set spring.devtools.livereload.enabled property to false.

|  |
| --- |
| **application.properties:** |
| spring.devtools.livereload.enabled  = false #Set false to disable live reload |
| Enable/disable logging of auto-configuration changes By default, each time your application restarts, a report showing the condition evaluation delta is logged. The report shows the changes to your application’s auto-configuration as you make changes such as adding or removing beans and setting configuration properties.  To disable the logging of the report, set the following property:   |  | | --- | | spring.devtools.restart.log-condition-evaluation-delta = false |  Disabling Restart To disable the restart of server on non-static code changes, use the property spring.devtools.restart.enabled.   |  | | --- | | spring.devtools.restart.enabled = false |  Using a Trigger File Automatic restarts may be desirable on every file change and sometimes can slower down development time due to frequent restarts. To solve this problem, you can use a trigger file. Spring boot will keep monitoring that file and once it will detect any modification in that file, it will restart the server and reload all your previous changes.  Use spring.devtools.restart.trigger-file property to mention the trigger file for your application. It can be any external or internal file.   |  | | --- | | spring.devtools.restart.trigger-file = c:/workspace/restart-trigger.txt |  Spring Boot war Packaging Example:In Spring boot applications, default packaging is jar which is deployed in embedded servers. If you want to generate a war file for deployment in separate application server instances such as Jboss, Weblogic or tomcat, then follow below instructions.Step 1) Declare packaging type ‘war’Step 2) Set embedded server dependency scope to ‘provided’Scope ‘provided’ indicates you expect the JDK or a container to provide the dependency at runtime. This scope is only available on the compilation and test classpath, and is not transitive.Spring Boot 2 REST API Example:In this ****Spring Boot 2 REST API**** tutorial, we will create two simple GET and POST APIs step by step and test them.1. Maven dependencies:The important dependencies are spring-boot-starter-parent ([read more](https://howtodoinjava.com/spring-boot2/spring-boot-starter-parent-dependency/)) and spring-boot-starter-web ([read more](https://howtodoinjava.com/spring-boot2/spring-boot-starter-templates/)). Starter web dependency transitively includes more dependencies to build a web application such as spring-webmvc, spring-web, hibernate-validator, tomcat-embed-core, tomcat-embed-el, tomcat-embed-websocket, jackson-databind, jackson-datatype-jdk8, jackson-datatype-jsr310 and jackson-module-parameter-names.2. Spring Boot 2 REST API Controller  * In Spring, a controller class, which is capable of serving REST API requests, is called rest controller. It should be annotated with **@RestController** annotation. * The resource uris are specified in **@RequestMapping** annotations. It can be applied at class level and method level both. Complete URI for an API is resolved after adding class level path and method level path. * We should always write **produces** and **consumes** attributes to specify the mediatype attributes for the API. Never reply on assumptions.   In given controller, we have two API methods. Feel free to add more methods as needed.   1. **HTTP GET /employees** – Returns list of the employees. 2. **HTTP POST /employees** – Add an employee in the employees collection.  3. @SpringBootApplication Our REST APIs skeleton is ready. Now we need to configure Spring to detect our rest controller (using auto scanning) and deploy apis in embedded tomcat server. Thankfully, Spring boot makes all these things very easy by using the concept of [auto configuration](https://howtodoinjava.com/spring-boot2/springbootapplication-auto-configuration/).  **Auto-configuration** attempts to guess and configure beans we you are likely to need. Auto-configuration classes are usually applied based on the jars in application classpath and the beans we have defined additionally in **@Configuration** classes.  In this case, it does following things.   1. It detects **spring-webmvc** so configure default spring mvc application beans. It help in scan and configure @RestController and similar annotations. 2. It detects embed tomcat jars so configure embedded tomcat for us. 3. It detects JSON jars so configure JSON support to APIs.    4. Model classes and DAO     DAO class uses a static list to store data. Here we need to implement actual database interaction.   5. Spring Boot REST Demo To start the application, run the main() method in SpringBootDemoApplication class. It will start the embedded tomcat server. In server logs, you will see that API have been registered in spring context.         Spring Boot 2 REST POST with Headers:HTTP POST API  1. It adds an employee in the employees collection. 2. It accept employee data in Employee object. 3. It accepts and creates JSON meda type. 4. It accepts two HTTP headers i.e. X-COM-PERSIST and X-COM-LOCATION. First header is required and second header is optional. 