Spring Boot Advance Concepts

Amit Kumar

Spring Boot Actuator

An actuator is a manufacturing term that refers to a mechanical device for moving or controlling something. Actuators can generate a large amount of motion from a small change.

Spring Boot Actuator is a sub-project of the Spring Boot Framework. It includes a number of additional features that help us to monitor and manage the Spring Boot application. It contains the actuator endpoints (the place where the resources live). We can use HTTP and JMX endpoints to manage and monitor the Spring Boot application. If we want to get production-ready features in an application, we should use the Spring Boot actuator.

There are three main features of Spring Boot Actuator:

- Endpoints
- Metrics
- Audit

Spring Boot Actuator

Endpoint: The actuator endpoints allows us to monitor and interact with the application. Spring Boot provides a number of built-in endpoints. We can also create our own endpoint. We can enable and disable each endpoint individually. Most of the application choose HTTP, where the Id of the endpoint, along with the prefix of /actuator, is mapped to a URL.

For example, the /health endpoint provides the basic health information of an application. The actuator, by default, mapped it to /actuator/health.

Metrics: Spring Boot Actuator provides dimensional metrics by integrating with the micrometer. The micrometer is integrated into Spring Boot. It is the instrumentation library powering the delivery of application metrics from Spring. It provides vendor-neutral interfaces for timers, gauges, counters, distribution summaries, and long task timers with a dimensional data model.

Audit: Spring Boot provides a flexible audit framework that publishes events to an AuditEventRepository. It automatically publishes the authentication events if spring-security is in execution.

Spring Boot Actuator

Sample Endpoints and Usage:

ENDPOINTS	USAGE
/metrics	To view the application metrics such as memory used, memory free, threads, classes, system uptime etc.
/env	To view the list of Environment variables used in the application.
/beans	To view the Spring beans and its types, scopes and dependency.
/health	To view the application health
/info	To view the information about the Spring Boot application.
/trace	To view the list of Traces of your Rest endpoints.

Spring Boot Actuator Endpoints

The actuator endpoints allow us to monitor and interact with our Spring Boot application.

Id	Usage	Default
actuator	It provides a hypermedia-based **discovery page** for the other endpoints. It requires Spring HATEOAS to be on the classpath.	True
auditevents	It exposes audit events information for the current application.	True
autoconfig	It is used to display an auto-configuration report showing all auto-configuration candidates and the reason why they 'were' or 'were not' applied.	True
beans	It is used to display a complete list of all the Spring beans in your application.	True
configprops	It is used to display a collated list of all @ConfigurationProperties.	True
dump	It is used to perform a thread dump.	True
env	It is used to expose properties from Spring's ConfigurableEnvironment.	True
flyway	It is used to show any Flyway database migrations that have been applied.	True
health	It is used to show application health information.	False
info	It is used to display arbitrary application info.	False
loggers	It is used to show and modify the configuration of loggers in the application.	True
liquibase	It is used to show any Liquibase database migrations that have been applied.	True
metrics	It is used to show metrics information for the current application.	True
mappings	It is used to display a collated list of all @RequestMapping paths.	True
shutdown	It is used to allow the application to be gracefully shutdown.	True
trace	It is used to display trace information.	True

Spring Boot actuator properties

Spring Boot enables security for all actuator endpoints. It uses form-based authentication that provides user Id as the user and a randomly generated password. We can also access actuator-restricted endpoints by customizing basicauth security to the endpoints. We need to override this configuration by management.security.roles property.

```
management.security.enabled=true

management.security.roles=ADMIN

security.basic.enabled=true

security.user.name=admin

security.user.passowrd=admin

management.endpoints.web.exposure.include=*
```

Spring Boot actuator

To add the actuator to a Maven based project, add the following ‘Starter’ dependency:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
</dependencies>
```

For Gradle, use the following declaration:

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-actuator'
}
```

Run method on Spring Boot startup

While developing a spring boot application, sometimes we need to run a method or a piece of code at startup. This code can be anything ranging from, logging certain information to setting up database, Cron jobs etc. We cannot just put this code in constructor, because required variables or services may not be initialized yet. This could lead to null pointers or some other exceptions

Why do we need to run code at spring boot startup?

We need to run method at application startup for many reasons like,

- Logging important things or message saying application is started
- Processing database or files, indexing, creating caches etc.
- Starting background process like sending notification, fetching data form some queue, etc.

Different ways to run method after startup in spring boot

1. Using CommandLineRunner interface

2. With ApplicationRunner interface

3. Spring boot Application events

4. @Postconstruct annotation on a method

5. The InitializingBean Interface

6. Init attribute of @bean annotation

Spring boot Runners

Spring Boot provides two runner interfaces, which are ApplicationRunner and CommandLineRunner. Both of these runners are used to execute piece of code when a Spring Boot Application starts. Both of these interfaces are Functional Interfaces, which means they have only one functional method. In order to execute specific piece of code when Spring Boot Application starts, we need to implement either of these functional interfaces and override the single method of run.

Spring boot CommandLine Runner

CommandLineRunner is a spring boot functional interface which is used to run code at application startup. It is present under package org.springframework.boot.

In startup process after the context is initialized, spring boot calls its run() method with command-line arguments provided to the application.

To inform spring boot about our commandlineRunner interface, we can either implement it and add @Component annotation above the class or create its bean using @bean.

Note: Spring Boot adds CommandLineRunner interface into the startup process. Hence throwing exception in commandlinerRunner will force Spring boot to abort startup.

Spring boot CommandLine Runner

```
@SpringBootApplication
public class Application {
        public static void main(String[] args) {
        SpringApplication.run(Application.class);
        }

        @Bean
        public CommandLineRunner CommandLineRunnerBean() {
                return (args) -> {
                        System.out.println("In CommandLineRunnerImpl ");
                        for (String arg : args) {
                                System.out.println(arg);
                        }
                };
            }
}
```

Spring boot Application Runner

ApplicationRunner interface provides run method with ApplicationArguments instead of raw string array. ApplicationArguments is an interface which is available from srping boot 1.3 under the package org.springframework.boot.

It provides different ways to access the arguments as below:

String[] GetSourceArgs()	Gives unprocessed arguments that were passed to the application
Set<String> getOptionNames()	Names of all optional arguments, optional arguments are preceded by **"—"** eg: –name= "stacktrace"
List<String> getNonOptionArgs()	Returns unprocessed non-optional arguments. Arguments without **"—"**
boolean containsOption(String name)	Checks if name is present in the optional arguments or not
List<String> getOptionValues(String name)	Gives the argument value by name

Spring boot Application Runner

```
@Component
public class ApplicationRunnerImpl implements ApplicationRunner {

   @Override
   public void run(ApplicationArguments args) throws Exception {

      System.out.println("ApplicationRunnerImpl Called");

//print all arguemnts: arg: status=running, arg: --mood=happy, 10, --20
      for (String arg : args.getSourceArgs()) {
         System.out.println("arg: "+arg);
      }
      System.out.println("NonOptionArgs: "+args.getNonOptionArgs()); //[status=running,10]
      System.out.println("OptionNames: "+args.getOptionNames());  //[mood, 20]

     System.out.println("Printing key and value in loop:");
      for (String key : args.getOptionNames()) {
         System.out.println("key: "+key);     //key: mood  //key: 20
         System.out.println("val: "+args.getOptionValues(key)); //val:[happy] //val:[]
      }
   }
}
```

Spring boot Application Runner

Output:

```
ApplicationRunnerImpl Called
arg: status=running
arg: --mood=happ
arg: 10
arg: --20
NonOptionArgs: [status=running , 10]
OptionNames: [mood, 20]
Printing key and value in loop:
key: mood
val: [happy]
key: 20
val: []
```

Spring boot Runners

CommandLineRunner and ApplicationRunner have similar features like

- An exception in the run() method will abort application startup
- Several ApplicationRunners can be ordered using Ordered interface or **@Order annotation**
- Most important point to note that the Order is shared between CommandLineRunners and ApplicationRunners. That means the execution order could be mixed between commandlinerRunner and applicationRunner.

Spring boot Runners examples

CommandLineRunner and ApplicationRunner Ordered: (@See – SpringApplication.callRunners())

```
@Order(1)
@Component
public class CommandLineRunnerImpl implements CommandLineRunner {
    @Override
    public void run(String... args) throws Exception {

@Order(2)
@Component
public class ApplicationRunnerImpl implements ApplicationRunner
@Override
    public void run(ApplicationArguments args) throws Exception {
```

Application event in Spring Boot

Spring framework triggers different events in different situations. It also triggers many events in startup process. We can use these events to execute our code, for example ApplicationReadyEvent can be used to execute code after spring boot application starts up.

```
@Component
public class RunAfterStartup{

@EventListener(ApplicationReadyEvent.class)
public void runAfterStartup() {
    System.out.println("Yaaah, I am running........");
}
```

@Postconstruct annotation

A method can be marked with @PostConstruct annotation. Whenever a method is marked with this annotation, it will be called immediately after the dependency injection.

```
@Component
public class PostContructImpl {
		public PostContructImpl() {
				System.out.println("PostContructImpl Constructor called");
		}
		@PostConstruct
		public void runAfterObjectCreated() {
				System.out.println("PostContruct method called");
		}
}
```

Note: Please note that @postConstruct annotation is part of Java EE module and it is marked as deprecated in Java 9 and removed in Java 11. We can still use it by adding java.se.ee into the application.

The InitializingBean Interface

The InitializingBean solution works exactly the similar to the postConstruct annotation. Instead of using annotation we have to implement an InitializingBean interface. Then we need to override void afterPropertiesSet() method.

```
@Component
public class InitializingBeanImpl implements InitializingBean {
    public InitializingBeanImpl() {
        System.out.println("InitializingBeanImpl Constructor called");
    }
    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("InitializingBeanImpl afterPropertiesSet method called");
    }
}
```

Init attribute of @bean annotation

```
@SpringBootApplication
public class ApplicationEventExampleApplication {

public static void main(String[] args) {
SpringApplication.run(ApplicationEventExampleApplication.class, args);
}

@Bean(initMethod="init")
    public BeanInitMethodImpl beanInitMethodImpl() {
         System.out.println("Bean Init called.");
        return new BeanInitMethodImpl();
    }
}
```

Execution flow of Startup methods

- Constructor
- afterPropertiesSet method (InitializingBean)
- Bean init Method
- ApplicationStartedEvent
- ApplicationRunner Or CommandLineRunner depends on Order
- ApplicationReadyEvent

Spring Boot Profile

- You can use **Spring Boot Profile** functionality when you need to provide application configuration for a specific environment. For example, when your application runs in a development environment, then your application will use a specific for that environment application.properties file. And when you are ready to move your application to a production environment, it will use a different application.properties file which is specific to production environment. Additionally to using a specific to an environment properties file, you can also use Spring Boot Profiles to configure your application to access specific to an environment Java Beans.

Spring Boot Profile

Profiles are a core feature of the framework — allowing us to map our beans to different profiles — for example, dev, test, and prod.

- **Use @Profile on a Bean:**

We use the @Profile annotation — we are mapping the bean to that particular profile; the annotation simply takes the names of one (or multiple) profiles.

Consider a basic scenario: We have a bean that should only be active during development but not deployed in production.

We annotate that bean with a dev profile, and it will only be present in the container during development. In production, the dev simply won't be active:

```
@Component

@Profile("dev")

public class DevDatasourceConfig
```

Spring Boot Profile

As a quick sidenote, profile names can also be prefixed with a NOT operator, e.g., !dev, to exclude them from a profile.

In the example, the component is activated only if dev profile is not active:

```
@Component
@Profile("!dev")
public class DevDatasourceConfig
```

Spring Boot Profile

Set Profiles:

Programmatically via WebApplicationInitializer Interface:

In web applications, WebApplicationInitializer can be used to configure the ServletContext programmatically.

