Spring Boot

By
Apparao

Spring Boot:

Spring Boot helps you to create stand-alone, production-grade Spring-based Applications that you can run

You can use Spring Boot to create Java applications that can be started by using java -jar or more traditional war deployments.

Advantages:

- Provide a radically faster and widely accessible getting-started experience for all Spring development.
- Spring boot avoids lots of maven imports and the various version conflicts.
- Provide a range of non-functional features that are common to large classes of projects (such as embedded servers, security, metrics, health checks, and externalized configuration).
- No Separate Web Server Needed. Which means that you no longer have to boot up Tomcat, Glassfish, or anything else.
- Absolutely no code generation and no requirement for XML configuration.

System Requirements:

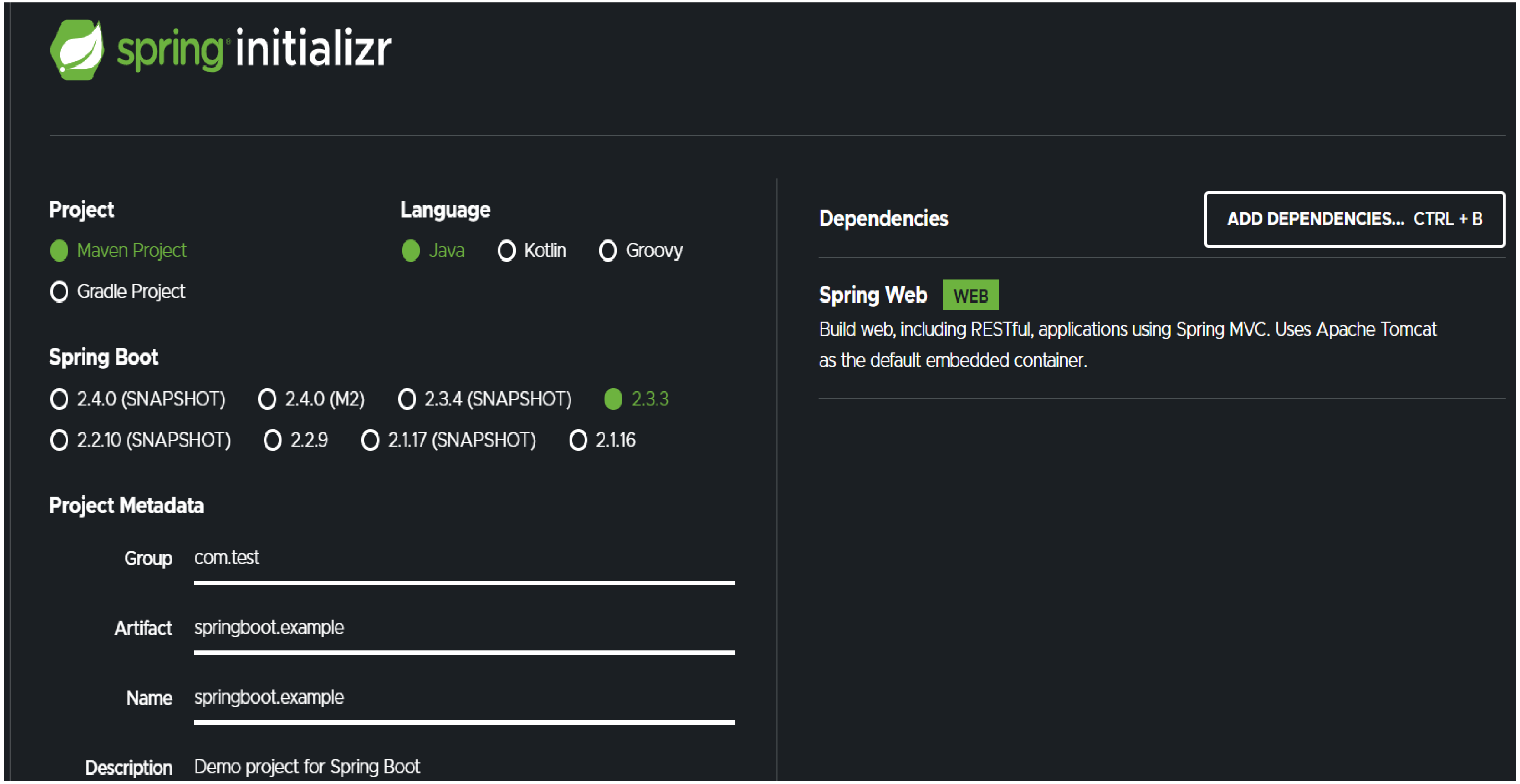
- Java 8+
- Maven 3.3+ (Spring Boot is compatible with Apache Maven 3.3 or above. If you do not already have Maven installed, you can follow the instructions at maven.apache.org.)
- Tomcat 9.0 (download the tomcat from https://tomcat.apache.org/download-90.cgi)

Creating Spring Boot Application:

1. Using spring Initializr
2. Spring Starter Project Wizard (Using STS)
3. Spring Boot CLI

Creating Spring boot project using spring Initializr

1. open the below link in your favarote brower
   https://start.spring.io/


spring initializr

Project

[✓ Maven Project] Gradle Project

Language

[✓ Java] Kotlin Groovy

Spring Boot

2.4.0 (SNAPSHOT) 2.4.0 (M2) 2.3.4 (SNAPSHOT) [✓ 2.3.3] 2.2.10 (SNAPSHOT) 2.2.9 2.1.17 (SNAPSHOT) 2.1.16

Project Metadata

Group	com.test
Artifact	springboot.example
Name	springboot.example
Description	Demo project for Spring Boot

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Web WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Provide required parameters in above screen.

1. select Maven Project

2. select Java

3. select Spring Boot version you need (2.3.3)

4. Provide Project Metadata

   Group : com.test

   Artifact: springboot.example

   Name : springboot.example

   Description : Demo project for Spring Boot

   Package name: com.test.springboot.example

   Package : war (select which packaging you want like jar or war)

   Java version: 8

5. Dependencies

   Click on "ADD DEPENDENCIES.."

   select Spring Web

6. Click on "GENERATE CTRL" button

   It will generate the spring boot project.