5. It returns the location of resource created.  3. Custom error handler A [good designed resi api](https://restfulapi.net/rest-api-design-tutorial-with-example/) must have consistent error messages as well. One way to achieve it in spring boot applications is using **controller advice**. Inside **@ControllerAdvice** class, use **@ExceptionHandler** annotated methods to return consistent responses in invalid scenarios. 4. @SpringBootApplication : Same as previous example5. Model classes and DAO: Employee.java, Employees.java and EmployeeDao.java is same as previous          Spring boot exception handling – @ExceptionHandler example:In this **spring boot exception handler** tutorial, we will learn to **validate request body** sent to PUT/POST REST APIs. We will also learn to add custom error messages in API responses for validation errors. In this spring boot example, we will see primarily two major validation cases –   1. HTTP POST /employees and request body does not contain valid values or some fields are missing. It will return *HTTP status code* 400 with proper message in response body. 2. HTTP GET /employees/{id} and INVALID ID is sent in request. It will return HTTP status code 404 with proper message in response body.   For status code details: <https://restfulapi.net/http-status-codes/> 1. Create REST APIs and model classes Given REST APIs are from employee management module. 2. Spring boot exception handling – REST request validation2.1. Default spring validation support To apply default validation, we only need to add relevant annotations in proper places. i.e.   1. **Annotate model class with required validation specific annotations such as @NotEmpty, @Email etc.**    2.2. Exception model classes It is always a good advise to create exceptions that are meaningful and describe the problem well enough. One way is to create seperate classes to denote specific business usecase failure and return them when that usecase fail.  e.g. I have created RecordNotFoundException class for all such scenarios where a resource is requested by it’s ID, and resource is not found in the system.    Similarly, I have wrote an special class which will be returned for all failure cases. Having consistent error message structure for all APIs, help the API consumers to write more robust code. 2.3. Custom ExceptionHandler Now add one class extending **ResponseEntityExceptionHandler** and annotate it with @ControllerAdvice annotation.  ResponseEntityExceptionHandler is a convenient base class for to provide centralized exception handling across all @RequestMapping methods through @ExceptionHandler methods. @ControllerAdvice is more for enabling auto-scanning and configuration at application startup.    Above class handles multiple exceptions including RecordNotFoundException; and it also handle request validation errors in @RequestBody annotated object. Let’s see how it works           5. Summary In this **spring REST validation tutorial**, we learned to –   * validate ID when fetching resource by ID. * validate request body fields in POST/PUT APIs. * send consistent and structured error response in API responses.  Spring @Async rest controller example – Spring @EnableAsync:Spring comes with @EnableAsync annotation and can be applied on application classes for asynchronous behavior. This annotation will look for methods marked with @Async annotation and run in background thread pools. The @Async annotated methods can return CompletableFuture to hold the result of an asynchronous computation.Spring boot caching tutorial with example :1. What is caching? Caching is a mechanism to enhance the performance of a system. It is a temporary memory that lies between the application and the persistent database. Cache memory stores recently used data items in order to reduce the number of database hits as much as possible. What data should be cached? It varies in different scenario and requirement on how much time we can tolerate stale data.  So caching candidates will vary on each project, still those are few examples of caching –   * List of products available in an eCommerce store * Any Master data which is not frequently changed * Any frequently used database read query, where result does not change in each call at least for a specific period.  2. Types of cache2.1. In-memory caching This is the most frequently used area where caching is used extensively to increase performance of the application. In-memory caches such as **Memcached** and **Radis** are key-value stores between your application and your data storage. Since the data is held in RAM, it is much faster than typical databases where data is stored on disk.  RAM is more limited than disk, so cache invalidation algorithms such as **least recently used (LRU)** can help invalidate ‘cold’ entries and keep ‘hot’ data in RAM. 2.2. Database caching One popular in this area is first level cache of Hibernate or any ORM frameworks. 2.3. Web server caching Web servers can also cache requests, returning responses without having to contact application servers 2.4. CDN caching Caches can be located on the client side (OS or browser), server side, or in a distinct cache layer. Mainly static data like css or javascript are cashed. 