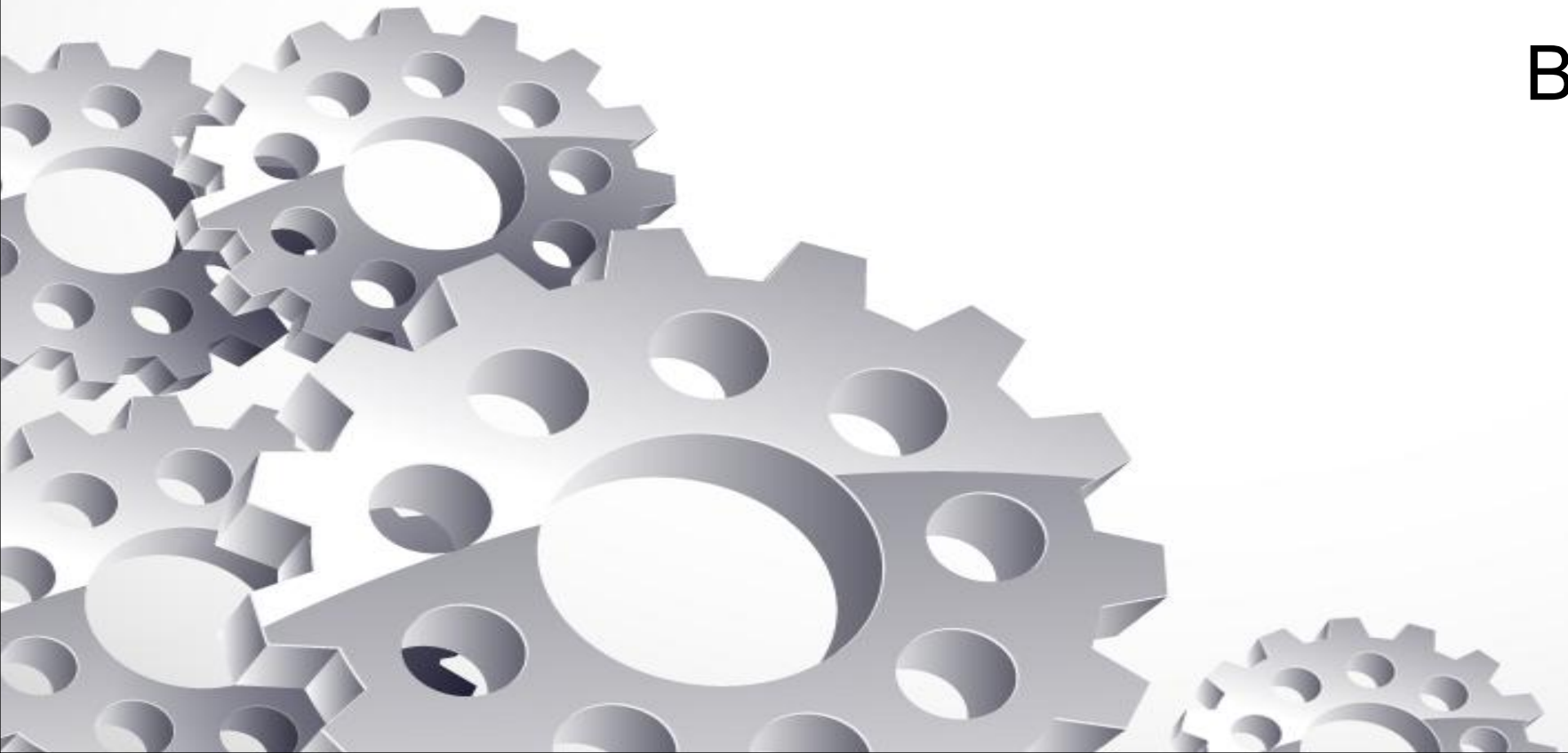
Integration Testing, Mocking, and Load Testing

BackEnd Testing



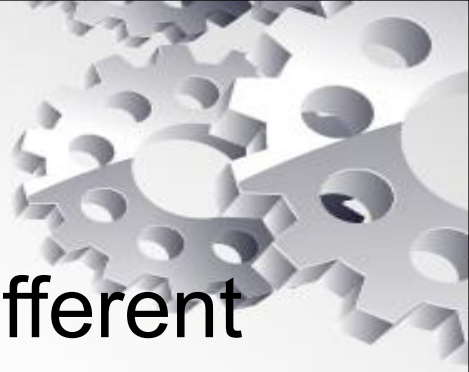
SPRING BOOT TESTING

- Testing in Spring Boot is crucial for ensuring the correctness and reliability of your applications. There are several types of tests you can write in a Spring Boot application, including unit tests, integration tests, and end-to-end tests. Here's a brief overview of how to perform testing in Spring Boot:
- Unit Testing:
- Unit tests focus on testing individual components or units of your application in isolation. In Spring Boot, you can use JUnit and Mockito for writing unit tests.

unit testing

```java
@ExtendWith(MockitoExtension.class)
public class MyServiceTest {

    @InjectMocks
    private MyService myService;

    @Mock
    private MyRepository myRepository;

    @Test
    public void testFindById() {
        Mockito.when(myRepository.findById(1L)).thenReturn(Optional.of(new MyEntity(
        MyEntity entity = myService.findById(1L);
        assertEquals("Test", entity.getName());
    }
}
```

Integration Testing



- Integration tests verify the interactions between different components or layers of your application. Spring Boot provides @SpringBootTest annotation for creating integration tests. You can use @Autowired to inject dependencies.

