

(/)

Pruebas en Spring Boot

Última modificación: 26 de abril de 2020

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

Bota de primavera (<https://www.baeldung.com/category/spring/spring-boot/>)

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

Fundamentos de arranque (<https://www.baeldung.com/tag/boot-basics/>)

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. Información general

En este artículo, veremos cómo **escribir pruebas usando el soporte de framework en Spring Boot** . Cubriremos las pruebas unitarias que pueden ejecutarse de forma aislada, así como las pruebas de integración que arrancarán el contexto Spring antes de ejecutar las pruebas.

Si es nuevo en Spring Boot, consulte nuestra introducción a Spring Boot (/spring-boot-start) .

Otras lecturas:

Explorando el Spring Boot TestRestTemplate (<https://www.baeldung.com/spring-boot-testresttemplate>)

Aprenda a usar el nuevo TestRestTemplate en Spring Boot para probar una API simple.

Leer más (<https://www.baeldung.com/spring-boot-testresttemplate>) →

Guía rápida de @RestClientTest en Spring Boot (<https://www.baeldung.com/restclienttest-in-spring-boot>)

Una guía rápida y práctica para la anotación @RestClientTest en Spring Boot

Leer más (<https://www.baeldung.com/restclienttest-in-spring-boot>) →

Injectando Mockito Mocks en Spring Beans (<https://www.baeldung.com/injecting-mocks-in-spring>)

Este artículo mostrará cómo usar la inyección de dependencia para insertar simulacros Mockito en Spring Beans para pruebas unitarias.

Leer más (<https://www.baeldung.com/injecting-mocks-in-spring>) →

2. Configuración del proyecto

La aplicación que vamos a utilizar en este artículo es una API que proporciona algunas operaciones básicas en un recurso de *empleado*. Esta es una arquitectura típica en niveles: la llamada API se procesa desde el *controlador* al *servicio* a la capa de *persistencia*.

3. Dependencias de Maven

Primero agreguemos nuestras dependencias de prueba:

```
1  <dependency>
2      <groupId>org.springframework.boot</groupId>
3      <artifactId>spring-boot-starter-test</artifactId>
4      <scope>test</scope>
5      <version>2.1.6.RELEASE</version>
6  </dependency>
7  <dependency>
8      <groupId>com.h2database</groupId>
9      <artifactId>h2</artifactId>
10     <scope>test</scope>
11     <version>1.4.194</version>
12 </dependency>
```

El *motor de arranque-pruebas de la primavera-arranque*

(<https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.boot%22%20AND%20a%3A%22spring-boot-starter-test%22>) es la dependencia principal que contiene la mayoría de los elementos necesarios para nuestras pruebas.

El H2 DB

(<https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22com.h2database%22%20AND%20a%3A%22h2%22>) es nuestra base de datos en memoria. Elimina la necesidad de configurar e iniciar una base de datos real para fines de prueba.

4. Pruebas de integración con *@DataJpaTest*

Vamos a trabajar con una entidad llamada *Empleado* que tiene una *identificación* y un *nombre* como sus propiedades:

```
1  @Entity
2  @Table(name = "person")
3  public class Employee {
4
5      @Id
6      @GeneratedValue(strategy = GenerationType.AUTO)
7      private Long id;
8
9      @Size(min = 3, max = 20)
10     private String name;
11
12     // standard getters and setters, constructors
13 }
```

Y aquí está nuestro repositorio: usando Spring Data JPA:

```
1  @Repository
2  public interface EmployeeRepository extends JpaRepository<Employee, Long> {
3
4      public Employee findByName(String name);
5
6  }
```

Eso es todo por el código de la capa de persistencia. Ahora vamos a escribir nuestra clase de prueba.

Primero, creemos el esqueleto de nuestra clase de prueba:

```
1  @RunWith(SpringRunner.class)
2  @DataJpaTest
3  public class EmployeeRepositoryIntegrationTest {
4
5      @Autowired
6      private TestEntityManager entityManager;
7
8      @Autowired
9      private EmployeeRepository employeeRepository;
10
11      // write test cases here
12
13 }
```

@RunWith (SpringRunner.class) se utiliza para proporcionar un puente entre las funciones de prueba de Spring Boot y JUnit. Siempre que usemos cualquier característica de prueba de Spring Boot en nuestras pruebas JUnit, se requerirá esta anotación.

