Docker Test Containers in Java Tests

Last updated: January 8, 2024

Written by: baeldung (https://www.baeldung.com/author/baeldung)

Reviewed by: José Carlos Valero Sánchez (https://www.baeldung.com/editor/jose-valero)

Docker (https://www.baeldung.com/category/docker)

Testcontainers (https://www.baeldung.com/tag/testcontainers)

Traditional keyword-based search methods rely on exact word matches, **often leading to irrelevant results** depending on the user's phrasing.

By comparison, using a vector store allows us to represent the data as vector embeddings, based on meaningful relationships. We can then compare the meaning of the user's query to the stored content, and **retrieve more relevant, context-aware results.**

Explore how to **build an intelligent chatbot using MongoDB Atlas, Langchain4j** and Spring Boot:

>> Building an AI Chatbot in Java With Langchain4j and MongoDB Atlas (/MongoDB-NPI-EA-5-cvdl)

1. Introduction

In this tutorial, we'll be looking at the Java *Testcontainers* (https://java.testcontainers.org/) library. It allows us to use Docker containers within our tests. As a result, we can write self-contained integration tests that depend on external resources.

We can use any resource in our tests that have a docker image. For example, there are images for databases, web browsers, web servers, and message queues. Therefore, we can run them as containers within our tests.

2. Requirements

The Testcontainers library can be used with Java 8 and higher. It's also compatible with the JUnit Rules API.

First, let's define the Maven dependency for the core functionality:

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers</artifactId>
    <version>1.19.6</version>
</dependency>
```

There are also modules for specialized containers. In this tutorial, we'll be using *PostgreSQL* and *Selenium.*

Let's add the relevant dependencies:

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>postgresql </artifactId>
    <version>1.19.6</version>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>selenium </artifactId>
    <version>1.19.6</version>
</dependency>
```

We can find the latest versions on Maven Central (https://mvnrepository.com/artifact/org.testcontainers/testcontainers).

Also, we need Docker to run containers. Refer to Docker documentation (https://docs.docker.com/install/) for installation instructions.

Make sure you're able to run Docker containers in your test environment.

3. Usage

Let's configure a generic container rule:

```
@ClassRule
public static GenericContainer simpleWebServer
 = new GenericContainer("alpine:3.2")
   .withExposedPorts(80)
   .withCommand("/bin/sh", "-c", "while true; do echo "
     + "\"HTTP/1.1 200 OK\n\nHello World!\" | nc -l -p 80; done");
```

We construct a *GenericContainer* test rule by specifying a docker image name. Then, we configure it with builder methods:

- We use *withExposedPorts* to expose a port from the container
- *withCommand* defines a container command. It will be executed when the container starts.

The rule is annotated with *@ClassRule*. As a result, it will start the Docker container before any test in that class runs. The container will be destroyed after all methods are executed.

If you apply *@Rule* annotation, the *GenericContainer* rule will start a new container for each test method. And it will stop the container when that test method finishes.

We can use an IP address and port to communicate with the process running in the container:

```
@Test
public void
givenSimpleWebServerContainer_whenGetReuqest_thenReturnsResponse()
  throws Exception {
    String address = "http://"
      + simpleWebServer.getContainerIpAddress()
      + ":" + simpleWebServer.getMappedPort(80);
    String response = simpleGetRequest(address);

    assertEquals(response, "Hello World!");
}
```

4. Usage Modes

There are several *usage modes* of the test containers. We saw an example of running a *GenericContainer.*

Testcontainers library has also rule definitions with specialized functionality. They are for containers of common databases like MySQL, PostgreSQL; and others like web clients.

Although we can run them as generic containers, the specializations provide extended convenience methods.

4.1. Databases

Let's assume we need a database server for data-access-layer integration tests. We can run databases in containers with the help of the Testcontainers library.