It's also a very handy location to set our active profiles programmatically:

```
@Configuration
public class MyWebApplicationInitializer
  implements WebApplicationInitializer {
    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        servletContext.setInitParameter(
          "spring.profiles.active", "dev");
    }
}
```

Spring Boot Profile

Set Profiles:

JVM System Parameter:

The profile names can also be passed in via a JVM system parameter. These profiles will be activated during application startup:

-Dspring.profiles.active=dev

Environment Variable:

In a Unix environment, profiles can also be activated via the environment variable:

export spring_profiles_active=dev

Spring Boot Profile

Set Profiles:

Maven Profile:

Spring profiles can also be activated via Maven profiles, by specifying the spring.profiles.active configuration property.

```
<profiles>
    <profile>
        <id>dev</id>
        <activation>
            <activeByDefault>true</activeByDefault>
        </activation>
        <properties>
            <spring.profiles.active>dev</spring.profiles.active>
        </properties>
    </profile>
</profiles>
```

Spring Boot Profile

Set Profiles: Maven Profile:

Its value will be used to replace the @spring.profiles.active@ placeholder in application.properties:
spring.profiles.active=@spring.profiles.active@
Now we need to enable resource filtering in *pom.xml*:

```
<build>
    <resources>
        <resource>
            <directory>src/main/resources</directory>
            <filtering>true</filtering>
        </resource>
    </resources>
    ...
</build>
```

and append a *-P* parameter to switch which Maven profile will be applied:

```
mvn clean package –Pprod
```

This command will package the application for prod profile. It also applies the spring.profiles.active value prod for this application when it is running.

Spring Boot Profile

Set Profiles:

@ActiveProfile in Tests

Tests make it very easy to specify what profiles are active using the @ActiveProfile annotation to enable specific profiles:
@ActiveProfiles("dev")

The Default Profile

Any bean that does not specify a profile belongs to the default profile.

Spring also provides a way to set the default profile when no other profile is active — by using the spring.profiles.default property.

Spring Boot Profile

Get Active Profiles:

Spring's active profiles drive the behavior of the @Profile annotation for enabling/disabling beans.

using Environment:

```
public class ProfileManager {
    @Autowired
    private Environment environment;
    public void getActiveProfiles() {
        for (String profileName : environment.getActiveProfiles()) {
            System.out.println("Currently active profile - " + profileName);
        }   } }
```

spring.active.profile:

```
@Value("${spring.profiles.active}")
private String activeProfile;
```

Spring Boot Profile

Separate Data Source Configurations Using Profiles

Spring's active profiles drive the behavior of the @Profile annotation for enabling/disabling beans.

```
@Component
@Profile("dev")
public class DevDatasourceConfig implements DatasourceConfig {
    @Override
    public void setup() {
        System.out.println("Setting up datasource for DEV environment. ");
    }
}
```

Spring Boot Profile

Profiles in Spring Boot:

Spring Boot supports all the profile configuration outlined so far, with a few additional features:

Activating or Setting a Profile:

```
spring.profiles.active=dev
```

Profile-specific Properties Files:

```
spring.profiles.active=dev
```

However, the most important profiles-related feature that Spring Boot brings is profile-specific properties files. These have to be named in the format application-{profile}.properties.

Spring Boot will automatically load the properties in an application.properties file for all profiles, and the ones in profile-specific .properties files only for the specified profile.

Spring Boot Profile

For example, we can configure different data sources for dev and production profiles by using two files named application-dev.properties and application-production.properties:

In the **application-production.properties** file, we can set up a MySql data source:

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/db
spring.datasource.username=root
spring.datasource.password=root
```

Then we can configure the same properties for the dev profile in **the application-dev.properties** file, to use an in-memory H2 database:

```
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.url=jdbc:h2:mem:db;DB_CLOSE_DELAY=-1
spring.datasource.username=sa
spring.datasource.password=sa
```

Spring Boot logging

Spring boot logging with application.properties:

spring boot logging configuration via application.properties file in simple and easy to follow instructions. In the default structure of a Spring Boot web application, we can locate the application.properties file under the resources folder.

application.properties

```
logging.level.org.springframework=DEBUG

#output to a file

logging.file.name=application.log

 # Logging pattern for the console

logging.pattern.console= %d{yyyy-MM-dd HH:mm:ss} - %msg%n

 # Logging pattern for file

logging.pattern.file=%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n
```

Spring boot default logging

Let’s first understand the log:

```
2017-03-02 23:33:51.318  INFO 3060 --- [nio-8080-exec-1] c.h.app.controller.IndexController       : info log statement printed
2017-03-02 23:33:51.319  WARN 3060 --- [nio-8080-exec-1] c.h.app.controller.IndexController       : warn log statement printed
2017-03-02 23:33:51.319 ERROR 3060 --- [nio-8080-exec-1] c.h.app.controller.IndexController       : error log statement printed
```

Note down the observation that Default logging level is INFO – because debug log message is not present.

There is fixed default log message pattern which is configured in different base configuration files.

Spring boot default logging

```
%clr{%d{yyyy-MM-dd HH:mm:ss.SSS}}{faint} %clr{${LOG_LEVEL_PATTERN}} %clr{${sys:PID}}{magenta}

%clr{---}{faint} %clr{[%15.15t]}{faint} %clr{%-40.40c{1.}}{cyan} %clr{:}{faint}
%m%n${sys:LOG_EXCEPTION_CONVERSION_WORD}
```

The above pattern print these listed log message parts with respective color coding applied:

- Date and Time — Millisecond precision.
- 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

Spring boot default logging

Spring boot logging levels:

When a message is logged via a Logger it is logged with a certain log level. In the application.properties file, you can define log levels of Spring Boot loggers, application loggers, Hibernate loggers, Thymeleaf loggers, and more. To set the logging level for any logger, add properties starting with logging.level.

Logging level can be one of one of TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF. The root logger can be configured using logging.level.root.

```
#logging.level.root=WARN

logging.level.org.springframework.web=ERROR

logging.level.com.amit=DEBUG
```

Spring boot default logging

Spring boot logging levels:

In above configuration, I upgraded log level for application classes to DEBUG (from default INFO). Now observe the logs:

```
2017-03-02 23:57:14.966 DEBUG 4092 --- [nio-8080-exec-1] c.h.app.controller.IndexController       : debug log statement printed
2017-03-02 23:57:14.967  INFO 4092 --- [nio-8080-exec-1] c.h.app.controller.IndexController       : info log statement printed
2017-03-02 23:57:14.967  WARN 4092 --- [nio-8080-exec-1] c.h.app.controller.IndexController       : warn log statement printed
2017-03-02 23:57:14.967 ERROR 4092 --- [nio-8080-exec-1] c.h.app.controller.IndexController       : error log statement printed
```

Spring boot logging patterns

To change the logging patterns, use logging.pattern.console and logging.pattern.file properties.

# Logging pattern for the console

```
logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} - %msg%n
```

# Logging pattern for file

```
logging.pattern.file=%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n
```

After changing console logging pattern in application, log statements are printed as below:

```
2017-03-03 12:59:13 - This is a debug message
2017-03-03 12:59:13 - This is an info message
2017-03-03 12:59:13 - This is a warn message
2017-03-03 12:59:13 - This is an error message
```

Spring boot logging patterns

Log output to file

To print the logs in file, use logging.file or logging.path property.

logging.file=c:/users/application-debug.log

Verify the logs in file.

```
2017-03-03 13:02:50.608 DEBUG 10424 --- [http-nio-8080-exec-1] c.h.app.controller.IndexController       : This is a debug message
2017-03-03 13:02:50.608  INFO 10424 --- [http-nio-8080-exec-1] c.h.app.controller.IndexController       : This is an info message
2017-03-03 13:02:50.608  WARN 10424 --- [http-nio-8080-exec-1] c.h.app.controller.IndexController       : This is a warn message
2017-03-03 13:02:50.609 ERROR 10424 --- [http-nio-8080-exec-1] c.h.app.controller.IndexController       : This is an error message
```

Spring boot logging patterns

Active profiles to load environment specific logging configuration

It is desirable to have multiple configurations for any application – where each configuration is specific to a particular runtime environment. In spring boot, you can achieve this by creating multiple application-{profile}.properties files in same location as application.properties file.

Profile specific properties always override the non-profile specific ones. If several profiles are specified, a last wins strategy applies.

If I have two environments for my application i.e. prod and dev. Then I will create two profile specific properties files.

Spring boot logging patterns

application-dev.properties

```
logging.level.com.amit=DEBUG
logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} - %msg%n
```

5.2. application-prod.properties

```
logging.level.com.amit=ERROR
logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} - %msg%n
```

To supply profile information to application, property spring.profiles.active is passed to runtime.

```
$ java -jar -Dspring.profiles.active=prod spring-boot-demo.jar
```

Spring boot logging patterns

Color-coded log output

If your terminal supports ANSI, color output will be used to aid readability. You can set spring.output.ansi.enabled value to either ALWAYS, NEVER or DETECT.

Color coding is configured using the %clr conversion word. In its simplest form the converter will color the output according to the log level.

FATAL and ERROR – Red

WARN – Yellow

INFO, DEBUG and TRACE – Green

Spring boot logging with Lombok

Project Lombok is very handy tool for removing the boilerplate code from application. Lombok can also be used to configure logging in spring boot applications and thus removing the boilerplate code for getting the logger instance.

@Log, @Log4j2, @Slf4j – Lombok annotations:

Instead, we can start writing log statements in a java class which is annotated with lombok's @Log annotations. Lombok supports following log annotations for spring boot –

@CommonsLog – Creates the logger with following statement:

```
private static final org.apache.commons.logging.Log log =
    org.apache.commons.logging.LogFactory.getLog(LogExample.class);
```

@Log – Creates the logger with following statement:

```
private static final java.util.logging.Logger log =
    java.util.logging.Logger.getLogger(LogExample.class.getName());
```

Spring boot logging with Lombok

@Log4j2 – Creates the logger with following statement:

private static final org.apache.logging.log4j.Logger log =

org.apache.logging.log4j.LogManager.getLogger(LogExample.class);

@Slf4j – Creates the logger with following statement:

Creates private static final org.slf4j.Logger log =

org.slf4j.LoggerFactory.getLogger(LogExample.class);

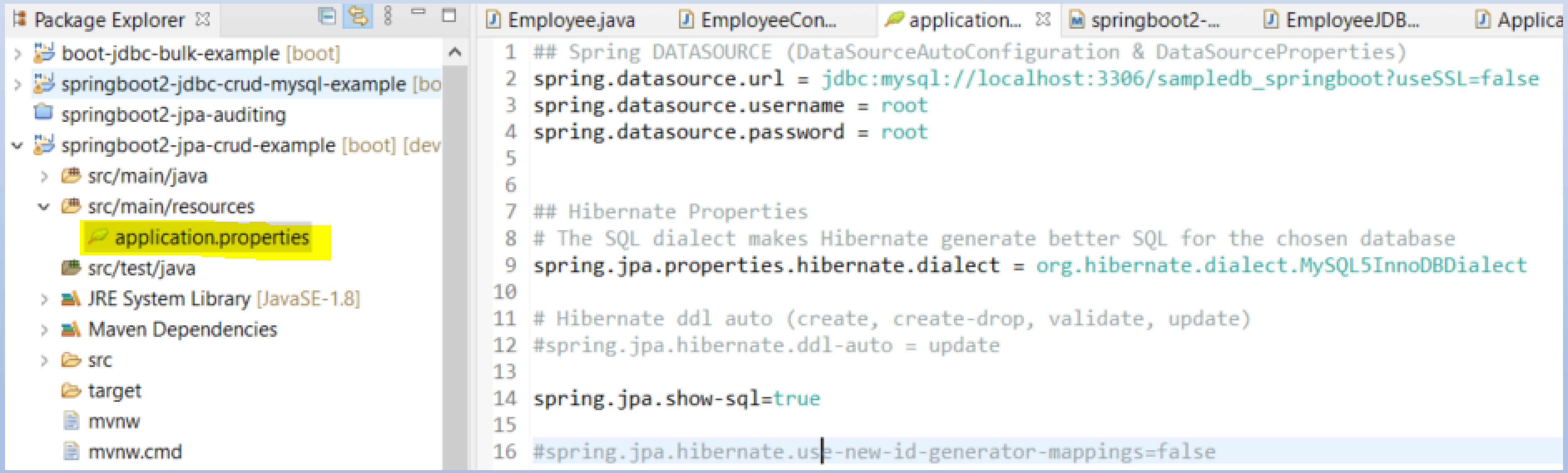
Spring boot logging with Lombok

