Porting to Spring Boot 3 / Spring 6

jim stafford

Fall 2023 v2023-02-28: Built: 2024-04-18 13:08 EST

Table of Contents

Chapter 1. Introduction

This write-up documents key aspects encountered when porting the Ejava course examples from Spring Boot 2.7.x/Spring 5 to version 3.x/Spring 6.

1.1. Goals

The student will learn:

- specific changes required to port from Spring Boot 2.x to 3.x
- recognize certain error messages and map them to solutions

1.2. Objectives

At the conclusion of this lecture, the student will be able to:

1. update package dependencies

Chapter 2. Background

Spring Boot 3.0.0 (with Spring 6) was released in late Nov 2022. I initially ported the course examples from Spring Boot 2.7.0 (with Spring 5) to Spring Boot 3.0.2 (released Jan 20, 2023). Incremental releases of Spring Boot have been released in 2 to 4 week time periods since then. This writeup documents issues encountered — to include the initial signal of error and resolution taken.

Spring provides an official Migration Guide for Spring Boot 3 and Spring 6 that should be used as primary references.

The Spring Migration Guide identifies ways to enable some backward compatibility with Spring 5 or force upcoming compliance with Spring 6 with their `BeanInfoFactory` setting. I will not be discussing those options.

The change from Oracle (`javax*`) to Jakarta (`jakarta.*`) enterprise APIs presents the simplest but most pervasive changes in the repository. Although the change is trivial and annoying — the enterprise `javax.*` APIs are frozen. All new enterprise API features will be added to the `jakarta.*` flavor of the libraries from here forward.

Refs:

- spring-boot-dependencies:2.7.0
- spring-boot-dependencies:3.0.2

Chapter 3. Preparation

There were two primary recommendations in the migration guide that where luckily addressed in the existing repository.

- migrate to Spring Boot 2.7.x
- use Java 17

Spring Boot 2.7.0 contained some deprecations that were also immediately addressed that significantly helped speed up the transition:

- Deprecation of `WebSecurityConfigurerAdapter` in favor of Component-based Web Security `WebSecurityConfigurerAdapter` is now fully removed from Spring Boot 3/Spring 6.

Chapter 4. Dependency Changes

4.1. Spring Boot Version

The first and most obvious change was to change the `springboot.version` from `2.7.x` to `3.x`. This setting was in both ejava-build-bom (identifying dependencyManagement) and ejava-build-parent (identifying pluginManagement)

Spring Boot 2 Setting	*Spring Boot 3 Setting*
`<springboot.version>2.7.0</springboot.version>`	`<springboot.version>3.0.2</springboot.version>`

The version setting is used to import the targeted dependency definitions from `spring-boot-dependencies`.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>${springboot.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
```

4.2. JAXB DependencyManagement

The JAXB dependency definitions had to be spelled out in Spring 2.x like the following:

Spring Boot 2 JAXB dependencyManagement

```
<properties>
    <jaxb-api.version>2.3.1</jaxb-api.version>
    <jaxb-core.version>2.3.0.1</jaxb-core.version>
    <jaxb-impl.version>2.3.2</jaxb-impl.version>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>javax.xml.bind</groupId>
            <artifactId>jaxb-api</artifactId>
            <version>${jaxb-api.version}</version>
        </dependency>
        <dependency>
            <groupId>com.sun.xml.bind</groupId>
            <artifactId>jaxb-core</artifactId>
```

```
            <version>${jaxb-core.version}</version>
        </dependency>
        <dependency>
            <groupId>com.sun.xml.bind</groupId>
            <artifactId>jaxb-impl</artifactId>
            <version>${jaxb-impl.version}</version>
        </dependency>
```

However, Spring Boot 3.x `spring-boot-dependencies` BOM includes a `jaxb-bom` that takes care of the JAXB dependencyManagement for us.

ejava-build-bom.xml

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId> ①
            <version>${springboot.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
```

① ejava-build-bom imports spring-boot-dependencies

org.springframework.boot:spring-boot-dependencies-bom.xml

```
<properties>
    <glassfish-jaxb.version>4.0.1</glassfish-jaxb.version>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.glassfish.jaxb</groupId>
            <artifactId>jaxb-bom</artifactId> ①
            <version>${glassfish-jaxb.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
```

① spring-boot-dependencies imports jaxb-bom

The `jaxb-bom` defines the replacement for the JAXB API using the `jakarta` naming. It also defines two versions of the `com.sun.xml.bind:jaxb-impl`. One uses the "old" `com.sun.xml.bind:jaxb-impl` naming construct and the other uses the "new" `org.glassfish.jaxb:jaxb-runtine` naming construct.

org.glassfish.jaxb:jaxb-bom

```
<dependencyManagement>
    <dependencies>
```

```
        <dependency> <!--JAXB-API-->
            <groupId>jakarta.xml.bind</groupId>
            <artifactId>jakarta.xml.bind-api</artifactId> ①
            <version>${xml.bind-api.version}</version>
            <classifier>sources</classifier>
        </dependency>

        <!-- new -->
        <dependency>
            <groupId>org.glassfish.jaxb</groupId>
            <artifactId>jaxb-runtime</artifactId> ②
            <version>${project.version}</version>
            <classifier>sources</classifier>
        </dependency>

        <!--OLD-->
        <dependency>
            <groupId>com.sun.xml.bind</groupId>
            <artifactId>jaxb-impl</artifactId> ②
            <version>${project.version}</version>
            <classifier>sources</classifier>
        </dependency>
```

① jaxb-bom defines artifact for JAXB API

② jaxb-bom defines old and new versions of artifact for JAXB runtime implementation

I am assuming the two ("old" and "new") are copies of the same artifact — and updated all runtime dependencies to the "new" `org.glassfish.jaxb` naming scheme.

4.3. Groovy

Class examples use a limited amount of groovy for Spock test framework examples. Version 3.0.x of `org.codehaus.groovy:groovy` was explicitly specified in `ejava-build-bom`.

Spring Boot 2.x ejava-build-bom Groovy Definition

```
<properties>
    <groovy.version>3.0.8</groovy.version>
<dependencyManagement>
    <dependencies>
        <dependency> ①
            <groupId>org.codehaus.groovy</groupId>
            <artifactId>groovy</artifactId>
            <version>${groovy.version}</version>
        </dependency>
```

① legacy `ejava-build-parent` explicitly defined groovy dependency

I noticed after the fact that Spring Boot 2.7.0 defined a version of `org.codehaus.groovy:groovy` that would have made the above unnecessary.

However, the move to Spring Boot 3 also caused a move in groups for groovy — from org.codehaus.groovy to org.apache.groovy. The explicit dependency was removed from ejava-build-bom and dependencies updated to groupId `org.apache.groovy`.