For example, we fire up a PostgreSQL container with *PostgreSQLContainer* rule. Then, we're able to use helper methods. **These are *getJdbcUrl, getUsername, getPassword* for database connection:**

```
@Rule                          (/)
public PostgreSQLContainer postgresContainer = new
PostgreSQLContainer();

@Test
public void whenSelectQueryExecuted_thenResulstsReturned()
  throws Exception {
    String jdbcUrl = postgresContainer.getJdbcUrl();
    String username = postgresContainer.getUsername();
    String password = postgresContainer.getPassword();
    Connection conn = DriverManager
      .getConnection(jdbcUrl, username, password);
    ResultSet resultSet =
      conn.createStatement().executeQuery("SELECT 1");
    resultSet.next();
    int result = resultSet.getInt(1);

    assertEquals(1, result);
}
```

It is also possible to run PostgreSQL as a generic container. But it'd be more difficult to configure the connection.

4.2. Web Drivers

Another useful scenario is to run containers with web browsers. *BrowserWebDriverContainer* rule enables running *Chrome* and *Firefox* in *docker-selenium* containers. Then, we manage them with *RemoteWebDriver.*

This is very useful for automating UI/Acceptance tests for web applications:

```
@Rule
public BrowserWebDriverContainer chrome = new
BrowserWebDriverContainer()
  .withCapabilities(new ChromeOptions());
@Test
public void whenNavigatedToPage_thenHeadingIsInThePage() {
    RemoteWebDriver driver = chrome.getWebDriver();
    driver.get("http://example.com");
    String heading = driver.findElement(By.xpath("/html/body/div/h1"))
      .getText();

    assertEquals("Example Domain", heading);
}
```

4.3. Docker Compose

If the tests require more complex services, we can specify them in a *docker-compose* file:

```
simpleWebServer:
  image: alpine:3.2
  command: ["/bin/sh", "-c", "while true; do echo 'HTTP/1.1 200
OK\n\nHello World!' | nc -l -p 80; done"]
```

Then, we use *DockerComposeContainer* rule. This rule will start and run services as defined in the compose file.

We use *getServiceHost* and *getServicePost* methods to build connection address to the service:

```
@ClassRule
public static DockerComposeContainer compose =
  new DockerComposeContainer(
    new File("src/test/resources/test-compose.yml"))
      .withExposedService("simpleWebServer_1", 80);

@Test
public void
givenSimpleWebServerContainer_whenGetReuqest_thenReturnsResponse()
  throws Exception {

    String address = "http://" +
compose.getServiceHost("simpleWebServer_1", 80) + ":" +
compose.getServicePort("simpleWebServer_1", 80);
    String response = simpleGetRequest(address);

    assertEquals(response, "Hello World");
}
```

5. Conclusion

We saw how we could use the *Testcontainers* library. It eases developing and running integration tests.

We used the *GenericContainer* rule for containers of given docker images. Then, we looked at *PostgreSQLContainer, BrowserWebDriverContainer* and *DockerComposeContainer* rules. They give more functionality for

specific use cases.

(/)

The code backing this article is available on GitHub. Once you're **logged in as a Baeldung Pro Member (/members/)**, start learning and coding on the project.

Traditional keyword-based search methods rely on exact word matches, **often leading to irrelevant results** depending on the user's phrasing.

By comparison, using a vector store allows us to represent the data as vector embeddings, based on meaningful relationships. We can then compare the meaning of the user's query to the stored content, and **retrieve more relevant, context-aware results.**

Explore how to **build an intelligent chatbot using MongoDB Atlas, Langchain4j** and Spring Boot:

>> Building an AI Chatbot in Java With Langchain4j and MongoDB Atlas (/MongoDB-NPI-EA-6-5tjp)

LEARN SPRING BOOT SERIES (/SPRING-BOOT)
SPRING TUTORIAL (/SPRING-TUTORIAL)
GET STARTED WITH JAVA (/GET-STARTED-WITH-JAVA-SERIES)
ALL ABOUT STRING IN JAVA (/JAVA-STRING)
SECURITY WITH SPRING (/SECURITY-SPRING)
JAVA COLLECTIONS (/JAVA-COLLECTIONS)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS/)
PARTNER WITH BAELDUNG (/PARTNERS/WORK-WITH-US)
EBOOKS (/LIBRARY/)
FAQ (/LIBRARY/FAQ)

BAELDUNG PRO (/MEMBERS/)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)

PRIVACY MANAGER