```
@Slf4j
@SpringBootApplication
public class Application
{
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
          log.info("Simple log statement with inputs {}, {} and {}", 1, 2, 3);
    }
}
```

By default, spring boot uses logback as logging provider.

Spring Boot Application Properties

Spring Boot Framework comes with a built-in mechanism for application configuration using a file called **application.properties**. It is located inside the **src/main/resources** folder:



```
## Spring DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
spring.datasource.url = jdbc:mysql://localhost:3306/sampledb_springboot?useSSL=false
spring.datasource.username = root
spring.datasource.password = root


## Hibernate Properties
# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect

# Hibernate ddl auto (create, create-drop, validate, update)
#spring.jpa.hibernate.ddl-auto = update

spring.jpa.show-sql=true

#spring.jpa.hibernate.use-new-id-generator-mappings=false
```

Spring Boot Application Properties

Spring Boot provides various properties that can be configured in the **application.properties** file. The properties have default values. We can set a property(s) for the Spring Boot application. Spring Boot also allows us to define our own property if required.

The **application.properties** file allows us to run an application in a **different environment.** In short, we can use the application.properties file to:

- Configure the Spring Boot framework
- define our application custom configuration properties

Example: application.properties:

```
#configuring application name
spring.application.name = demoApplication
#configuring port
server.port = 8081
```

In the above example, we have configured the **application name** and **port**. The port 8081 denotes that the application runs on port **8081**.

Spring Boot Application Properties

YAML Properties File

Spring Boot provides another file to configure the properties is called **yml** file. The Yaml file works because the **Snake YAML** jar is present in the classpath. Instead of using the application.properties file, we can also use the application.yml file, but the **Yml** file should be present in the classpath.

Example of application.yml

```yaml
spring:
  application:
    name: demoApplication
server:
  port: 8081
```

In the above example, we have configured the **application name** and **port**. The port 8081 denotes that the application runs on port **8081**.

Spring Boot Application Properties

Spring Boot Property Categories

There are **sixteen** categories of Spring Boot Property are as follows:

1. Core Properties
2. Cache Properties
3. Mail Properties
4. JSON Properties
5. Data Properties
6. Transaction Properties
7. Data Migration Properties
8. Integration Properties
9. Web Properties
10. Templating Properties
11. Server Properties
12. Security Properties
13. RSocket Properties
14. Actuator Properties

Spring Boot Application Properties

Spring Boot Property Categories

There are **sixteen** categories of Spring Boot Property are as follows:

1. Core Properties
2. Cache Properties
3. Mail Properties
4. JSON Properties
5. Data Properties
6. Transaction Properties
7. Data Migration Properties
8. Integration Properties
9. Web Properties
10. Templating Properties
11. Server Properties
12. Security Properties
13. RSocket Properties
14. Actuator Properties

Spring Boot Application Properties

Application Properties Table

The following tables provide a list of common Spring Boot properties:

Property	Default Values	Description
Debug	false	It enables debug logs.
spring.application.name		It is used to set the application name.
spring.application.admin.enabled	false	It is used to enable admin features of the application.
spring.config.name	application	It is used to set config file name.
spring.config.location		It is used to config the file name.
server.port	8080	Configures the HTTP server port
server.servlet.context-path		It configures the context path of the application.
logging.file.path		It configures the location of the log file.
spring.banner.charset	UTF-8	Banner file encoding.
spring.banner.location	classpath:banner.txt	It is used to set banner file location.
logging.file.name		It is used to set log file name. For example, data.log.
spring.application.index		It is used to set application index.

Spring Boot Application Properties

Application Properties Table

Property	Default Values	Description
spring.config.location		It is used to config the file locations.
spring.config.name	application	It is used to set config the file name.
spring.mail.default-encoding	UTF-8	It is used to set default MimeMessage encoding.
spring.mail.host		It is used to set SMTP server host. For example, smtp.example.com.
spring.mail.password		It is used to set login password of the SMTP server.
spring.mail.port		It is used to set SMTP server port.
spring.mail.test-connection	false	It is used to test that the mail server is available on startup.
spring.mail.username		It is used to set login user of the SMTP server.
spring.main.sources		It is used to set sources for the application.
server.address		It is used to set network address to which the server should bind to.
server.connection-timeout		It is used to set time in milliseconds that connectors will wait for another HTTP request before closing the connection.
server.context-path		It is used to set context path of the application.
server.port	8080	It is used to set HTTP port.
server.server-header		It is used for the Server response header (no header is sent if empty)
server.servlet-path	/	It is used to set path of the main dispatcher servlet
server.ssl.enabled		It is used to enable SSL support.
spring.http.multipart.enabled	True	It is used to enable support of multi-part uploads.
spring.servlet.multipart.max-file-size	1MB	It is used to set max file size.

Spring Boot Application Properties

Application Properties Table

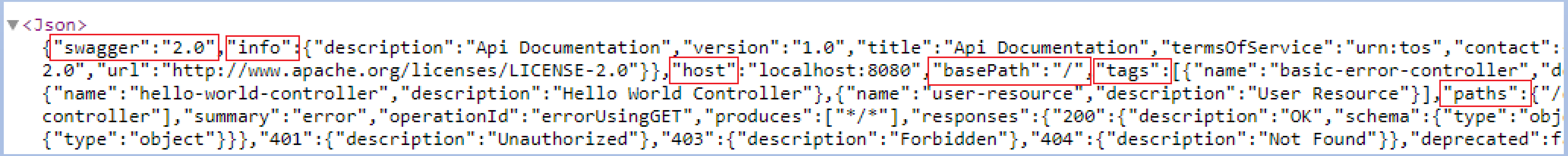
Property	Default Values	Description
spring.mvc.async.request-timeout		It is used to set time in milliseconds.
spring.mvc.date-format		It is used to set date format. For example, dd/MM/yyyy.
spring.mvc.locale		It is used to set locale for the application.
spring.social.facebook.app-id		It is used to set application's Facebook App ID.
spring.social.linkedin.app-id		It is used to set application's LinkedIn App ID.
spring.social.twitter.app-id		It is used to set application's Twitter App ID.
security.basic.authorize-mode	role	It is used to set security authorize mode to apply.
security.basic.enabled	true	It is used to enable basic authentication.
Spring.test.database.replace	any	Type of existing DataSource to replace.
Spring.test.mockmvc.print	default	MVC Print option
spring.freemaker.content-type	text/html	Content Type value
server.server-header		Value to use for the server response header.
spring.security.filter.dispatcher-type	async, error, request	Security filter chain dispatcher types.
spring.security.filter.order	-100	Security filter chain order.
spring.security.oauth2.client.registration.*		OAuth client registrations.
spring.security.oauth2.client.provider.*		OAuth provider details.

Swagger

Swagger is a specification for documenting REST API. It specifies the format (URL, method, and representation) to describe REST web services. It also provides tools to generate/compute the documentation from the application code.

As an application developer, we write web services using a framework, Swagger scans application code, and exposes the documentation on URL. A client can consume this URL and learn how to use REST web services: which HTTP methods to call on which URL, which input documents to send, which status code to expect, etc.

We're going to see what is inside the Swagger documentation. When we have a close look at the documentation of the web API, we see some important elements in the starting of the documentation, as shown in the following image.



```
▼<Json>
   {"swagger":"2.0","info":{"description":"Api Documentation","version":"1.0","title":"Api Documentation","termsOfService":"urn:tos","contact":
   2.0","url":"http://www.apache.org/licenses/LICENSE-2.0"}},"host":"localhost:8080","basePath":"/","tags":[{"name":"basic-error-controller","d
   {"name":"hello-world-controller","description":"Hello World Controller"},{"name":"user-resource","description":"User Resource"}],"paths":{"/
   controller"],"summary":"error","operationId":"errorUsingGET","produces":["*/*"],"responses":{"200":{"description":"OK","schema":{"type":"obj
   {"type":"object"}}},"401":{"description":"Unauthorized"},"403":{"description":"Forbidden"},"404":{"description":"Not Found"}},"deprecated":f
```

Swagger

There are following important swagger elements that are present in the Swagger documentation.

swagger: It specifies the version specification of Swagger, which we are using.

info: The info tab contains the information about API like description, version of API, the title of API, termOfServices, and URL.

host: It specifies the host where we are hosting the service.

basePath: It is used in URI after the port number and before the API.

tags: We can assign tags to our resources. It is used to group resources in multiple categories.

paths: It specifies the path of resources that we are exposing and the different operations that can be performed on these resources.

definitions: It includes the different elements that we have used in our API.

We will discuss three elements info, paths, and definitions in detail.

Swagger

Let's expand the path element. It contains all the path that we are exposing.

```
paths: {
/error: {-}
/hello-world: {-}
/hello-world-bean: {-}
/hello-world-internationalized: {-}
/hello-world/path-variable/{name}: {-}
/users: {-}
/users/{id}: {-}
},
```

The two most important resources are "**/users**" and "**/users/{id}**". These resources exposes the group of users. Let's expand these two resources one by one.

Swagger

Expanding "/users" resource:

"/users":{"get":{"tags":["user-resource"],"summary":"retriveAllUsers","operationId":"retriveAllUsersUsingGET","produces":["*/*"],"responses":{"200":{"description":"OK","schema":{"type":"array","items":{"$ref":"#/definitions/User"}}},"401":{"description":"Unauthorized"},"403":{"description":"Forbidden"},"404":{"description":"Not Found"}},"deprecated":**false**},"post":{"tags":["user-resource"],"summary":"createUser","operationId":"createUserUsingPOST","consumes":["application/json"],"produces":["*/*"],"parameters":[{"in":"body","name":"user","description":"user","required":**true**,"schema":{"$ref":"#/definitions/User"}}],"responses":{"200":{"description":"OK","schema":{"type":"object"}},"201":{"description":"Created"},"401":{"description":"Unauthorized"},"403":{"description":"Forbidden"},"404":{"description":"Not Found"}},"deprecated":**false**}},

The above resource contains the two operations **get** and **post** that can be performed. We can use get operation to retrieve all the users and post operation to post a user.

Inside the get operation, we get all the response status present there. Response status 200 denotes the successful creation of a user, 401 denotes the unauthorized access of resources, 404 denotes not found, and 403 denotes the forbidden. When we look at the status 200, there is a schema definition. Schema definition shows that we are sending an array of the user as a response. An array of the user is present in the definitions. Similarly, we can also expand the definitions tag to see the definition of the user.

In addition to a POST request, we have parameters that send as part of the body of the request. We accept an input type **user** as the body of the request.

Swagger

Expand "/users/{id}" resource:

"/users/{id}":{"get":{"tags":["user-
resource"],"summary":"retriveUser","operationId":"retriveUserUsingGET","produces":["*/*"],"parameters":[{"name":"id","in":"path
","description":"id","required":true,"type":"integer","format":"int32"}],"responses":{"200":{"description":"OK","schema":{"$ref":"#
/definitions/Resource«User»"}},"401":{"description":"Unauthorized"},"403":{"description":"Forbidden"},"404":{"description":"Not
Found"}},"deprecated":false},"delete":{"tags":["user-
resource"],"summary":"deleteUser","operationId":"deleteUserUsingDELETE","produces":["*/*"],"parameters":[{"name":"id","in":"p
ath","description":"id","required":true,"type":"integer","format":"int32"}],"responses":{"200":{"description":"OK"},"204":{"descrip
tion":"No Content"},"401":{"description":"Unauthorized"},"403":{"description":"Forbidden"}},"deprecated":false}}},

The /users/{id} resource allows two operations **get** and **delete**. Inside the delete method, there is a parameter called id. This id we are accepting in the path while in the post request, we put content as a part of the body of the request.

Definition defines different kinds of objects that are being used. These definitions are used in the different operations exposed by each resource. When we perform get operation on /users, it returns a list of users. This resource of the user sending back to get the operation of the resource **/user/{id}** and the resource of the user contains the additional links. The definition of links is also present in the resource of user type.

Links contains a rel and href. A rel is all **-users**, and href is the link to a particular resource.

Swagger

There are two ways to expose documentation to the client:

- Download the documentation from http://localhost:8080/v2/api-docs as JSON and send it to clients.
- Share the link of Swagger UI http://localhost:8080/swagger-ui.html. It is a UI that describes all the operations that are ready to expose.

Swagger

