## **SPRING BOOT**

**Spring Boot**

Spring Boot is a process to develop spring based application with very less configuration and it is built on top of core spring framework to remove tedious work from the developer end and allow developers to focus on the business logic with minimal or zero configurations.

**Features of Spring boot**

* **Auto-Configuration** - No need to manually configure dispatcher servlet, static resource mappings, property source loader, message converters etc.
* **Dependency Management** - The different versions of commonly used libraries are pre-selected and grouped in different starter POMs that we can include in your project. By selecting one Spring Boot version we are implicitly selecting dozens of dependencies that we would have to otherwise select and harmonize ourself. Example-
* **Advanced Externalized Configuration** - There is a large list of bean properties that can be configured through application. properties file without touching java or xml config.

**Advantages Of Spring Boot:**

* Spring boot helps in resolving dependency conflict. It identifies required dependencies and import them for you.
* It provides a lot of default configurations which help you to create Spring application faster
* It has information of compitable version for all dependencies. It minimizes the runtime classloader issues.
* It avoids writing lots of boilerplate Code, Annotations and XML Configuration.
* It is very easy to integrate Spring Boot Application with its Spring Ecosystem like Spring JDBC, Spring ORM, Spring Data, Spring Security etc.
* It provides embedded HTTP server Tomcat so that you can develop and test quickly.

**Limitation of Spring Boot:**

* It is a very tough and time-consuming process to convert existing or legacy Spring Framework projects into Spring Boot Applications. It is applicable only for brand new/Greenfield Spring Projects.

**Spring Boot Project**

There are multiple approaches to create Spring Boot project. We can use any of the following approach to create application.

* Spring Maven Project
* Spring Starter Project Wizard
* Spring Initializr
* Spring Boot CLI

**Spring Initializr: Create Spring Boot Projects**

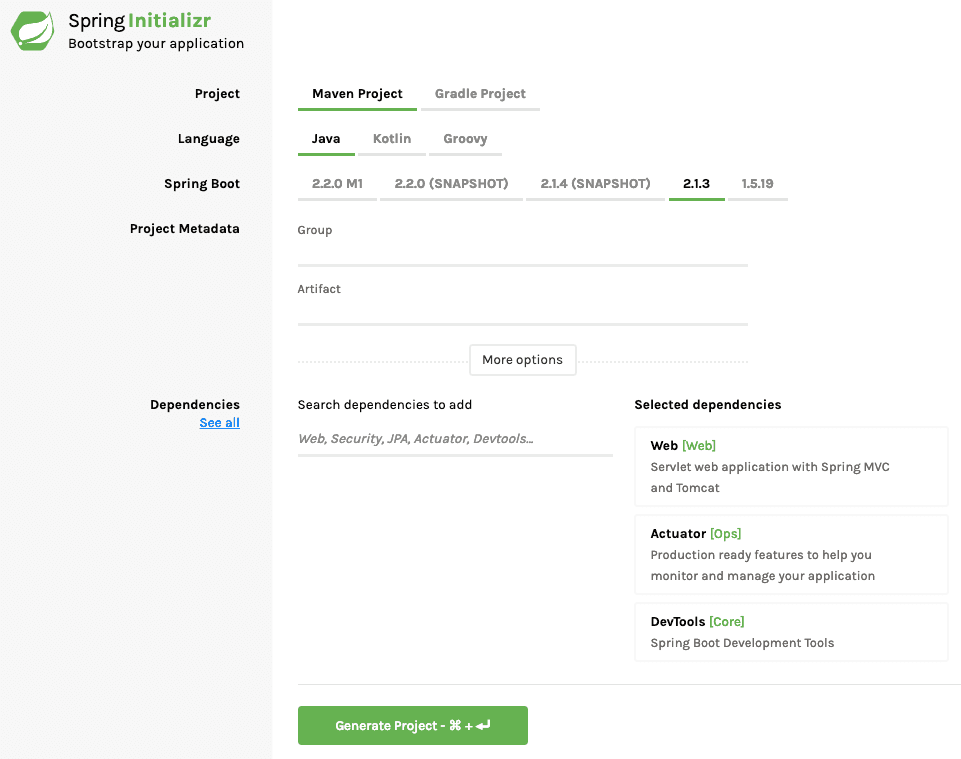
Spring Boot Initializr provides a simple interface to quickly bootstrap a Spring Boot application

[It](http://start.spring.io/) is great tool to bootstrap your Spring Boot projects.

It allows you to create a varied range of Spring Boot-based applications from a very simple UI. Some of the types of applications you can bootstrap are:

* Web applications
* Restful applications
* Batch applications

Let’s take the example of creating a web application with Spring Initializr.



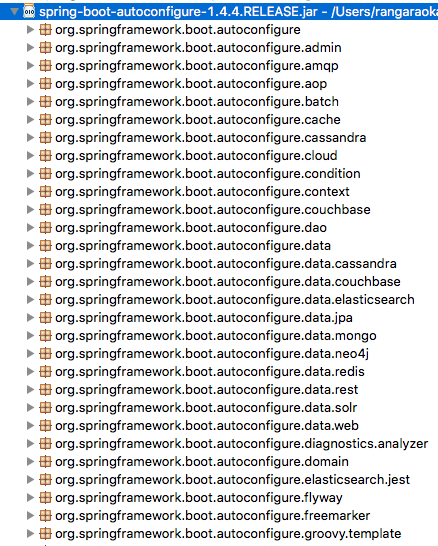
As shown in the image above, the following steps have to be taken:

* Launch Spring Initializr (<http://start.spring.io/>) and choose the following
  + Choose com.springboot as the Group
  + Choose student-services as the Artifact
  + Choose following dependency
    - Web
* Click the Generate Project button at the bottom of the page.
* Import the project into Eclipse.

As soon as we added in Spring Boot Starter Web as a dependency in our project, Spring Boot Autoconfiguration sees that Spring MVC is on the classpath. It autoconfigures dispatcherServlet, a default error page and webjars.

