Testcontainers

jim stafford

Fall 2024 v2020-08-10: Built: 2024-11-19 21:43 EST

Table of Contents

Chapter 1. Introduction

In a previous section we implemented "unit integration tests" with in-memory instances for back-end resources. We later leveraged Docker and Docker Compose to implement "integration tests" with real resources operating in a virtual environment. We self-integrated Docker Compose in that later step, using several Maven plugins and Maven’s integration testing phases.

In this lecture I will demonstrate an easier, more seamless way to integrate Docker Compose into our testing using Testcontainers. This will allow us to drop back into the Maven test phase and implement the integration tests using straight forward unit test constructs.

1.1. Goals

You will learn:

- how to better integrate Docker and DockerCompose into unit tests
- how to inject dynamically assigned values into the application context startup

1.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. implement an integration unit test using Docker Compose and Testcontainers library
2. implement a `Spring DynamicPropertySource` to obtain dynamically assigned port numbers in time for concrete URL injections
3. execute shell commands from a JUnit test into a running Docker container using Testcontainers library
4. establish client connection to back-end resources to inspect state as part of the test

Chapter 2. Testcontainers Overview

Testcontainers is a Java library that supports running Docker containers within JUnit tests and other test frameworks.

Testcontainers provides a layer of integration that is well aware of the integration challenges that are present when testing with Docker images and can work both outside and inside a Docker container itself.

Spring making changes to support Testcontainers

As a self observation — by looking at documentation, articles, and timing of feature releases — it is my opinion that Spring and Spring Boot are very high on Testcontainers and have added features to their framework to help make testing with Testcontainers as seamless as possible.

Chapter 3. Example

This example builds on the previous Docker Compose lecture that uses the same Votes and Elections services. The main difference is that we will be directly interfacing with the Docker images using Testcontainers in the `test` phase versus starting up the resources at the beginning of the tests and shutting down at the end.

By having such direct connect with the containers — we can control what gets reused from test to test. Sharing reused container state between tests can be error prone. Starting up and shutting down containers takes a noticeable amount of time to complete. Alternatively, we want to have more control over when we do which approach without going through extreme heroics.

3.1. Maven Dependencies

The following lists the Testcontainers Maven dependencies. The core library calls are within the `testcontainers` artifact and JUnit-specific capabilities are within the `junit-jupiter` artifact. I have declared `junit-jupiter` dependency at the `test` scope and `testcontainers` at `compile` (default) scope because

- this is a pure test module — with no packaged implementation code
- helper methods have been placed in `src/main`
- as the test suite grows larger, this allows the helper code and other test support features to be shared among different testing modules

Testcontainers Maven Dependencies

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers</artifactId> ①
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId> ②
    <scope>test</scope>
</dependency>
```

① core Testcontainers calls will be placed in `src/main` to begin to form a test helper library

② JUnit-specific calls will be placed in `src/test`

3.2. Main Tree

The module's main tree contains a source copy of the Docker Compose file describing the network of services, a helper class that encapsulates initialization and configuration status of the network, and a JMS listener that can be used to subscribe to the JMS messages between the Voters and Elections services.

Module Main Tree

```
src/main/
|-- java
|   `-- info
|       `-- ejava
|           `-- examples
|               `-- svc
|                   `-- docker
|                       `-- votes
|                           |-- ClientTestConfiguration.java
|                           `-- VoterListener.java
`-- resources
    `-- docker-compose-votes.yml
```

3.3. Test Tree

The test tree contains artifacts that are going to pertain to this test only. The JUnit test will rely heavily on the artifacts in the `src/main` tree and we should try to work like that might come in from a library shared by multiple integration unit tests.

Module Test Tree

```
src/test/
|-- java
|   `-- info
|       `-- ejava
|           `-- examples
|               `-- svc
|                   `-- docker
|                       `-- votes
|                           `-- ElectionCNTest.java
`-- resources
    |-- application.properties
    `-- junit-platform.properties
```

Chapter 4. Example: Main Tree Artifacts

The main tree contains artifacts that are generic to serving up the network for specific tests hosted in the `src/test` tree. This division has nothing directly related to do with Testcontainers — except to show that once we get one of these going, we are going to want more.

4.1. Docker Compose File

Our Docker Compose file is tucked away within the test module since it is primarily meant to support testing. I have purposely removed all external port mapping references because they are not needed. Testcontainers will provide another way to map and locate the host port#. I have eliminated the build of the image. It should have been built by now based on Maven module dependencies. However, if we can create a resolvable source reference to the module — Testcontainers will make sure it is built.

Docker Compose File For Test