7. Unzip the generated project code.

8. Open the eclipse/sts

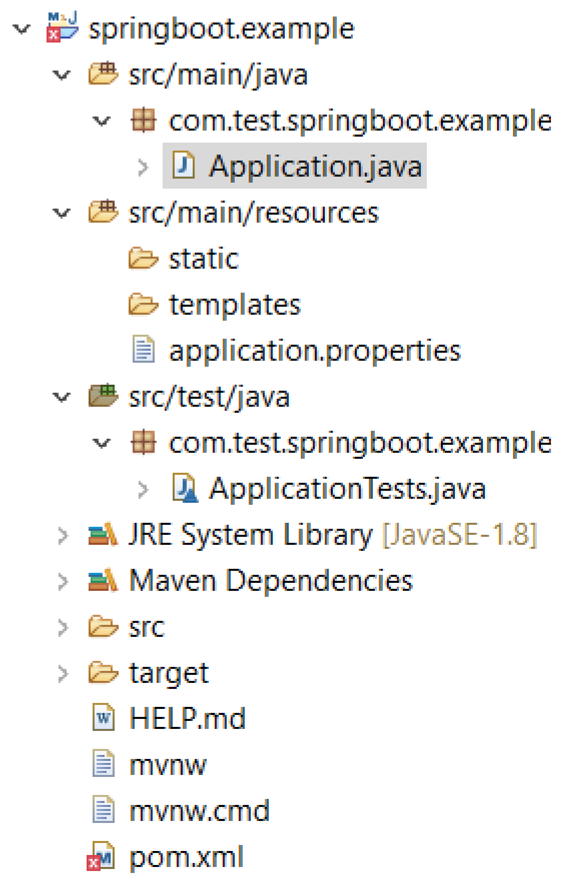
   Go to File --> Import --> Maven --> select "Existing Maven Project"

   click on "Next"

   Browse the project folder and select the project

   Click on "Finish" It will import project to eclipse.

Project structure:



```
springboot.example
  src/main/java
    com.test.springboot.example
      Application.java
  src/main/resources
    static
    templates
    application.properties
  src/test/java
    com.test.springboot.example
      ApplicationTests.java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

1. Application class:

```
package com.test.springboot.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

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

}
```

@SpringBootApplication is a convenience annotation that adds all of the following:

@Configuration: Tags the class as a source of bean definitions for the application context.

@EnableAutoConfiguration: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings. For example, if spring-webmvc is on the classpath, this annotation flags the application as a web application and activates key behaviors, such as setting up a DispatcherServlet.

@ComponentScan: Tells Spring to look for other components, configurations, and services in the com.test package, letting it find the controllers.

The main() method uses Spring Boot's SpringApplication.run() method to launch an application.

Note: There is no web.xml file and xml configuration files.. This web application is 100% pure Java and you did not have to deal with configuring any plumbing or infrastructure.

2. pom.xml file: It has following dependencies

```
<properties>
        <java.version>1.8</java.version>
</properties>
<dependencies>
                <dependency>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-starter-web</artifactId>
                </dependency>
</dependencies>

<build>
                <plugins>
                                <plugin>
                                                <groupId>org.springframework.boot</groupId>
                                                <artifactId>spring-boot-maven-plugin</artifactId>
                                </plugin>
                </plugins>
</build>
```

3. Create the controller class.

```
package com.test.controller;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

	@RequestMapping("/")
	public String message() {

		return "Welcome to spring boot technology";
	}
}
```

The class is flagged as a @RestController, meaning it is ready for use by Spring MVC to handle web requests.

@RequestMapping maps / to the message() method. When invoked from a browser or by using curl on the command line, the method returns pure text.

That is because @RestController combines @Controller and @ResponseBody, two annotations that results in web requests returning data rather than a view.

4. Build the maven project using below command.

```
mvn clean install
```

It will generate war file. Copy the war file into webapps directory of tomcat.

5. Run the tomcat server.

6. Launch the your favorite browser and access the url like below

http://localhost:8080/

How to Change the Default Port in Spring Boot?

By default, the embedded server start on port 8080. If you want to change different value then use server.port property in application.properties file

server.port=8085

How to change context path in spring boot application?

By default, serves content on the root context path ("/"). If you want to change different context path, use following property.

server.servlet.context-path=/springbootexample

Log4j Configuration:

log4j is a reliable, fast and flexible logging framework (APIs) written in Java, which is distributed under the Apache Software License.

log4j is highly configurable through external configuration files at runtime. It views the logging process in terms of levels of priorities and offers mechanisms to direct logging information to a great variety of destinations, such as a database, file, console etc.

log4j has three main components:

loggers: Responsible for capturing logging information.

appenders: Responsible for publishing logging information to various preferred destinations.

layouts: Responsible for formatting logging information in different styles.

It uses multiple levels, namely ALL, TRACE, DEBUG, INFO, WARN, ERROR and FATAL

Below table describes the log levels.

Level	Description
ALL	All levels including custom levels.
DEBUG	Designates fine-grained informational events that are most useful to debug an application.
INFO	Designates informational messages that highlight the progress of the application at coarse-grained level.
WARN	Designates potentially harmful situations.
ERROR	Designates error events that might still allow the application to continue running.
FATAL	Designates very severe error events that will presumably lead the application to abort.
OFF	The highest possible rank and is intended to turn off logging.
TRACE	Designates finer-grained informational events than the DEBUG.

Log4j configuration:

1. Add log4j dependency in pom.xml file.