If you add Spring Boot Data JPA Starter, you will see that Spring Boot Auto Configuration auto configures a datasource and an Entity Manager.

**Where Is Spring Boot Auto Configuration Implemented?**

* All auto configuration logic is implemented in spring-boot-autoconfigure.jar. All auto configuration logic for MVC, data, JMS, and other frameworks is present in a single JAR.
* 

**Spring Boot & Spring MVC**

Spring Boot is well suited for web application development. You can easily create a self-contained HTTP server using embedded Tomcat, Jetty, or Undertow. Most web applications will use the spring-boot-starter-web module to get up and running quickly.

Spring MVC lets you create special **@Controller** or **@RestController** beans to handle incoming HTTP requests. Methods in your controller are mapped to HTTPusing **@RequestMapping** annotations.

**Template engines Support**

Spring Boot includes auto-configuration support for the following templating engines mention in this Spring Boot Tutorial.

* [FreeMarker](http://freemarker.org/docs/" \t "_top)
* [Thymeleaf](http://www.thymeleaf.org/)

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.

**Embedded containers does Spring Boot support:**

Spring Boot includes support for embedded **Tomcat**, **Jetty**, and **Undertow** servers.

**Reload changes on Spring Boot without having to restart server:**

Applications that use **spring-boot-devtools** will automatically restart whenever files on the classpath change. This can be a useful feature when working in an IDE as it gives a very fast feedback loop for code changes. By default, any entry on the classpath that points to a folder will be monitored for changes.

**<dependency>**

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

**<artifactId>spring-boot-devtools</artifactId>**

**<optional>true</optional>**

**</dependency>**

This can be achieved using DEV Tools. With this dependency any changes you save, the embedded tomcat will restart. Spring Boot has a Developer tools (DevTools) module which helps to improve the productivity of developers. One of the key challenge for the Java developers is to auto deploy the file changes to server and auto restart the server. Developers can reload changes on Spring Boot without having to restart my server. This will eliminate the need for manually deploying the changes every time. Spring Boot doesn’t have this feature when it has released it’s first version. This was a most requested features for the developers. The module DevTools does exactly what is needed for the developers. This module will be disabled in the production environment.

**Actuator**   
**Spring Boot Actuator** help you to monitor and manage your application health when you push it to production. It adds several production grade services to your application with little effort on your part. There are also has many features added to your application out-of-the-box for managing the service in a production (or other) environment. They’re mainly used to expose different types of information about the running application – health, metrics, info, dump, env etc.

In **Spring Boot 2, Actuator** comes with most endpoints disabled.

Thus, the only two available by default are **/health and /info**.

We need to add the following configuration to expose all endpoints :

|  |  |
| --- | --- |
|  | **management.endpoints.web.exposure.include=\*** |

To explicitly enable a specific endpoint (for example /shutdown), we use:

|  |  |
| --- | --- |
|  | **management.endpoint.shutdown.enabled=true** |

To expose all enabled endpoints except one (for example /loggers), we use:

|  |  |
| --- | --- |
|  | **management.endpoints.web.exposure.include=\***  **management.endpoints.web.exposure.exclude=loggers** |

By default, all Actuator endpoints are now placed under the **/actuator** path

* */beans – r*eturns all available beans in our *BeanFactory*. Unlike */auditevents*, it doesn't support filtering
* */env –*returns the current environment properties. Additionally, we can retrieve single properties
* */health –*summarises the health status of our application
* */info –*returns general information. It might be custom data, build information or details about the latest commit
* */metrics –*details metrics of our application. This might include generic metrics as well as custom ones
* */shutdown –*performs a graceful shutdown of the application
* */threaddump –*dumps the thread information of the underlying JVM

**Configure datasource using Spring boot:**

• Use either spring-boot-starter-jdbc or spring-boot-starterdata-jpa and include a JDBC driver on classpath

• Declare properties

**spring.datasource.url=jdbc:mysql://localhost/test**

**spring.datasource.username=dbuser**

**spring.datasource.password=dbpass**

**spring.datasource.driver-class-name=com.mysql.jdbc.Driver**

– Spring Boot will create a DataSource with properties set

**Disable a specific auto-configuration:**

If we want to disable a specific auto-configuration, we can indicate it using the exclude attribute of the @EnableAutoConfiguration annotation. For instance, this code snippet neutralizes DataSourceAutoConfiguration:

// other annotations

**@EnableAutoConfiguration(exclude = DataSourceAutoConfiguration.class)**

public class MyConfiguration { }

If we enabled auto-configuration with the @SpringBootApplication annotation — which has @EnableAutoConfiguration as a meta-annotation — we could disable auto-configuration with an attribute of the same name:

// other annotations

**@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)**

public class MyConfiguration { }

We can also disable an auto-configuration with the spring.autoconfigure.exclude environment property. This setting in the **application.properties** file does the same thing as before:

**spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration**

**Customizing the Banner using Image file**

* In addition to banner.txt file, you can use the image file to customize the banner of Spring Boot application. You need to add the banner.png or banner.jpg or banner.gif file to your classpath, Spring Boot converts the image file to ASCII art representation automatically and print it above the text banner.
* In this example, I am using the following image file. You can create your own image(png/jpg/gif) file for banner.
* Add the following image file in src/main/resources folder.

**Hikari Connection Pool with Spring Boot2**

[**HikariCP**](https://brettwooldridge.github.io/HikariCP/) is a lightweight and highly optimized JDBC connection pool. A **connection poo**l is a cache of database connections maintained so that the connections can be reused when future requests to the database are required. Connection pools may significantly reduce the overall resource usage.

***Java Database Connectivity (JDBC)*** is a Java API for accessing relational databases. It provides methods for querying and updating data in a database.

