

(/)

Probar un trabajo de Spring Batch

Última modificación: 27 de octubre de 2019

por [baeldung](https://www.baeldung.com/author/baeldung/) (<https://www.baeldung.com/author/baeldung/>)

Primavera (<https://www.baeldung.com/category/spring/>) +

Pruebas (<https://www.baeldung.com/category/testing/>)

Lote de primavera (<https://www.baeldung.com/tag/spring-batch/>)

Acabo de anunciar el nuevo curso *Learn Spring*, centrado en los fundamentos de Spring 5 y Spring Boot 2:

>> VER EL CURSO (</ls-course-start>)

1. Introducción

A diferencia de otras aplicaciones basadas en Spring, probar trabajos por lotes presenta algunos desafíos específicos, principalmente debido a la naturaleza asincrónica de cómo se ejecutan los trabajos.

En este tutorial, vamos a explorar las diversas alternativas para probar un trabajo Spring Batch (<https://www.baeldung.com/introduction-to-spring-batch>).

2. Dependencias requeridas

Estamos usando *spring-boot-starter-batch*

(<https://search.maven.org/classic/#search%7Cga%7C1%7Cspring-boot-starter-batch>) , así que primero configuremos las dependencias requeridas en nuestro *pom.xml*:

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-batch</artifactId>
4   <version>2.1.9.RELEASE</version>
5 </dependency>
6 <dependency>
7   <groupId>org.springframework.boot</groupId>
8   <artifactId>spring-boot-starter-test</artifactId>
9   <version>2.1.9.RELEASE</version>
10  <scope>test</scope>
11 </dependency>
12 <dependency>
13   <groupId>org.springframework.batch</groupId>
14   <artifactId>spring-batch-test</artifactId>
15   <version>4.2.0.RELEASE</version>
16   <scope>test</scope>
17 </dependency>
```

Incluimos la *prueba* (<https://search.maven.org/classic/#search%7Cga%7C1%7Ca%3A%22spring-boot-starter-test%22>) *spring-boot-starter-test*

(<https://search.maven.org/classic/#search%7Cga%7C1%7Ca%3A%22spring-boot-starter-test%22>) y *spring-batch-test* (<https://search.maven.org/classic/#search%7Cga%7C1%7Ca%3A%22spring-batch-test%22>) que aportan algunos métodos auxiliares, oyentes y corredores necesarios para probar las aplicaciones Spring Batch.

3. Definiendo el trabajo de Spring Batch

Creemos una aplicación simple para mostrar cómo Spring Batch resuelve algunos de los desafíos de las pruebas.

Nuestra aplicación utiliza una de dos pasos *de empleo* que lee un archivo CSV de entrada con información estructurada y salidas libro libros y detalles del libro.

3.1. Definiendo los pasos del trabajo

Los dos *pasos* siguientes extraen información específica de *BookRecords* y luego los asignan a *Books* (paso1) y *BookDetails* (paso2):

```

1  @Bean
2  public Step step1(
3      ItemReader<BookRecord> csvItemReader, ItemWriter<Book> jsonItemWriter) throws IOException {
4      return stepBuilderFactory
5          .get("step1")
6          .<BookRecord, Book> chunk(3)
7          .reader(csvItemReader)
8          .processor(bookItemProcessor())
9          .writer(jsonItemWriter)
10         .build();
11 }
12
13 @Bean
14 public Step step2(
15     ItemReader<BookRecord> csvItemReader, ItemWriter<BookDetails> listItemWriter) {
16     return stepBuilderFactory
17         .get("step2")
18         .<BookRecord, BookDetails> chunk(3)
19         .reader(csvItemReader)
20         .processor(bookDetailsItemProcessor())
21         .writer(listItemWriter)
22         .build();
23 }

```

3.2. Definición del lector de entrada y el escritor de salida

Configuremos ahora el lector de entrada de archivos CSV usando un *FlatFileItemReader* para **deserializar** la información estructurada del libro en objetos *BookRecord*:

```

1  private static final String[] TOKENS = {
2      "bookname", "bookauthor", "bookformat", "isbn", "publishyear" };
3
4  @Bean
5  @StepScope
6  public FlatFileItemReader<BookRecord> csvItemReader(
7      @Value("#{jobParameters['file.input']}") String input) {
8      FlatFileItemReaderBuilder<BookRecord> builder = new FlatFileItemReaderBuilder<>();
9      FieldSetMapper<BookRecord> bookRecordFieldSetMapper = new BookRecordFieldSetMapper();
10     return builder
11         .name("bookRecordItemReader")
12         .resource(new FileSystemResource(input))
13         .delimited()
14         .names(TOKENS)
15         .fieldSetMapper(bookRecordFieldSetMapper)
16         .build();
17 }

```

Hay un par de cosas importantes en esta definición, que tendrán implicaciones en la forma en que probamos.

En primer lugar, **anotamos el bean *FlatItemReader* con *@StepScope*** y, como resultado, **este objeto compartirá su vida útil con *StepExecution***.

Esto también nos permite inyectar valores dinámicos en tiempo de ejecución para que podamos pasar nuestro archivo de entrada desde *JobParameters* en la línea 4. Por el contrario, los tokens utilizados para *BookRecordFieldSetMapper* se configuran en tiempo de compilación.

Luego definimos de manera similar el escritor de salida *JsonFileItemWriter*:

```

1  @Bean
2  @StepScope
3  public JsonFileItemWriter<Book> jsonItemWriter(
4      @Value("#{jobParameters['file.output']}") String output) throws IOException {
5      JsonFileItemWriterBuilder<Book> builder = new JsonFileItemWriterBuilder<>();
6      JacksonJsonObjectMarshaller<Book> marshaller = new JacksonJsonObjectMarshaller<>();
7      return builder
8          .name("bookItemWriter")
9          .jsonObjectMarshaller(marshaller)
10         .resource(new FileSystemResource(output))
11         .build();
12 }

```

Utilizamos cookies para mejorar su experiencia con el sitio. Para obtener más información, puede leer la [Política de privacidad y cookies completa \(/privacy-policy/\)](/privacy-policy/)

Ok

Para el segundo *paso*, usamos un *ListItemWriter* proporcionado por Spring Batch que simplemente volca cosas en una lista en memoria.

3.3. Definir el *JobLauncher* personalizado

A continuación, desactivemos la configuración predeterminada de inicio de *trabajos* de Spring Boot Batch estableciendo *spring.batch.job.enabled = false* en nuestra *application.properties*.

Configuramos nuestro propio *JobLauncher* para pasar una instancia personalizada de *JobParameters* al iniciar el *trabajo*:

```
1  @SpringBootApplication
2  public class SpringBatchApplication implements CommandLineRunner {
3
4      // autowired jobLauncher and transformBooksRecordsJob
5
6      @Value("${file.input}")
7      private String input;
8
9      @Value("${file.output}")
10     private String output;
11
12     @Override
13     public void run(String... args) throws Exception {
14         JobParametersBuilder paramsBuilder = new JobParametersBuilder();
15         paramsBuilder.addString("file.input", input);
16         paramsBuilder.addString("file.output", output);
17         jobLauncher.run(transformBooksRecordsJob, paramsBuilder.toJobParameters());
18     }
19
20     // other methods (main etc.)
21 }
```

4. Prueba del trabajo de Spring Batch

La dependencia *spring-batch-test* proporciona un conjunto de métodos y escuchas útiles que pueden usarse para configurar el contexto Spring Batch durante las pruebas.

Creemos una estructura básica para nuestra prueba:

```

1  @RunWith(SpringRunner.class)
2  @SpringBatchTest
3  @EnableAutoConfiguration
4  @ContextConfiguration(classes = { SpringBatchConfiguration.class })
5  @TestExecutionListeners({ DependencyInjectionTestExecutionListener.class,
6      DirtiesContextTestExecutionListener.class})
7  @DirtiesContext(classMode = ClassMode.AFTER_CLASS)
8  public class SpringBatchIntegrationTest {
9
10     // other test constants
11
12     @Autowired
13     private JobLauncherTestUtils jobLauncherTestUtils;
14
15     @Autowired
16     private JobRepositoryTestUtils jobRepositoryTestUtils;
17
18     @After
19     public void cleanUp() {
20         jobRepositoryTestUtils.removeJobExecutions();
21     }
22
23     private JobParameters defaultJobParameters() {
24         JobParametersBuilder paramsBuilder = new JobParametersBuilder();
25         paramsBuilder.addString("file.input", TEST_INPUT);
26         paramsBuilder.addString("file.output", TEST_OUTPUT);
27         return paramsBuilder.toJobParameters();
28     }

```

La anotación **@SpringBatchTest** proporciona las clases auxiliares **JobLauncherTestUtils** y **JobRepositoryTestUtils**. Los usamos para activar el *trabajo* y los *pasos* en nuestras pruebas.

Nuestra aplicación utiliza la **configuración automática de Spring Boot** (<https://www.baeldung.com/spring-boot-annotations>), que habilita un **JobRepository** predeterminado en memoria. Como resultado, **ejecutar múltiples pruebas en la misma clase requiere un paso de limpieza después de cada ejecución de prueba**.

Finalmente, **si queremos ejecutar múltiples pruebas desde varias clases de prueba, debemos marcar nuestro contexto como sucio** (<https://www.baeldung.com/spring-dirtiescontext>). Esto es necesario para evitar el choque de varias instancias de **JobRepository** que usan la misma fuente de datos.

4.1. Prueba de la de extremo a extremo de empleo

Lo primero que vamos a probar es una completa de extremo a extremo *de empleo* con una pequeña entrada de conjunto de datos.

Luego podemos comparar los resultados con una salida de prueba esperada:

```

1  @Test
2  public void givenReferenceOutput_whenJobExecuted_thenSuccess() throws Exception {
3      // given
4      FileSystemResource expectedResult = new FileSystemResource(EXPECTED_OUTPUT);
5      FileSystemResource actualResult = new FileSystemResource(TEST_OUTPUT);
6
7      // when
8      JobExecution jobExecution = jobLauncherTestUtils.launchJob(defaultJobParameters());
9      JobInstance actualJobInstance = jobExecution.getJobInstance();
10     ExitStatus actualJobExitStatus = jobExecution.getExitStatus();
11
12     // then
13     assertThat(actualJobInstance.getJobName(), is("transformBooksRecords"));
14     assertThat(actualJobExitStatus.getExitCode(), is("COMPLETED"));
15     AssertFile.assertFileEquals(expectedResult, actualResult);
16 }

```

Spring Batch Test proporciona un **método útil de comparación de archivos para verificar los resultados utilizando la clase AssertFile**.

Utilizamos cookies para mejorar su experiencia con el sitio. Para obtener más información, puede leer la [Política de privacidad y cookies completa \(/privacy-policy/\)](/privacy-policy/)

Ok

4.2. Probar pasos individuales

A veces es bastante costoso probar el *trabajo* completo de principio a fin, por lo que tiene sentido probar los *pasos* individuales en su lugar:

```

1  @Test
2  public void givenReferenceOutput_whenStep1Executed_thenSuccess() throws Exception {
3      // given
4      FileSystemResource expectedResult = new FileSystemResource(EXPECTED_OUTPUT);
5      FileSystemResource actualResult = new FileSystemResource(TEST_OUTPUT);
6
7      // when
8      JobExecution jobExecution = jobLauncherTestUtils.launchStep(
9          "step1", defaultJobParameters());
10     Collection actualStepExecutions = jobExecution.getStepExecutions();
11     ExitStatus actualJobExitStatus = jobExecution.getExitStatus();
12
13     // then
14     assertThat(actualStepExecutions.size(), is(1));
15     assertThat(actualJobExitStatus.getExitCode(), is("COMPLETED"));
16     AssertFile.assertFileEquals(expectedResult, actualResult);
17 }
18
19 @Test
20 public void whenStep2Executed_thenSuccess() {
21     // when
22     JobExecution jobExecution = jobLauncherTestUtils.launchStep(
23         "step2", defaultJobParameters());
24     Collection actualStepExecutions = jobExecution.getStepExecutions();
25     ExitStatus actualExitStatus = jobExecution.getExitStatus();
26
27     // then
28     assertThat(actualStepExecutions.size(), is(1));
29     assertThat(actualExitStatus.getExitCode(), is("COMPLETED"));
30     actualStepExecutions.forEach(stepExecution -> {
31         assertThat(stepExecution.getWriteCount(), is(8));
32     });
33 }

```

Tenga en cuenta que **utilizamos el método *launchStep* para activar pasos específicos**.

Recuerde que **también diseñamos nuestro *ItemReader* y *ItemWriter* para usar valores dinámicos en tiempo de ejecución**, lo que significa **que podemos pasar nuestros parámetros de E / S a *JobExecution*** (líneas 9 y 23).

Para la primera prueba de *Paso*, comparamos la salida real con la salida esperada.

Por otro lado, **en la segunda prueba, verificamos el *StepExecution* para los elementos escritos esperados**.

4.3. Prueba de componentes de ámbito escalonado

Probemos ahora el *FlatFileItemReader*. **Recuerde que lo *expusimos* como *@StepScope* bean, por lo que **querremos usar el soporte dedicado de Spring Batch para esto**:**

Utilizamos cookies para mejorar su experiencia con el sitio. Para obtener más información, puede leer la [Política de privacidad y cookies completa \(/privacy-policy\)](#)

Ok

```

1  // previously autowired itemReader
2
3  @Test
4  public void givenMockedStep_whenReaderCalled_thenSuccess() throws Exception {
5      // given
6      StepExecution stepExecution = MetadataInstanceFactory
7          .createStepExecution(defaultJobParameters());
8
9      // when
10     StepScopeTestUtils.doInStepScope(stepExecution, () -> {
11         BookRecord bookRecord;
12         itemReader.open(stepExecution.getExecutionContext());
13         while ((bookRecord = itemReader.read()) != null) {
14
15             // then
16             assertThat(bookRecord.getBookName(), is("Foundation"));
17             assertThat(bookRecord.getBookAuthor(), is("Asimov I."));
18             assertThat(bookRecord.getBookISBN(), is("ISBN 12839"));
19             assertThat(bookRecord.getBookFormat(), is("hardcover"));
20             assertThat(bookRecord.getPublishingYear(), is("2018"));
21         }
22         itemReader.close();
23         return null;
24     });
25 }

```

El *MetadataInstanceFactory* crea una costumbre *StepExecution* que se necesita para inyectar nuestra Paso de ámbito de *ItemReader*.

Debido a esto, podemos verificar el comportamiento del lector con la ayuda del método *doInTestScope*.

A continuación, probemos el *JsonFileItemWriter* y verifiquemos su salida:

```

1  @Test
2  public void givenMockedStep_whenWriterCalled_thenSuccess() throws Exception {
3      // given
4      FileSystemResource expectedResult = new FileSystemResource(EXPECTED_OUTPUT_ONE);
5      FileSystemResource actualResult = new FileSystemResource(TEST_OUTPUT);
6      Book demoBook = new Book();
7      demoBook.setAuthor("Grisham J.");
8      demoBook.setName("The Firm");
9      StepExecution stepExecution = MetadataInstanceFactory
10         .createStepExecution(defaultJobParameters());
11
12     // when
13     StepScopeTestUtils.doInStepScope(stepExecution, () -> {
14         jsonItemWriter.open(stepExecution.getExecutionContext());
15         jsonItemWriter.write(Arrays.asList(demoBook));
16         jsonItemWriter.close();
17         return null;
18     });
19
20     // then
21     AssertFile.assertFileEquals(expectedResult, actualResult);
22 }

```

A diferencia de las pruebas anteriores, ahora tenemos el control total de nuestros objetos de prueba. Como resultado, somos responsables de apertura y cierre de los flujos de E / S.

5. Conclusión

En este tutorial, hemos explorado los diversos enfoques para probar un trabajo Spring Batch.

Las pruebas de extremo a extremo verifican la ejecución completa del trabajo. Probar pasos individuales puede ayudar en escenarios complejos.

Finalmente, cuando se trata de componentes de alcance escalonado, podemos usar un montón de métodos auxiliares proporcionados por *spring-batch-test*. Nos ayudarán a tropezar y burlarse de los objetos del dominio Spring Batch.

Ok

Como de costumbre, podemos explorar la base de código completa en GitHub (<https://github.com/eugenp/tutorials/tree/master/spring-batch>) .

Acabo de anunciar el nuevo curso *Learn Spring* , centrado en los fundamentos de Spring 5 y Spring Boot 2:

>> VER EL CURSO (/ls-course-end)



¿Estás aprendiendo a construir tu API
con Spring ?


>> Obtenga el libro electrónico

Deja una respuesta



Start the discussion...

☒ Suscribirse ▼

 **ezoic** (<https://www.ezoic.com/what-is-ezoic/>)

reportar este anuncio

CATEGORÍAS

- PRIMAVERA (<https://www.baeldung.com/category/spring/>)
- DESCANSO (<https://www.baeldung.com/category/rest/>)
- JAVA (<https://www.baeldung.com/category/java/>)
- SEGURIDAD (<https://www.baeldung.com/category/security-2/>)
- PERSISTENCIA (<https://www.baeldung.com/category/persistence/>)
- JACKSON (<https://www.baeldung.com/category/json/jackson/>)
- HTTP DEL LADO DEL CLIENTE (<https://www.baeldung.com/category/http/>)
- KOTLIN (<https://www.baeldung.com/category/kotlin/>)

[TUTORIAL DE JAVA 'VOLVER A LO BÁSICO' \(/JAVA-TUTORIAL\)](#)
[JACKSON JSON TUTORIAL \(/JACKSON\)](#)
[HTTPCLIENT 4 TUTORIAL \(/HTTPCLIENT-GUIDE\)](#)
[RESTO CON SPRING TUTORIAL \(/REST-WITH-SPRING-SERIES\)](#)
[TUTORIAL SPRING PERSISTENCE \(/PERSISTENCE-WITH-SPRING-SERIES\)](#)
[SEGURIDAD CON PRIMAVERA \(/SECURITY-SPRING\)](#)

ACERCA DE

[SOBRE BAELDUNG \(/ABOUT\)](#)
[LOS CURSOS \(HTTPS://COURSES.BAELDUNG.COM\)](#)
[TRABAJO DE CONSULTORÍA \(/CONSULTING\)](#)
[META BAELDUNG \(HTTP://META.BAELDUNG.COM/\)](#)
[EL ARCHIVO COMPLETO \(/FULL_ARCHIVE\)](#)
[ESCRIBIR PARA BAELDUNG \(/CONTRIBUTION-GUIDELINES\)](#)
[EDITORES \(/EDITORS\)](#)
[NUESTROS COMPAÑEROS \(/PARTNERS\)](#)
[ANUNCIE EN BAELDUNG \(/ADVERTISE\)](#)

[TÉRMINOS DE SERVICIO \(/TERMS-OF-SERVICE\)](#)
[POLÍTICA DE PRIVACIDAD \(/PRIVACY-POLICY\)](#)
[INFORMACIÓN DE LA COMPAÑÍA \(/BAELDUNG-COMPANY-INFO\)](#)
[CONTACTO \(/CONTACT\)](#)