```
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
```

2. Add the below log4j.xml file under resource folder

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration PUBLIC "-//APACHE//DTD LOG4J 1.2//EN" "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
	<appender name="ASYNC" class="org.apache.log4j.AsyncAppender">
		<param name="BufferSize" value="500" />
		<param name="LocationInfo" value="true" />
		<appender-ref ref="stdoutAppender" />
	</appender>
```

```
<appender name="fileAppender" class="org.apache.log4j.RollingFileAppender">
                <param name="Threshold" value="ALL" />
                <param name="MaxFileSize" value="100MB" />
                <param name="MaxBackupIndex" value="20" />
                <param name="File”  value="${catalina.base}/logs/springbootexample.log" />
                <layout class="org.apache.log4j.PatternLayout">
                        <param name="ConversionPattern" value="%d %t %-5p %c{1}:%L %m %n" />
                </layout>
        </appender>
        <appender name="stdoutAppender" class="org.apache.log4j.ConsoleAppender">
                <param name="Target" value="System.out" />
                <layout class="org.apache.log4j.PatternLayout">
                        <param name="ConversionPattern" value="%d %t %-5p %c{1}:%L %m %n" />
                </layout>
        </appender>
        <logger name="org.springframework">
                <level value="info" />
        </logger>
        <logger name="com.test">
                <level value="debug" />
        </logger>
        <logger name="org.hibernate.SQL">
                <level value="debug" />
        </logger>
        <root>
                <priority value="info"></priority>
                <appender-ref ref="ASYNC" />
        </root>
</log4j:configuration>
```

3. Usage of logger class.

```
@RestController
public class VegetableBagController {
	private static final Logger logger = Logger.getLogger(VegetableBagController.class);

	@RequestMapping(value = "/v1/vegetableBag", method = RequestMethod.POST, produces = MediaType.APPLICATION_JSON_VALUE,
			consumes = MediaType.APPLICATION_JSON_VALUE)
	public Response<VegetableBagDetails> saveVegetableBag(@Valid @RequestBody VegetableBagDetails vegetableBagDetails) {
		Response<VegetableBagDetails> response = new Response<VegetableBagDetails>();
		try {
			logger.info("saveVegetableBag request::" + vegetableBagDetails.toString());
			VegetableBagDetails vegetableBag = vegetableBagBusiness.saveVegetableBag(vegetableBagDetails);
			response.setResult(vegetableBag);
			response.setMetaData(new MetaData(200, VegetableBagConstant.SUCCESS));
		} catch (Exception e) {
			logger.error("Exception occurred while saveVegetableBag::" + e.getMessage());
			response.setErrorData(exceptionUtil.getErrorData(e));
			response.setMetaData(new MetaData(500, VegetableBagConstant.FAILURE));
		}
		return response;
	}
}
```

Database Integration using Spring Data JPA:

Spring Data JPA, part of the larger Spring Data family, makes it easy to easily implement JPA based repositories.

Implementing a data access layer of an application has been cumbersome for quite a while. Too much boilerplate code has to be written to execute simple queries as well as perform pagination, and auditing.

Spring Data JPA aims to significantly improve the implementation of data access layers by reducing the effort to the amount that’s actually needed. As a developer you write your repository interfaces, including custom finder methods, and Spring will provide the implementation automatically.

step1. Configure the Data source parameters in application.properties file.

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.driver-class-name=org.h2.Driver

# Add below properties for Statistics for query execution
spring.jpa.properties.hibernate.generate_statistics=true
logging.level.org.hibernate.stat=debug
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
logging.level.org.hibernate.type=trace
```

step2. Create the entity class for corresponding table.

```
@Entity
@Table(name = "vegetable_bag")
public class VegetableBag extends AbstractAuditable implements Serializable {
            @Id
            @GeneratedValue(generator = "system-uuid")
            @GenericGenerator(name = "system-uuid", strategy = "uuid2")
            @Column(name = "id", columnDefinition = "CHAR(36)")
            private String id;
            @Column(name = "quantity")
            private Integer quantity;
            @Column(name = "suppiler_name")
            private String supplierName;
            @Temporal(TemporalType.TIMESTAMP)
            @Column(name = "package_date")
            private Date packageDate;
            @Column(name = "price")
            private Double price;

           // generate setters and geters methods.
}
```

step3. Create the repository

```
  @Repository
  public interface VegetableBagRepository extends JpaRepository<VegetableBag, String> {


  }
```

step4. Invoke the repository from DAO class

```
 @Repository
 public class VegetableBagDaoImpl implements VegetableBagDao {
        @Autowired
        private VegetableBagRepository vegetableBagRepository;
        @Override
        public void saveVegetableBag(VegetableBagDetails vegetableBagDetails) {
                VegetableBag vegetableBag = new VegetableBag();
                BeanUtils.copyProperties(vegetableBagDetails, vegetableBag);
                vegetableBagRepository.save(vegetableBag);
        }
}
```

step5. Add the below annatation for scanning the repositoreis and entities.