**JdbcTemplate** is a library that helps programmers create applications that work with relational databases and JDBC

[**HikariCP**](https://brettwooldridge.github.io/HikariCP/) is much faster, lightweight and have better performance as compare to other connection pool API( [**c3p0**](http://www.mchange.com/projects/c3p0/)**,**[**dbcp2**](https://commons.apache.org/proper/commons-dbcp/)). Because of all these compelling reasons, **HikariCP is now the default pool implementation in Spring Boot 2**.

In Spring Boot 2, Hikari is the default DataSource implementation.

This means we need not add explicit dependency in the pom.xml. **The spring-boot-starter-jdbc and spring-boot-starter-data-jpa** resolve it by default.

**dbcp2 Connection Pool with Spring Boot**

**dbcp2** (Data Base Connection Pooling) is a very popular library to manage the connection pool, **dbcp2** is the project of [**apache**](https://commons.apache.org/proper/commons-dbcp/).

Let’s try to understand requirements of connection pooling in simple words, To communicate with the database requires the database connection and create the connection with the database is heavy operation. While performing each operation requires to open connection and after perform, database operation closes the connection but this process will take too much time for opening and closing connection every database operations. To solve this issue database connection pooling is used.

The **connection pool** can maintain multiple connections with the database when the demand of  Connection object at that time return object from the pool instead of creating a new connection every time.

So now let’s try to understand how to configure **dbcp2** connection pooling with spring boot application

**Step 1: Remove tomcat-jdbc connection pool**

While working with **spring-boot-starter-jdbc**, Spring boot will default used a **tomcat-jdbc** connection pool. so first of all need to remove this dependency:

**<dependency>**

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

**<artifactId>spring-boot-starter-jdbc</artifactId> <!--It contains database base related classes-->**

**<exclusions>**

**<exclusion>**

**<groupId>org.apache.tomcat</groupId>**

**<artifactId>tomcat-jdbc</artifactId>**

**</exclusion>**

**</exclusions>**

**</dependency>**

**Step 2: Add commons-dbcp2 dependency**

Add **commons-dbcp2** in maven dependency or classpath as like:

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-dbcp2</artifactId>

<version>2.2.0</version>

</dependency>

@Component

public class EmployeeDAO {

@Autowired

private JdbcTemplate jdbcTemplate;

public java.util.List<Employee> getEmployeeList() {

List<Map<String, Object>> employees = jdbcTemplate.queryForList("select \* from employee");

}

}

**Use Jetty instead of the tomcat in our web application?**

Spring Boot web starters use Tomcat as the default embedded servlet container. When switching to a different HTTP server, we need to exclude the default dependencies besides including the one we need. Spring Boot provides separate starters for HTTP servers to help make this process as easy as possible. To use Jetty, we need to exclude Tomcat and include Jetty in our application’s pom.xml file.

<dependency>

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

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

<exclusions>

<!-- Exclude the Tomcat dependency -->

<exclusion>

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

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

</exclusion>

</exclusions>

</dependency>

<!-- Use Jetty instead -->

<dependency>

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

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

</dependency>

**Swagger:**

Swagger is a tool that generates visual representations of RESTful web services.

In the Maven we need the swagger dependency

**<dependency>**

**<groupId>io.springfox</groupId>**

**<artifactId>springfox-swagger2</artifactId>**

**<version>2..4.0</version>**

**</dependency>**

**<dependency>**

**<groupId>io.springfox</groupId>**

**<artifactId>springfox-swagger-ui</artifactId>**

**<version>2..4.0</version>**

**</dependency>**

To enable the **Swagger 2** we use the annotation **@EnableSwagger2** in the main Spring Boot class.

To run Swagger API type the below url in browser:

**http://localhost:8080/swagger-ui.html**

A custom banner is generally used as one of the following in the Spring Boot applications;

• Text Banner: For text banner just create a file named as banner.txt with desired text and keep it at the location src\main\resources.   
• Image Banner: For image banner just create a file named as banner.gif and keep it at the location src\main\resources.   
• In the application.properties we can configure following banner properties.   
• banner.charset: It configures banner encoding. Default is UTF-8.  
• banner.location: It is banner file location. Default is classpath:banner.txt   
• banner.image.location: It configures banner image file location. Default is classpath:banner.gif. File can also be jpg, png.   
• banner.image.width: It configures width of the banner image in char. Default is 76.   
• banner.image.height: It configures height of the banner image in char. Default is based on image height.   
• banner.image.margin: It is left hand image margin in char. Default is 2.   
• banner.image.invert: It configures if images should be inverted for dark terminal themes. Default is false.

**Externalized Properties**

Instead of keeping the properties file under classpath, we can keep the properties in different location or path. While running the JAR file, we can specify the properties file path. You can use the following command to specify the location of properties file while running the JAR −

-Dspring.config.location = C:\application.properties

Externalized Properties

## **Spring Boot Active Profile**

**Active profile in Spring Boot:**

There are two ways to set the active profile in Spring Boot.

* Pass in the active profile as an argument while launching the application.
* Use the application.properties file to set the active profile.

**java -jar -Dspring.profiles.active=production application-1.0.0-RELEASE.jar //pass as command line argument**

**spring.profiles.active=production**

Spring Boot supports different properties based on the Spring active profile. For example, we can keep two separate files for development and production to run the Spring Boot application.

### Spring active profile in application.properties

Let us understand how to have Spring active profile in application.properties. By default, application. properties will be used to run the Spring Boot application. If you want to use profile based properties, we can keep separate properties file for each profile as shown below –

**application.properties**

server.port = 8080

spring.application.name = demoservice

**application-dev.properties**

server.port = 9090

spring.application.name = demoservice

**application-prod.properties**

server.port = 4431

spring.application.name = demoservice

While running the JAR file, we need to specify the spring active profile based on each properties file. By default, Spring Boot application uses the application.properties file. The command to set the spring active profile is shown below −



**Spring Boot Starters:**

Starter is like a small spring project for each module such as web MVC, JDBC,ORM and so on.For your spring application ,you just add the starters of the respective module in the classpath and Spring Boot will ensure that the necessary libraries are added to the build by using Maven.

