(/)

Pruebas en Spring Boot

Última modificación: 2 de septiembre de 2018

por baeldung (/author/baeldung/) (/author/baeldung/)

Primavera (/category/spring/) + Pruebas (/category/testing/)

Arranque de primavera (/tag/spring-boot/)

1. Información general

En este artículo, echaremos un vistazo a las **pruebas de escritura utilizando el soporte de framework en Spring Boot** . Cubriremos las pruebas unitarias que pueden ejecutarse de forma aislada, así como las pruebas de integración que iniciarán el contexto Spring antes de ejecutar las pruebas.

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

Otras lecturas:

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

Aprenda cómo usar la nueva 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 de la anotación @RestClientTest en Spring Boot

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

Inyectar Mockito se burla de los frijoles (https://www.baeldung.com/injecting-mocks-in-spring)

Este artículo mostrará cómo usar la inyección de dependencia para insertar burlas de 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 de niveles típica: la llamada API se procesa desde el *controlador* al *servicio* en la capa de *persistencia* .

3. Dependencias de Maven

Primero agreguemos nuestras dependencias de prueba:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <version>1.5.3.RELEASE</version>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
    <version>1.4.194</version>
</dependency>
```

El *spring-boot-starter-test* (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 *Employee* que tiene un *id* y un *nombre* como sus propiedades:

```
@Entity
@Table(name = "person")
public class Employee {

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

    @Size(min = 3, max = 20)
    private String name;

    // standard getters and setters, constructors
}
```

Y aquí está nuestro repositorio, utilizando Spring Data JPA:

```
@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Lo

    public Employee findByName(String name);

}
```

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:

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class EmployeeRepositoryIntegrationTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private EmployeeRepository employeeRepository;

    // write test cases here

}
```

@RunWith (SpringRunner.class) se usa para proporcionar un puente entre las funciones de prueba Spring Boot y JUnit. Siempre que utilicemos funciones de prueba Spring Boot en pruebas JUnit, esta anotación será necesaria.

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

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

Para llevar a cabo alguna operación de base de datos, necesitamos algunos registros ya configurados en nuestra base de datos. Para configurar tales 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 vamos a escribir nuestro primer caso de prueba:

```
@Test
public void whenFindByName_thenReturnEmployee() {
    // given
    Employee alex = new Employee("alex");
    entityManager.persist(alex);
    entityManager.flush();

    // when
    Employee found = employeeRepository.findByName(alex.getName());

    // then
    assertThat(found.getName())
      .isEqualTo(alex.getName());
}
```

En la prueba anterior, usamos *TestEntityManager* para insertar un *empleado* en el DB y leerlo a través de la API de búsqueda por nombre.

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

5. *Burlarse* con *@MockBean*

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

```
@Service
public class EmployeeServiceImpl implements EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepository;

    @Override
    public Employee getEmployeeByName(String name) {
        return employeeRepository.findByName(name);
    }
}
```

Idealmente, deberíamos poder escribir y probar nuestro código de la capa de servicio sin cableado en nuestra capa de persistencia completa.

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

Echemos un vistazo al esqueleto de la clase de prueba primero:

```
@RunWith(SpringRunner.class)
public class EmployeeServiceImplIntegrationTest {

    @TestConfiguration
    static class EmployeeServiceImplTestContextConfiguration {

        @Bean
        public EmployeeService employeeService() {
            return new EmployeeServiceImpl();
        }
    }

    @Autowired
    private EmployeeService employeeService;

    @MockBean
    private EmployeeRepository employeeRepository;

    // write test cases here
}
```

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 accidentalmente se recogen en todas partes. Para ayudar a prevenir eso, **Spring Boot proporciona la anotación *@TestConfiguration* que se puede usar en las clases en *src / test / java* para indicar que no deben recogerse escaneando.**

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

```
@Before
public void setUp() {
    Employee alex = new Employee("alex");

    Mockito.when(employeeRepository.findByName(alex.getName()))
      .thenReturn(alex);
}
```