```
  @EnableJpaRepositories(basePackages = { "com.test.vegetable.dao.repository" })
  @EntityScan(basePackages = "com.test.vegetable.dao.entity")
```

Note : For complete example click on below github link

https://github.com/apparao1289/marketservice

Transaction Management:

Spring provide convenient annotation for managing the transaction using @Transactional annotation.

@Transactional : It has following important attributes.

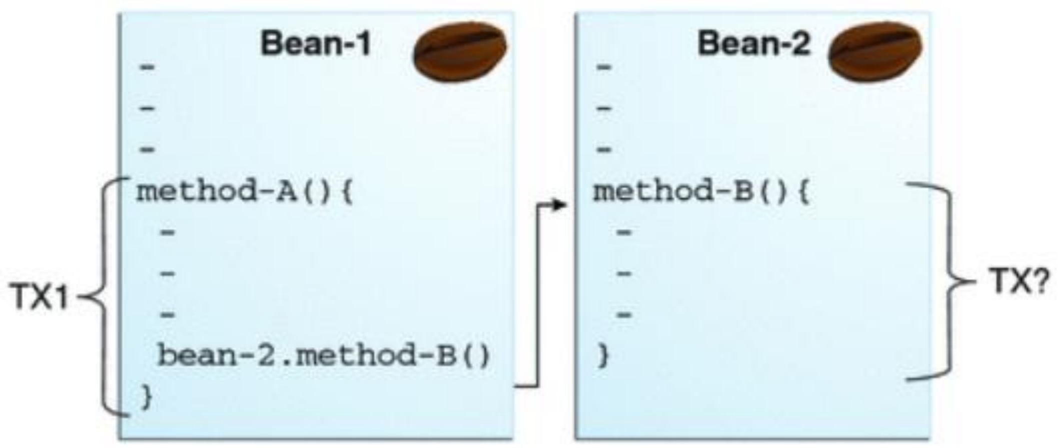
Isolation: The degree to which this transaction is isolated from the work of other transactions. For example, can this transaction see uncommitted writes from other transactions?

Propagation: Typically, all code executed within a transaction scope will run in that transaction. However, you have the option of specifying the behavior in the event that a transactional method is executed when a transaction context already exists. For example, code can continue running in the existing transaction (the common case); or the existing transaction can be suspended and a new transaction created. Spring offers all of the transaction propagation options familiar from EJB CMT.

Timeout: How long this transaction runs before timing out and being rolled back automatically by the underlying transaction infrastructure.

Read-only status: A read-only transaction can be used when your code reads but does not modify data. Read-only transactions can be a useful optimization in some cases, such as when you are using Hibernate.

A transaction attribute controls the scope of a transaction. Below illustrates why controlling the scope is important. In the diagram, method-A begins a transaction and then invokes method-B of Bean-2. When method-B executes, does it run within the scope of the transaction started by method-A, or does it execute with a new transaction? The answer depends on the transaction attribute of method-B.



A transaction attribute can have one of the following values:

Required

RequiresNew

Mandatory

NotSupported

Supports

Never

Required Attribute:

If the client is running within a transaction and invokes the spring bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container starts a new transaction before running the method.

Example : Propagation.REQUIRED

RequiresNew Attribute:

If the client is running within a transaction and invokes the spring bean's method, the container takes the following steps:

Suspends the client's transaction

Starts a new transaction

Delegates the call to the method

Resumes the client's transaction after the method completes

If the client is not associated with a transaction, the container starts a new transaction before running the method.

You should use the RequiresNew attribute when you want to ensure that the method always runs within a new transaction.
Example :

Propagation.REQUIRES_NEW

Mandatory Attribute:

If the client is running within a transaction and invokes the spring bean’s method, the method executes within the client’s transaction. If the client is not associated with a transaction, the container throws a TransactionRequiredException.

Use the Mandatory attribute if the spring bean’s method must use the transaction of the client.
Example : Propagation.MANDATORY

NotSupported Attribute:

If the client is running within a transaction and invokes the spring bean’s method, the container suspends the client’s transaction before invoking the method. After the method has completed, the container resumes the client’s transaction.

If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Use the NotSupported attribute for methods that don’t need transactions. Because transactions involve overhead, this attribute may improve performance.

Example : Propagation.NOT_SUPPORTED

Supports Attribute:

If the client is running within a transaction and invokes the spring bean’s method, the method executes within the client’s transaction. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Because the transactional behavior of the method may vary, you should use the Supports attribute with caution.

Example : Propagation.SUPPORTS

Never Attribute:

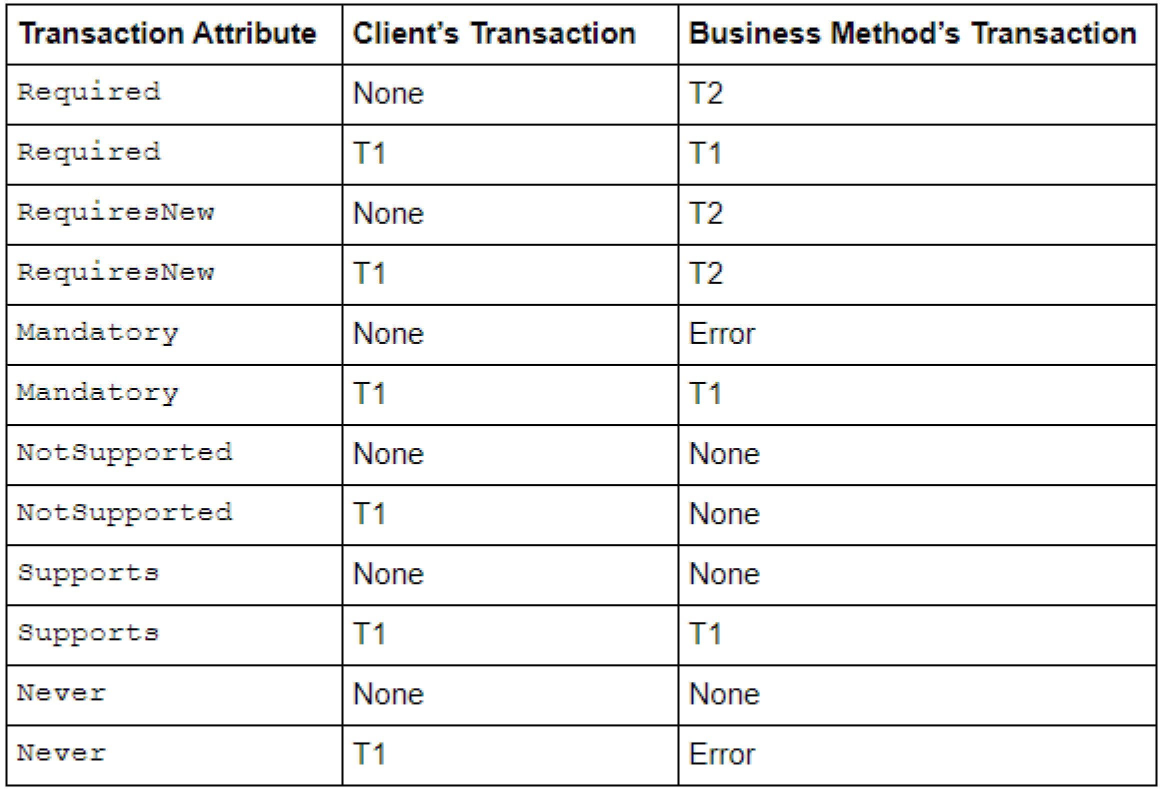
If the client is running within a transaction and invokes the spring bean’s method, the container throws a RemoteException. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Example : Propagation.NEVER

Summary of Transaction Attributes:

Below table summarizes the effects of the transaction attributes. Both the T1 and the T2 transactions are controlled by the container. A T1 transaction is associated with the client that calls a method in the enterprise bean. In most cases, the client is another enterprise bean. A T2 transaction is started by the container just before the method executes.