Spring Boot **Starters** reduces a build’s dependencies and Spring Boot auto configuration reduces the Spring configuration.

If you want to exclude auto-configuration for the some of the modules ,then you use the exclude property of @SpringBootApplication.Lets look at the following

**@SpringBootApplication(exclude={DataSourceAutoConfiguration.class,HibernateJpaAutoConfiguration.class})**

**@RestController annotation:** It indicates that this is the controller class and its result writes in to the response body and does not want to render view.

**@GetMapping** annotation:It is a shortform of @RequestMapping(method=RequestMethod.GET)

**@SpringBootApplication :** This annotation tells Spring Boot,when launched,to scan recursively for Spring components inside this package and register them.It also tells Spring Boot to enable auto-configuration.

SpringApplication.run():The SpringApplication class is responsible for creating the **ApplicationContext** from the classpath, scan the configuration classes and launch the application.

## **Steps Executed under this method –**

So in short when the main method runs following **steps** occur:

1. **Application Context** is started.
2. Using application context **autodiscovery** occurs: @ComponentScan
3. All default configurations are set up ie based on dependencies mentioned spring boot automatically sets up defaults. It makes use of intelligence that if we have included spring-web starter then dispatcher servlet is auto-configured. (**@EnableAutoConfiguration**)
4. An embedded servlet container is started. ( No need to set up a separate web server ) . Note embedded servlet container is launched only if the web is mentioned in a dependency

When **SpringApplication.run()** command is invoked, the Application Context is created by calling the method below:

public ConfigurableApplicationContext run(String... args) {

// Create, load, refresh, and run the ApplicationContext

context = createApplicationContext();

return context ; // handle to the context object for the developer

}

**What exactly is the type of this context?**

**Answer:**

The createApplicationContext method checks if it is a web or standalone application based on the type it creates for the context. I was creating a REST-based controller for which a context of type **AnnotationConfigEmbeddedWebApplicationContext** was initialized. In the case of a standalone application, **AnnotationConfigApplicationContext** will be initialized.

How are the beans created once the context is initialized?

**Answer:**

When the constructor of the context is invoked, it will register the annotated class beans with the context. That's why no XML configurations are required. All your **@Repository**, **@Component, @Service**, and Controller beans will be registered and the context is returned. The following lines of code are executed for context initialization and bean creation for a web application.

public AnnotationConfigEmbeddedWebApplicationContext(Class<?>... annotatedClasses) {

this();

register(annotatedClasses);

refresh(); // Refreshing org.springframework.boot.context.embedded. This log appears in the console

}

Which servlet acts as a front controller?

**Answer:**

No prizes for guessing that: DispatcherServlet.

The **AnnotationConfigEmbeddedWebApplicationContext** class extends the EmbeddedWebApplicationContext, which registers the dispatcher servlet.

public static final String DISPATCHER\_SERVLET\_NAME = ServletContextInitializerBeans.DISPATCHER\_SERVLET\_NAME;

 What about the embedded Tomcat?

**Answer:**

Normally, starting an embedded Tomcat is as easy as instantiating the Tomcat class.

Include the following dependencies in Maven

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-core</artifactId>

<version>${tomcat.version}</version>

</dependency>

And write a class to bootstrap Tomcat:

Tomcat tomcat = new Tomcat();

tomcat.setPort(8080);

// Create context object and set it

tomcat.addContext ("/mycontext);

tomcat.start();

tomcat.getServer().await();

So with regards to Spring Boot. the **EmbeddedWebApplicationContext** creates an instance of **org.springframework.boot.context.embedded.tomcat.TomcatEmbeddedServletContainer** and adds the context.

**TomcatEmbeddedServletContainer** class has Tomcat as an instance variable.

Check the **selfInitialize**() method and **prepareEmbeddedWebApplicationContext** of the **EmbeddedWebApplicationContext** class:

prepareEmbeddedWebApplicationContext() {

servletContext.log("Initializing Spring embedded WebApplicationContext"); // these logs are printed in your STS console.

logger.info("Root WebApplicationContext: initialization completed in " );  // these logs are printed in your STS console.

}

# **How to rename application.properties file in Spring Boot application?**

By default, [**Spring Boot**](https://projects.spring.io/spring-boot/) will look for your externalized properties in a file named **application.properties** located in **resources** folders :

Once we create a Spring Boot project,

* Let’s delete the ***application.properties*** file from the resources folder of our project.
* Then, let’s create a ***customapp.properties***file in the resources folder of our project. We’ll provide all our application configurations in this properties file.

***spring.config.name*** property holds the name of our properties file in the Spring Boot application.

## System Property:

@SpringBootApplication

public class MasteringSpringBootApplication {

public static void main(String[] args) {

**System.setProperty("spring.config.name", "myapp");**

SpringApplication.run(MasteringSpringBootApplication.class, args);

}

}

## **SpringBootApplication scanBasePackages:**

By default SpringApplication scans the configuration class package and all it’s sub-pacakges. So if our SpringBootRestApplication class is in com.spring.main package, then it won’t scan com.spring.controller package. We can fix this situation using SpringBootApplication scanBasePackages property.

@SpringBootApplication(scanBasePackages="com.spring")

public class SpringBootRestApplication {

}

## Spring Boot Auto-Configured Beans

Since Spring Boot provides auto-configuration, there are a lot of beans getting configured by it. We can get a list of these beans using below code snippet.

ApplicationContext ctx = SpringApplication.run(SpringBootRestApplication.class, args);

String[] beans = ctx.getBeanDefinitionNames();

for(String s : beans) System.out.println(s);

Below is the list of beans configured by our spring boot application.

**Spring Boot Devtools Tutorial**

## **Enabling Dev Tools Module**

To enable dev tools in spring boot application is very easy. Just add the **spring-boot-devtools** dependency in your build file.