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

```
@Test
public void whenValidName_thenEmployeeShouldBeFound() {
    String name = "alex";
    Employee found = employeeService.getEmployeeByName(name);

     assertThat(found.getName())
      .isEqualTo(name);
 }
```

6. Pruebas unitarias con *@WebMvcTest*

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

```
@RestController
@RequestMapping("/api")
public class EmployeeRestController {

    @Autowired
    private EmployeeService employeeService;

    @GetMapping("/employees")
    public List<Employee> getAllEmployees() {
        return employeeService.getAllEmployees();
    }
}
```

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

```
@RunWith(SpringRunner.class)
@WebMvcTest(EmployeeRestController.class)
public class EmployeeRestControllerIntegrationTest {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private EmployeeService service;

    // write test cases here
}
```

Para probar los *Controladores* , podemos usar *@WebMvcTest* . Configurará 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.

Habiendo dicho eso, escribamos nuestro caso de prueba:

```
@Test
public void givenEmployees_whenGetEmployees_thenReturnJsonArray()
  throws Exception {

    Employee alex = new Employee("alex");

    List<Employee> allEmployees = Arrays.asList(alex);

    given(service.getAllEmployees()).willReturn(allEmployees);

    mvc.perform(get("/api/employees")
      .contentType(MediaType.APPLICATION_JSON))
      .andExpect(status().isOk())
      .andExpect(jsonPath("$", hasSize(1)))
      .andExpect(jsonPath("$[0].name", is(alex.getName())));
}
```

La llamada al método *get (...)* puede ser reemplazada 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 para usarlo.

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 se trata de burlarse.

Idealmente, deberíamos mantener las pruebas de integración separadas de las pruebas unitarias y no deberíamos ejecutar junto con las pruebas unitarias. Podemos hacerlo utilizando un perfil diferente para ejecutar solo

las pruebas de integración. Un par de razones para hacer esto podría ser que las pruebas de integración consumen mucho tiempo y es posible que necesiten una base de datos real para ejecutar.

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:

```
@RunWith(SpringRunner.class)
@SpringBootTest(
  SpringBootTest.WebEnvironment.MOCK,
  classes = Application.class)
@AutoConfigureMockMvc
@TestPropertySource(
  locations = "classpath:application-integrationtest.properties")
public class EmployeeRestControllerIntegrationTest {

    @Autowired
    private MockMvc mvc;

    @Autowired
    private EmployeeRepository repository;

    // write test cases here
}
```

La anotación *@SpringBootTest* se puede usar cuando necesitamos iniciar el contenedor completo. La anotación funciona al crear 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 funcione en un entorno de servlet simulado.

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

La *aplicación*- integrationtest.properties contiene los detalles para configurar el almacenamiento de persistencia:

```
spring.datasource.url = jdbc:h2:mem:test
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.H2Dia
```

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 parecerse a las pruebas de la unidad de capa del *Controlador* :

```
@Test
public void givenEmployees_whenGetEmployees_thenStatus200()
  throws Exception {

    createTestEmployee("bob");

    mvc.perform(get("/api/employees")
      .contentType(MediaType.APPLICATION_JSON))
      .andExpect(status().isOk())
      .andExpect(content()
      .contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
      .andExpect(jsonPath("$[0].name", is("bob")));
}
```

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

8. Conclusión

En este tutorial, hicimos una profunda inmersión 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) . El código fuente contiene muchos más ejemplos y varios casos de prueba.

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

Huésped

Stéphane Nicoll (https://www.about.me/snicoll)

Gracias por el artículo. No estoy seguro de entender la parte `@TestPropertySource` en la prueba de integración. Si desea habilitar un perfil `integrationtest` (así es como se ve su archivo), puede agregar` @ActiveProfiles ("integrationtest") `y Spring Boot cargará ese archivo automáticamente. Además, no necesita hacer eso si quiere usar H2. Simplemente agregue `@AutoconfigureTestDatabase` y reemplazaremos su` DataSource` con una base de datos integrada para usted. También me interesa saber por qué necesita referirse a `Application` en su prueba de integración. ¿Tiene otros `@SpringBootApplication` en este proyecto? Si no lo hace, vamos a ... Leer más »

+ 0 −

hace 1 año

Huésped

Dusan Odalovic (https://odalinjo.wordpress.com/)

@snicoll: disqus Stéphane, ¿sería posible proporcionar muchas más pequeñas aplicaciones de muestra para que podamos verificarlas y aprender con ejemplos? Spring Boot ayuda mucho, pero la documentación de mi humilde opinión no está en el mismo nivel.