In the last column of below table, the word "None" means that the business method does not execute within a transaction controlled by the container. However, the database calls in such a business method might be controlled by the transaction manager of the database management system.



Transaction Attribute	Client's Transaction	Business Method's Transaction
`Required`	None	T2
`Required`	T1	T1
`RequiresNew`	None	T2
`RequiresNew`	T1	T2
`Mandatory`	None	Error
`Mandatory`	T1	T1
`NotSupported`	None	None
`NotSupported`	T1	None
`Supports`	None	None
`Supports`	T1	T1
`Never`	None	None
`Never`	T1	Error

Isolation Levels:

DEFAULT:

Use the default isolation level of the underlying datastore.

All other levels correspond to the JDBC isolation levels.

READ_UNCOMMITTED :

A constant indicating that dirty reads, non-repeatable reads and phantom reads can occur. This level allows a row changed by one transaction to be read by another transaction before any changes in that row have been committed (a "dirty read"). If any of the changes are rolled back, the second transaction will have retrieved an invalid row.

READ_COMMITTED: A constant indicating that dirty reads are prevented; non-repeatable reads and phantom reads can occur. This level only prohibits a transaction from reading a row with uncommitted changes in it.

REPEATABLE_READ: A constant indicating that dirty reads and non-repeatable reads are prevented; phantom reads can occur. This level prohibits a transaction from reading a row with uncommitted changes in it, and it also prohibits the situation where one transaction reads a row, a second transaction alters the row, and the first transaction rereads the row, getting different values the second time (a "non-repeatable read").

SERIALIZABLE: A constant indicating that dirty reads, non-repeatable reads and phantom reads are prevented. This level includes the prohibitions in {@code ISOLATION_REPEATABLE_READ} and further prohibits the situation where one transaction reads all rows that satisfy a {@code WHERE} condition, a second transaction inserts a row that satisfies that {@code WHERE} condition, and the first transaction rereads for the same condition, retrieving the additional "phantom" row in the second read.

The default @Transactional settings are as follows:

Propagation setting is PROPAGATION_REQUIRED.

Isolation level is ISOLATION_DEFAULT.

Transaction is read/write.

Transaction timeout defaults to the default timeout of the underlying transaction system, or to none if timeouts are not supported.

Any RuntimeException triggers rollback, and any checked Exception does not.

These default settings can be changed; the various properties of the @Transactional annotation are summarized in the following table.

Property	Type	Description
value	String	Optional qualifier specifying the transaction manager to be used.
propagation	enum: `Propagation`	Optional propagation setting.
`isolation`	enum: `Isolation`	Optional isolation level.
`readOnly`	boolean	Read/write vs. read-only transaction
`timeout`	int (in seconds granularity)	Transaction timeout.
`rollbackFor`	Array of `Class` objects, which must be derived from `Throwable.`	Optional array of exception classes that *must* cause rollback.
`rollbackForClassName`	Array of class names. Classes must be derived from `Throwable.`	Optional array of names of exception classes that *must* cause rollback.
`noRollbackFor`	Array of `Class` objects, which must be derived from `Throwable.`	Optional array of exception classes that *must not* cause rollback.
`noRollbackForClassName`	Array of `String` class names, which must be derived from `Throwable.`	Optional array of names of exception classes that *must not* cause rollback.

API Documentation using swagger:

Swagger UI allows anyone — be it your development team or your end consumers — to visualize and interact with the API’s resources without having any of the implementation logic in place.

It’s automatically generated from your OpenAPI (formerly known as Swagger) Specification, with the visual documentation making it easy for back end implementation and client side consumption.

1. Below is the sample swagger configurations class.

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {
		@Bean
		public Docket api() {
			return new Docket(DocumentationType.SWAGGER_2).select()

		.apis(RequestHandlerSelectors.basePackage("com.test.vegetable.controller")).paths(PathSelectors.any())
								.build().apiInfo(apiInfo());
		}


		private ApiInfo apiInfo() {
			return new ApiInfoBuilder().title("Vegetable API")
								.description("Below api's having all functionality related vegetable bag").build();
		}
}
```