```
"info":{"description":"Api Documentation","version":"1.0","title":"Api Documentation","termsOfService":"urn:tos","contact":{},"license":{"name":"Apache 2.0","url":"http://www.apache.org/licenses/LICENSE-2.0"}},
```

- **description:** It contains the high level information of API.
- **version:** It shows the version of API which we are exposing.
- **title:** It specifies the title of API.
- **termOfService:** It specifies the term of service, if any.
- **contact:** It specifies the contact detail of a person, if any.
- **license:** It specifies the default license Apache 2.0.

Spring boot i18n

Internationalization is the process of designing web applications or services in such a way that it can provide support for various countries, various languages automatically without making the changes in the application. It is also known as I18N because the word internationalization has total 18 characters starting from I to N.

Localization is performed by adding locale-specific components such as translated text, data describing locale-specific behavior, etc. It supports full integration into the classes and packages that provide language-or-culture-dependent functionality.

Spring boot i18n

Java provides the foundation for internationalization for desktop and server applications. There are following important internationalized areas of functionality.

Text Representation: Java is based on the Unicode character set, and several libraries implement the Unicode standard.

Locale identification and localization: Locale in Java are identifiers that can be used to request locale-specific behavior in different areas of functionality.

Date and time handling: Java provides various calendars. It supports conversion to and from calendar independent Date objects. Java supports all the time zones in the world.

Text processing: It includes character analysis, case mapping, string comparison, breaking text into words, formatting numbers, dates, and time values into strings or parsing them back from strings. Most of these functions are locale-dependent.

Character encoding: It supports converting text between Unicode and other character encodings when reading incoming text from the streams or writing outgoing text to the streams.

Spring boot i18n

spring.http.encoding.charset=UTF-8 # Charset of HTTP requests and responses. Added to the "Content-Type" header if not set explicitly.

spring.http.encoding.enabled=true # Enable http encoding support.

spring.http.encoding.force= # Force the encoding to the configured charset on HTTP requests and responses.

spring.http.encoding.force-request= # Force the encoding to the configured charset on HTTP requests. Defaults to true when "force" has not been specified.

spring.http.encoding.force-response= # Force the encoding to the configured charset on HTTP responses

i18N

When developing a web application, we tend to code it using a collection of the most efficient, the most popular, and the most sought-after programming languages for both our front end and back end. But what about spoken languages? Most of the time, with or without our knowledge, we depend on the built-in translation engines of our customers’ browsers to handle the required translations.

In the ever-globalizing world we live in, we need our web applications to reach as wide an audience as possible. Here enters the much-required concept of internationalization. In this article, we will be looking at how i18n works on the popular Spring Boot framework.

i18N

- I18n on Spring Boot.
- MessageSource interface and its uses.
- Locale resolving through LocaleResolver, LocaleChangeInterceptor classes.
- Storing the user-preferred locale in cookies.
- Switching between languages.
- Interpolation using placeholders.
- Pluralization with the help of ICU4J standards.
- Date-time localization using @DateTimeFormat annotation.

i18N

LocaleResolver

The LocaleResolver interface deals with locale resolution required when localizing web applications to specific locales. Spring aptly ships with a few LocaleResolver implementations that may come in handy in various scenarios:

FixedLocaleResolver

Always resolves the locale to a singular fixed language mentioned in the project properties. Mostly used for debugging purposes.

AcceptHeaderLocaleResolver

Resolves the locale using an “accept-language” HTTP header retrieved from an HTTP request.

SessionLocaleResolver

Resolves the locale and stores it in the HttpSession of the user. But as you might have wondered, yes, the resolved locale data is persisted only for as long as the session is live.

i18N

CookieLocaleResolver

Resolves the locale and stores it in a cookie stored on the user’s machine. Now, as long as browser cookies aren’t cleared by the user, once resolved the resolved locale data will last even between sessions. Cookies save the day!

Use CookieLocaleResolver

Let’s see how we can use CookieLocaleResolver in our java-i18n-spring-boot application. Simply add a LocaleResolver bean within the JavaI18nSpringBootApplication class annotated with @SpringBootApplication and set a default locale. For instance:

```
@Bean // <--- 1
public LocaleResolver localeResolver() {
    CookieLocaleResolver localeResolver = new CookieLocaleResolver(); // <--- 2
    localeResolver.setDefaultLocale(Locale.US); // <--- 3
    return localeResolver;
}
```

i18N

Bean annotation is added to mark this method as a Spring bean.

LocaleResolver interface is implemented using Spring’s built-in CookieLocaleResolver implementation.

The default locale is set for this locale resolver to return in the case that no cookie is found.

Add LocaleChangeInterceptor

Okay, now our application knows how to resolve and store locales. However, when users from different locales visit our app, who’s going to switch the application’s locale accordingly? Or in other words, how do we localize our web application to the specific locales it supports?

For this, we’ll add an interceptor – or interceptor? – bean that will intercept each request that the application receives, and eagerly check for a localeData parameter on the HTTP request. If found, the interceptor uses the localeResolver we coded earlier to register the locale it found as the current user’s locale. Let’s add this bean within the JavaI18nSpringBootApplication class:

i18N

```
@Bean
public LocaleChangeInterceptor localeChangeInterceptor() {
    LocaleChangeInterceptor localeChangeInterceptor = new LocaleChangeInterceptor();
    // Defaults to "locale" if not set
    localeChangeInterceptor.setParamName("localeData");
    return localeChangeInterceptor;
}
```

i18N

Now, to make sure this interceptor properly intercepts all incoming requests, we should add it to the Spring InterceptorRegistry:

1. Set the main class in your project, which is the JavaI18nSpringBootApplication class annotated with @SpringBootApplication, to implement WebMvcConfigurer, like so:

```
@SpringBootApplication

public class JavaI18nSpringBootApplication implements WebMvcConfigurer {..}
```

2. Override the addInterceptors method and add our locale change interceptor to the registry. We can do this simply by passing its bean localeChangeInterceptor as a parameter to interceptorRegistry.addInterceptor method. Let's add this overriding method to our main class JavaI18nSpringBootApplication. For example:

```
@Override

public void addInterceptors(InterceptorRegistry interceptorRegistry) {

    interceptorRegistry.addInterceptor(localeChangeInterceptor());

}
```

i18N

Create a Controller

Add a class named HelloController within the same package and annotate it with @Controller. This will mark this class as a Spring Controller which holds Controller endpoints on Spring MVC architecture, as below:

```
import org.springframework.stereotype.Controller;
@Controller
public class HelloController {
}
```

Now, let’s add a GET mapping to the root URL. Add this to HelloController:

```
@GetMapping("/")
public String hello() { // <--- 1
```

i18N

The method name is insignificant here since the Spring IoC Container resolves the mapping by looking at the annotation type, method parameters, and method return value.

The hello View is called by the Controller.

Implement a View

Next, it’s time to create a simple View on our java-i18n-spring-boot application. Let’s make a hello.html file within the project’s resources/templates directory, like so:

i18N

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">  <!-- 1 -->
<head>
    <meta charset="UTF-8">
    <title th:text="#{welcome}"></title>  <!-- 2 -->
</head>
<body>
    <span th:text="#{hello}"></span>!<br>
    <span th:text="#{welcome}"></span><br>
    <button type="button" th:text="#{switch-en}"
onclick="window.location.href='http://localhost:8080/?localeData=en'"></button>
    <button type="button" th:text="#{switch-it}"
onclick="window.location.href='http://localhost:8080/?localeData=it'"></button>  <!--
3 -->
```

i18N

Make sure to declare Thymeleaf namespace in order to support th:* attributes.

The value for the welcome key is retrieved from the applicable language resource file of the specified locale and displayed as the title.

The button has the value of a switch-it property key of the specified locale.

Upon clicking the button, the page is reloaded with an additional localeData=it parameter. This in turn causes our LocaleChangeInterceptor to kick in and resolve the template in the Italian language.

Test Functionality

Let’s see if our Spring Boot application correctly performs internationalization. Run the project, then open up a browser and hit the GET mapping URL we coded on our application’s Controller, which in this case would be the root URL localhost:8080/. Click different ‘language switch’ buttons to see if the page now reloads with its content properly localized in the requested locale.

i18N

As a nifty bonus, switch to one locale, close and reopen the browser, and navigate to the root URL again; since we used CookieLocaleResolver as our LocaleResolver implementation, you'll see that the chosen locale choice has been retained.

Scour Spring Boot I18n

Let's skim through a few more features that could turn out to be useful when internationalizing our Spring Boot application.

Interpolation

Additionally, to make our java-i18n-spring-boot application a tiny bit more interesting, let's add a name path variable to our GET mapping. Then, we'll add it to the MVC model object to eventually be passed on to our View. Open up the HelloController class and insert a new mapping method to it as follows:

i18N

```
@GetMapping("/{name}") // <--- 1
public String hello(@PathVariable String name, Model model) { // <--- 2
    model.addAttribute("name", name); // <--- 3
    return "hello"; // <--- 4
}
```

Placing name inside brackets lets Spring identify it as a URI template variable.

@PathVariable annotating the name variable here binds it to a URI template variable of the same name.

The name variable is added as a Model attribute.

The hello View is called by the Controller passing the Model named model along with it.

Note that {name} here acts as a placeholder. The user of this application can replace it by calling the root URL suffixed with an additional text.

i18N

Pluralization

With the internationalization of our Spring Boot app aiming to support various locales, pluralization can become a somewhat overlooked, yet crucial step.

To demonstrate the point, let’s suppose we need to handle text representing some apples based on a provided quantity. So, for the English language, it would take this form:

0 apples

1 apple

2 apples

i18N

In order to handle pluralization, we can take the help of the spring-icu library which introduces ICU4J message formatting features into Spring. Since the project on GitHub is a Gradle project, we'll have to take its Maven project available on the JitPack repository. Follow the steps mentioned there to add the spring-icu dependency onto our java-i18n-spring-boot application.