+ 0 −

hace 1 año

Huésped

Eugen Paraschiv (http://www.baeldung.com/)

Hola, @snicoll: disqus, gracias por los comentarios. Le preguntaré al autor y también echaré un vistazo a tus puntos y, potencialmente, me lanzaré a abordarlos. Tienes razón, la terminología necesita un poco de limpieza / aclaración aquí.
Saludos,
Eugen.

+ 0 −

hace 1 año

Huésped

Fernando Fradegrada

Intento seguir el test @DataJpaTest y no logro ejecutar la prueba. Es como si todo el contexto de mi aplicación se intenta cargar y falla al cargar mis controladores, servicios, etc. ¿Qué ocurre?

+ 2 −

hace 1 año

Huésped

Grzegorz Piwowarek

¿Puedes compartir tu stacktrace? Sin esto, solo podríamos adivinar a ciegas

+ 2 −

hace 1 año

Huésped

Fernando Fradegrada

Así que he encontrado cuál era el problema, pero todavía no entiendo por qué: en mi clase principal de Spring Boot, he reemplazado el @ComponentScan con esto, porque necesito @Autowire una utilidad en otro jar. Así que eso está anulando algo que hace que mi prueba cargue todo el contexto de la aplicación. Si elimino @ComponentScan, la prueba se ejecutará correctamente, pero luego no tendré mi componente de autoenlace cuando ejecute mi aplicación. Sé que esta pregunta no tiene nada que ver aquí, pero ¿pueden enviarme un enlace para entender esto? ¡Lo siento por mi ingles! @SpringBootApplication @ComponentScan ({"ar.com.myapp.utils" ... Leer más »

+ 0 −

hace 1 año

Huésped

Fernando Fradegrada

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

+ 1 −

hace 1 año

Huésped

Grzegorz Piwowarek

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

+ 0 −

hace 1 año

CATEGORÍAS

PRIMAVERA (/CATEGORY/SPRING/)
DESCANSO (/CATEGORY/REST/)
JAVA (/CATEGORY/JAVA/)
SEGURIDAD (/CATEGORY/SECURITY-2/)
PERSISTENCIA (/CATEGORY/PERSISTENCE/)
JACKSON (/CATEGORY/JACKSON/)
HTTPCLIENT (/CATEGORY/HTTP/)
KOTLIN (/CATEGORY/KOTLIN/)

SERIE

TUTORIAL "VOLVER A LO BÁSICO" DE JAVA (HTTPS://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTPS://WWW.BAELDUNG.COM/JACKSON)
TUTORIAL DE HTTPCLIENT 4 (HTTPS://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST CON SPRING TUTORIAL (/REST-WITH-SPRING-SERIES/)
TUTORIAL DE SPRING PERSISTENCE (/PERSISTENCE-WITH-SPRING-SERIES/)
SEGURIDAD CON SPRING (HTTPS://WWW.BAELDUNG.COM/SECURITY-SPRING)

ACERCA DE

ACERCA DE 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)

CONTACTO (/CONTACT)

EDITORES (/EDITORS)

KIT DE MEDIOS (PDF) (HTTPS://S3.AMAZONAWS.COM/BAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF)

TÉRMINOS DE SERVICIO (/TERMS-OF-SERVICE)

POLÍTICA DE PRIVACIDAD (/PRIVACY-POLICY)

INFORMACIÓN DE LA COMPAÑÍA (/BAELDUNG-COMPANY-INFO)