Spring Boot 3 spring-boot-dependencies.pom

```
<properties>
    <groovy.version>4.0.7</groovy.version>
<dependencyManagement>
    <dependencies>
      <dependency> ①
            <groupId>org.apache.groovy</groupId>
            <artifactId>groovy-bom</artifactId>
            <version>${groovy.version}</version>
            <type>pom</type>
            <scope>import</scope>
      </dependency>
```

① `spring-boot-dependencies` imports `groovy-bom`

groovy-bom

```
<dependencyManagement>
    <dependencies>
      <dependency> ①
        <groupId>org.apache.groovy</groupId>
        <artifactId>groovy</artifactId>
        <version>4.0.7</version>
      </dependency>
```

① `groovy-bom` now used to explicitly define groovy dependency

```
[ERROR]     'dependencies.dependency.version' for org.codehaus.groovy:groovy:jar is
missing. @ info.ejava.examples.build:ejava-build-parent:6.1.0-SNAPSHOT,
/Users/jim/proj/ejava-javaee/ejava-springboot/build/ejava-build-parent/pom.xml, line
438, column 29
```

4.4. Spock

Using Spock test framework with Spring Boot 3.x requires the use of Spock version >= 2.4-M1 and groovy 4.0. I also noted that the `M1` version for `2.4` was required to work with `@SpringBootTest`.

Spring Boot 2 ejava-build-bom Spock property spec

```
<properties>
  <spock.version>2.0-groovy-
3.0</spock.version>
```

Spring Boot 3 ejava-build-bom Spock property spec

```
<properties>
  <spock.version>2.4-M1-groovy-
4.0</spock.version>
```

The above property definition is seamlessly used to define the necessary dependencies in the following snippet in order to use Spock test framework.

ejava-build-bom Spock Definition

```
<properties>
  <spock.version>...</spock.version>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.spockframework</groupId>
            <artifactId>spock-bom</artifactId>
            <version>${spock.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>org.spockframework</groupId>
            <artifactId>spock-spring</artifactId>
            <version>${spock.version}</version>
            <scope>test</scope>
        </dependency>
```

4.5. Flapdoodle

Direct support for the Flapdoodle embedded Mongo database was removed from Spring Boot 3, but can be manually brought in with the following definition for "spring30x".

Spring Boot 3.x ejava-build-bom

```
<properties>
    <flapdoodle.spring30x.version>4.5.2</flapdoodle.spring30x.version>
<dependencyManagement>
    <dependencies>
        <dependency> ①
            <groupId>de.flapdoodle.embed</groupId>
            <artifactId>de.flapdoodle.embed.mongo.spring30x</artifactId>
            <version>${flapdoodle.spring30x.version}</version>
        </dependency>
```

① `spring-boot-dependencies` no longer defines the flapdoodle dependency. New Spring Boot 3.x flapdoodle dependency now defined by `ejava-build-bom`

Of course the dependency declaration groupId must be changed from `de.flapdoodle.embed.mongo` to `de.flapdoodle.embed.mongo.spring30x` in the child projects as well.

Spring Boot 2.x Flapdoodle Declaration

```
<dependency>
    <groupId>de.flapdoodle.embed</groupId>
```

```
    <artifactId>de.flapdoodle.embed.mongo</artifactId>
    <scope>test</scope>
</dependency>
```

Spring Boot 3.x Flapdoodle Spring 30x Declaration

```
<dependency>
    <groupId>de.flapdoodle.embed</groupId>
    <artifactId>de.flapdoodle.embed.mongo.spring30x</artifactId>
    <scope>test</scope>
</dependency>
```

4.6. HttpClient / SSL

The ability to define property features to outgoing HTTP connections requires use of the `org.apache.httpcomponents` libraries.

Spring Boot 2.x HttpComponents Dependency Definition

```
<properties>
        <httpclient.version>4.5.13</httpclient.version>
<dependencyManagement>
    <dependencies>
        <dependency> ①
            <groupId>org.apache.httpcomponents</groupId>
            <artifactId>httpclient</artifactId>
            <version>${httpclient.version}</version>
        </dependency>
```

① Spring Boot 2.x `spring-boot-dependencies` defined dependency on `httpclient`

Spring Boot 3 has migrated to "client5" version of the libraries. The older version gets replaced with the following. The dependencyManagement definition for httpclient(anything) can be removed from our local `ejava-build-bom`.

Spring Boot 3.x HttpComponents Client 5 Dependency Definition

```
<properties>
    <httpclient5.version>5.1.4</httpclient5.version>
<dependencyManagement>
    <dependencies>
        <dependency> ①
            <groupId>org.apache.httpcomponents.client5</groupId>
            <artifactId>httpclient5</artifactId>
            <version>${httpclient5.version}</version>
        </dependency>
```

① Spring Boot 3.x `spring-boot-dependencies` defines dependency on `httpclient5`

However, `httpclient5` requires a secondary library to configure SSL connections. `ejava-build-bom` now defines the dependency for that.

Spring Boot 3.x HttpComponents Client 5 Dependency Definition - ejava-build-bom

```
<properties>
    <sslcontext-kickstart.version>7.4.9</sslcontext-kickstart.version>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.github.hakky54</groupId> ①
            <artifactId>sslcontext-kickstart-for-apache5</artifactId>
            <version>${sslcontext-kickstart.version}</version>
        </dependency>
```

① `ejava-build-bom` defines necessary dependency to configure `httpclient5` SSL connections

4.7. Javax /Jakarta Artifact Dependencies

With using the spring-boot-starters, there are very few direct dependencies on enterprise artifacts. However, for the direct API references — simply change the `javax.*` groupId to `jakarta.*`.

Spring Boot 2.x API Dependency Definition

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-
api</artifactId>
  <scope>provided</scope>
</dependency>
```

Spring Boot 3.x API Dependency Definition

```
<dependency>
  <groupId>jakarta.servlet</groupId>
  <artifactId>jakarta.servlet-
api</artifactId>
  <scope>provided</scope>
</dependency>
```

4.8. jakarta.inject

`javax.inject` does not have a straight replacement and is not defined within the Spring Boot BOM. Replace any `javax.inject` dependencies with `jakarta.inject-api`.

javax.inject-api

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

jakarta.inject-api

```
<dependency>
    <groupId>jakarta.inject</groupId>
```

```
    <artifactId>jakarta.inject-api</artifactId>
    <version>2.0.1</version>
</dependency>
```

Since `spring-boot-dependencies` does not provide a dependencyManagement entry for inject — it was difficult to determine which version would be best appropriate. I went with `2.0.1` and added to the `ejava-build-bom`.

ejava-build-bom: jakarta.inject-api