Firstly, head over to the JavaI18nSpringBootApplication class of your project, and add a new ICUMessageSource bean. Make sure to set its base name correctly with a classpath: prefix, like so:

```
@Bean
public ICUMessageSource messageSource() {
    ICUReloadableResourceBundleMessageSource messageSource = new
ICUReloadableResourceBundleMessageSource();
    messageSource.setBasename("classpath:lang/res");
    return messageSource;
}
```

i18N

Secondly, add a plural property to the res.properties file indicating how to deal with particular quantities of apples:

```
plural={0} {0, plural, zero{apples}one{apple}other{apples}}
```

Note that this follows the FormatElement: { ArgumentIndex , FormatType , FormatStyle } pattern mentioned on MessageFormat with a 'plural' FormatType added by the spring-icu library.

i18N

Finally, add these lines to the hello.html after the main body:

```
<button type="button" th:text="#{switch-en}"
onclick="window.location.href='http://localhost:8080/?localeData=en'"></button>

<button type="button" th:text="#{switch-it}"
onclick="window.location.href='http://localhost:8080/?localeData=it'"></button>

<br><span th:text="#{plural(0)}"></span>

<br><span th:text="#{plural(1)}"></span>

<br><span th:text="#{plural(22)}"></span>
```

Run the java-i18n-spring-boot application and test out how the plurals are calculated when representing zero apples, one apple, and multiple quantities of apples.

i18N

Date and Time

We can use the @DateTimeFormat Spring annotation to parse – or in other terms, deserialize – a String date-time input into a LocalDate or LocalDateTime object.

Open up HelloController in our java-i18n-spring-boot application and add a new GET mapping:

```
@GetMapping("/datetime")

@ResponseBody

public String dateTime(@RequestParam("date") @DateTimeFormat(iso =
DateTimeFormat.ISO.DATE) LocalDate date,

                        @RequestParam("datetime") @DateTimeFormat(iso =
DateTimeFormat.ISO.DATE_TIME) LocalDateTime datetime) {

    return date.toString() + "<br>" + datetime.toString();

}
```

i18N

Run the application and call the /datetime GET endpoint passing parameters as follows:

date param: String in the most common ISO Date Format – yyyy-MM-dd

e.g. 1993-12-16

datetime param: String in the most common ISO DateTime Format – yyyy-MM-dd'T'HH:mm:ss.SSSXXX

e.g. 2018-11-22T01:30:00.000-05:00

Project Lombok

"Boilerplate" is a term used to describe code that is repeated in many parts of an application with little alteration. One of the most frequently voiced criticisms of the Java language is the volume of this type of code that is found in most projects. This problem is frequently a result of design decisions in various libraries, but it's exacerbated by limitations in the language itself. Project Lombok aims to reduce the prevalence of some of the worst offenders by replacing them with a simple set of annotations.

While it is not uncommon for annotations to be used to indicate usage, to implement bindings, or even to generate code used by frameworks, they are generally not used for the generation of code that is directly utilized by the application. This is partly because doing so would require that the annotations be eagerly processed at development time. Project Lombok does just that. By integrating into the IDE, Project Lombok is able to inject code that is immediately available to the developer.

Project Lombok

For example, simply adding the @Data annotation to a data class, as below, results in a number of new methods in the IDE under project customer-service:

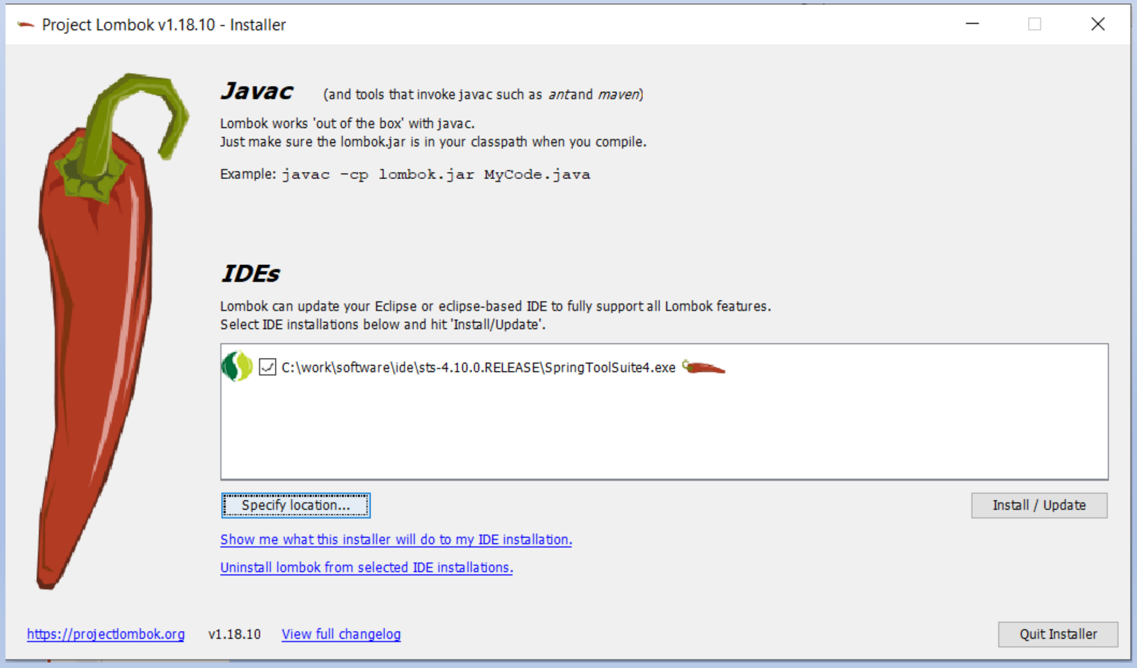
```
@Data
@Entity
@Table(name = "customer")
public class Customer {

@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private long id;
```

Project Lombok

INSTALLATION:

Run: java -jar lombok.jar

Specify the installation directory of your IDE:



Project Lombok v1.18.10 - Installer

Javac (and tools that invoke javac such as *ant* and *maven*)

Lombok works 'out of the box' with javac.
Just make sure the lombok.jar is in your classpath when you compile.

Example: `javac -cp lombok.jar MyCode.java`

IDEs

Lombok can update your Eclipse or eclipse-based IDE to fully support all Lombok features.
Select IDE installations below and hit 'Install/Update'.

[✓] C:\work\software\ide\sts-4.10.0.RELEASE\SpringToolSuite4.exe

Specify location...
Install / Update
Show me what this installer will do to my IDE installation.
Uninstall lombok from selected IDE installations.
https://projectlombok.org
v1.18.10
View full changelog
Quit Installer

Project Lombok

The jar file will still need to be included in the classpath of any projects that will use Project Lombok annotations. Maven users can include Lombok as a dependency by adding this to the project pom.xml file:

Pom.xml:

```
<dependency>
<groupId>org.projectlombok</groupId>
<artifactId>lombok</artifactId>
<optional>true</optional>
</dependency>
```

Project Lombok

The project Lombok is a popular and widely used Java library that is used to minimize or remove the boilerplate code. It saves time and effort. Just by using the annotations, we can save space and readability of the source code. It is automatically plugging into IDEs and build tools to spice up our Java application.

Here, a question arises that **does project Lombok and IDEs do the same work? If yes, then what is the use of Lombok?**

The answer is **no**, IDEs and Lombok do different works but are closely similar to each other. When we use IDEs to generate these boilerplate codes (getters and setters), we save ourselves from writing getters and setters manually but it actually exists in the source code that increases the lines of code, and reduces maintainability and readability. While the project Lombok adds all these boilerplate codes at the compile-time in the class file instead of adding these boilerplate code in original source code.

Project Lombok

The Lombok Java API includes the following packages that can be used for different purposes.

- lombok
- experimental
- extern.apachecommons
- extern.flogger
- extern.java
- extern.jbosslog
- extern.log4j
- extern.slf4j

Project Lombok

Features of Lombok Project

- It reduces the **boilerplate**
- It replaces boilerplate code with easy-to-use **annotations**.
- It makes code easier to read and **less error-prone**.
- By using Lombok the developers becomes more **productive**.
- It works well with all popular IDEs.
- Also provides **delombok** utility (adding back all the boilerplate code).
- Provide annotation for checking null values.
- Concise data objects
- Easy cleanup
- Locking safely
- Effortless logging

Project Lombok

Java Lombok Package

The package contains all the annotations and classes required to use Lombok. All other packages are only applicable to those who are extending Lombok for their own uses, except the following two packages:

- **lombok.extern.*:** The packages contain Lombok annotations that are useful to reduce boilerplate issues for libraries. It is not part of the JRE itself.
- **lombok.experimental:** The package contains Lombok features that are new or likely to change before committing to long-term support.

The Java Lombok package contains the following classes:

Classes	
Class	**Description**
ConfigurationKeys	A container class containing all Lombok configuration keys that do not belong to a specific annotation.
Lombok	Useful utility methods to manipulate Lombok-generated code.
Enum	
AccessLevel	Represents an AccessLevel.

Project Lombok

Annotations:

Annotations	
Annotations	**Description**
AllArgsConstructor	Generates an all-args constructor.
Builder	The builder annotation creates a so-called 'builder' aspect to the class that is annotated or the class that contains a member which is annotated with @Builder.
Builder.Default	The field annotated with @Default must have an initializing expression; that expression is taken as the default to be used if not explicitly set during building.
Builder.ObtainVia	Put on a field (in case of @Builder on a type) or a parameter (for @Builder on a constructor or static method) to indicate how Lombok should obtain a value for this field or parameter given an instance; this is only relevant if toBuilder is true.
Cleanup	Ensures the variable declaration that you annotate will be cleaned up by calling its close method, regardless of what happens.
CustomLog	Causes Lombok to generate a logger field based on a custom logger implementation.
Data	Generates getters for all fields, a useful toString method, and hashCode and equals implementations that check all non-transient fields.
EqualsAndHashCode	Generates implementations for the equals and hashCode methods inherited by all objects, based on relevant fields.
EqualsAndHashCode.Exclude	If present, do not include this field in the generated equals and hashCode methods.
EqualsAndHashCode.Include	Configure the behavior of how this member is treated in the equals and hashCode implementation; if on a method, include the method's return value as part of calculating hashCode/equality.

Project Lombok

Annotations:

Annotations	
Annotations	**Description**
NoArgsConstructor	Generates a no-args constructor.
NonNull	If put on a parameter, Lombok will insert a null-check at the start of the method /constructor's body, throwing a NullPointerException with the parameter's name as a message.
RequiredArgsConstructor	Generates a constructor with required arguments.
Setter	Put on any field to make Lombok build a standard setter.
Singular	The singular annotation is used together with @Builder to create single element 'add' methods in the builder for collections.
SneakyThrows	@SneakyThrow will avoid javac's insistence that you either catch or throw onward any checked exceptions that statements in your method body declare they generate.
Synchronized	Almost exactly like putting the 'synchronized' keyword on a method, except will synchronize on a private internal Object, so that other code not under your control doesn't meddle with your thread management by locking on your own instance.
ToString	Generates an implementation for the toString method inherited by all objects, consisting of printing the values of relevant fields.
ToString.Exclude	If present, do not include this field in the generated toString.
ToString.Include	Configure the behavior of how this member is rendered in the toString; if on a method, include the method's return value in the output.
val	Use val as the type of any local variable declaration (even in a for-each statement), and the type will be inferred from the initializing expression.
Value	Generates a lot of code that fits with a class that is a representation of an immutable entity.
var	Use var as the type of any local variable declaration (even in a for statement), and the type will be inferred from the initializing expression (any further assignments to the variable are not involved in this type inference).
With	Put on any field to make Lombok build a 'with' - a withX method that produces a clone of this object (except for 1 field which gets a new value).

Project Lombok

Purpose to Use Lombok Project

There are several reasons to use Lombok but some of them are as follows:

Check for Nulls

It is the most basic utility that Lombok offers. The library offers **@NonNull** annotation that can be used to generate a null check on a setter field. It throws the **NullPointerException** if the annotated class field contains a null value. Note that we cannot annotate the **primitive** parameter. With @NonNull annotation. For example, consider the following code snippet.

```
@NonNull @Setter

private String studentId;
```

Project Lombok

Concise Data Object

Generating getters and setters can be laborious work if there are several private fields in the POJO file. The task can be performed easily with the Lombok by using the **@Getter** and **@Setter** annotation. For example, consider the following code.

```
@Getter @Setter private String studentName;
```

```
@Data

public class Customer {
```

The **@Data** annotation can be used to apply usefulness behind all the annotations. It means that annotate a class with the @Data annotation, Lombok produces getters and setters for every one of the non-static class fields and a class constructor. It is the same as **toString(), equivalents(), and hashCode() strategies. It makes the coding of a POJO exceptionally simple.**

Project Lombok

Generate Constructor Automatically

It provides two annotations to generate constructors i.e. **@AllArgsConstructor** and **@NoArgsConstructor**. The **@AllArgsConstructor** annotation generates a constructor with all fields that are declared. If any field is added or removed, the constructor is also revised for the changes. The **@NoArgsConstructor** annotation simply generates the constructor without any argument.