@DataJpaTest proporciona una configuración estándar necesaria para probar la capa de persistencia:

- configurar H2, una base de datos en memoria
- configurar Hibernate, Spring Data y *DataSource*
- realizar un *@EntityScan*
- activar el registro de SQL

Para llevar a cabo alguna operación de DB, necesitamos algunos registros ya configurados en nuestra base de datos. Para configurar estos datos, podemos usar *TestEntityManager*. **El *TestEntityManager* proporcionado por Spring Boot es una alternativa al JPA *EntityManager* estándar que proporciona métodos comúnmente utilizados al escribir pruebas.**

EmployeeRepository es el componente que vamos a probar. Ahora escribamos nuestro primer caso de prueba:

```
1  @Test
2  public void whenFindByName_thenReturnEmployee() {
3      // given
4      Employee alex = new Employee("alex");
5      entityManager.persist(alex);
6      entityManager.flush();
7
8      // when
9      Employee found = employeeRepository.findByName(alex.getName());
10
11     // then
12     assertThat(found.getName())
13         .isEqualTo(alex.getName());
14 }
```

En la prueba anterior, estamos utilizando *TestEntityManager* para insertar un *empleado* en la base de datos y leerlo a través de la API de búsqueda por nombre.

La parte *afirmar que (...)* proviene de la biblioteca Assertj (/introduction-to-assertj) que viene incluida con Spring Boot.

5. Burlándose de *@MockBean*

Nuestro código de capa de *servicio* depende de nuestro *repositorio*. Sin embargo, para probar la capa de *Servicio*, no necesitamos saber ni preocuparnos por cómo se implementa la capa de persistencia:

```
1  @Service
2  public class EmployeeServiceImpl implements EmployeeService {
3
4      @Autowired
5      private EmployeeRepository employeeRepository;
6
7      @Override
8      public Employee getEmployeeByName(String name) {
9          return employeeRepository.findByName(name);
10     }
11 }
```

Idealmente, deberíamos poder escribir y probar nuestro código de capa de Servicio sin necesidad de cablear nuestra capa de persistencia completa.

Para lograr esto, **podemos utilizar el soporte de burla proporcionado por Spring Boot Test**.

Veamos primero el esqueleto de la clase de prueba:


```
1  @RunWith(SpringRunner.class)
2  public class EmployeeServiceImplIntegrationTest {
3
4      @TestConfiguration
5      static class EmployeeServiceImplTestContextConfiguration {
6
7          @Bean
8          public EmployeeService employeeService() {
9              return new EmployeeServiceImpl();
10         }
11     }
12
13     @Autowired
14     private EmployeeService employeeService;
15
16     @MockBean
17     private EmployeeRepository employeeRepository;
18
19     // write test cases here
20 }
```

Para verificar la clase de *servicio* , necesitamos tener una instancia de clase de *servicio* creada y disponible como *@Bean* para que podamos *@Autowire* en nuestra clase de prueba. Esta configuración se logra utilizando la anotación *@TestConfiguration* .

Durante el escaneo de componentes, podemos encontrar componentes o configuraciones creadas solo para pruebas específicas que se recogen accidentalmente en todas partes. Para ayudar a evitar eso, **Spring Boot proporciona la anotación *@TestConfiguration* que se puede usar en las clases en *src / test / java* para indicar que no se deben recoger mediante el escaneo.**

Otra cosa interesante aquí es el uso de *@MockBean* . Se crea una maqueta (*/mockito-mock-methods*) para el *EmployeeRepository* que puede utilizarse para omitir la llamada a la actual *EmployeeRepository* :

```
1  @Before
2  public void setUp() {
3      Employee alex = new Employee("alex");
4
5      Mockito.when(employeeRepository.findByName(alex.getName()))
6          .thenReturn(alex);
7  }
```

Como la configuración está hecha, el caso de prueba será más simple:

```
1  @Test
2  public void whenValidName_thenEmployeeShouldBeFound() {
3      String name = "alex";
4      Employee found = employeeService.getEmployeeByName(name);
5
6      assertThat(found.getName())
7          .isEqualTo(name);
8  }
```