**Maven**

|  |
| --- |
| pom.xml |
| **<dependencies>**  **<dependency>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-devtools</artifactId>**  **<optional>true</optional>**  **</dependency>**  **</dependencies>** |

## **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 extention for it.

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 |

## **Excluding Resources from auto-reload**

By default, Auto-reload works on these paths:

1. /META-INF/maven
2. /META-INF/resources
3. /resources
4. /static
5. /public
6. /templates

If you want to disable auto-reload in browser for files in few of these paths, then use spring.devtools.restart.exclude property. e.g.

|  |
| --- |
| spring.devtools.restart.exclude=static/\*\*,public/\*\* |

## **Watching/Excluding Additional Paths**

There may be few files not in classpath, but you still may want to watch those addtional files/paths to reload the application. To do so, use the spring.devtools.restart.additional-paths property.

|  |
| --- |
| spring.devtools.restart.additional-paths=script/\*\* |

Similarily, If you want to keep those defaults and **add additional exclusions**, use the spring.devtools.restart.additional-exclude property instead.

|  |
| --- |
| spring.devtools.restart.additional-exclude=styles/\*\* |

## **Automatic server restart**

Auto-restart means reloading the java classes and consiguration at server side. After the server side changes are re-deployed dynamically, server restart happen and load the modified code and configutation.

## **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** |

## **Logging in spring boot**

**Logging in spring boot** is very flexible and easy to configure. Spring boot active enabled logging is determined by [spring-boot-starter-logging](https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot-starters/spring-boot-starter-logging/pom.xml) artifact

If we do not provide any logging specific configuration, we will still see logs printed in “console”. These are because of **default logging support** provided in spring boot which uses **Logback**.

Spring boot’s internal logging is written with Apache Commons Logging so it is one and only mandatory dependency. Till, boot 1.x – we had to import it manually. Since boot 2.x, it is downloaded transitively. To be more precise, spring-boot-starter-web depends on spring-boot-starter-logging, which pulls in spring-jcl for us.

## **Add log statements**

To add log statements in application code, use org.slf4j.Logger and org.slf4j.LoggerFactory from SLF4J. It provides lots of useful methods for logging anf also decouple the logging implementation from application.

|  |
| --- |
| Application.java |
| import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;    @SpringBootApplication  public class Application  {   private static final Logger LOGGER=LoggerFactory.getLogger(Application.class);        public static void main(String[] args) {          SpringApplication.run(Application.class, args);            LOGGER.info("Simple log statement with inputs {}, {} and {}", 1,2,3);      }  } |
| Console |
| 2019-07-28 12:16:57.129  INFO 3416 --- [main]  com.demo.Application: Simple log statement with inputs 1, 2 and 3 |

## 1.2. **Logging Level**

Logback supports ERROR, WARN, INFO, DEBUG, or TRACE as logging level. By default, logging level is set to **INFO**. It means that code>DEBUG and TRACE messages are not visible.

To enable debug or trace logging, we can set the logging level in application.properties file. Also, we can pass the –debug or –trace arguments on the command line while starting the application.

|  |
| --- |
| Configuration |
| # In properties file  debug=true    # In Console  $ java -jar target/my-app-0.0.1-SNAPSHOT.jar --trace |

We can apply logging levels to specific packages as well. It can be done either in console or application.properties file.

|  |
| --- |
| Configuration |
| # In Console  -Dlogging.level.org.springframework=ERROR  -Dlogging.level.com.howtodoinjava=TRACE    # In properties file  logging.level.org.springframework=ERROR  logging.level.com.howtodoinjava=TRACE |

If the log level for a package is defined multiple times with different log levels, the lowest level will be used. TRACE is lowest and ERROR is highest.

## 1.3. **Log format**

The default log statement formatting is mentioned in [defaults.xml](https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot/src/main/resources/org/springframework/boot/logging/logback/defaults.xml) file.

|  |
| --- |
| defaults.xml |
| <conversionRule conversionWord="clr"  converterClass="org.springframework.boot.logging.logback.ColorConverter" />    <conversionRule conversionWord="wex"  converterClass="org.springframework.boot.logging.logback.WhitespaceThrowableProxyConverter" />    <conversionRule conversionWord="wEx"  converterClass="org.springframework.boot.logging.logback.ExtendedWhitespaceThrowableProxyConverter" />    <property name="CONSOLE\_LOG\_PATTERN" value="${CONSOLE\_LOG\_PATTERN:-%clr(%d{${LOG\_DATEFORMAT\_PATTERN:-yyyy-MM-dd HH:mm:ss.SSS}})  {faint} %clr(${LOG\_LEVEL\_PATTERN:-%5p}) %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint} %clr(%-40.40logger{39})  {cyan} %clr(:){faint} %m%n${LOG\_EXCEPTION\_CONVERSION\_WORD:-%wEx}}"/>    <property name="FILE\_LOG\_PATTERN" value="${FILE\_LOG\_PATTERN:-%d{${LOG\_DATEFORMAT\_PATTERN:-yyyy-MM-dd HH:mm:ss.SSS}}  ${LOG\_LEVEL\_PATTERN:-%5p} ${PID:- } --- [%t] %-40.40logger{39} : %m%n${LOG\_EXCEPTION\_CONVERSION\_WORD:-%wEx}}"/> |

It outputs following information.

* **Date and Time**: Millisecond precision and easily sortable.
* **Log Level**: ERROR, WARN, INFO, DEBUG, or TRACE.
* Process ID.
* A --- separator to distinguish the start of actual log messages.
* **Thread name**: Enclosed in square brackets (may be truncated for console output).
* **Logger name**: This is usually the source class name (often abbreviated).
* The log message.

To **customize the log format**, use logging.pattern.console and logging.pattern.file properties.

|  |
| --- |
| application.properties |
| # Logging pattern for the console  logging.pattern.console= %d{yyyy-MM-dd HH:mm:ss} - %logger{36} - %msg%n    # Logging pattern for file  logging.pattern.file= %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg% |

## 1.4. **Logging to file**

By default spring boot logs to console only. If we want to enable file logging, we can easily do it using simple property logging.file or logging.path.

