CURSO PROGRAMACIÓN FULLSTACK

GUÍA SPRING TEST - API REST

EGG

INTRODUCCIÓN

Las pruebas son una parte fundamental del desarrollo de software. Proporcionan una garantía de calidad al verificar que el código funciona como se espera y se mantiene libre de errores a medida que se realizan cambios. En el contexto de Spring Boot, las pruebas se pueden realizar en varios niveles, desde simples pruebas unitarias hasta pruebas de integración y pruebas de extremo a extremo.

Esta guía nos centraremos en la escritura de pruebas unitarias específicamente para servicios y controladores REST. Cubriremos las mejores prácticas y herramientas proporcionadas por el marco Spring Boot Test para garantizar la robustez y la fiabilidad de estas capas cruciales de la aplicación.

¿QUÉ ES SPRING BOOT TEST?

Spring Boot Test es un módulo de Spring Boot que proporciona soporte para escribir y ejecutar pruebas en aplicaciones Spring Boot. Cuando creas un proyecto de spring se suele añadir por defecto, si no la encuentras en tu pom.xml puedes agregarla, aquí tienes la dependencia de Maven:

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

CLASES DE PRUEBA EN UN PROYECTO SPRING BOOT

Cuando creas un nuevo proyecto Spring Boot, automáticamente se creará un paquete y ruta en donde estará tu primera clase test “/src/test/java/com/nombre/NombreApplicationTests.java”. Y se verá de la siguiente manera:

```
@SpringBootTest
class NameApplicationTests {

	@Test
	void contextLoads() {
	}

}
```

El “@Test” ya lo conocemos de Junit, pero veamos que función cumple “@SpringBootTest”, esta anotación le indica a Spring Boot que cargue y configure el contexto de la aplicación completa para la ejecución de la prueba, es decir, nos permite automatizar el proceso de configuración de la clase de prueba en el caso de que necesitemos usar algún componente de la aplicación.

PRUEBAS UNITARIAS DE CLASES DE SERVICIO

Al realizar pruebas unitarias en nuestras clases de servicio en Spring Boot, es crucial comprender que estas clases a menudo dependen de otros componentes, como repositorios o servicios externos. Para garantizar que nuestras pruebas se centren únicamente en la lógica de la clase de servicio y no en la funcionalidad de sus dependencias, recurrimos a la creación de mocks.

¿QUÉ SON LOS MOCKS?

Un "mock" es una simulación de un componente real que se utiliza en las pruebas para reemplazar la implementación real de una dependencia. Estos mocks nos permiten controlar el comportamiento de las dependencias y aislar la lógica de la clase que estamos probando. Utilizamos la anotación @MockBean para crear y agregar mocks al contexto de Spring Boot.

ESTRUCTURA BÁSICA DE LAS PRUEBAS CON MOCKS

Ten en cuenta esta estructura básica que debes seguir a la hora de hacer pruebas unitarias con mocks:

- A nivel de clase:
    - Inyección de Dependencias: La clase de servicio puede depender de otros componentes, como repositorios. En las pruebas unitarias se utilizan mocks para estas dependencias.
    - Configuración de Mocks: Utilizamos la anotación @MockBean para crear mocks de las dependencias de la clase de servicio. Estos mocks se agregan automáticamente al contexto de Spring.
- A nivel de método:
    - Definición del Comportamiento del Mock: Utilizamos el método when de Mockito para definir el comportamiento esperado de los mocks. Esto puede incluir respuestas a métodos específicos o excepciones que deberían arrojarse.

- Ejecución de la Lógica de la Clase de Servicio: invocamos a los métodos de la clase de servicio.
- Verificación de Resultados: Utilizamos afirmaciones (assert) para verificar que los resultados de las operaciones de la clase de servicio sean los esperados.

EJEMPLO DE PRUEBAS UNITARIAS DE SERVICIOS CON MOCKS

A continuación, se presenta un ejemplo de cómo podríamos estructurar las pruebas unitarias para una clase de servicio UserService que interactúa con un repositorio UserRepository:

UserService:

```
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public User createUser(User user) {
        return userRepository.save(user);
    }

    public Optional<User> findUserById(Long userId) {
        return userRepository.findById(userId);
    }
}
```

UserRepository:

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    // Métodos del repositorio
}
```

UserServiceTest:

```
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.when;
import java.util.Optional;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;

@SpringBootTest
public class UserServiceTest {

    @Autowired
    private UserService userService;

    @MockBean
    private UserRepository userRepository;

    @Test
    public void deberiaCrearUsuario() {
        // Configuración del mock
        when(userRepository.save(any(User.class)))
          .thenAnswer(invocation -> {
            User user = invocation.getArgument(0);
            user.setId(1L);
            return user;
            });