- Example:

Integration testing

```java
@SpringBootTest
public class MyIntegrationTest {

    @Autowired
    private MyService myService;

    @Autowired
    private MyRepository myRepository;

    @Test
    public void testFindById() {
        MyEntity entity = new MyEntity(1L, "Test");
        myRepository.save(entity);
        MyEntity result = myService.findById(1L);
        assertEquals("Test", result.getName());
    }
}
```

End-to-End Testing:

- End-to-end tests validate the behavior of your entire application, including external dependencies. Tools like Selenium or REST Assured are commonly used for end-to-end testing in Spring Boot.

- Example (with REST Assured):

end-to-end testing

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

    @LocalServerPort
    private int port;

    @Test
    public void testEndpoint() {
        given()
            .port(port)
            .when()
            .get("/api/myendpoint")
            .then()
            .statusCode(200);
    }
}
```

Mocking Dependencies

- Mockito is commonly used for mocking dependencies in Spring Boot tests. You can use @MockBean annotation to replace beans with mocks in the Spring application context.

- Example:

mocking dependencies

```java
@SpringBootTest
public class MyMockTest {

    @Autowired
    private MyService myService;

    @MockBean
    private MyRepository myRepository;

    @Test
    public void testFindById() {
        Mockito.when(myRepository.findById(1L)).thenReturn(Optional.of(new MyEntity(
        MyEntity entity = myService.findById(1L);
        assertEquals("Test", entity.getName());
    }
}
```

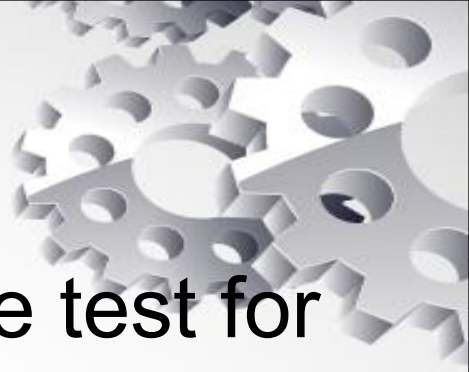
mockito fundamentals

```
import static org.mockito.Mockito.*;

    //mock creation
    LinkedList mockedList = mock(LinkedList.class);


    //using mock object
    mockedList.add("one");
    mockedList.clear();