```
<properties>
    <jakarta.inject-api.version>2.0.1</jakarta.inject-api.version>
<build>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>jakarta.inject</groupId>
                <artifactId>jakarta.inject-api</artifactId>
                <version>${jakarta.inject-api.version}</version>
            </dependency>
```

4.9. ActiveMQ / Artemis Dependency Changes

ActiveMQ does not yet support `jakarta` packaging, but its `artemis` sibling does. Modify all local pom dependency definitions to the `artemis` variant to first get things compiling. Dependency management will be taken care of by the Spring Boot Dependency POM.

Spring Boot 2.x ActiveMQ Dependency - local child pom.xml

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

Spring Boot 3.x Artemis Dependency - local child pom.xml

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

Chapter 5. Package Changes

Once maven artifact dependencies are addressed, resolvable Java package imports can be put in place. `javax` APIs added to the Java language (e.g., `javax.sql.*`) are still in the `javax` structure. `javax` APIs that are part of independent Enterprise APIs have been moved to `jakarta`.

The table below summarizes the APIs encountered in the EJava course examples repository. Whether through command-line (e.g., `find`, `sed`) or IDE search/replace commands — it is well worth the time to identify a global way to make these mindless `javax.(package)` to `jakarta.(package)` package name changes.

Spring Boot 2.x Enterprise Package Imports	*Spring Boot 3.x Enterprise Package Imports*

`import javax.annotation.*;`
`import javax.persistence.*;`
`import javax.jms.*;`
`import javax.servlet.*;`
`import javax.validation.*;`
`import javax.xml.bind.annotation.*;`
	`import jakarta.annotation.*;`
	`import jakarta.jms.*;`
	`import jakarta.persistence.*;`
	`import jakarta.servlet.*;`
	`import jakarta.validation.*;`
	`import jakarta.xml.bind.annotation.*;`

For example, the following bash script will locate all Java source files with `import javax` and change those occurrences to `import jakarta`.

Baseline changes prior to bulk file changes

Baseline all in-progress changes prior to making bulk file changes so that you can easily revert to the previous state.

change occurrences of "import javax" with "import jakarta"

```
$ for file in `find . -name "*.java" -exec grep -l 'import javax' {} \;`; do sed -i ''
's/import javax/import jakarta/' $file; done
```

However, not all `javax` packages are part of JavaEE. We now need to execute the following to correct `javax.sql`, `javax.json`, and `javax.net`, imports caught up in the mass change.

revert occurrences of "import jakarta" back to "import javax"

```
$ for file in `find . -name "*.java" -exec grep -l 'import jakarta.sql' {} \;`; do sed
-i '' 's/import jakarta.sql/import javax.sql/' $file; done

$ for file in `find . -name "*.java" -exec grep -l 'import jakarta.json' {} \;`; do
sed -i '' 's/import jakarta.json/import javax.json/' $file; done

$ for file in `find . -name "*.java" -exec grep -l 'import jakarta.net' {} \;`; do sed
-i '' 's/import jakarta.net/import javax.net/' $file; done
```

Chapter 6. AssertJ Template Changes

AssertJ test assertion library has the ability to generate type-specific assertions. However, some of the generated classes make reference to deprecated `javax.*` packages ...

AssertJ assertion generator, build-in template

```
@javax.annotation.Generated(value="assertj-assertions-generator") ①
public class BddAssertions extends org.assertj.core.api.BDDAssertions {
    ...
```

① `javax` package name prefix must be renamed

... and must be updated to `jakarta.*`.

Required template modification

```
@jakarta.annotation.Generated(value="assertj-assertions-generator") ①
public class BddAssertions extends org.assertj.core.api.BDDAssertions {
    ...
```

① `jakarta` package name prefix must be used in place of `javax`

However, AssertJ assertions generator releases have been idle since Feb 2021 (version `2.2.1`) and our only option is to manually edit the templates ourself.

The testing with AssertJ assertions lecture notes covers how to customize the generator.

Review: Assertj assertion generator template customizations

```
$ ls app/app-testing/apptesting-testbasics-example/src/test/resources/templates/ |
sort
ejava_bdd_assertions_entry_point_class_template.txt ①
```

① template was defined for custom type

Review: Assertj assertion generator maven configuration

```
<!-- generate custom AssertJ assertions -->
<plugin>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-assertions-generator-maven-plugin</artifactId>
    <configuration>
        <classes> ①
            <param>info.ejava.examples.app.testing.testbasics.Person</param>
        </classes>
        <templates>
            <!-- local customizations -->
            <templatesDirectory>
${basedir}/src/test/resources/templates/</templatesDirectory>
```

```
<bddEntryPointAssertionClass>ejava_bdd_assertions_entry_point_class_template.txt</bddE
ntryPointAssertionClass>
        </templates>
    </configuration>
</plugin>
```

① custom template and type was declared with `AssertJ` plugin

The following listing show we can host downloaded and modified template files.

Modified AssertJ generator plugin template files

```
$ ls app/app-testing/apptesting-testbasics-example/src/test/resources/templates/ |
sort
ejava_bdd_assertions_entry_point_class_template.txt
jakarta_bdd_soft_assertions_entry_point_class_template.txt
jakarta_custom_abstract_assertion_class_template.txt
jakarta_custom_assertion_class_template.txt
jakarta_custom_hierarchical_assertion_class_template.txt
jakarta_junit_soft_assertions_entry_point_class_template.txt
jakarta_soft_assertions_entry_point_class_template.txt
jakarta_standard_assertions_entry_point_class_template.txt
```

The following snippet shows how we can configure the plugin to use the additional custom template files.

Declared modified AssertJ generator plugin template fiels

```
<!-- Spring Boot 3.x / AspectJ jakarta customizations -->
<!-- https://github.com/assertj/assertj-assertions-generator-maven-plugin/issues/93
-->
<assertionClass>jakarta_custom_assertion_class_template.txt</assertionClass>
<assertionsEntryPointClass>jakarta_standard_assertions_entry_point_class_template.txt<
/assertionsEntryPointClass>
<hierarchicalAssertionAbstractClass>jakarta_custom_abstract_assertion_class_template.t
xt</hierarchicalAssertionAbstractClass>
<hierarchicalAssertionConcreteClass>jakarta_custom_hierarchical_assertion_class_templa
te.txt</hierarchicalAssertionConcreteClass>
<softEntryPointAssertionClass>jakarta_soft_assertions_entry_point_class_template.txt</
softEntryPointAssertionClass>
<junitSoftEntryPointAssertionClass>jakarta_junit_soft_assertions_entry_point_class_tem
plate.txt</junitSoftEntryPointAssertionClass>
```

Chapter 7. Spring Boot Configuration Property

@ConstructorBinding is used to designate how to populate the properties object with values. In Spring Boot 2.x, the annotation could be applied to the class or constructor.

Spring Boot 2 ConfigurationProperties Constructor Binding Option