3. Spring boot cache annotations3.1. @EnableCaching It enables Spring’s annotation-driven cache management capability. In spring boot project, we need to add it to the boot application class annotated with @SpringBootApplication. 3.2. @Cacheable It is used on the method level to let spring know that the response of the method are cacheable.  Eg:  @Cacheable(value="books", key="#isbn")  public Book findStoryBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)  We can also use **conditional caching** as well. For example,   |  | | --- | | @Cacheable(value="book", condition="#name.length < 50")  public Book findStoryBook (String name) 3.3. @CachePut Sometimes we need to manipulate the cacheing manually to put (update) cache before method call. This will allow us to update the cache and will also allow the method to be executed.  **Note**: It supports the same options as @Cacheable and should be used for cache population rather then method flow optimization.  Note that using @CachePut and @Cacheable annotations on the same method is generally discouraged because they have different behaviors. 3.4. @CacheEvict It is used when we need to evict (remove) the cache previously loaded of master data. When **CacheEvict** annotated methods will be executed, it will clear the cache.  We can specify key here to remove cache, if we need to remove all the entries of the cache then we need to use allEntries=true. 3.5. @Caching This annotation is required when we need both CachePut and CacheEvict at the same time. 5. Spring boot caching example5.1 Create Spring Boot project5.2 Create HTTP GET REST API: Create one REST service which will be a search service using GET request. Our main target is to cache the response of the method in the service layer where we will introduce an intentional delay to simulate the actual backend service call to get the result. In the first hit, the response will be delayed as we will have some simulated delay in the application, but in the subsequent calls, we will get much faster response. |     Notice that –   * The service layer method is annotated with @Cacheable("student"), as explained above, this annotation is enabling caching in this particular method and cache name is student. * In the getStudentByID() method we have an intentional 5 seconds delay using Thread.sleep(1000\*5). This is just to understand whether response is coming from cache or real backend.   To test, go to url  http://localhost:8080/student/1  and you will get one JSON response of a Student object. To notice, first time, the response will take at least 5 seconds to response, then subsequent responses of the same url will be faster. If you have difficulty understanding the differences, you can change the delay time in the service class.  Now change the url to get the Student id 2 by http://localhost:8080/student/2, you will again experience the delay, but in the subsequent calls the response will be served from Cache.   Spring retry module example with spring boot:Here we will learn how we can build applications using ****spring retry module**** facility where we have to call some methods where exception is sometimes expected and we have to retry the request.In such cases, if we have to implement any retry functionality on any backend service call, generally we implement using loop and a break condition and we keep on retrying to certain retry limits. But this solution is error prone. Spring has provided us one simple framework around this called spring-retry which can be configured using annotations. We can define the retry limits, fallback method etc.1. Why we need to retry?In most our projects we usually have some scenerio to retry few operation if it falls first time. For example let’s say the during invoking any backend external service, that service might be down for few reasons like network outage, server down, network glitch, deadlock etc. In this case usually we try to retry the operation for few times before we send any specific error to the client programs to make processing more robust and less prone to failure. Sometimes it helps to automatically retry a failed operation in case it might succeed on a subsequent attempt. I guess all of you have faced this and as workaround, you most probably solved this by having a loop and breaked that loop once you reached the retry limit, but now with the help of spring-retry module, we don’t have to write such code to handle the retry scenerio.2. Spring retry2.1. Spring retry annotations  * **@EnableRetry** – to enable spring retry in spring boot project * **@Retryable** – to indicate any method to be a candidate of retry * **@Recover** – to specify fallback method!  3. Spring retry example3.1. Development environment We will use the following technology stack to try out spring-retry in pur sample application below.   * Java, Eclipse, Maven as Developmnet Environment * Spring-boot as application framework * spring-retry module as retry module * AspectJ as depedency of spring-retry  3.2. Demo overview  1. Create one Spring boot project to expose once sample Rest API which will call one backend operation which is prone to failure, We will simulate this failure conditions to initiate the retry. 2. One service class which will actually invoke the remote api and this will send exception in case of failure, we will design the retry based on this custom exception, like once we receive this exception, we will retry for 3 times and finally return to client.   In those 3 attempts, if we get success response from backend service then that success response will be returned else a standard fallback method will be called. 3.3. Create Spring-boot project To do this we need to go to https://start.spring.io/ and select dependencies **web** and **retry**. 3.5. Create rest api endpoint Create one sample Rest controller which will call the backend service class where we will simulate the exception and spring-retry module will automatically retry.    In the Rest Api we will add two optional request parameters.   * **simulateretry** – parameter to simulate the exception scenario, so that spring can retry. * **simulateretryfallback** – as we are simulating the exception, after retry certain times we can either expect a successful backend call or all retry falls. In this case we will go to the fall back method to get hardcoded/error response. Now this parameter will ensure all the retry will fail and we will go to fall back path only.  3.6. @EnableRetry annotation To enable spring-retry we need to put one annotation in the Spring Boot Application calss. So open SpringRetryApplication class and add @EnableRetry in class level.   3.7. Backend Service Adapter interface and impl Now we will create one interface/implementation for calling the external service. Here we will not actually call any external service call, rather will simulate the success/failure scenerios by adding some random logic, as below       * **@Retryable** – This is the main annotation after @EnableRetry. This annotation tells that if we get RemoteServiceNotAvailableException from the method then retry maximum 3 times before sending the response. Also we are introducting delay of 1 second in each retry. * **@Recover** – in the fallback method indicates that if we don’t get any success response afer 3 retry, response will come from this fallback method. Make sure you pass expected exception as parameter, else spring will have hard time finding the exact method. * In the actual method from where the Remote service will be invoked, we have added some custom logic to control the Exception based on simulateretry and simulateretryfallback parameters. The code is simple, just returning the expected exception for retry if the conditions are met, else we will return the success response. Also we have added some randim logic based on the Random number to mimic the randomness of the failure. * The fallback method implementation sends simple fallback response.  4. Test the application The testing section is pretty straightforward. We will pass proper parameter in the REST request to simulate the retry requests. 4.1. Test retry – success or fallback Let’s start with 'http://localhost:8080/retry?simulateretry=true&simulateretryfallback=false' in browser. Based on the parameter, we are expecting exception in the backend service call and at the same time as simulateretryfallback=false, we are depending on the random logic (random % 2 == 0 –> even random number) we can expect a success response while retry.  So once we hit the request in browser, we might get exception in backend and spring will retry the same method multiple times. The outcome could be the Success response from backend. Here are the few lines of log from one of my request where spring is trying retry.     Spring boot – CommandLineRunner interface example:[CommandLineRunner](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/CommandLineRunner.html) interface is used to run a code block only once in application’s lifetime – after application is initialized.How to use CommandLineRunner You can use CommandLineRunner interface in three ways: 3) Using CommandLineRunner as Bean You can define a bean in SpringBootApplication which return the class that implements CommandLineRunner interface. Using @Order if multiple CommandLineRunner interface implementations You may have multiple implementations of CommandLineRunner interface. By default, spring boot to scan all its run() methods and execute it. But if you want to force some ordering in them, use [@Order](https://docs.spring.io/spring-framework/docs/4.3.7.RELEASE/javadoc-api/org/springframework/core/annotation/Order.html?is-external=true) annotation. Why use CommandLineRunner interface  * Command line runners are a useful functionality to execute the various types of code that only have to be run once, right after application startup. * FYI, Spring Batch relies on these runners in order to trigger the execution of the jobs. * We can use the dependency injection to our advantage in order to wire in whatever dependencies that we need and in whatever way we want – in run() method implementation. |