2. Usage of the swagger annatations:

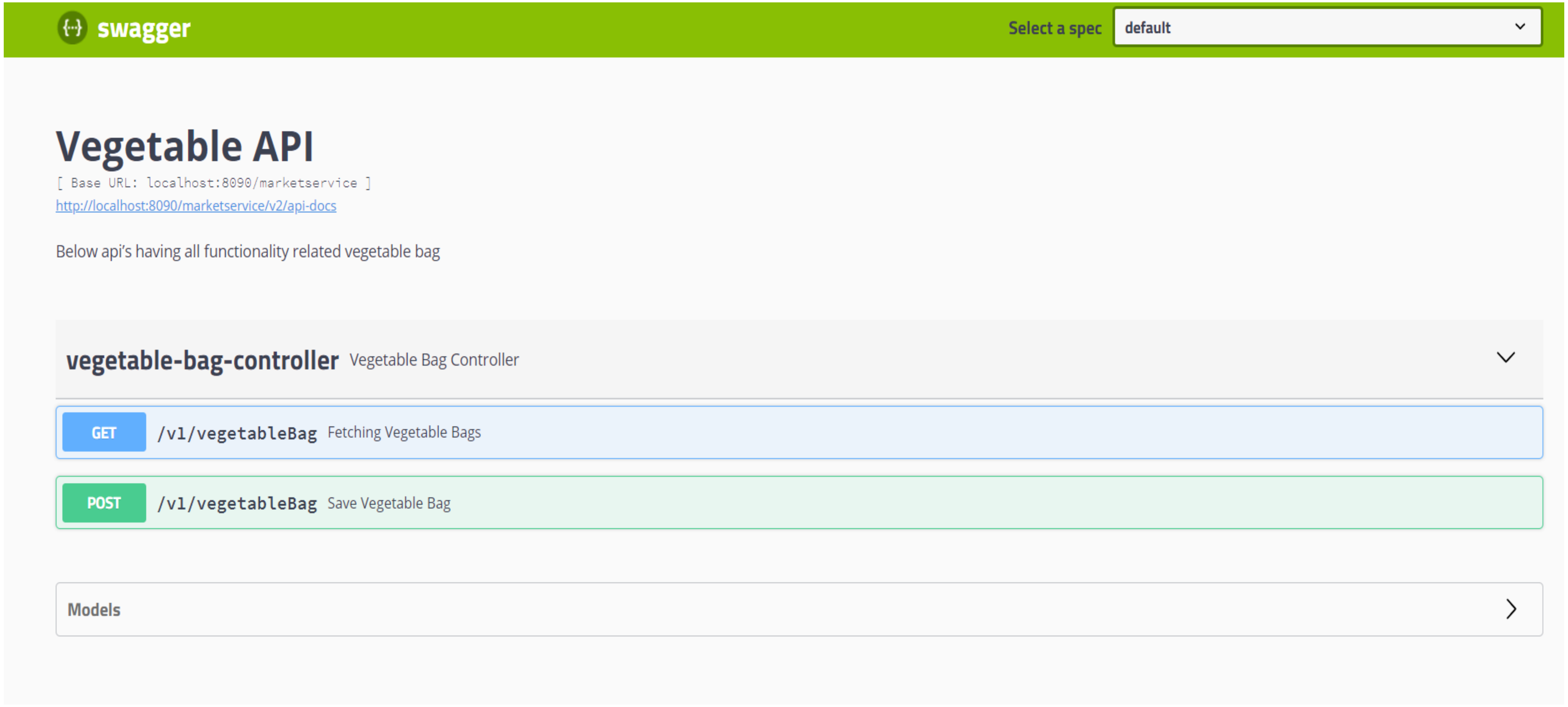
```
@RestController
@Api(value = "Vegetable Bag")
public class VegetableBagController {
        @RequestMapping(value = "/v1/vegetableBag", method = RequestMethod.GET, produces = MediaType.APPLICATION_JSON_VALUE)
        @ApiOperation(value = "Fetching Vegetable Bags", response = Response.class)
        @ApiImplicitParams({ @ApiImplicitParam(name = "pageNumber", value = "Page Number", required = false, dataType = "Integer", paramType = "query"),
                    @ApiImplicitParam(name = "pageSize", value = "Page Size", required = false, dataType = "Integer", paramType = "query") })
        public Response<List<VegetableBagDetails>> getVegetableBags(
                        @RequestParam(value = "pageNumber", required = false) final Integer pageNumber,
                        @RequestParam(value = "pageSize", required = false) final Integer pageSize) {
                Response<List<VegetableBagDetails>> response = new Response<List<VegetableBagDetails>>();
                try {
                        logger.info("getVegetableBags with pageNumber::" + pageNumber + "::pageSize::" + pageSize);
                        List<VegetableBagDetails> vegetableBagDetailsList = VegetableBagBusiness.getVegetableBags(pageNumber,
                                        pageSize);
                        response.setResult(vegetableBagDetailsList);
                        response.setMetaData(new MetaData(200, VegetableBagConstant.SUCCESS));
                } catch (Exception e) {
                }
                return response;
        }
}
```

3. Add following swagger dependencies in pom.xml file.

```
        <dependency>
                        <groupId>io.springfox</groupId>
                        <artifactId>springfox-swagger2</artifactId>
                        <version>${swagger.version}</version>
                </dependency>
                <dependency>
                        <groupId>io.springfox</groupId>
                        <artifactId>springfox-swagger-ui</artifactId>
                        <version>${swagger.version}</version>
        </dependency>
```

4. Swagger UI : Access the swagger link like below.

http://localhost:8090/marketservice/swagger-ui.html



Vegetable API

[Base URL: localhost:8090/marketservice]

http://localhost:8090/marketservice/v2/api-docs

Below api's having all functionality related vegetable bag

vegetable-bag-controller Vegetable Bag Controller

GET /v1/vegetableBag Fetching Vegetable Bags

POST /v1/vegetableBag Save Vegetable Bag

Models

Scheduling:

Spring provides @Scheduled annotation for scheduling the tasks.

@Scheduled annotation having below attributes.

fixedRate: which specifies the interval between method invocations, measured from the start time of each invocation.

Example: @Scheduled(fixedRate = 5000)

fixedDelay: which specifies the interval between invocations measured from the completion of the task.

Example : @Scheduled(fixedDelay = 5000)

cron: we can also use @Scheduled(cron=". . .") expressions for more sophisticated task scheduling.

Example:

```
@Component
public class ScheduledEmailTasks {

        private static final Logger log = LoggerFactory.getLogger(ScheduledEMailTasks.class);

        @Scheduled(fixedRate = 5000)
        public void processEmails() {
                log.info("Write email process logic here");
        }
}
```

Enable Scheduling:

@EnableScheduling annotation ensures that a background task executor is created. Without it, nothing gets scheduled.