Generating Getters for Final Fields

The **@Value** annotation is the same as the @Data annotation. It is a class-level annotation. The only difference is that it generates an **immutable** class. It invokes the automatic generation of getters only for all **private** and **final** fields. Note that it does not generate **setters** for any field, and marked the class as final.

Annotate Our Class to Get a Logger

Many of us add logging statements to our code sparingly by creating an instance of a Logger from our framework of choice. Example:

@Slf4j @Log @CommonsLog @Log4j @Log4j2

Spring AOP

Aspect Oriented Programming (AOP) compliments OOPs in the sense that it also provides modularity. But the key unit of modularity is aspect than class.

AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by **cross-cutting concerns**.

A **cross-cutting concern** is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

Spring AOP module lets interceptors intercept an application. For example, when a method is executed, you can add extra functionality before or after the method execution.

Spring AOP

Why use AOP?

It provides the pluggable way to dynamically add the additional concern before, after or around the actual logic. Suppose there are 10 methods in a class as given below:

```
class A{
public void m1(){...}
public void m2(){...}
public void m3(){...}
public void m4(){...}
public void m5(){...}
public void n1(){...}
public void n2(){...}
public void p1(){...}
public void p2(){...}
public void p3(){...}
}
```

Spring AOP

Why use AOP?

- There are 5 methods that starts from m, 2 methods that starts from n and 3 methods that starts from p.
- **Understanding Scenario** I have to maintain log and send notification after calling methods that starts from m.
- **Problem without AOP** We can call methods (that maintains log and sends notification) from the methods starting with m. In such scenario, we need to write the code in all the 5 methods.
- But, if client says in future, I don't have to send notification, you need to change all the methods. It leads to the maintenance problem.
- **Solution with AOP** We don't have to call methods from the method. Now we can define the additional concern like maintaining log, sending notification etc. in the method of a class. Its entry is given in the xml file.
- In future, if client says to remove the notifier functionality, we need to change only in the xml file. So, maintenance is easy in AOP.

Spring AOP

Where use AOP?

AOP is mostly used in following cases:

- to provide declarative enterprise services such as declarative transaction management.
- It allows users to implement custom aspects.

AOP Concepts and Terminology:

Join point

Advice

Pointcut

Introduction

Target Object

Aspect

Interceptor

AOP Proxy

Weaving

Spring AOP

Join point

Join point is any point in your program such as method execution, exception handling, field access etc. Spring supports only method execution join point.

Advice

Advice represents an action taken by an aspect at a particular join point. There are different types of advices:

Before Advice: it executes before a join point.

After Returning Advice: it executes after a joint point completes normally.

After Throwing Advice: it executes if method exits by throwing an exception.

After (finally) Advice: it executes after a join point regardless of join point exit whether normally or exceptional return.

Around Advice: It executes before and after a join point.

Spring AOP

Pointcut

It is an expression language of AOP that matches join points.

Introduction

It means introduction of additional method and fields for a type. It allows you to introduce new interface to any advised object.

Target Object

It is the object i.e. being advised by one or more aspects. It is also known as proxied object in spring because Spring AOP is implemented using runtime proxies.

Aspect

It is a class that contains advices, joinpoints etc.

Interceptor

It is an aspect that contains only one advice.

Spring AOP

AOP Proxy

It is used to implement aspect contracts, created by AOP framework. It will be a JDK dynamic proxy or CGLIB proxy in spring framework.

Weaving

It is the process of linking aspect with other application types or objects to create an advised object. Weaving can be done at compile time, load time or runtime. Spring AOP performs weaving at runtime.

AOP Implementations

AspectJ

Spring AOP

JBoss AO

Spring AOP

PointCut Methods:

JoinPoint

A JoinPoint represents a point in your application where you can plug-in AOP aspect. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework. Consider the following examples −

All methods classes contained in a package(s).

A particular methods of a class.

PointCut

PointCut is a set of one or more JoinPoint where an advice should be executed. You can specify PointCuts using expressions or patterns as we will see in our AOP examples. In Spring, PointCut helps to use specific JoinPoints to apply the advice. Consider the following examples −

@PointCut("execution(com.tutorialspoint.*.*(..))")*

@PointCut("execution(com.tutorialspoint.Student.getName(..))")*

Spring AOP

PointCut Methods:

```
@Aspect
public class Logging {
   @PointCut("execution(* com.tutorialspoint.*.*(..))")
   private void selectAll(){}
}
```

Where,

@Aspect − Mark a class as a class containing advice methods.

@PointCut − Mark a function as a PointCut

execution(expression) − Expression covering methods on which advice is to be applied.

Spring AOP

@Before is an advice type which ensures that an advice runs before the method execution. Following is the syntax of @Before advice.

```
@PointCut("execution(* com.tutorialspoint.Student.getName(..))")
private void selectGetName(){}
@Before("selectGetName()")
public void beforeAdvice(){
   System.out.println("Going to setup student profile.");
}
```

Where,

@PointCut − Mark a function as a PointCut

execution(expression) − Expression covering methods on which advice is to be applied.

@Before − Mark a function as an advice to be executed before method(s) covered by PointCut.

Spring AOP

@After is an advice type which ensures that an advice runs after the method execution. Following is the syntax of @After advice.

```
@PointCut("execution(* com.tutorialspoint.Student.getAge(..))")
private void selectGetName(){}
@After("selectGetAge()")
public void afterAdvice(){
   System.out.println("Student profile setup completed.");
}
```

Where,

@PointCut − Mark a function as a PointCut

execution(expression) − Expression covering methods on which advice is to be applied.

@After − Mark a function as an advice to be executed before method(s) covered by PointCut.

Spring boot Cache

Caching is a part of temporary memory (RAM). It lies between the application and persistence database. It stores the recently used data that reduces the number of database hits as much as possible.

The primary reason for using cache is to make data access faster and less expensive. When the highly requested resource is requested multiple times, it is often beneficial for the developer to cache resources so that it can give responses quickly. Using cache in an application enhances the performance of the application. Data access from memory is always faster in comparison to fetching data from the database. It reduces both monetary cost and opportunity cost.

Spring boot Cache annotations

@Cacheable: Triggers cache population.

@CacheEvict: Triggers cache eviction.

@CachePut: Updates the cache without interfering with the method execution.

@Caching: Regroups multiple cache operations to be applied on a method.

@CacheConfig: Shares some common cache-related settings at class-level.

Spring boot Cache

@EnableCaching

It is a class-level annotation. We can enable caching in the Spring Boot application by using the annotation @EnableCaching. It is defined in org.springframework.cache.annotation package. It is used together with @Configuration class.

The auto-configuration enables caching and setup a CacheManager, if there is no already defined instance of CacheManager. It scans for a specific provider, and when it does not find, it creates an in-memory cache using concurrent HashMap.

Spring boot Cache

@CacheConfig

It is a class-level annotation that provides a common cache-related setting. It tells the Spring where to store cache for the class. When we annotate a class with the annotation, it provides a set of default settings for any cache operation defined in that class. Using the annotation, we need not to declare things multiple times.

```
@CacheConfig(cacheNames={"employee"})
public class UserService
{
//some code
}
```

Spring boot Cache

@Caching

It is type. used when we need both annotations @CachePut or @CacheEvict at the same time on the same method. In other words, it is used when we want to use multiple annotations of the same

But Java does not allow multiple annotations of the same type to be declared for a given method. To avoid this problem, we use @Caching annotation.

```
@Caching(evict = {@CacheEvict("phone_number"),
@CacheEvict(value="directory", key="#student.id")
})
public String getAddress(Student student)
{
//some code
}
```

Spring boot Cache

@Cacheable

It is a method level annotation. It defines a cache for a method's return value. The Spring Framework manages the requests and responses of the method to the cache that is specified in the annotation attribute. The @Cacheable annotation contains more options. For example, we can provide a cache name by using the value or cacheNames attribute.

```
@Cacheable(value = "customerInfo")
public List customerInformation() {{
//some code
return customerDetails;
}
```

Spring boot Cache

@CacheEvict

It is a method level annotation. It is used when we want to remove stale or unused data from the cache. It requires one or multiple caches that are affected by the action.

Evict the whole cache:

```
@CacheEvict(allEntries=true)
```

Evict an entry by key:

```
@CacheEvict(key="#student.stud_name")
```

```
@CacheEvict(value="student_data", allEntries=true)
//removing all entries from the cache
public String getNames(Student student)
{
//some code
}
```

Spring boot Cache

@CachePut

It is a method level annotation. It is used when we want to update the cache without interfering the method execution. It means the method will always execute, and its result will be placed into the cache. It supports the attributes of @Cacheable annotation.

```
@CachePut(cacheNames="employee", key="#id")
//updating cache
public Employee updateEmp(ID id, EmployeeData
data)
{
//some code
}
```

Spring boot EHCache

EhCache is an open-source, Java-based cache used to boost performance.

Features:

- It is fast, lightweight, Scalable, and Flexible.
- It allows us to perform Serializable and Object
- It offers cache eviction policies such as LRU, LFU, FIFO,
- It stores the cache in memory and disk (SSD).
- It depends on SLF4J for logging.
- It has a full implementation of JSR-107 and Jcache
- It supports distributed caching via JGroups or JMS and RMI.
- It uses fluent query language for distributed search.

Spring boot Redis Cache

Redis cache provides Snapshots, Replication, Transactions, Pub/Sub, Lua scripting, Geospatial support features which are not provided by Memcached. Memcached could be preferable when caching relatively small and static data, such as HTML code, images and small metadata sets.

Redis cache limits:

It has no limit on storing data on a 64-bit system but, on the 32-bit system, it can only store 3 GB. So, once cache reaches its memory limit, we should remove the old data to make some space for the new one.

Spring Batch

Spring Batch is a lightweight framework which is used to develop Batch Applications that are used in Enterprise Applications. This tutorial explains the fundamental concepts of Spring Batch and shows how you can use it in practical environment.

Batch processing is a processing mode which involves execution of series of automated complex jobs without user interaction. A batch process handles bulk data and runs for a long time.

Several Enterprise applications require to process huge data to perform operations involving −

Time-based events such as periodic calculations.

Periodic applications that are processed repetitively over large datasets.

Applications that deals with processing and validation of the data available in a transactional manner.

Therefore, batch processing is used in enterprise applications to perform such transactions.

Spring Batch

In addition to bulk processing, this framework provides functions for −

- Including logging and tracing
- Transaction management
- Job processing statistics
- Job restart
- Skip and Resource management

You can also scale spring batch applications using its portioning techniques.

Features of Spring Batch

- Following are the notable features of Spring Batch −
- **Flexibility** − Spring Batch applications are flexible. You simply need to change an XML file to alter the order of processing in an application.
- **Maintainability** − Spring Batch applications are easy to maintain. A Spring Batch job includes steps and each step can be decoupled, tested, and updated, without effecting the other steps.
- **Scalability** − Using the portioning techniques, you can scale the Spring Batch applications. These techniques allow you to −
    - Execute the steps of a job in parallel.
    - Execute a single thread in parallel.
- **Reliability** − In case of any failure, you can restart the job from exactly where it was stopped, by decoupling the steps.
- **Support for multiple file formats** − Spring Batch provides support for a large set of readers and writers such as XML, Flat file, CSV, MYSQL, Hibernate, JDBC, Mongo, Neo4j, etc.
- **Multiple ways to launch a job** − You can launch a Spring Batch job using web applications, Java programs, Command Line, etc.