        // Ejecución de la lógica de la clase de servicio
        User user = new User("John Doe");
        User usuarioCreado = userService.createUser(user);

        // Verificación de resultados
        assertNotNull(usuarioCreado.getId());
        assertEquals("John Doe", usuarioCreado.getUsername());
    }

    @Test
    public void deberiaBuscarUsuarioPorId() {
        // Configuración del mock
        Long userId = 1L;
        User user = new User("Jane Doe");

when(userRepository.findById(userId)).thenReturn(Optional.of(user));

        // Ejecución de la lógica de la clase de servicio
    Optional<User> usuarioEncontrado = userService.findUserById(userId);

        // Verificación de resultados
        assertTrue(usuarioEncontrado.isPresent());
        assertEquals("Jane Doe", usuarioEncontrado.get().getUsername());
    }
}
```

¿CUÁL ES LA DIFERENCIA ENTRE THENRETURN Y THENANSWER?

La elección entre thenReturn y thenAnswer en Mockito depende del escenario de prueba y de lo que estás tratando de lograr.

- thenReturn: Se utiliza para devolver un valor específico cada vez que se llama al método en el que se aplica. Funciona bien cuando el resultado es determinista y no cambia en función de las entradas o el estado interno.
- thenAnswer: Permite proporcionar una implementación más dinámica del método. Puedes realizar lógica personalizada basada en los argumentos recibidos. Es útil cuando el comportamiento de la respuesta depende de la entrada o cuando necesitas realizar acciones más complejas.

En el caso de la prueba deberiaCrearUsuario, estamos utilizando thenAnswer porque queremos realizar una acción personalizada: establecer el id del usuario creado antes de devolverlo. Esto es más complejo que simplemente devolver un valor fijo. La implementación con thenAnswer nos permite manipular el objeto antes de devolverlo como resultado.

PRUEBAS UNITARIAS DE CONTROLADORES REST

Las pruebas unitarias de controladores REST en Spring Boot son fundamentales para garantizar que las API web se comporten correctamente y devuelvan las respuestas esperadas. En estas pruebas, nos centraremos en cómo utilizar @WebMvcTest para crear un entorno de prueba ligero y cómo simular peticiones HTTP para evaluar el comportamiento de nuestros controladores.

ANOTACIÓN @WEBMVCTEST

La anotación @WebMvcTest es clave al realizar pruebas unitarias para controladores en Spring Boot. Esta anotación se utiliza para cargar solo la configuración relevante para las pruebas de controladores y para deshabilitar la carga de todo el contexto de la aplicación.

```
@WebMvcTest(UsuarioRestController.class)
public class UsuarioRestControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UsuarioService usuarioService;

    // Métodos de prueba
}
```

@WebMvcTest(UsuarioController.class): Especifica que solo se cargarán las configuraciones necesarias para probar UsuarioController. Esto mejora la eficiencia al cargar solo las partes relevantes de la aplicación.

MockMvc: Es una clase proporcionada por Spring para simular solicitudes HTTP y evaluar las respuestas. Se inyecta automáticamente cuando se utiliza @WebMvcTest. Aquí te proporciono una descripción de algunos de los métodos más comunes disponibles en dicha clase:

perform(MockHttpServletRequestBuilder requestBuilder): Inicia una ejecución de solicitud simulada basada en el requestBuilder proporcionado. Devuelve un objeto ResultActions que permite realizar aserciones sobre la respuesta.

```
...
mockMvc.perform(MockMvcRequestBuilders.get("/example"))
       .andExpect(status().isOk());
...
```

andExpect(ResultMatcher resultMatcher): Permite especificar expectativas sobre la respuesta, como el estado HTTP, contenido, encabezados, etc. Puedes encadenar múltiples expectativas para realizar verificaciones más complejas.

```
...
mockMvc.perform(MockMvcRequestBuilders.get("/example"))
       .andExpect(status().isOk())
       .andExpect(content().string("Hello, World!"));
...
```

andDo(ResultHandler resultHandler): Permite realizar acciones adicionales después de que se haya procesado la solicitud, como imprimir detalles de depuración.

```
...
mockMvc.perform(MockMvcRequestBuilders.get("/example"))
       .andDo(print());
...
```

andReturn(): Ejecuta la solicitud simulada y devuelve el resultado como un objeto MvcResult que tiene detalles sobre la respuesta, como el contenido, los encabezados, etc.

```
...
MvcResult result = mockMvc
                        .perform(MockMvcRequestBuilders.get("/example"))
                        .andReturn();
...
```

MockMvcRequestBuilders:

MockHttpServletRequestBuilder es una interfaz que representa una solicitud HTTP simulada. Es una interfaz para construir objetos que representan diversas operaciones HTTP como GET, POST, PUT, DELETE, etc. Los métodos en MockHttpServletRequestBuilder te permiten configurar detalles de la solicitud como la URL, los parámetros, los encabezados, el contenido, etc.