Example:

```
@SpringBootApplication
@EnableScheduling
public class EmailTasksApplication {

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

Asynchronous calls configuration and invocation:

@Async annotation, indicating that it should run on a separate thread.

The @EnableAsync annotation switches on Spring’s ability to run @Async methods in a background thread pool. This class also customizes the Executor by defining a new bean. Here, the method is named taskExecutor, since this is the specific method name for which Spring searches

Example:

```
@SpringBootApplication

@EnableAsync

public class AsyncMethodApplication {

  public static void main(String[] args) {

    // close the application context to shut down the custom ExecutorService

    SpringApplication.run(AsyncMethodApplication.class, args).close();

  }

  @Bean

  public Executor taskExecutor() {

    ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();

    executor.setCorePoolSize(2);

    executor.setMaxPoolSize(2);

    executor.setQueueCapacity(500);

    executor.setThreadNamePrefix("TestTask-");

    executor.initialize();

    return executor;

  }

}
```

Externalized Configuration:

Spring Boot lets you externalize your configuration so that you can work with the same application code in different environments. You can use properties files, YAML files, environment variables, and command-line arguments to externalize configuration. Property values can be injected directly into your beans by using the @Value annotation, accessed through Spring’s Environment abstraction, or be bound to structured objects through @ConfigurationProperties.

Define the properties as key and value pairs with same prefix in application.properties file

```
mail.host=google.com
mail.port=8090
mail.from=test@gmail.com
```

Example 1:

```
@ConfigurationProperties(prefix = "mail")

public class MailConfigProperties {

   private String host;

   private int port;

   private String from;

   // setters and getters

}
```

Example 2:

```
mail.triggred.time=5000
```

We can read the above property using @Value annatation in java class.

```
@Value("${triggred.time}")

private Integer triggerTime;
```

Profile management:

Enterprise application development is complex. You have multiple environments

- Dev
- QA
- Preview
- Production

We want to have different application configuration in each of the environments.

1. Define environment specific application.properties files

    Example: application-dev.properties, application-prod.properties

2. Set the active profile for a specific environment in application.properties file

    Example: spring.profiles.active=prod

Spring Boot would pick up the application configuration based on the active profile that is set in a specific environment.

We would want to customize the application.properties for dev profile. We would need to create a file with name application-dev.properties and override the properties that we would want to customize.

application-dev.properties

mail.triggred.time=5000

Similarly you can configure properties for prod profile.

application-prod.properties

mail.triggred.time=8000

Once you have profile specific configuration, you would need to set the active profile in an environment.

There are multiple ways of doing this

- Using -Dspring.profiles.active=prod in VM Arguments
- Use spring.profiles.active=prod in application.properties

How to reload my changes on Spring Boot without having to restart server?

This can be achieved using DEV Tools. With this dependency any changes you save, the embedded tomcat will restart.

Spring Boot has a Developer tools (DevTools) module which helps to improve the productivity of developers. One of the key challenge for the Java developers is to auto deploy the file changes to server and auto restart the server.

Developers can reload changes on Spring Boot without having to restart my server. This will eliminates the need for manually deploying the changes every time.

This module will be disabled in the production environment. It also provides H2-database console for better testing the application.

Maven :

```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <optional>true</optional>
   </dependency>
</dependencies>
```

Gradle :

```
dependencies {
   developmentOnly("org.springframework.boot:spring-boot-devtools")
}
```

Actuators:

Spring boot actuator helps you to access the current state of the running application in production environment. There are several metrics that has to be checked and monitored in the production environment.

Even some external applications may be using those services to trigger the alert message to concerned person. Actuator module exposes set of REST endpoints that can be directly accessed as a HTTP URL to check the status.

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>
```

Actuator endpoints let you monitor and interact with your application. Spring Boot includes a number of built-in endpoints and lets you add your own. For example, the health endpoint provides basic application health information.

Each individual endpoint can be enabled or disabled and exposed (made remotely accessible) over HTTP or JMX. An endpoint is considered to be available when it is both enabled and exposed.