Features of Spring Batch

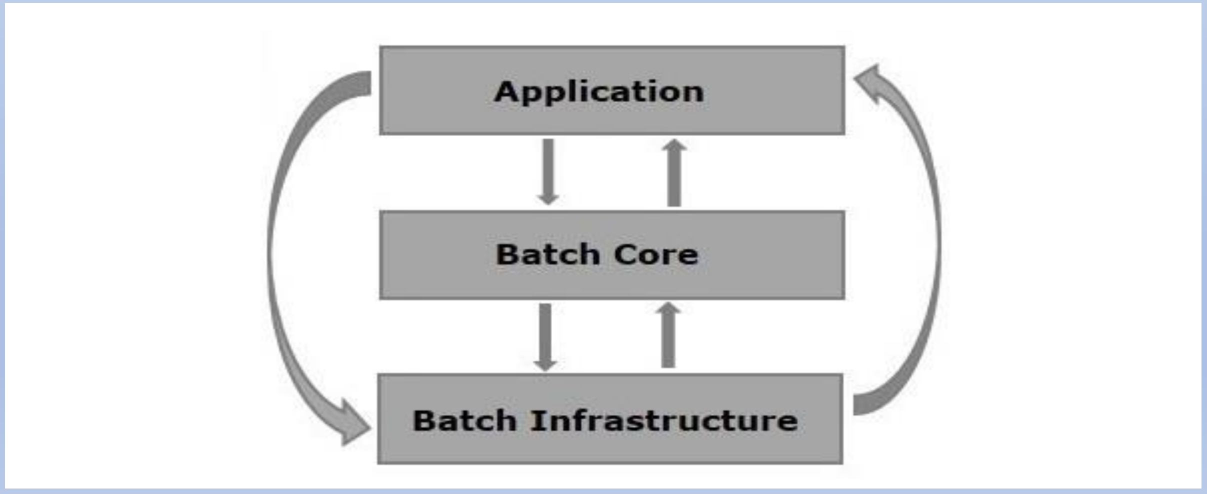
In addition to these, Spring Batch applications support −

- Automatic retry after failure.
- Tracking status and statistics during the batch execution and after completing the batch processing.
- To run concurrent jobs.
- Services such as logging, resource management, skip, and restarting the processing.

Spring Batch

Three main components:

- **Application** − This component contains all the jobs and the code we write using the Spring Batch framework.
- **Batch Core** − This component contains all the API classes that are needed to control and launch a Batch Job.
- **Batch Infrastructure** − This component contains the readers, writers, and services used by both application and Batch core components.

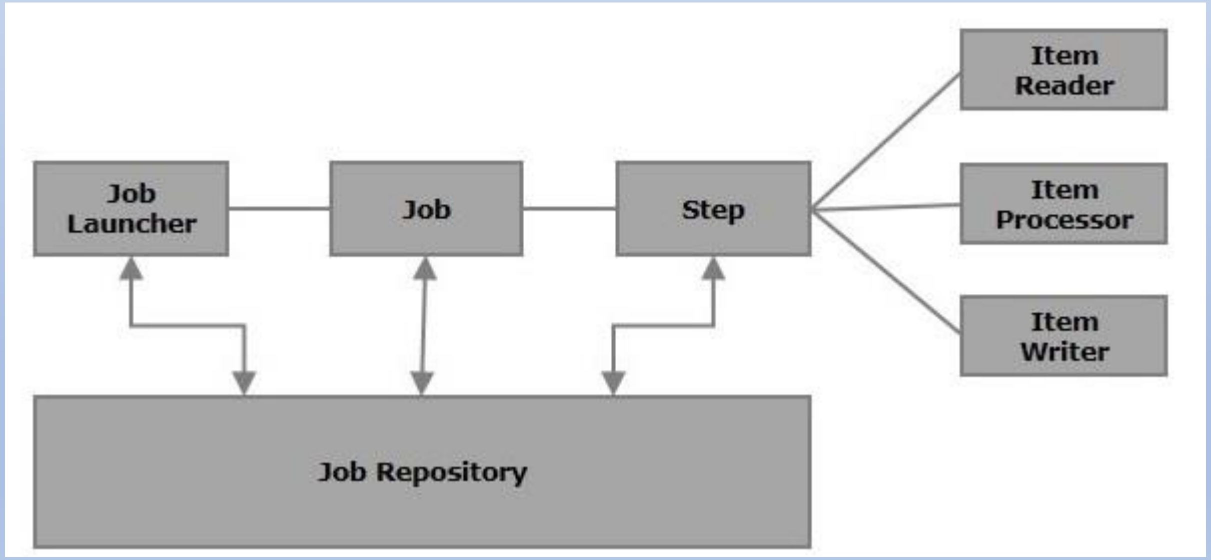


Spring Batch

Different components of Spring Batch

Job - Job is the batch process that is to be executed. It runs from start to finish without interruption. This job is further divided into steps (or a job contains steps).

Step - A step is an independent part of a job which contains the necessary information to define and execute the job (its part).



Spring Batch

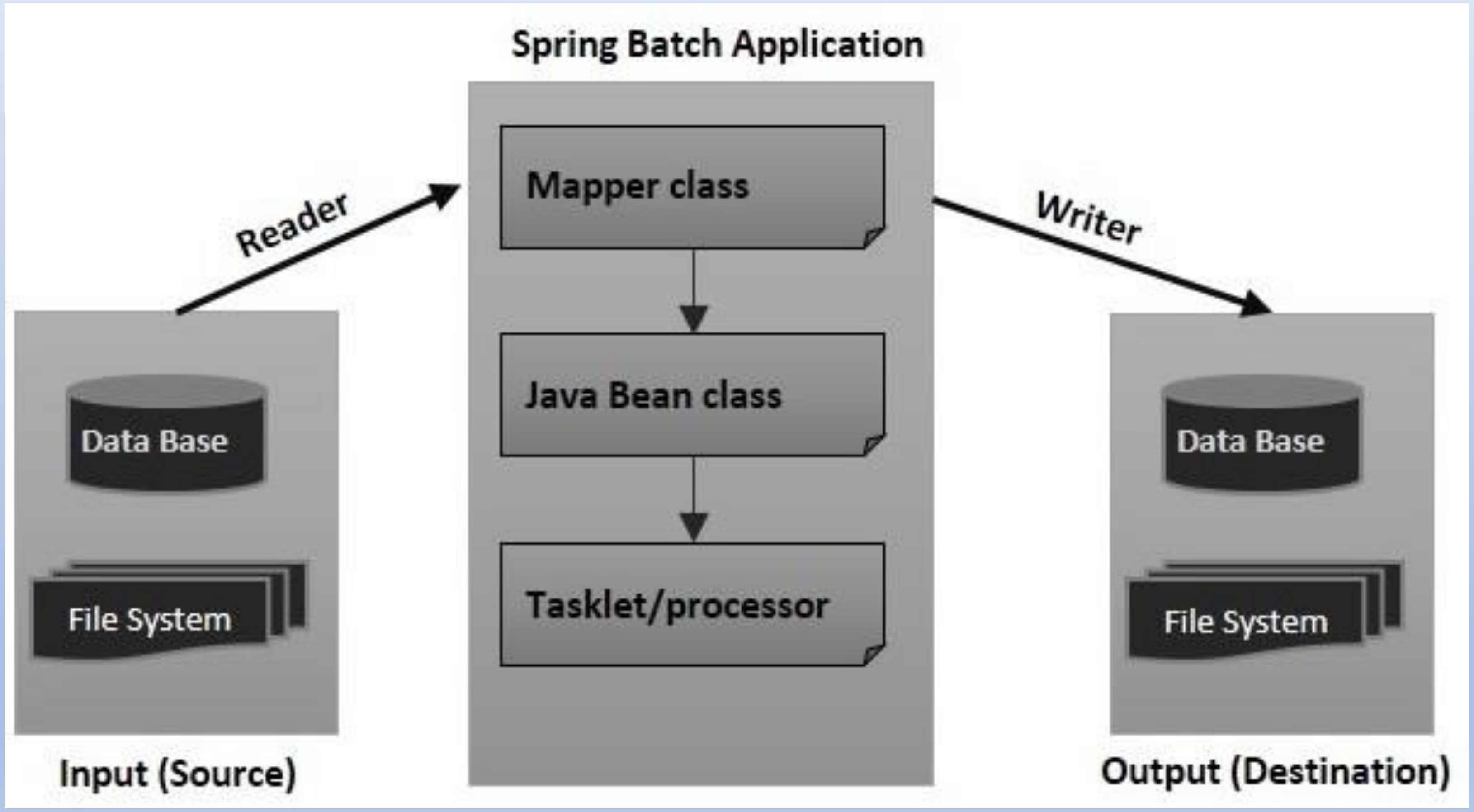
Readers, Writers, and Processors –

- An **item reader** reads data into a Spring Batch application from a particular source, whereas an **item writer** writes data from the Spring Batch application to a particular destination.
- An **Item processor** is a class which contains the processing code which processes the data read into the spring batch. If the application reads **"n"** records, then the code in the processor will be executed on each record.
- When no reader and writer are given, a **tasklet** acts as a processor for SpringBatch. It processes only a single task. For example, if we are writing a job with a simple step in it where we read data from MySQL database and process it and write it to a file (flat), then our step uses −
    - A **reader** which reads from MySQL database.
    - A **writer** which writes to a flat file.
    - A **custom processor** which processes the data as per our wish.

Spring Batch

- **JobRepository -** A Job repository in Spring Batch provides Create, Retrieve, Update, and Delete (CRUD) operations for the JobLauncher, Job, and Step implementations.
- **In-Memory Repository** − In case you don’t want to persist the domain objects of the Spring Batch in the database, you can configure the in-memory version of the jobRepository as shown below.
- **JobLauncher -** JobLauncher is an interface which launces the Spring Batch job with the given set of parameters. SampleJoblauncher is the class which implements the JobLauncher interface.
- **JobInstance** - represents the logical run of a job; it is created when we run a job. Each job instance is differentiated by the name of the job and the parameters passed to it while running. If a JobInstance execution fails, the same JobInstance can be executed again. Hence, each JobInstance can have multiple job executions.
- **JobExecution and StepExecution -** JobExecution and StepExecution are the representation of the execution of a job/step. They contain the run information of the job/step such as start time (of job/step), end time (of job/step).

Spring Batch



Spring Boot Request Timeout

Spring provides us with a property called spring.mvc.async.request-timeout. This property allows us to define a request timeout with millisecond precision:

```
spring.mvc.async.request-timeout=750
```

This property is global and externally configurable.

The response is returned immediately once the timeout has been reached, and it even returns a more descriptive 503 HTTP error instead of a generic 500.

Spring Boot Flyway

- **Flyway** is a database migration and version control tool
- It has Java API, command-line client, a plugin for Maven and Gradle
- Supports most of the relational databases such as MySQL, PostgreSQL, SQL Server, and Oracle
- Migration scripts can be written in either SQL or Java
- Spring Boot can autorun database migration at the application startup with a variety of mechanisms such as javax.sql.DataSource, JPA and Hibernate, Flyway and Liquibase

Spring Boot Flyway

- Spring Boot, via JPA and Hibernate, can auto-export schema DDL to a database via your definition on @Entity classes. You can turn it off by setting validate or none (none is the default value for non-embedded databases) to the spring.jpa.hibernate.ddl-auto property

- Spring Boot can autorun classpath:schema.sql and classpath:data.sql script files for your DataSource. This feature can be controlled via spring.datasource.initialization-mode property. Its value is embedded (only apply for embedded databases) by default

Spring Boot Flyway

Flyway migration scripts :

Unlike JPA and Hibernate, database migration in Flyway is not based on the definition of @Entity classes, you have to manually write the migration scripts in either SQL or Java, SQL is the most commonly used

- Typically, SQL scripts are in the form V<VERSION>__<DESCRIPTION>.sql
- <VERSION> is a dot or underscore separated version, such as '1.0' or '1_1'). <VERSION> must be unique
- <DESCRIPTION> should be informative for you able to remember what each migration does
- The following gives you some example SQL scripts [V1.0__create_book.sql]
- By default, Spring Boot looks for them in classpath:db/migration folder, you can modify that location by setting spring.flyway.locations

Spring Boot Flyway

Flyway works:

Flyway applies migration scripts to the underlying database in the order based on the version number specified in the script file naming

At each execution, only pending migrations are applied. Flyway manages this via creating (if not exists) and updating a metadata table.