When using logging.path, it will create a file named spring.log in mentioned package.

|  |
| --- |
| application.properties |
| # Output to a temp\_folder/file  logging.file=c:/temp/application.log    #logging.path=/my-folder/    # Logging pattern for file  logging.pattern.file= %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg% |

## 2. **Logback Logging**

The default logging is good enough for most usecases. But sometimes in enterprise applications, we need more fine control over logging with other complex requirements. In that case, having a dedicated logging configuration is suitable.

Spring boot by default uses logback, so to customize it’s behavior, all we need to add only **logback.xml in classpath** and define customization over the file.

|  |
| --- |
| logback.xml |
| <?xml version="1.0" encoding="UTF-8"?>  <configuration>        <property name="LOG\_LOCATION" value="c:/temp" />        <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">          <encoder>              <pattern>%d{yyyy-MM-dd HH:mm:ss} - %logger{36} - %msg%n</pattern>          </encoder>      </appender>        <appender name="FILE" class="ch.qos.logback.core.FileAppender">          <File>{LOG\_LOCATION}/mylog.log</File>          <encoder>               <pattern>%d{yyyy-MM-dd HH:mm:ss} - %logger{36} - %msg%n</pattern>          </encoder>          <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">              <fileNamePattern>${LOG\_LOCATION}/archived/mylog-%d{yyyy-MM-dd}.%i.log              </fileNamePattern>              <timeBasedFileNamingAndTriggeringPolicy                  class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">                  <maxFileSize>10MB</maxFileSize>              </timeBasedFileNamingAndTriggeringPolicy>          </rollingPolicy>      </appender>        <root level="INFO">          <appender-ref ref="CONSOLE"/>          <appender-ref ref="FILE"/>      </root>        <!-- Application logs at trace level -->      <logger name="com.howtodoinjava" level="trace" additivity="false">          <appender-ref ref="RollingFile" />          <appender-ref ref="Console" />      </logger>    </configuration> |

## 3. **Log4j2 Logging**

## Step 1: Exclude logback and include log4j2

As mentioned earlier, spring boot uses logback as default. So if we have to use any other logging framework e.g. log4j2, we must **exclude logback** from classpath of the application. Also, add **spring-boot-starter-log4j2** to classpath.

|  |
| --- |
| 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-logging</artifactId>          </exclusion>      </exclusions>  </dependency>    <dependency>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-starter-log4j2</artifactId>  </dependency> |

## Step 2: **Add log4j2 configuration file**

Now, add log4j2 specific configuration file in classpath (typically in **resources** folder). It can be named as any of the following:

* log4j2-spring.xml
* log4j2.xml

If we have logging configuration in any other file (e.g. log4j2.properties, applogs.xml etc), we can use logging.file property to specify it’s path application.properties file.

|  |
| --- |
| log4j2.xml |
| <?xml version="1.0" encoding="UTF-8"?>  <Configuration status="WARN" monitorInterval="30">      <Properties>          <Property name="LOG\_PATTERN">%d{yyyy-MM-dd'T'HH:mm:ss.SSSZ} %p %m%n</Property>          <Property name="APP\_LOG\_ROOT">c:/temp</Property>      </Properties>      <Appenders>          <Console name="console" target="SYSTEM\_OUT">              <PatternLayout pattern="${LOG\_PATTERN}" />          </Console>            <RollingFile name="file"              fileName="${APP\_LOG\_ROOT}/SpringBoot2App/application.log"              filePattern="${APP\_LOG\_ROOT}/SpringBoot2App/application-%d{yyyy-MM-dd}-%i.log">              <PatternLayout pattern="${LOG\_PATTERN}" />              <Policies>                  <SizeBasedTriggeringPolicy size="19500KB" />              </Policies>              <DefaultRolloverStrategy max="1" />          </RollingFile>        </Appenders>      <Loggers>          <Root level="info">              <AppenderRef ref="console" />              <AppenderRef ref="file" />          </Root>      </Loggers>  </Configuration> |

## Step 3: **With or without Slf4j**

By default, if you are using SLF4J logger classes i.e. org.slf4j.Logger and org.slf4j.LoggerFactory, nothing needs to be changed in application code and all log statement will continue printing in target appenders.

If you are targeting to use log4j2 specific classes only, use org.apache.logging.log4j.Logger and org.apache.logging.log4j.LogManager.

I will recommend to use SLF4J logger classes.

|  |
| --- |
| SLF4J logger classes |
| import org.slf4j.Logger;  import org.slf4j.LoggerFactory;    @SpringBootApplication  public class Application  {  private static final Logger LOGGER = LoggerFactory.getLogger(Application.class);        public static void main(String[] args) {          SpringApplication.run(Application.class, args);            LOGGER.info("Simple log statement with inputs {}, {} and {}", 1,2,3);      }  } |
| LOG4J2 logger classes |
| import org.apache.logging.log4j.LogManager;  import org.apache.logging.log4j.Logger;    @SpringBootApplication  public class Application  {      private static Logger LOGGER = LogManager.getLogger(Application.class);        public static void main(String[] args) {          SpringApplication.run(Application.class, args);            LOGGER.info("Simple log statement with inputs 1, 2 and 3");      }  } |

## **Spring Boot with H2 Database**

Learn to configure H2 database with [Spring boot](https://howtodoinjava.com/spring-boot-tutorials/) to create and use an in-memory database in runtime, generally for [unit testing](https://howtodoinjava.com/junit-4/) or POC purposes. Remember an in-memory database is created/initialized when an application starts up; and destroyed when the application shuts down.

## **What is H2 Database?**

H2 is one of the popular in-memory databases written in Java. It can be embedded in Java applications or run in the client-server mode.