```
import org.springframework.boot.context.properties.ConstructorBinding;

@ConstructorBinding ①
public class AddressProperties {
private final String street;
    public AddressProperties(String street, String city, String state, String zip) {
... }
```

① annotation could be applied to class or constructor

In Spring Boot 3.x, the annotation has been moved one Java package level lower — into the bind package — and the new definition can only be legally applied to constructors.

Spring Boot 3 ConfigurationProperties Constructor Binding Option

```
import org.springframework.boot.context.properties.bind.ConstructorBinding; ①

public class AddressProperties {
private final String street;
    @ConstructorBinding ②
    public AddressProperties(String street, String city, String state, String zip) {
... }
```

① annotation moved to new package

② annotation can only be applied to a specific constructor

However, it is technically only needed when there are multiple constructors.

@ConstructorBinding is only needed when having multiple constructors

```
@ConstructorBinding //only required for multiple constructors
public BoatProperties(String name) {
    this.name = name;
}
//not used for ConfigurationProperties initialization
public BoatProperties() { this.name = "default"; }
```

Chapter 8. HttpStatus

`HttpStatus` represents the status returned from an HTTP call. Responses are primarily in the 1xx, 2xx, 3xx, 4xx, and 5xx ranges with some well-known values.

When updating to Spring Boot 3, you may encounter the following compilation problem:

HttpStatus compilation error

```
incompatible types: org.springframework.http.HttpStatusCode cannot be converted to
org.springframework.http.HttpStatus
```

8.1. Spring Boot 2.x

Spring Boot 2.x used an Enum type to represent these well-known values and properties and all interfaces accepted and returned that enum type.

Spring 5.x HttpStatus enum

```
public enum HttpStatus {
    OK(200, Series.SUCCESSFUL, "OK")
    CREATED(201, Series.SUCCESSFUL, "Created"),
...
```

The following are two code examples for acquiring an HttpStatus object:

Getting HttpStatus enum from ClientHttpResponse

```
public class ResponseEntity<T> extends HttpEntity<T> {
    public HttpStatus getStatusCode() {

- - -
ClientHttpResponse response = ...
HttpStatus status = response.getStatusCode();
```

Getting HttpStatus enum from ClientHttpResponse and HttpStatusCodeException

```
//when
HttpStatus status;
try {
    status = homesClient.hasHome("anId").getStatusCode();
} catch (HttpStatusCodeException ex) {
    status = ex.getStatusCode();
}
```

The following is an example of inspecting the legacy HttpStatus object.

Inspecting HttpStatus enum

```
then(response.getStatusCode().series()).isEqualTo(HttpStatus.Series.SUCCESSFUL);
```

The problem was that the HttpStatus enum could not represent custom HTTP status values.

8.2. Spring Boot 3.x

Spring 6 added a breaking change by having methods accept and return a new `HttpStatusCode` interface. The `HttpStatus` enum now implements that interface but cannot be directly resolved to an `HttpStatus` without an additional lookup.

Spring 6 HttpStatus Implements HttpStatusCode

```
public enum HttpStatus implements HttpStatusCode {
    OK(200, Series.SUCCESSFUL, "OK")
    CREATED(201, Series.SUCCESSFUL, "Created"),
...
```

One needs to call a lookup method to convert to an `HttpStatus` instance if the `HttpStatusCode` object is needed. Use `resolve()` if null is acceptable (e.g., log statements) and `valueOf()` if required.

Getting HttpStatus type from ClientHttpResponse

```
public class ResponseEntity<T> extends HttpEntity<T> {
    public HttpStatusCode getStatusCode() {

- - -
ClientHttpResponse response = ...
HttpStatus status = HttpStatus.resolve(response.getStatusCode().value()); ①
//or
HttpStatus status = HttpStatus.valueOf(response.getStatusCode().value()); ②
```

① returns null if value is not resolved

② throws IllegalArgumentException is value is not resolved

Getting HttpStatusCode enum from ClientHttpResponse and HttpStatusCodeException

```
HttpStatusCode status;
try {
    status = homesClient.hasHome("anId").getStatusCode();
} catch (HttpStatusCodeException ex) {
    status = ex.getStatusCode();
}
```

Accessing HttpStatus methods

```
then(response.getStatusCode().is2xxSuccessful()).isTrue();
```

Most of the same information is available — just not as easy to get to.

Chapter 9. HttpMethod

The common values for HttpMethod are also very well-known.

9.1. Spring Boot 2.x

Spring Boot 2.x used an enum to represent these well-known values and associated properties.

Spring Boot 2.x/Spring 5 HttpMethod as an Enum

```
public enum HttpMethod {
    GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS, TRACE;
...
```

Leveraging HttpMethod enum Definition for @ParameterizedTest value source

```
@ParameterizedTest
@EnumSource(value = HttpMethod.class, names = {"GET", "POST"})
void anonymous_user(HttpMethod method) {
```

The rigid aspects of the enum made it not usable for custom HTTP methods.

9.2. Spring Boot 3.x

Spring Boot 3.x changed HttpMethod from an enum to a regular class as well.

```
public final class HttpMethod implements Comparable<HttpMethod>, Serializable {
    public static final HttpMethod GET = new HttpMethod("GET");
    public static final HttpMethod POST = new HttpMethod("POST");
...
```

The following shows the `@ParameterizedTest` from above, updated to account for the change. @EnumSource was changed to `@CsvSource` and the provided String was converted to an HttpMethod type within the method.

Leveraging HttpMethod String values and Conversion for @ParameterizedTest value source

```
@ParameterizedTest
@ValuesSource(strings={"GET","POST"})
void anonymous_user(String methodName) {
    HttpMethod method = HttpMethod.valueOf(methodName);
```

Chapter 10. Spring Factories Changes

The location for AutoConfiguration bootstrap classes has changed from the general-purpose `META-INF/spring.factories`...

META-INF/spring.factories

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
  info.ejava.examples.app.hello.HelloAutoConfiguration, \
  info.ejava.examples.app.hello.HelloResourceAutoConfiguration
```

to the bootstrap-specific `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` file

META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports

```
info.ejava.examples.app.hello.HelloAutoConfiguration
info.ejava.examples.app.hello.HelloResourceAutoConfiguration
```

The same information is conveyed in the `import` file — just expressed differently.

`META-INF/spring.factories` still exists. It is no longer used to express this information.

Chapter 11. Spring WebSecurityConfigurerAdapter

Spring had deprecated `WebSecurityConfigurerAdapter` by the time Spring Boot 2.7.0 was released.