The migration scripts can not be changed after applied. Flyway compares the checksum of each script in every execution and throws an exception if there's a mismatch.

Config Flyway DataSource

Spring Boot uses either annotations or external properties to connect Flyway to the underlying data source

@Primary DataSource or @FlywayDataSource annotation

spring.datasource.[url, username, password], or spring.flyway.[url, user, password] properties

Spring Boot Flyway

Run Flyway with Spring Boot

Spring Boot auto enable and trigger Flyway at the application startup when you include the Flyway core library into the project. In case you'd like to turn it off, update this setting spring.flyway.enabled to false (true is the default value)

The application startup may be failed if there's an exception (such as the checksum mismatch error of migration scripts mentioned in the previous section) thrown by Flyway during migration

Each migration script is run within a single transaction. You can configure to run all pending migrations in a single transaction with spring.flyway.group=true (the default value is false)

Query migration status and history

You can query migration status and history in web interface with Spring Boot Actuator by enabling it in this property management.endpoints.web.exposure.include=info,health,flyway and access to {endpoints}/actuator/flyway

Spring Rest Template

Accessing the REST apis inside a Spring application revolves around the use of the Spring RestTemplate class. The RestTemplate class is designed on the same principles as the many other Spring *Template classes (e.g., JdbcTemplate, JmsTemplate), providing a simplified approach with default behaviors for performing complex tasks.

Given that the RestTemplate class is a synchronous client that is designed to call REST services. It should come as no surprise that its primary methods are closely tied to REST's underpinnings, which are the HTTP protocol's methods HEAD, GET, POST, PUT, DELETE, and OPTIONS.

Building RestTemplate Bean

The given below are few examples to create RestTemplate bean in the application. We are only looking at very simple bean definitions. For extensive configuration options, please refer to RestTemplate Configuration with HttpClient.

Spring RestTemplate

RestTemplateBuilder

```
@Bean
public RestTemplate restTemplate(RestTemplateBuilder builder) {
    return builder
            .setConnectTimeout(Duration.ofMillis(3000))
            .setReadTimeout(Duration.ofMillis(3000))
            .build();
}
```

Spring RestTemplate

SimpleClientHttpRequestFactory

```
@Bean
public RestTemplate restTemplate() {

    var factory = new SimpleClientHttpRequestFactory();
    factory.setConnectTimeout(3000);
    factory.setReadTimeout(3000);
    return new RestTemplate(factory);
}
```

Spring RestTemplate

Apache HTTPClient

```
@Autowired
CloseableHttpClient httpClient;
 @Bean
public RestTemplate restTemplate() {
     RestTemplate restTemplate = new RestTemplate(clientHttpRequestFactory());
    return restTemplate;
}
@Bean
public HttpComponentsClientHttpRequestFactory clientHttpRequestFactory() {
     HttpComponentsClientHttpRequestFactory clientHttpRequestFactory
                        = new HttpComponentsClientHttpRequestFactory();
    clientHttpRequestFactory.setHttpClient(httpClient);
    return clientHttpRequestFactory;
}
```

Spring RestTemplate

Injecting RestTemplate bean

To inject the RestTemplate bean, use the well known @Autowired annotation. If you have multiple beans of type RestTemplate with different configurations, use the @Qualifier annotation as well.

```
@Autowired
private RestTemplate restTemplate;
```

3. Spring RestTemplate – HTTP GET Example

Available methods are:

getForObject(url, classType) – retrieve a representation by doing a GET on the URL. The response (if any) is unmarshalled to given class type and returned.

getForEntity(url, responseType) – retrieve a representation as ResponseEntity by doing a GET on the URL.

exchange(requestEntity, responseType) – execute the specified RequestEntity and return the response as ResponseEntity.

execute(url, httpMethod, requestCallback, responseExtractor) – execute the httpMethod to the given URI template, preparing the request with the RequestCallback, and reading the response with a ResponseExtractor.

Spring RestTemplate

HTTP GET REST APIs

```
@GetMapping(value = "/employees",
        produces = {MediaType.APPLICATION_XML_VALUE, MediaType.APPLICATION_JSON_VALUE})
public EmployeeListVO getAllEmployees(
        @RequestHeader(name = "X-COM-PERSIST", required = true) String headerPersist,
        @RequestHeader(name = "X-COM-LOCATION", defaultValue = "ASIA") String headerLocation)
{
    LOGGER.info("Header X-COM-PERSIST :: " + headerPersist);
    LOGGER.info("Header X-COM-LOCATION :: " + headerLocation);

    EmployeeListVO employees = getEmployeeList();
    return employees;
}
```

Spring RestTemplate

HTTP GET REST APIs

```
@GetMapping(value = "/employees/{id}",
        produces = {MediaType.APPLICATION_XML_VALUE, MediaType.APPLICATION_JSON_VALUE})
public ResponseEntity<EmployeeVO> getEmployeeById (@PathVariable("id") Integer id)
{
    LOGGER.info("Requested employee id :: " + id);

    if (id != null && id > 0) {
        //TODO: Fetch the employee and return from here
        EmployeeVO employee = new EmployeeVO(id, "Lokesh","Gupta", "howtodoinjava@gmail.com");
        return new ResponseEntity<EmployeeVO>(employee, HttpStatus.OK);
    }
    return new ResponseEntity<EmployeeVO>(HttpStatus.NOT_FOUND);
}
```

Spring RestTemplate

Spring RestTemplate example to consume REST API

In the given example, we are fetching the API response as a JSON string. We need to use ObjectMapper to parse it to the POJO before using it in the application.

getForObject() method is pretty useful when we are getting an unparsable response from the server, and we have no control to get it fixed on the server-side. Here, we can get the response as String, and use a custom parser or use a string replacement function to fix the response before handling it to the parser.

```
private static void getEmployees()
{
    final String uri = "http://localhost:8080/springrestexample/employees";
    //TODO: Autowire the RestTemplate in all the examples
    RestTemplate restTemplate = new RestTemplate();
    String result = restTemplate.getForObject(uri, String.class);
    System.out.println(result);
}
```

Spring RestTemplate

Spring RestTemplate example to consume API response into POJO
In the given example, we are fetching the API response directly into the domain object.

Using getForObject() Method

```
private static void getEmployees()
{
    final String uri = "http://localhost:8080/springrestexample/employees";
    RestTemplate restTemplate = new RestTemplate();
    EmployeeListVO result = restTemplate.getForObject(uri, EmployeeListVO.class);
    //Use the response
}
```

Spring RestTemplate

Spring RestTemplate example to consume API response into POJO

Using getForEntity() Method

```
private static void getEmployees()
{
    final String uri = "http://localhost:8080/springrestexample/employees";
    RestTemplate restTemplate = new RestTemplate();

    ResponseEntity<EmployeeListVO> response = restTemplate.getForEntity(uri,
EmployeeListVO.class);

    //Use the response.getBody()
}
```

Spring RestTemplate

Sending HTTP Headers using RestTemplate

```
private static void getEmployees()
{
    final String uri = "http://localhost:8080/springrestexample/employees";
    RestTemplate restTemplate = new RestTemplate();
    HttpHeaders headers = new HttpHeaders();
    headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
    headers.set("X-COM-PERSIST", "NO");
    headers.set("X-COM-LOCATION", "USA");
    HttpEntity<String> entity = new HttpEntity<String>(headers);
    ResponseEntity<String> response = restTemplate.exchange(uri, HttpMethod.GET, entity, 
String.class);
    //Use the response.getBody()
}
```

Spring RestTemplate

Sending URL Parameters using RestTemplate

```
private static void getEmployeeById()
{
    final String uri = "http://localhost:8080/springrestexample/employees/{id}";
    RestTemplate restTemplate = new RestTemplate();

    Map<String, String> params = new HashMap<String, String>();
    params.put("id", "1");

    EmployeeVO result = restTemplate.getForObject(uri, EmployeeVO.class, params);

    //Use the result
}
```

Spring RestTemplate

Spring RestTemplate – HTTP POST Example

Available methods are:

postForObject(url, request, classType) – POSTs the given object to the URL, and returns the representation found in the response as given class type.

postForEntity(url, request, responseType) – POSTs the given object to the URL, and returns the response as ResponseEntity.

postForLocation(url, request, responseType) – POSTs the given object to the URL, and returns returns the value of the Location header.

exchange(url, requestEntity, responseType)

execute(url, httpMethod, requestCallback, responseExtractor)

Spring RestTemplate

HTTP POST REST API

The POST API, we will consume in this example:

```
PostMapping(value = "/employees")
public ResponseEntity<String> createEmployee(EmployeeVO employee)
{
    //TODO: Save employee details which will generate the employee id
    employee.setId(111);
URI location = ServletUriComponentsBuilder.fromCurrentRequest()
                          .path("/{id}")
                          .buildAndExpand(employee.getId())
                          .toUri();
    return ResponseEntity.created(location).build();
}
```

Spring RestTemplate

Spring RestTemplate example to consume POST API
Spring REST client using RestTemplate to access HTTP POST api requests.

```
private static void createEmployee()
{
    final String uri = "http://localhost:8080/springrestexample/employees";
    RestTemplate restTemplate = new RestTemplate();

    EmployeeVO newEmployee = new EmployeeVO(-1, "Adam", "Gilly", "test@email.com");

    EmployeeVO result = restTemplate.postForObject( uri, newEmployee, EmployeeVO.class);

    System.out.println(result);
}
```

Spring RestTemplate

Spring RestTemplate – HTTP PUT Method Example

Available methods are:

put(url, request) – PUTs the given request object to URL.

```
@PutMapping(value = "/employees/{id}")
public ResponseEntity<EmployeeVO> updateEmployee(@PathVariable("id") int id
                ,EmployeeVO employee)
{
    //TODO: Save employee details
    return new ResponseEntity<EmployeeVO>(employee, HttpStatus.OK);
}
```

Spring RestTemplate

Spring RestTemplate example to consume PUT API

```
private static void updateEmployee()
{
    final String uri = "http://localhost:8080/springrestexample/employees/{id}";
    RestTemplate restTemplate = new RestTemplate();

    Map<String, String> params = new HashMap<String, String>();
    params.put("id", "2");

    EmployeeVO updatedEmployee = new EmployeeVO(2, "New Name", "Gilly", "test@email.com");

    restTemplate.put ( uri, updatedEmployee, params );
}
```

Spring RestTemplate

Spring RestTemplate – HTTP DELETE Method Example
Available methods are:
delete(url) – deletes the resource at the specified URL

```
@DeleteMapping(value = "/employees/{id}")
public ResponseEntity<String> deleteEmployee(@PathVariable("id") int id)
{
    //TODO: Delete the employee record
    return new ResponseEntity<String>(HttpStatus.OK);
}
```

Spring RestTemplate

Spring RestTemplate example to consume DELETE API

```
private static void deleteEmployee()
{
    final String uri = "http://localhost:8080/springrestexample/employees/{id}";
    RestTemplate restTemplate = new RestTemplate();

    Map<String, String> params = new HashMap<String, String>();
    params.put("id", "2");

    restTemplate.delete ( uri,  params );
}
```