6. Pruebas unitarias con *@WebMvcTest*

Nuestro *controlador* depende de la capa de *servicio* ; incluyamos un solo método para simplificar:

```
1  @RestController
2  @RequestMapping("/api")
3  public class EmployeeRestController {
4
5      @Autowired
6      private EmployeeService employeeService;
7
8      @GetMapping("/employees")
9      public List<Employee> getAllEmployees() {
10         return employeeService.getAllEmployees();
11     }
12 }
```

Como solo estamos enfocados en el código del *Controlador*, es natural burlarse del código de la capa de *Servicio* para nuestras pruebas unitarias:

```
1  @RunWith(SpringRunner.class)
2  @WebMvcTest(EmployeeRestController.class)
3  public class EmployeeRestControllerIntegrationTest {
4
5      @Autowired
6      private MockMvc mvc;
7
8      @MockBean
9      private EmployeeService service;
10
11     // write test cases here
12 }
```

Para probar los *controladores*, podemos usar `@WebMvcTest`. Configurarán automáticamente la infraestructura Spring MVC para nuestras pruebas unitarias.

En la mayoría de los casos, `@WebMvcTest` se limitará a arrancar un solo controlador. Se usa junto con `@MockBean` para proporcionar implementaciones simuladas para las dependencias requeridas.

`@WebMvcTest` también configura automáticamente `MockMvc`, que ofrece una forma poderosa de probar fácilmente los controladores MVC sin iniciar un servidor HTTP completo.

Dicho esto, escribamos nuestro caso de prueba:

```
1  @Test
2  public void givenEmployees_whenGetEmployees_thenReturnJsonArray()
3      throws Exception {
4
5      Employee alex = new Employee("alex");
6
7      List<Employee> allEmployees = Arrays.asList(alex);
8
9      given(service.getAllEmployees()).willReturn(allEmployees);
10
11     mvc.perform(get("/api/employees")
12         .contentType(MediaType.APPLICATION_JSON))
13         .andExpect(status().isOk())
14         .andExpect(jsonPath("$", hasSize(1)))
15         .andExpect(jsonPath("$[0].name", is(alex.getName())));
16 }
```

La llamada al método `get (...)` se puede reemplazar por otros métodos correspondientes a verbos HTTP como `put ()`, `post ()`, etc. Tenga en cuenta que también estamos configurando el tipo de contenido en la solicitud.

MockMvc es flexible, y podemos crear cualquier solicitud con él.

7. Pruebas de integración con *@SpringBootTest*

Como su nombre indica, las pruebas de integración se centran en la integración de diferentes capas de la aplicación. Eso también significa que no hay burlas involucradas.

Idealmente, deberíamos mantener las pruebas de integración separadas de las pruebas unitarias y no deberíamos ejecutarlas junto con las pruebas unitarias. Podemos hacerlo utilizando un perfil diferente para ejecutar solo las pruebas de integración. Un par de razones para hacerlo podría ser que las pruebas de integración requieren mucho tiempo y pueden necesitar una base de datos real para ejecutarse.

Sin embargo, en este artículo, no nos centraremos en eso y en su lugar haremos uso del almacenamiento de persistencia H2 en memoria.

Las pruebas de integración deben iniciar un contenedor para ejecutar los casos de prueba. Por lo tanto, se requiere una configuración adicional para esto; todo esto es fácil en Spring Boot:

```
1  @RunWith(SpringRunner.class)
2  @SpringBootTest(
3      SpringBootTest.WebEnvironment.MOCK,
4      classes = Application.class)
5  @AutoConfigureMockMvc
6  @TestPropertySource(
7      locations = "classpath:application-integrationtest.properties")
8  public class EmployeeRestControllerIntegrationTest {
9
10     @Autowired
11     private MockMvc mvc;
12
13     @Autowired
14     private EmployeeRepository repository;
15
16     // write test cases here
17 }
```

La anotación *@SpringBootTest* se puede usar cuando necesitamos arrancar todo el contenedor. La anotación funciona creando el *ApplicationContext* que se utilizará en nuestras pruebas.