Spring Boot 2 Deprecated WebSecurityConfigurerAdapter

```
@Configuration
@Order(100)
@ConditionalOnClass(WebSecurityConfigurerAdapter.class)
public static class SwaggerSecurity extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatchers(cfg->cfg
                .antMatchers("/swagger-ui*", "/swagger-ui/**", "/v3/api-docs/**"));
        http.csrf().disable();
        http.authorizeRequests(cfg->cfg.anyRequest().permitAll());
    }
}
```

The deprecated `WebSecurityConfigurerAdapter` approach was replaced with the Component-based `@Bean` factory mechanism. Spring 6 has completely eliminated the adapter approach.

Spring Boot Component-based

```
@Bean
@Order(100)
public SecurityFilterChain swaggerSecurityFilterChain(HttpSecurity http) throws
Exception {
        http.securityMatchers(cfg->cfg
                .requestMatchers("/swagger-ui*", "/swagger-ui/**", "/v3/api-docs/**")
);
        http.csrf().disable();
        http.authorizeHttpRequests(cfg->cfg.anyRequest().permitAll());
        return http.build();
}
```

11.1. SecurityFilterChain securityMatcher

One or more request matchers can be applied to the SecurityFilterChain, primarily for the cases when there are multiple SecurityFilterChains. Lacking a request matcher — the highest priority SecurityFilterChain will likely process all requests. For Spring Boot 2.x/Spring 5, this was expressed with a `requestMatchers()` builder call on the injected `HttpSecurity` object.

```
@Bean
@Order(50)
public SecurityFilterChain h2Configuration(HttpSecurity http) throws Exception {
```

```
    http.requestMatchers(cfg->...); ①
    ...
```

① `SecurityFilterChain.requestMatchers()` determined what filter chain will process

In Spring Boot 3/Spring 6, the request matcher for the SecurityFilterChain is now expressed with a `securityMatchers()` call. They function the same with a different name to help distinguish the call from the ones made to configure `RequestMatcher`.

```
@Bean
@Order(50)
public SecurityFilterChain h2Configuration(HttpSecurity http) throws Exception {
    http.securityMatchers(cfg->...); ①
    ...
```

① `securityMatchers()` replaces `requestMatchers()` for `SecurityFilterChain`

A simple search for `requestMatchers` and replace with `securityMatchers` is a suitable solution.

Search/Replace requestMatchers with securityMatchers Builder

```
for file in `find . -name "*.java" -exec grep -l 'requestMatchers(' {} \;`; do sed -i
'' 's/requestMatchers(/securityMatchers(/' $file; done
```

11.2. antMatchers/requestMatchers

The details of the `RequestMatcher` for both the `SecurityFilterChain` and `WebSecurityCustomizer` were defined by a `antMatchers()` builder. The `mvcMatchers()` builder also existed, but were not used in the course examples.

```
@Bean
@Order(50)
public SecurityFilterChain h2Configuration(HttpSecurity http) throws Exception {
    http.requestMatchers(cfg->cfg.antMatchers( ①
        "/h2-console/**","/login","/logout"));
    ...
@Bean
public WebSecurityCustomizer authzStaticResources() {
    return (web) -> web.ignoring().antMatchers( ①
        "/content/**");
}
```

① legacy `antMatchers()` defined details of legacy `requestMatchers()`

Documentation states that legacy `antMatchers()` have simply been replaced with `requestMatchers()` and then warn that `/foo` matches no longer match `/foo/` URIs. One must explicitly express `/foo` and `/foo/` to make that happen.

```
@Bean
@Order(50)
public SecurityFilterChain h2Configuration(HttpSecurity http) throws Exception {
    http.securityMatchers(cfg->cfg.requestMatchers(①
        "/h2-console/**","/login","/logout"));
    ...
@Bean
public WebSecurityCustomizer authzStaticResources() {
    return (web) -> web.ignoring().requestMatchers( ①
        "/content/**");
}
```

① `requestMatchers()` now defines the match details

In reality, the `requestMatchers()` will resolve to the `mvcMatchers()` when using WebMVC and that is simply how the mvcMatchers() work. I assume that is what Spring Security wants you to use. Otherwise the convenient alternate builders would not have been removed or at least the instructions would have more prominently identified how to locate the explicit builders in the new API.

Using antMatcher anb regexMatcher

```
import org.springframework.security.web.util.matcher.AntPathRequestMatcher;
import org.springframework.security.web.util.matcher.RegexRequestMatcher;

http.authorizeHttpRequests(cfg->cfg
    .requestMatchers(AntPathRequestMatcher.antMatcher("...")).hasRole("...")
    .requestMatchers(RegexRequestMatcher.regexMatcher("...")).hasRole("...")
    .requestMatchers("/h2-console/**").authenticated()); //MvcRequestMatcher
```

A simple search and replace can be performed for this update as long as `mvcMatchers()` is a suitable solution.

Search/Replace antMatchers with requestMatchers Builder

```
for file in `find . -name "*.java" -exec grep -l 'antMatchers(' {} \;`; do sed -i ''
's/antMatchers(/requestMatchers(/' $file; done
```

11.3. ignoringAntMatchers/ignoringRequestMatchers

The same is true for the ignoring case. Just replace the `ignoringAntMatchers()` builder method with `ignoringRequestMatchers()`.

Replace ignoringAntMatchers with ignoringRequestMatchers Builder

```
http.csrf(cfg->cfg.ignoringAntMatchers("/h2-console/**"));
...
```

```
http.csrf(cfg->cfg.ignoringRequestMatchers("/h2-console/**"));
```

Search/Replace ignoringAntMatchers with ignoringRequestMatchers Builder

```
for file in `find . -name "*.java" -exec grep -l 'ignoringAntMatchers(' {} \;`; do sed
-i '' 's/ignoringAntMatchers(/ignoringRequestMatchers(/' $file; done
```

11.4. authorizeRequests/authorizeHttpRequests

Spring Boot 2.x/Spring 5 used the `authorizeRequests()` builder to define access restrictions for a URI.

```
http.authorizeRequests(cfg->cfg.requestMatchers(
        "/api/whoami", "/api/authorities/paths/anonymous/**").permitAll());
```

The builder still exists, but has been deprecated for `authorizeHttpRequests()`.

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
        "/api/whoami", "/api/authorities/paths/anonymous/**").permitAll());
```

A simple search and replace can address this issue.

Search/Replace authorizeRequests with authorizeHttpRequests Builder

```
for file in `find . -name "*.java" -exec grep -l 'authorizeRequests(' {} \;`; do sed
-i '' 's/authorizeRequests(/authorizeHttpRequests(/' $file; done
```

Chapter 12. Role Hierarchy

Early Spring Security 3.x omission left off automatic support for role inheritance.

12.1. Spring Boot 2.x Role Inheritance

The following shows the seamless integration of role access constraints and role hierarchy definition for security mechanisms that support hierarchies.