## **Maven Dependency**

To use H2 in Spring boot application, all we need to do is adding H2 runtime jar into dependencies. The best way to add is through maven.

|  |
| --- |
| pom.xml |
| **<dependency>**  **<groupId>com.h2database</groupId>**  **<artifactId>h2</artifactId>**  **<scope>runtime</scope>**  **</dependency>** |

## H2 Configuration Options

#### 3.1. **Simple configuration**

Spring provides very easy configuration options to connect to any database using simple properties. Below are the configuration properties, we shall have in application.properties file.

|  |
| --- |
| **application.properties** |
| **spring.datasource.url=jdbc:h2:mem:testdb**  **spring.datasource.driverClassName=org.h2.Driver**  **spring.datasource.username=sa**  **spring.datasource.password=**  **spring.jpa.database-platform=org.hibernate.dialect.H2Dialect** |

Please note by default, Spring Boot configures the in-memory database connection with the username 'sa' and an empty password ' '. If you wish to change these values, override them in above properties options.

## **H2 Console**

#### 5.1. **Enable H2 console**

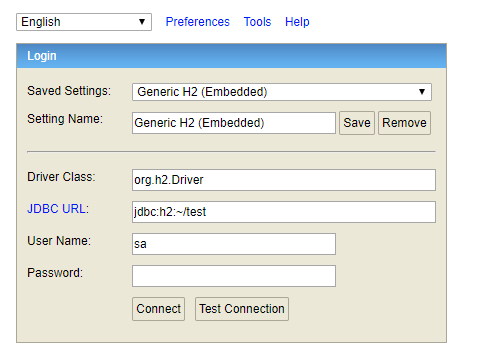
By default, the console view of H2 database is disabled. We must enable it to view and access it in browser. Note that we can customize the URL of H2 console which, by default, is '/h2'.

|  |
| --- |
| application.properties |
| # Enabling H2 Console  **spring.h2.console.enabled=true**    # Custom H2 Console URL  **spring.h2.console.path=/h2** |

#### 5.2. **Accessing H2 console**

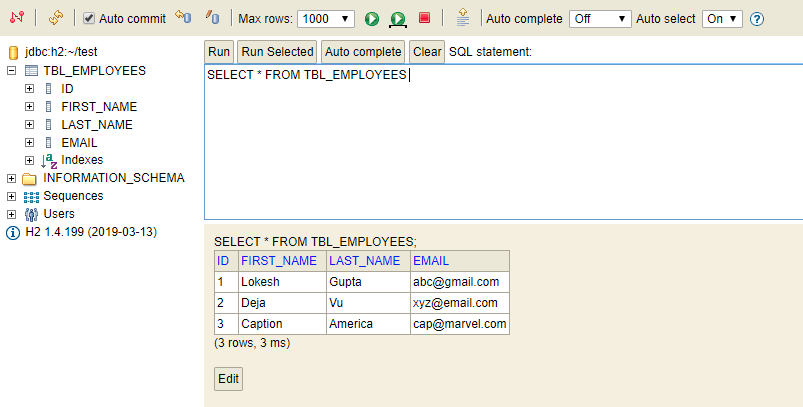
Start the spring boot application and access the console in browser with **URL : http://localhost:8080/h2**.

We can see the console like this.



**H2 Database Console Login Window**

Now enter the configured username and password. We can verify the table structure and default data inserted through SQL files.

**H2 Console View**

#### 5.3. **Other configuration options**

Spring boot provides two more properties to further customize the behavior of H2 console. i.e. we can enable/disable the database trace logs and we can enable/disable the remote access of H2 console.

By default both properties are false.

|  |
| --- |
| application.properties |
| # Whether to enable trace output.  **spring.h2.console.settings.trace=false**    # Whether to enable remote access.  **spring.h2.console.settings.web-allow-others=false** |

#### **Working With hashcode() and equals():**

By default, the Java super class **java.lang.Object**provides two important methods for comparing objects: **equals()** and **hashcode()**. These methods become very useful when implementing interactions between several classes in large projects. In this article, we will talk about the relationship between these methods, their default implementations, and the circumstances that force developers to provide a custom implementation for each of them.

Method Definition and Default Implementation

* **equals(Object obj):** a method provided by **java.lang.Object** that indicates whether some other object passed as an argument is **"equal to"** the current instance. The default implementation provided by the JDK is based on memory location — two objects are equal if and only if they are stored in the same memory address.
* **hashcode():**a method provided by **java.lang.Object** that returns an integer representation of the object memory address. By default, this method returns a random integer that is unique for each instance. This integer might change between several executions of the application and won't stay the same.

## The Contract Between equals() and hashcode()

The default implementation is not enough to satisfy business needs, especially if we're talking about a huge application that considers two objects as equal when some business fact happens. In some business scenarios, developers provide their own implementation in order to force their own equality mechanism regardless the memory addresses.

As per the Java documentation, developers should override both methods in order to achieve a fully working equality mechanism — it's not enough to just implement the **equals()** method.

**If two objects are equal according to the equals(Object) method, then calling the hashcode() method on each of the two objects must produce the same integer result.**

In the following sections, we provide several examples that show the importance of overriding both methods and the drawbacks of overriding **equals()** without **hashcode()**.

## Practical Example

We define a class called **Student** as the following:

package com.programmer.gate.beans;

public class Student {

private int id;

private String name;

public Student(int id, String name) {

this.name = name;

this.id = id;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

For testing purposes, we define a main class **HashcodeEquals**that checks whether two instances of **Student** (who have the exact same attributes) are considered as equal.

public class HashcodeEquals {

public static void main(String[] args) {

Student alex1 = new Student(1, "Alex");

Student alex2 = new Student(1, "Alex");

System.out.println("alex1 hashcode = " + alex1.hashCode());

System.out.println("alex2 hashcode = " + alex2.hashCode());

System.out.println("Checking equality between alex1 and alex2 = " + alex1.equals(alex2));

}

}

Output:

alex1 hashcode = 1852704110

alex2 hashcode = 2032578917

Checking equality between alex1 and alex2 = false

Although the two instances have exactly the same attribute values, they are stored in different memory locations. Hence, they are not considered equal as per the default implementation of **equals()**. The same applies for **hashcode()** — a random unique code is generated for each instance.