```
...
mockMvc.perform(MockMvcRequestBuilders.get("/example")
                            .param("param1", "value1")
                            .header("Authorization", "Bearer token")
                ).andDo(print());
...
```

puedes hacer una importación estatica de los métodos con “import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;”

MockMvcResultMatchers:

ResultMatcher es una interfaz que te permite expresar expectativas sobre la respuesta de la solicitud. Se utiliza con el método andExpect de MockMvc para realizar aserciones sobre la respuesta obtenida después de realizar una solicitud simulada.

Hay diversas implementaciones de ResultMatcher que te permiten verificar el estado HTTP, el contenido, los encabezados, etc., de la respuesta.

```
...
mockMvc.perform(MockMvcRequestBuilders.get("/example"))
       .andExpect(MockMvcResultMatchers.status().isOk())
                .andExpect(MockMvcResultMatchers.content().string("Hello,
World!"));
...
```

puedes hacer una importación estatica de los métodos con “import static org.springframework.test.web.servlet.request.MockMvcResultMatchers.*;”

EJEMPLO DE PRUEBAS UNITARIAS DE CONTROLADORES REST

Supongamos que tenemos un controlador UsuarioRestController que utiliza un servicio UsuarioService para gestionar operaciones relacionadas con usuarios.

UsuarioRestController:

```
@RestController
@RequestMapping("/usuarios")
public class UsuarioRestController {

    @Autowired
    private UsuarioService usuarioService;

    @PostMapping
       public  ResponseEntity<Usuario>  crearUsuario(@RequestBody  Usuario
usuario) {
        Usuario usuarioCreado = usuarioService.guardarUsuario(usuario);
        return new ResponseEntity<>(usuarioCreado, HttpStatus.CREATED);
    }

    @GetMapping("/{id}")
     public ResponseEntity<Usuario> obtenerUsuario(@PathVariable Long id)
{
        Optional<Usuario> usuario = usuarioService.buscarPorId(id);
              return  usuario.map(value  ->  new  ResponseEntity<>(value,
HttpStatus.OK))
                                                    .orElseGet(()   ->   new
ResponseEntity<>(HttpStatus.NOT_FOUND));
    }

    // Otros métodos del controlador
}
```

Pruebas Unitarias (UsuarioRestControllerTest):

```
@WebMvcTest(UsuarioController.class)
public class UsuarioControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UsuarioService usuarioService;

    @Test
    public void deberiaCrearUsuario() throws Exception {
        Usuario usuario = new Usuario("John Doe");


when(usuarioService.guardarUsuario(any(Usuario.class))).thenReturn(usuar
io);
```

```
        mockMvc.perform(post("/usuarios")
                .contentType(MediaType.APPLICATION_JSON)
                .content(asJsonString(usuario)))
                .andExpect(status().isCreated())
                .andExpect(jsonPath("$.username", is("John Doe")));
    }

    @Test
    public void deberiaObtenerUsuarioExistente() throws Exception {
        Long userId = 1L;
        Usuario usuario = new Usuario("Jane Doe");

when(usuarioService.buscarPorId(userId)).thenReturn(Optional.of(usuario)
);

        mockMvc.perform(get("/usuarios/{id}", userId))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.username", is("Jane Doe")));
    }

    @Test
      public  void  deberiaObtenerNotFoundParaUsuarioNoExistente()  throws
Exception {
        Long userId = 2L;

when(usuarioService.buscarPorId(userId)).thenReturn(Optional.empty());

        mockMvc.perform(get("/usuarios/{id}", userId))
                .andExpect(status().isNotFound());
    }

    // Otros métodos de prueba para el controlador
}
```

EJERCICIOS

1) Utilizando la API REST desarrollada para gestionar clientes, pedidos y productos de la guía anterior, aplica los conocimientos adquiridos sobre pruebas unitarias para los controladores y servicios de dichas entidades.

    a) En el servicio ClientService, implementa pruebas unitarias para los métodos de creación, lectura, actualización y eliminación de clientes. Asegúrate de cubrir

diversos escenarios, como crear clientes válidos, intentar leer un cliente que no existe, actualizar un cliente existente y eliminar un cliente que no está presente.

b) Repite el proceso para los servicios OrderService y ProductService, abordando los métodos relacionados con la gestión de pedidos y productos.

c) En el controlador ClientRestController, escribe pruebas unitarias para los métodos asociados a las operaciones CRUD (GET, POST, PUT, DELETE). Utiliza MockMvc para simular las solicitudes HTTP y verifica que las respuestas sean coherentes con las operaciones realizadas.

d) Repite el proceso para los controladores OrderRestController y ProductRestController, cubriendo los métodos asociados a las operaciones CRUD.