```
@Bean
public RoleHierarchy roleHierarchy() {
    RoleHierarchyImpl roleHierarchy = new RoleHierarchyImpl();
    roleHierarchy.setHierarchy(StringUtils.join(List.of(
            "ROLE_ADMIN > ROLE_CLERK",
            "ROLE_CLERK > ROLE_CUSTOMER"
    ),System.lineSeparator()));
    return roleHierarchy;
```

```
http.authorizeRequests(cfg->cfg.antMatchers(
        "/api/authorities/paths/customer/**")
        .hasAnyRole("CUSTOMER"));
http.authorizeRequests(cfg->cfg.antMatchers(HttpMethod.GET,
        "/api/authorities/paths/price")
        .hasAnyAuthority("PRICE_CHECK", "ROLE_ADMIN", "ROLE_CLERK"));
```

12.2. Spring Boot 3.x Role Inheritance

The role hierarchies are optionally stored within an AuthorizationManager. Early Spring Boot 3 left that automatic registration out but was available in an up-coming merge request. An interim solution was to manually supply the `SecurityFilterChain` an `AuthorizationManager` pre-registered with a `RoleHierarchy` definition.

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
        "/api/authorities/paths/customer/**")
        .access(anyRoleWithRoleHierarchy(roleHierarchy, "CUSTOMER"))
);
http.authorizeHttpRequests(cfg->cfg.requestMatchers(HttpMethod.GET,
        "/api/authorities/paths/price")
        .access(anyAuthorityWithRoleHierarchy(roleHierarchy, "PRICE_CHECK",
"ROLE_ADMIN", "ROLE_CLERK"))
);
```

The following snippets show the definition of the `RoleHierarchy` injected into the `SecurityChainFilter` builder. Two have been defined — one for `roleInheritance` profile and one for otherwise.

```
@Bean
@Profile("roleInheritance")
public RoleHierarchy roleHierarchy() {
    RoleHierarchyImpl roleHierarchy = new RoleHierarchyImpl();
    roleHierarchy.setHierarchy(StringUtils.join(List.of(
            "ROLE_ADMIN > ROLE_CLERK",
            "ROLE_CLERK > ROLE_CUSTOMER"
    ),System.lineSeparator()));
    return roleHierarchy;
}

@Bean
@Profile("!roleInheritance")
public RoleHierarchy nullHierarchy() {
    return new NullRoleHierarchy();
}
```

Temporary Builder code to supply Injected Authorization Manager

```
//temporary work-around until this fix is available
//https://github.com/spring-projects/spring-security/issues/12473
private AuthorizationManager anyRoleWithRoleHierarchy(RoleHierarchy roleHierarchy,
String...roles) {
    AuthorityAuthorizationManager<Object> authzManager =
AuthorityAuthorizationManager.hasAnyRole(roles);
    authzManager.setRoleHierarchy(roleHierarchy);
    return authzManager;
}
private AuthorizationManager anyAuthorityWithRoleHierarchy(RoleHierarchy
roleHierarchy, String...authorities) {
    AuthorityAuthorizationManager<Object> authzManager =
AuthorityAuthorizationManager.hasAnyAuthority(authorities);
    authzManager.setRoleHierarchy(roleHierarchy);
    return authzManager;
}
```

Chapter 13. Annotated Method Security

`@EnableGlobalMethodSecurity` has been renamed to `@EnableMethodSecurity` and `prePostEnabled` has been enabled by default.

Spring Boot 2.x @EnableGlobalMethodSecurity

```
@EnableGlobalMethodSecurity(prePostEnabled = true) ①
public class AuthoritiesTestConfiguration {
```

① `prePostEnabled` had to be manually enabled

Spring Boot 3.x @EnableMethodSecurity

```
@EnableMethodSecurity //(prePostEnabled = true) now default
public class AuthoritiesTestConfiguration {
}
```

A simple search and replace solution should be enough to satisfy the deprecation.

Search/Replace EnableGlobalMethodSecurity with EnableMethodSecurity Annotation

```
for file in `find . -name "*.java" -exec grep -l 'EnableGlobalMethodSecurity(' {} \;`;
do sed -i '' 's/EnableGlobalMethodSecurity(/EnableMethodSecurity(/' $file; done
```

Chapter 14. @Secured

Spring Boot 3.x `@Secured` annotation now supports non-ROLE authorities

```
@Secured({"ROLE_ADMIN", "ROLE_CLERK", "PRICE_CHECK"})
@GetMapping(path = "price", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> checkPrice(
```

Chapter 15. JSR250 RolesAllowed

Spring Boot 2.x Jsr250 `ROLE`-s started with the `ROLE_` prefix when defined. Permissions (`PRICE_CHECK`) did not.

15.1. Spring Boot 2.x

Spring Boot 2.x Method Security using @Secured

```
@RolesAllowed("ROLE_CLERK")
public ResponseEntity<String> doClerk(

@RolesAllowed("ROLE_CUSTOMER")
public ResponseEntity<String> doCustomer(

@RolesAllowed({"ROLE_ADMIN", "ROLE_CLERK", "PRICE_CHECK"})
public ResponseEntity<String> checkPrice(
```

15.2. Spring Boot 3.x

Spring Boot 3.x Jsr250 `ROLE`-s definition no longer start with `ROLE_` prefix — just like Permissions (`PRICE_CHECK`).

Spring Boot 3.x Method Security using @Secured

```
@RolesAllowed("CLERK")
public ResponseEntity<String> doClerk(

@RolesAllowed("CUSTOMER")
public ResponseEntity<String> doCustomer(

@RolesAllowed({"ADMIN", "CLERK", "PRICE_CHECK"})
public ResponseEntity<String> checkPrice(
```

Chapter 16. HTTP Client

Lower-level client networking details for `RestTemplate` is addressed using HTTP Client. This primarily includes TLS (still referred to as SSL) but can also include other features like caching and debug logging.

16.1. Spring Boot 2 HTTP Client

Spring Boot 2 used `httpclient`. The following snippet shows how the TLS could be optionally configured for HTTPS communications.

HttpClient and SSLContext imports

```
import org.apache.http.client.HttpClient;
import org.apache.http.impl.client.HttpClientBuilder;
import org.apache.http.ssl.SSLContextBuilder;
import javax.net.ssl.SSLContext;
```

We first need to establish an `SSLContext` with the definition of protocols and an optional trustStore. The trustStore is optional to communicate with globally trusted sites, but necessary if we communicate using HTTPS with self-generated certs.

The following Spring Boot 2 example, uses an injected definition of the external server to load the trustStore and build the `SSLContext`.