Podemos usar el atributo *webEnvironment* de *@SpringBootTest* para configurar nuestro entorno de tiempo de ejecución; estamos usando *WebEnvironment.MOCK* aquí, para que el contenedor opere en un entorno de servlet simulado.

Podemos usar la anotación *@TestPropertySource* para configurar ubicaciones de archivos de propiedades específicos para nuestras pruebas. Tenga en cuenta que el archivo de propiedades cargado con *@TestPropertySource* anulará el archivo *application.properties* existente .

Los *application-integrationtest.properties* contiene los detalles para configurar el almacenamiento de persistencia:

```
1 | spring.datasource.url = jdbc:h2:mem:test
2 | spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.H2Dialect
```

Si queremos ejecutar nuestras pruebas de integración contra MySQL, podemos cambiar los valores anteriores en el archivo de propiedades.

Los casos de prueba para las pruebas de integración pueden ser similares a las pruebas de unidad de capa del *controlador*:

```
1 | @Test
2 | public void givenEmployees_whenGetEmployees_thenStatus200()
3 |     throws Exception {
4 |
5 |     createTestEmployee("bob");
6 |
7 |     mvc.perform(get("/api/employees")
8 |         .contentType(MediaType.APPLICATION_JSON))
9 |         .andExpect(status().isOk())
10 |        .andExpect(content()
11 |            .contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
12 |        .andExpect(jsonPath("$.name", is("bob"))));
13 | }
```

La diferencia con las pruebas de unidad de capa de *controlador* es que aquí no se burla nada y se ejecutarán escenarios de extremo a extremo.

8. Pruebas autoconfiguradas

Una de las características sorprendentes de las anotaciones autoconfiguradas de Spring Boot es que ayuda a cargar partes de la aplicación completa y probar capas específicas de la base de código.

Además de las anotaciones mencionadas anteriormente, aquí hay una lista de algunas anotaciones ampliamente utilizadas:

- *@WebFluxTest*: podemos usar la anotación *@WebFluxTest* para probar los controladores Spring Webflux. A menudo se usa junto con *@MockBean* para proporcionar implementaciones simuladas para las dependencias requeridas.
- *@JdbcTest*: podemos usar la anotación *@JdbcTest* para probar aplicaciones JPA, pero es para pruebas que solo requieren un *DataSource*. La anotación configura una base de datos integrada en memoria y un *JdbcTemplate*.
- *@JooqTest*: para probar las pruebas relacionadas con jOOQ, podemos usar la anotación *@JooqTest*, que configura un *DSLContext*.
- *@DataMongoTest*: para probar aplicaciones MongoDB, *@DataMongoTest* es una anotación útil. De manera predeterminada, configura un MongoDB incorporado en memoria si el controlador está disponible a través de dependencias, configura un *MongoTemplate*, escanea las clases *@Document* y configura los repositorios Spring Data MongoDB.
- *@DataRedisTest*: hace que sea más fácil probar las aplicaciones de *Redis*. Analiza las clases *@RedisHash* y configura los repositorios Spring Data Redis de forma predeterminada.
- *@DataLdapTest*: configura un *LDAP* incorporado en la memoria (si está disponible), configura un *LdapTemplate*, escanea las clases de *@Entry* y configura los repositorios Spring Data *LDAP* de forma predeterminada.
- *@RestClientTest*: generalmente utilizamos la anotación *@RestClientTest* para probar clientes REST. Configura automáticamente diferentes dependencias como Jackson, GSON y Jsonb, configura *RestTemplateBuilder* y agrega soporte para *MockRestServiceServer* de forma predeterminada.

9. Conclusión

En este tutorial, profundizamos en el soporte de pruebas en Spring Boot y mostramos cómo escribir pruebas unitarias de manera eficiente.

El código fuente completo de este artículo se puede encontrar en GitHub (<https://github.com/eugenp/tutorials/tree/master/spring-boot-modules/spring-boot>) . El código fuente contiene muchos más ejemplos y varios casos de prueba.

Y, si desea seguir aprendiendo sobre las pruebas, tenemos artículos separados relacionados con las pruebas de integración (/integration-testing-in-spring) y las pruebas unitarias en JUnit 5 (/junit-5) .

