Unit Testing

How does Spring Framework Make Unit Testing Easy?

Answer: The Spring Framework makes unit testing easy by providing a variety of features that facilitate the creation of unit tests:

- **Dependency Injection:** Spring's IoC container manages the dependencies of beans, which allows for easier setup and configuration of unit tests.
- **Mocking Support:** Spring supports the use of various mocking frameworks like Mockito, allowing developers to mock dependencies and isolate the unit of work.
- **Test Context Framework:** Spring's TestContext Framework integrates with JUnit and TestNG, providing support for loading Spring contexts and injecting dependencies directly into test cases.
- **Annotations:** Spring provides several annotations like `@MockBean`, `@WebMvcTest`, `@SpringBootTest`, and others to simplify the configuration of test contexts and mocking.

What is Mockito?

Answer: Mockito is a popular open-source mocking framework for Java. It allows you to create mock objects for your unit tests, which can mimic the behavior of real objects in a controlled way. Mockito helps in isolating the unit of work being tested and provides functionality to stub methods, verify interactions, and assert method calls.

What is your favorite mocking framework?

Answer: My favorite mocking framework is Mockito because it is widely used, well-documented, and integrates seamlessly with JUnit. Mockito's API is intuitive, and it provides powerful features for stubbing, verifying, and handling complex mocking scenarios.

How do you do mock data with Mockito?

Answer: To mock data with Mockito, you typically follow these steps:

1. **Create Mocks:** Use the `@Mock` annotation or `Mockito.mock()` method to create mock objects.
2. **Stub Methods:** Define the behavior of mock objects using `when(...).thenReturn(...)`.
3. **Inject Mocks:** Use `@InjectMocks` to inject mock dependencies into the class under test.
4. **Verify Interactions:** Use `verify(...)` to ensure methods on mock objects were called as expected.

Example:

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    @Test
    public void testGetUserById() {
        User mockUser = new User(1, "John");

when(userRepository.findById(1)).thenReturn(Optional.of(mockUser));

        User user = userService.getUserById(1);

        assertNotNull(user);
        assertEquals("John", user.getName());
        verify(userRepository).findById(1);
    }
}
```

What are the different mocking annotations that you worked with?

Answer:

- `@Mock`: Creates a mock object.
- `@InjectMocks`: Injects mock objects into the class under test.
- `@Spy`: Creates a spy object, which allows partial mocking.
- `@Captor`: Captures argument values for further assertions.
- `@MockBean`: Used in Spring tests to add mock objects to the Spring ApplicationContext.

What is MockMvc?

Answer: `MockMvc` is a Spring component that provides support for testing Spring MVC controllers. It allows you to simulate HTTP requests and assert the results without starting a web server. `MockMvc` is used for testing the web layer in isolation.

What is @WebMvcTest?

Answer: `@WebMvcTest` is a Spring Boot annotation used to configure Spring MVC tests. It is used to test the web layer and auto-configures Spring MVC infrastructure components, focusing only on the web layer.

Example:

```
@WebMvcTest(UserController.class)
public class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService;

    @Test
    public void testGetUser() throws Exception {
        User mockUser = new User(1, "John");
        when(userService.getUserById(1)).thenReturn(mockUser);

        mockMvc.perform(get("/users/1"))
               .andExpect(status().isOk())
               .andExpect(jsonPath("$.name").value("John"));
    }
}
```

What is @MockBean?

Answer: `@MockBean` is a Spring Boot annotation used to add mock objects to the Spring ApplicationContext. It allows for the replacement of existing beans with mocks during testing, making it useful for isolating the unit of work.

How do you write a unit test with MockMVC?

Answer: To write a unit test with `MockMvc`, follow these steps:

1. Annotate the test class with `@WebMvcTest`.
2. Autowire `MockMvc`.
3. Use `mockMvc.perform()` to simulate HTTP requests.
4. Assert the response using methods like `andExpect()`.

Example:

```
@WebMvcTest(UserController.class)
public class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService;

    @Test
    public void testGetUser() throws Exception {
        User mockUser = new User(1, "John");
        when(userService.getUserById(1)).thenReturn(mockUser);

        mockMvc.perform(get("/users/1"))
               .andExpect(status().isOk())
               .andExpect(jsonPath("$.name").value("John"));
    }
}
```

What is JSONAssert?

Answer: `JSONAssert` is a library for asserting JSON content in tests. It allows for flexible and lenient comparisons of JSON strings, making it easier to write assertions for JSON responses in unit tests.

Example:

java
Copy code

```
String actualJson = "{ \"name\": \"John\", \"age\": 30 }";
String expectedJson = "{ \"name\": \"John\", \"age\": 30 }";
JSONAssert.assertEquals(expectedJson, actualJson, false);
```

How do you write an integration test with Spring Boot?

Answer: To write an integration test with Spring Boot, use the `@SpringBootTest` annotation. This annotation loads the complete application context and allows you to test the entire application.

Example:

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class UserControllerIntegrationTest {
```

```
    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void testGetUser() {
        ResponseEntity<User> response =
restTemplate.getForEntity("/users/1", User.class);
        assertEquals(HttpStatus.OK, response.getStatusCode());
        assertEquals("John", response.getBody().getName());
    }
}
```

What is @SpringBootTest?

Answer: `@SpringBootTest` is a Spring Boot annotation used for integration testing. It loads the complete application context and can be configured to run with a random port, defined port, or without a web environment.

What is @LocalServerPort?

Answer: `@LocalServerPort` is a Spring Boot annotation used to inject the port number of the running server when the `@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)` configuration is used. It allows for dynamic assignment and retrieval of the port number.

What is TestRestTemplate?

Answer: `TestRestTemplate` is a Spring Boot component that simplifies writing integration tests for RESTful services. It is a convenient alternative to `RestTemplate` with additional methods to set the base URL, handle cookies, and perform assertions on HTTP responses.

Example:

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class UserControllerIntegrationTest {

    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void testGetUser() {
```

```
        String baseUrl = "http://localhost:" + port + "/users/1";
        ResponseEntity<User> response =
restTemplate.getForEntity(baseUrl, User.class);
        assertEquals(HttpStatus.OK, response.getStatusCode());
        assertEquals("John", response.getBody().getName());
    }
}
```