Building Customized SSLContext

```
@Bean
public SSLContext sslContext(ServerConfig serverConfig)  {
    try {
        URL trustStoreUrl = null;
        if (serverConfig.getTrustStore()!=null) {
            trustStoreUrl = ClientITConfiguration.class.getResource("/" +
serverConfig.getTrustStore());
            if (null==trustStoreUrl) {
                throw new IllegalStateException("unable to locate truststore:/" +
serverConfig.getTrustStore());
            }
        }
        SSLContextBuilder builder = SSLContextBuilder.create()
                .setProtocol("TLSv1.2");
        if (trustStoreUrl!=null) {
            builder.loadTrustMaterial(trustStoreUrl, serverConfig
.getTrustStorePassword());
        }
        return builder.build();
    } catch (Exception ex) {
        throw new IllegalStateException("unable to establish SSL context", ex);
    }
```

```
}
```

The `SSLContext` and remote server definition are used to build a `HttpClient` to insert into the `ClientRequestFactory` used to establish client connections.

Building Customized ClientHttpRequestFactory with TLS Capability

```
@Bean
public ClientHttpRequestFactory httpsRequestFactory(SSLContext sslContext,
                                                    ServerConfig serverConfig) {
    HttpClient httpsClient = HttpClientBuilder.create()
            .setSSLContext(serverConfig.isHttps() ? sslContext : null)
            .build();
    return new HttpComponentsClientHttpRequestFactory(httpsClient);
}
```

16.2. Spring Boot 3.x HttpClient5

Spring Boot updated the networking in `RestTemplate` to use `httpclient5` and a custom SSL Context library.

HttpClient5 and SSL Factory imports

```
import nl.altindag.ssl.SSLFactory;
import nl.altindag.ssl.util.Apache5SslUtils;
import org.apache.hc.client5.http.classic.HttpClient;
import org.apache.hc.client5.http.impl.classic.HttpClients;
import org.apache.hc.client5.http.impl.io.PoolingHttpClientConnectionManager;
import org.apache.hc.client5.http.impl.io.PoolingHttpClientConnectionManagerBuilder;
import org.springframework.boot.autoconfigure.condition.ConditionalOnExpression;
```

`httpclient5` uses a `SSLFactory` TLS definition that is similar to its `httpclient` counterpart. The biggest difference in the `@Bean` factory in the example code is that we have decided to disable the bean if the `ServerConfig` trustStore property is empty.

Building Customized SSLContext

```
@Bean ①
@ConditionalOnExpression("!T(org.springframework.util.StringUtils).isEmpty('${it.serve
r.trust-store:}')")
public SSLFactory sslFactory(ServerConfig serverConfig) throws IOException {
    try (InputStream trustStoreStream = Thread.currentThread()

.getContextClassLoader().getResourceAsStream(serverConfig.getTrustStore())) {
        if (null==trustStoreStream) {
            throw new IllegalStateException("unable to locate truststore: " +
serverConfig.getTrustStore());
        }
        return SSLFactory.builder()
```

```
            .withProtocols("TLSv1.2")
            .withTrustMaterial(trustStoreStream, serverConfig.getTrustStorePassword())
            .build();
    }
}
```

① SSLFactory will not be created when `it.server.trust-store` is empty

With our design change, we then make the injected `SSLFactory` into the `ClientRequestFactory` @Bean method optional. From there we use `httpsclient5` constructs to build the proper components.

Building Customized ClientHttpRequestFactory with TLS Capability

```
@Bean
public ClientHttpRequestFactory httpsRequestFactory(
        @Autowired(required = false) SSLFactory sslFactory) { ①
    PoolingHttpClientConnectionManagerBuilder builder =
        PoolingHttpClientConnectionManagerBuilder.create();
    PoolingHttpClientConnectionManager connectionManager =
        Optional.ofNullable(sslFactory)
            .map(sf -> builder.setSSLSocketFactory(Apache5SslUtils.toSocketFactory(sf
)))
            .orElse(builder)
            .build();

    HttpClient httpsClient = HttpClients.custom()
            .setConnectionManager(connectionManager)
            .build();
    return new HttpComponentsClientHttpRequestFactory(httpsClient);
}
```

① SSLFactory defined to be optional and checked for null during ConnectionManager creation

Note that `httpclient5` and its TLS extensions require two new dependencies.

httpclient5 and TLS dependencies

```
<dependency>
    <groupId>org.apache.httpcomponents.client5</groupId>
    <artifactId>httpclient5</artifactId>
</dependency>
<dependency>
    <groupId>io.github.hakky54</groupId>
    <artifactId>sslcontext-kickstart-for-apache5</artifactId>
</dependency>
```

The caching extensions caching extensions are made available through the following dependency. Take a look at CachingHttpClientBuilder.

```
<dependency>
```

```
    <groupId>org.apache.httpcomponents.client5</groupId>
    <artifactId>httpclient5-cache</artifactId>
</dependency>
```

Chapter 17. Subject Alternative Name (SAN)

Java HTTPS has always had a hostname check that verified the reported hostname matched the DN within the SSL certificate. For local testing, that use to mean only having to supply a `CN=localhost`. Now, the SSL security matches against the subject alternative name ("SAN").

We will get the following error when the service we are calling using HTTPS returns a certificate that does not list a valid subject alternative name (SAN) consistent with the hostname used to connect.

Subject Alternative Name Error

```
ResourceAccess I/O error on GET request for "https://localhost:63848/api/authn/hello":
Certificate for <localhost> doesn't match any of the subject alternative names: []
```

A valid subject alternative name (SAN) can be generated with the `-ext` parameter within keytool.

Using keytool to supply Subject Alternative Name (SAN)

```
#https://stackoverflow.com/questions/50928061/certificate-for-localhost-doesnt-match-
any-of-the-subject-alternative-names
#https://ultimatesecurity.pro/post/san-certificate/

keytool -genkeypair -keyalg RSA -keysize 2048 -validity 3650 \
-ext "SAN:c=DNS:localhost,IP:127.0.0.1" \①
-dname "CN=localhost,OU=Unknown,O=Unknown,L=Unknown,ST=Unknown,C=Unknown" \
-keystore keystore.p12  -alias https-hello \
-storepass password
```

① clients will accept `localhost` or `127.0.0.1` returned from the SSL connection provided by this trusted certificate

Chapter 18. Swagger Changes

18.1. Spring Doc

18.1.1. Spring Boot 2.x

Spring Doc supported Spring Boot 2.x with their 1.x version.

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-ui</artifactId>
    <version>1.6.9</version>
</dependency>
```

18.2. Spring Boot 3.x

Spring Doc supports Spring Boot 3.x with their 2.x version.

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.1.0</version>
</dependency>
```

Chapter 19. JPA Dependencies

19.1. Spring Boot 3.x/Hibernate 6.x

Spring Boot 3.x/Hibernate 6.x requires a dependency on a Validator.