## Overriding equals()

For business purposes, we consider that two students are equal if they have the same ID, so we override the **equals()** method and provide our own implementation as the following:

@Override

public boolean equals(Object obj) {

if (obj == null) return false;

if (!(obj instanceof Student))

return false;

if (obj == this)

return true;

return this.getId() == ((Student) obj).getId();

}

In the above implementation, we are saying that two students are equal if and only if they are stored in the same memory address **OR**they have the same ID. Now if we try to run **HashcodeEquals**,we will get the following output:

alex1 hashcode = 2032578917

alex2 hashcode = 1531485190

Checking equality between alex1 and alex2 = true

As you noticed, overriding **equals()**with our custom business forces Java to consider the ID attribute when comparing two **Student** objects.

### equals() With ArrayList

A very popular usage of **equals()**is defining an array list of **Student** and searching for a particular student inside it. So we modified our testing class in order the achieve this.

public class HashcodeEquals {

public static void main(String[] args) {

Student alex = new Student(1, "Alex");

List < Student > studentsLst = new ArrayList < Student > ();

studentsLst.add(alex);

System.out.println("Arraylist size = " + studentsLst.size());

System.out.println("Arraylist contains Alex = " + studentsLst.contains(new Student(1, "Alex")));

}

}

After running the above test, we get the following output:

Arraylist size = 1

Arraylist contains Alex = true

## Overriding hashcode()

Okay, so we override **equals()**and we get the expected behavior — even though the hash code of the two objects are different. So, what's the purpose of overriding **hashcode()**?

### equals() With HashSet

Let's consider a new test scenario. We want to store all the students in a **HashSet**, so we update **HashcodeEquals**as the following:

public class HashcodeEquals {

public static void main(String[] args) {

Student alex1 = new Student(1, "Alex");

Student alex2 = new Student(1, "Alex");

HashSet < Student > students = new HashSet < Student > ();

students.add(alex1);

students.add(alex2);

System.out.println("HashSet size = " + students.size());

System.out.println("HashSet contains Alex = " + students.contains(new Student(1, "Alex")));

}

}

If we run the above test, we get the following output:

HashSet size = 2

HashSet contains Alex = false

WAIT!! We already override **equals()**and verify that alex1 and alex2 are equal, and we all know that **HashSet** stores unique objects, so why did it consider them as different objects ?

**HashSet** stores its elements in memory buckets. Each bucket is linked to a particular hash code. When calling students.add(alex1), Java stores alex1 inside a bucket and links it to the value of alex1.hashcode(). Now any time an element with the same hash code is inserted into the set, it will just replace alex1.However, since alex2 has a different hash code, it will be stored in a separate bucket and will be considered a totally different object.

Now when **HashSet** searches for an element inside it, it first generates the element's hash code and looks for a bucket which corresponds to this hash code.

Here comes the importance of overriding **hashcode()**, so let's override it in **Student** and set it to be equal to the ID so that students who have the same ID are stored in the same bucket:

@Override

public int hashCode() {

return id;

}

Now if we try to run the same test, we get the following output:

HashSet size = 1

HashSet contains Alex = true

See the magic of **hashcode()**! The two elements are now considered as equal and stored in the same memory bucket, so any time you call contains() and pass a student object holding the same hash code, the set will be able to find the element.

The same is applied for **HashMap, HashTable**, or any data structure that uses a hashing mechanism for storing elements.

## Conclusion

In order to achieve a fully working custom equality mechanism, it is mandatory to override **hashcode()** each time you override **equals().** Follow the tips below and you'll never have leaks in your custom equality mechanism:

* If two objects are equal, they MUST have the same hash code.
* If two objects have the same hash code, it doesn't mean that they are equal.
* Overriding **equals()**alone will make your business fail with hashing data structures like: **HashSet, HashMap, HashTable** ... etc.
* Overriding **hashcode()**alone doesn't force Java to ignore memory addresses when comparing two objects.

#### **How to Iterate a HashMap of ArrayLists of String in Java.**

This Java program shows how to iterate a HashMap that contains arraylists of String.

In the Java program to iterate a HashMap containing ArrayLists there is a method getMap() where 3 lists are created and stored in the HashMap.

First you need to iterate the HashMap, though there are several ways to iterate over a HashMap, but here I have used the for-each loop for iterating the created HashMap. Each Map.Entry object is a key-value pair where value is the ArrayList stored with the given key. That's the list retrieved using listEntry.getValue() method.

In the second for-each loop List that is retrieved using listEntry.getValue() is iterated and the elements that are in the list are displayed.

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

public class MapLoop {

public static void main(String[] args) {

MapLoop mapLoop = new MapLoop();

Map<String, List<String>> cityMap = mapLoop.getMap();

int i = 0;

// iterating over a map

for(Map.Entry<String, List<String>> listEntry : cityMap.entrySet()){

System.out.println("Iterating list number - " + ++i);

// iterating over a list

for(String cityName : listEntry.getValue()){

System.out.println("City - " + cityName);

}

}

}

/\*\*

\* A method to create a list and store it in a Map

\* @return

\*/

private Map<String, List<String>> getMap(){

Map<String, List<String>> cityMap = new HashMap<String, List<String>>();

// First List

List<String> temp = new ArrayList<String>();

temp.add("Delhi");

temp.add("Mumbai");

// Putting first list in the map

cityMap.put("1", temp);

// Second List

temp = new ArrayList<String>();

temp.add("Hyderabad");

temp.add("Bangalore");

// Putting second list in the map

cityMap.put("2", temp);

// Third List

temp = new ArrayList<String>();

temp.add("Kolkata");

temp.add("Chennai");

// Putting third list in the map

cityMap.put("3", temp);

return cityMap;

}

}

**Checked Exception:**

By extending Exception, you can create a checked exception:

class NotEnoughBalance extends Exception {

// Implementation

}

**Unchecked Exception:**

By extending RuntimeException, you can create unchecked exception:

class NotEnoughBalance extends RuntimeException {

// Implementation

}