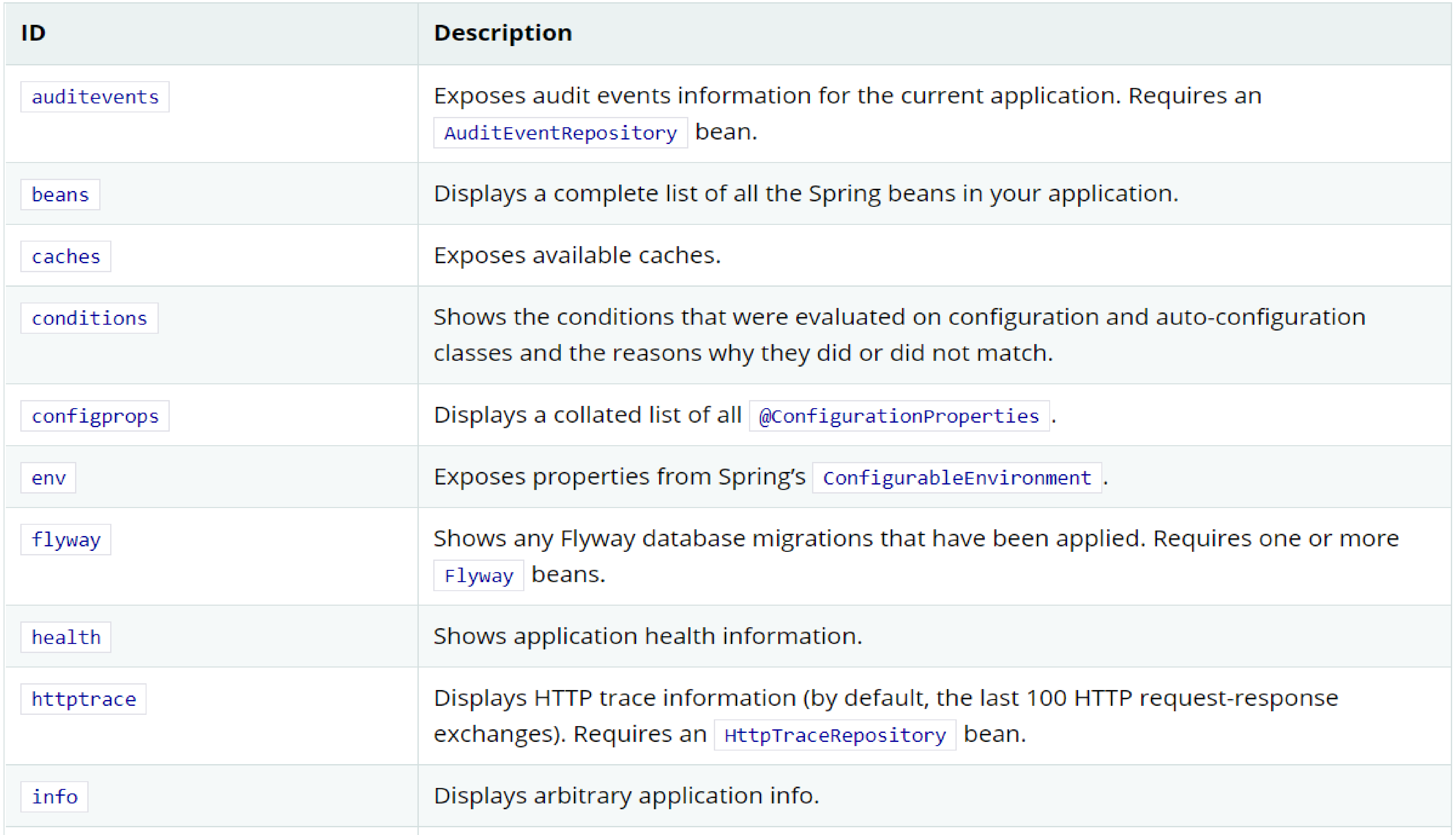
The built-in endpoints will only be auto-configured when they are available.

Most applications choose exposure via HTTP, where the ID of the endpoint along with a prefix of /actuator is mapped to a URL. For example, by default, the health endpoint is mapped to /actuator/health.

Example :

http://localhost:8090/marketservice/actuator/health

The following technology-agnostic endpoints are available:



ID	Description
`auditevents`	Exposes audit events information for the current application. Requires an `AuditEventRepository` bean.
`beans`	Displays a complete list of all the Spring beans in your application.
`caches`	Exposes available caches.
`conditions`	Shows the conditions that were evaluated on configuration and auto-configuration classes and the reasons why they did or did not match.
`configprops`	Displays a collated list of all `@ConfigurationProperties`.
`env`	Exposes properties from Spring's `ConfigurableEnvironment`.
`flyway`	Shows any Flyway database migrations that have been applied. Requires one or more `Flyway` beans.
`health`	Shows application health information.
`httptrace`	Displays HTTP trace information (by default, the last 100 HTTP request-response exchanges). Requires an `HttpTraceRepository` bean.
`info`	Displays arbitrary application info.

`integrationgraph`	Shows the Spring Integration graph. Requires a dependency on `spring-integration-core`.
`loggers`	Shows and modifies the configuration of loggers in the application.
`liquibase`	Shows any Liquibase database migrations that have been applied. Requires one or more `Liquibase` beans.
`metrics`	Shows ‘metrics’ information for the current application.
`mappings`	Displays a collated list of all `@RequestMapping` paths.
`scheduledtasks`	Displays the scheduled tasks in your application.
`sessions`	Allows retrieval and deletion of user sessions from a Spring Session-backed session store. Requires a Servlet-based web application using Spring Session.
`shutdown`	Lets the application be gracefully shutdown. Disabled by default.
`threaddump`	Performs a thread dump.

Enabling Endpoints:

By default, all endpoints except for shutdown are enabled. To configure the enablement of an endpoint, use its management.endpoint.<id>.enabled property. The following example enables the shutdown endpoint:

```
management.endpoint.shutdown.enabled=true
```

If you prefer endpoint enablement to be opt-in rather than opt-out, set the management.endpoints.enabled-by-default property to false and use individual endpoint enabled properties to opt back in. The following example enables the info endpoint and disables all other endpoints:

```
management.endpoints.enabled-by-default=false

management.endpoint.info.enabled=true
```

Filters:

- A filter is an object that is invoked at the preprocessing and postprocessing of a request.
- A filter is an object that performs filtering tasks on either the request to a resource (a servlet or static content), or on the response from a resource, or both.
- Filters perform filtering in the doFilter method. Every Filter has access to a FilterConfig object from which it can obtain its initialization parameters, a reference to the ServletContext which it can use, for example, to load resources needed for filtering tasks.
- Filters are configured in the deployment descriptor of a web application

Examples:

1) Authentication Filters

2) Logging and Auditing Filters

3) Image conversion Filters

4) Data compression Filters

5) Encryption Filters

6) Tokenizing Filters

7) Filters that trigger resource access events

8) XSL/T filters

9) Mime-type chain Filter

Example :

```
@Configuration
public class LoginFilter implements Filter {
private static final Logger LOGGER = LoggerFactory.getLogger(LoginFilter.class);

@Override  public void init(FilterConfig filterConfig) throws ServletException {
   LOGGER.info("Initiating LoginFilter ...");
}

@Override  public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws
IOException, ServletException {
   LOGGER.info("Pre processing logic..");
   filterChain.doFilter(request, response);
   LOGGER.info("Post process logic here..");
}

@Override  public void destroy() {
   LOGGER.info("Calling the destroy method..");
}


}
```

Reference Links:

Spring Data JPA:

https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#reference

Spring Boot:

https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/