Hibernate 6.x requires Validator

```
jakarta.validation.NoProviderFoundException: Unable to create a Configuration, because
no Jakarta Bean Validation provider could be found. Add a provider like Hibernate
Validator (RI) to your classpath.
```

To correct, add the validation starter.

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

Chapter 20. JPA Default Sequence

One mechanism for generating a primary key value is to use a sequence.

20.1. Spring Boot 2.x/Hibernate 5.x

Spring Boot 2.x/Hibernate 5.x used to default the sequence to `hibernate_sequence`.

Example Sequence Generator Definition without a Name

```
@Entity
public class Song {
    @Id @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private int id;
```

Spring Boot2.x/Hibernate 5.x Defaulted to hibernate_sequence

```
enum Dialect {
    H2("call next value for hibernate_sequence"),
    POSTGRES("select nextval('hibernate_sequence')");
```

```
drop sequence IF EXISTS hibernate_sequence;
create sequence hibernate_sequence start with 1 increment 1;
```

20.2. Spring Boot 3.x/Hibernate 6.x

Spring Boot 3.x/Hibernate 6.x no longer permit an unnamed sequence generator. It must be named.

Default increment has changed

The default increment for the sequence has also changed from 1 to 50.

Example Named Sequence Generator Definition

```
@Entity
public class Song {
    @Id @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
"REPOSONGS_SONG_SEQUENCE")
    private int id;
```

Spring Boot3.x/Hibernate 6.x with Named Sequence

```
enum Dialect {
    H2("call next value for REPOSONGS_SONG_SEQUENCE"),
    POSTGRES("select nextval('REPOSONGS_SONG_SEQUENCE')");
```

```
drop sequence IF EXISTS REPOSONGS_SONG_SEQUENCE;
create sequence REPOSONGS_SONG_SEQUENCE start with 1 increment 50; ①
```

① default increment is 50

Chapter 21. JPA Property Changes

21.1. Spring Boot 2.x/Hibernate 5.x

The legacy JPA persistence properties used a `javax` prefix.

Spring Boot 2.x/Hibernate 5.x JPA javax Property Prefix

```
spring.jpa.properties.javax.persistence.schema-generation.create-source=metadata
spring.jpa.properties.javax.persistence.schema-generation.scripts.action=drop-and-
create
spring.jpa.properties.javax.persistence.schema-generation.scripts.create-
target=target/generated-sources/ddl/drop_create.sql
spring.jpa.properties.javax.persistence.schema-generation.scripts.drop-
target=target/generated-sources/ddl/drop_create.sql
```

21.2. Spring Boot 3.x/Hibernate 6.x

With Spring Boot 3.x/Hibernate 6.x, the property prefix has changed to `jakarta`.

Spring Boot 3.x/Hibernate 6.x JPA jakarta Property Prefix

```
spring.jpa.properties.jakarta.persistence.schema-generation.create-source=metadata
spring.jpa.properties.jakarta.persistence.schema-generation.scripts.action=drop-and-
create
spring.jpa.properties.jakarta.persistence.schema-generation.scripts.create-
target=target/generated-sources/ddl/drop_create.sql
spring.jpa.properties.jakarta.persistence.schema-generation.scripts.drop-
target=target/generated-sources/ddl/drop_create.sql
```

Chapter 22. Embedded Mongo

22.1. Embedded Mongo AutoConfiguration

Spring Boot 3.x has removed direct support for Flapdoodle in favor of configuring it yourself or using `testcontainers`.

The legacy `EmbeddedMongoAutoConfiguration` can now be found in a flapdoodle package.

Spring Boot 2.x Flapdoodle AutoConfiguration

```
import
org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration;
```

Spring Boot 3.x Flapdoodle AutoConfiguration

```
import de.flapdoodle.embed.mongo.spring.autoconfigure.EmbeddedMongoAutoConfiguration;
```

22.2. Embedded Mongo Properties

The mandatory `mongodb.embedded.version` property has been renamed

- from: `spring.mongodb.embedded.version`
- to: `de.flapdoodle.mongodb.embedded.version`

It works exactly the same as before.

Spring Boot 2.x Mandatory Flapdoodle Property

```
spring.mongodb.embedded.version=4.4.0
```

Spring Boot 3.x Mandatory Flapdoodle Property

```
de.flapdoodle.mongodb.embedded.version=4.4.0
```

A simple search and replace of property files addresses this change. YAML file changes would have been more difficult.

```
for file in `find . -name "*.properties" -exec egrep -l
'spring.mongodb.embedded.version' {} \;`; do sed -i ''
's/spring.mongodb.embedded.version/de.flapdoodle.mongodb.embedded.version/' $file;
done
```

Chapter 23. ActiveMQ/Artemis

ActiveMQ and Artemis are two branches within the ActiveMQ baseline. I believe Artemis came from a JBoss background. Only Artemis has been updated to support `jakarta` constructs.

23.1. Spring Boot 2.x

The following snippet shows the Maven dependency for ActiveMQ, with its `javax.jms` support.

ActiveMQ Dependency

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

The following snippet shows a few example ActiveMQ properties used in the simple example within the course tree.

Example ActiveMQ Properties

```
spring.activemq.broker-url=tcp://activemq:61616

spring.activemq.in-memory=true
spring.activemq.pool.enabled=false
```

23.2. Spring Boot 3.x

The following snippet shows the Maven dependencies for Artemis and its `jakarta.jms` support. The Artemis server dependency had to be separately added in order to run an embedded JMS server. JMSTemplate also recommended the Pooled JMS dependency to tune the use of connections.

Artemis Dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-artemis</artifactId>
</dependency>
<!-- jmsTemplate connection polling -->
<dependency>
    <groupId>org.messaginghub</groupId>
    <artifactId>pooled-jms</artifactId>
</dependency>
<!-- dependency added a runtime server to allow running with embedded topic -->
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>artemis-jakarta-server</artifactId>
```

```
</dependency>
```

The following snippet shows a few example Artemis properties and configuring the JMS connection pool.

Artemis Replacement Properties

```
spring.artemis.broker-url=tcp://activemq:61616

#requires org.messaginghub:pooled-jms dependency
#https://activemq.apache.org/spring-support
spring.artemis.pool.enabled=true
spring.artemis.pool.max-connections=5
```

Chapter 24. Summary

In this module we:

- Identified dependency definition and declaration changes between Spring Boot 2.x and 3.x
- Identified code changes required to migrate from Spring Boot 2.x to 3.x

With a Spring Boot 3/Spring 6 baseline, we can now move forward with some of the latest changes in the Spring Boot ecosystem.