    //verification
    verify(mockedList).add("one");
    verify(mockedList).clear();
```

Repository Testing

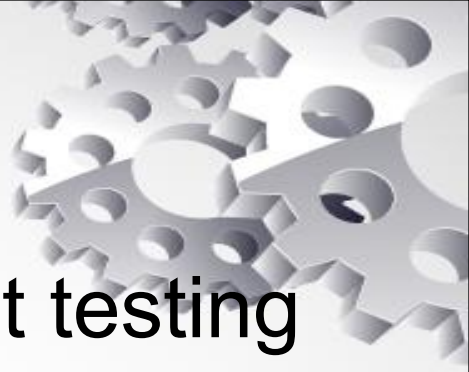
- Testing repositories in Spring Boot typically involves testing CRUD (Create, Read, Update, Delete) operations along with custom query methods. You can use Spring Data JPA's @DataJpaTest annotation to focus your tests on JPA components only, which will speed up the test execution by loading only the relevant parts of the Spring context.

Repository Testing



- We use @DataJpaTest annotation to configure the test for JPA components only. It will set up an in-memory database and only load components relevant for JPA testing.
- We autowire the XXXXRepository to perform CRUD operations.
- In each test method, we perform a specific operation (e.g., save a user, find a user by username) and then assert the expected outcomes.

Service Testing



- Testing Spring Boot services typically involves unit testing the individual methods within the service class.
- Annotate with @SpringBootTest,
- You can use JUnit along with Mockito to mock dependencies and verify the behavior of your service methods. Here's an example of how to write unit tests for a Spring Boot service:

Service Testing

- We're using @Mock to mock the XXXXRepository dependency.
- @InjectMocks is used to inject the mocked XXXRepository into the XXXXService.
- In each test method, we set up behavior for the mocked repository using Mockito.when().
- We then call methods on the XXXXService and assert on the expected behavior.

Controller/API Testing

- MockMvc for Integration Testing:
  - Use Spring's MockMvc to test your controllers in isolation. Write tests to verify the correct handling of HTTP requests and responses.

- Request and Response Validation:
  - Check if the request parameters are correctly mapped and if the response is as expected.

MockMvc

- In Spring Boot testing with MockMvc, several frequently used methods are available to perform various actions and assertions on HTTP requests and responses. Here's a list of some commonly used methods in MockMvc:

MockMvc - Performing HTTP Requests

- perform(MockHttpServletRequestBuilder requestBuilder): Performs an HTTP request and returns a ResultActions object for further assertions.
  - get(String urlTemplate), post(String urlTemplate), put(String urlTemplate), delete(String urlTemplate): Convenience methods to create GET, POST, PUT, and DELETE requests, respectively.
- contentType(MediaType mediaType): Sets the content type of the request.

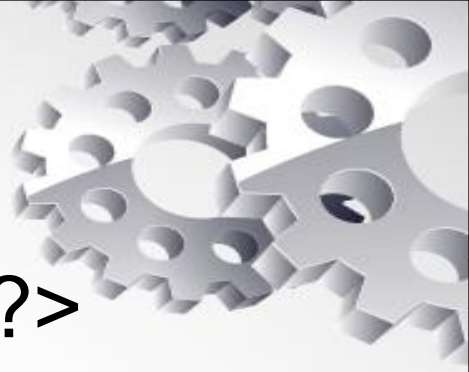
MockMvc - Asserting Response Status

- andExpect(status().isOk()): Asserts that the HTTP response status is 200 (OK).
- andExpect(status().isNotFound()): Asserts that the HTTP response status is 404 (Not Found).
- andExpect(status().isCreated()): Asserts that the HTTP response status is 201 (Created).
- andExpect(status().isBadRequest()): Asserts that the HTTP response status is 400 (Bad Request).

MockMvc - Asserting Response Content

- andExpect(content().json(String jsonContent)): Asserts that the response body matches the provided JSON content.
- andExpect(content().contentType(MediaType mediaType)): Asserts that the content type of the response is the specified media type.

MockMvc - Asserting Response Body with JSONPath

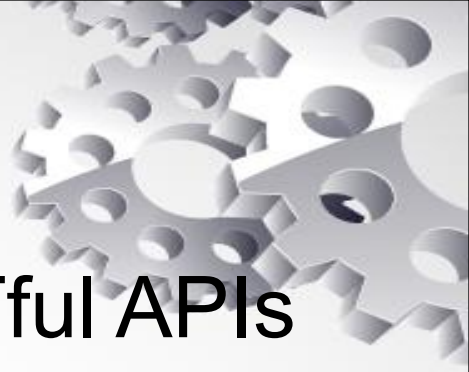


- andExpect(jsonPath(String expression, Matcher<?> matcher)): Asserts that the specified JSONPath expression matches the expected value using a Hamcrest matcher.

MockMvc - Others

- Setting Request Content:
  - content(String content): Sets the content of the request.
- Setting Request Parameters:
  - param(String name, String... values): Sets request parameters.
- Setting Request Headers:
  - header(String name, String... values): Sets request headers.

REST ASSURED



- Rest Assured is a popular library for testing RESTful APIs in Java. It provides a fluent interface for making HTTP requests and validating responses. In the context of Spring Boot, you can use Rest Assured to write integration tests for your RESTful endpoints.

REST ASSURED -methods

- given(): This method initializes the given state for the request. It's typically used at the beginning of a Rest Assured test to specify the request details such as headers, parameters, and body.

```
given()
    .contentType(ContentType.JSON)
    .header("Authorization", "Bearer token")
    .param("key", "value")
    .body(requestBody)
```

REST ASSURED -methods

- when(): This method performs the HTTP request to the API endpoint. It's usually chained after the given() method.
  - when()
    .get("/api/resource")
- then(): This method validates the response received from the API. It's typically used to assert the response status code, body, headers, etc.
  - then()
  - .statusCode(200)
  - .body("data.name", equalTo("John Doe"))
  - .header("Content-Type", containsString("application/json"))

REST ASSURED -methods

- contentType(): This method sets the content type of the request or specifies the expected content type of the response.
- header(): This method adds headers to the request or validates headers in the response.
- param(): This method adds query parameters to the request.
- body(): This method specifies the request body for POST, PUT, or PATCH requests.

REST ASSURED -methods

- get(), post(), put(), delete(), etc.: These methods perform HTTP requests of the specified type (GET, POST, PUT, DELETE) to the specified endpoint.
- statusCode(): This method asserts the status code of the response.
- body(): This method performs assertions on the response body using JSONPath expressions, XMLPath expressions, or plain text.
- extract(): This method extracts data from the response to use in subsequent requests or assertions.