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 ?

Ingrese su dirección de correo electrónico

>> Obtenga el libro electrónico

8 COMENTARIOS



Más antiguo ▾

**Stéphane Nicoll** (<https://www.about.me/snicoll>) hace 3 años

Thanks for the article. I am not sure I understand the `@TestPropertySource` part on the integration test. If you want to enable an `integrationtest` profile (that's really what your file looks like), you can just add `@ActiveProfiles("integrationtest")` and Spring Boot will load that file automatically. Also, you do not need to do that if you want to use H2. Just add `@AutoconfigureTestDatabase` and we'll replace your `DataSource` with an embedded database for you. I am also curious why you need to refer to `Application` in your integration test. Do you have others `@SpringBootApplication` in this project? If you don't, we'll... Read more »

+ 0 -

**Dusan Odalovic** (<https://odalinjo.wordpress.com/>) 3 years ago

Reply to Stéphane Nicoll

@snicoll:disqus Stéphane, would it be possible to provide lots more small sample apps so that we can just check them out and learn by examples? Spring Boot helps a lot but IMHO documentation is not at the same level.

+ 0 -

**Eugen Paraschiv** (<https://www.baeldung.com/>) 3 years ago

Reply to Stéphane Nicoll

Hey @snicoll:disqus – thanks for the feedback – I'll ask the author and also have a look at your points and potentially jump in and address them. You're right – the terminology needs a bit of cleanup/clarification here.

Cheers,

Eugen.

+ 0 -



Fernando Fradegrada 3 years ago

I am trying to follow the `@DataJpaTest` and I cannot achieve to run the test. It's like all of my application context is being tried to load, and fails to load my controllers, services, etc. What's wrong??

+ 2 -



Grzegorz Piwowarek 3 years ago

Reply to *Fernando Fradegrada*

Can you share you stacktrace? Without this we could only guess blindly

+ 2 -



Fernando Fradegrada 3 years ago

Reply to *Grzegorz Piwowarek*

So I've found what was the problem, but I still not understand why: In my Spring Boot main class I have override the `@ComponentScan` with this, because I need to `@Autowire` a util in another jar. So that's is overriding something that makes my test to load all the App Context. If I remove the `@ComponentScan`, the test runs ok, but then I will not have my autowired component when running my app. I know that this question has nothing to do here, but can you send me a link to understand this? Sorry for my english!

`@SpringBootApplication @ComponentScan("ar.com.myapp.utils"...` Read more »

+ 0 -




Fernando Fradegrada 3 years ago

¿Cómo puedo lidiar con la seguridad de primavera en las pruebas de integración? Recibo una respuesta 401. ¿Hay alguna manera de evitar la seguridad? ¿O tal vez la buena práctica es iniciar sesión antes de realizar la solicitud?

+ 1 -



Grzegorz Piwowarek  hace 3 años

 Respuesta a *Fernando Fradegrada*

El enfoque general es configurar su restTemplate antes de la prueba y luego usarla libremente. Eche un vistazo a TestRestTemplate porque tiene algunos métodos útiles adicionales

+ 0 0 -

¡Los comentarios están cerrados en este artículo!

CATEGORIAS

PRIMAVERA ([HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/](https://www.baeldung.com/category/spring/))

DESCANSO ([HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/](https://www.baeldung.com/category/rest/))

JAVA ([HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/](https://www.baeldung.com/category/java/))

SEGURIDAD ([HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/](https://www.baeldung.com/category/security-2/))

PERSISTENCIA ([HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/](https://www.baeldung.com/category/persistence/))

JACKSON ([HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/](https://www.baeldung.com/category/json/jackson/))

HTTP DEL LADO DEL CLIENTE ([HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/](https://www.baeldung.com/category/http/))

KOTLIN ([HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/](https://www.baeldung.com/category/kotlin/))

SERIE

[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 \(HTTPS://COURSES.BAELDUNG.COM\)](https://courses.baeldung.com)

[TRABAJOS \(/TAG/ACTIVE-JOB/\)](#)

[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 \(/CONTACT\)](#)