```yaml
version: '3.8'
services:
  mongo:
    image: mongo:4.4.0-bionic
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: secret
  postgres:
    image: postgres:12.3-alpine
    environment:
      POSTGRES_PASSWORD: secret
  activemq:
    image: rmohr/activemq:5.15.9
  api:
    image: dockercompose-votes-api:latest
    depends_on:
      - mongo
      - postgres
      - activemq
    environment:
      - spring.profiles.active=integration
      - MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin
      - DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
```

4.2. Docker Compose File Reference

Testcontainers will load one to many layered Docker Compose files — but insists that they each be expressed as a `java.io.File`. If we assume the code in the `src/main` tree is always going to be in source form — then we can make a direct reference there. However, assuming that this could be coming from a JAR — I decided to copy the data from classpath and into a referencable file in the target tree.

Obtaining Portable File Reference from Classpath

```
import java.io.File;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
...
public static File composeFile() {
    Path targetPath = Paths.get("target/docker-compose-votes.yml"); ②
    try (InputStream is = ClientTestConfiguration.class ①
                        .getResourceAsStream("/docker-compose-votes.yml")) {
        Files.copy(is, targetPath, StandardCopyOption.REPLACE_EXISTING);
    } catch (IOException ex) {
        Assertions.fail("error creating source Docker Compose file", ex);
    }
    return targetPath.toFile();
}
```

① assuming worse case that the file will be coming in from a test support JAR

② placing referencable file in target path — actual name does not matter

The following shows the source and target locations of the Docker Compose file written out.

Writing Out Docker Compose File

```
target/
| `-- classes/
|       `-- docker-compose-votes.yml ①
`-- docker-compose-votes.yml ②
```

① source coming from classpath

② target written as a known file in target directory

4.3. DockerComposeContainer

Testcontainers provides many containers — including a generic Docker container, image-specific containers, and a Docker Compose container. We are going to leverage our knowledge of Docker Compose and the encapsulation of details of the Docker Compose file here and have Testcontainers directly parse the Docker Compose file.

The example shows us supplying a project name, file reference(s), and then exposing individual container ports from each of the services. Originally — only the API port needed to be exposed. However, because of the simplicity to do more with Testcontainers, I am going to expose the other ports as well. Testcontainers will also conveniently wait for activity on each of the ports when the network is started — before returning control back to our test. This can eliminate the need for "is server ready?" checks.

```
public static DockerComposeContainer testEnvironment() {
    DockerComposeContainer env =
        new DockerComposeContainer("testcontainers-votes", composeFile())
            .withExposedService("api", 8080) ①
            .withExposedService("activemq", 61616) ②
            .withExposedService("postgres", 5432) ②
            .withExposedService("mongo", 27017) ②
            .withLocalCompose(true); ③
    return env;
}
```

① exposing container ports using random port and will wait for container port to become active

② optionally exposing lower level resource services to demonstrate further capability

③ indicates whether this is a host machine that will run the images as children or whether this is running as a Docker image and the images will be tunneled (wormholed) out as sibling containers

4.4. Obtaining Runtime Port Numbers

At runtime, we can obtain the assigned hostname and port numbers by calling `getServiceHost()` and `getServicePort()` with the service name and container port we exposed earlier.

Obtaining Runtime Port Numbers

```
DockerComposeContainer env = ClientTestConfiguration.testEnvironment(); ①
...
env.start(); ②

env.getServicePort("api", 8080)); ③
env.getServiceHost("mongo", null); ④
env.getServicePort("mongo", 27017);
env.getServiceHost("activemq", null);
env.getServicePort("activemq", 61616);
env.getServiceHost("postgres", null);
env.getServicePort("postgres", 5432);
```

① Docker Compose file is parsed

② network/services must be started in order to determine mapped host port numbers

③ referenced port must have been listed with `withExposedService()` earlier

④ hostname is available as well if ever not available on `localhost`. Second param not used.

Chapter 5. Example: Test Tree Artifacts

5.1. Primary NTest Setup

We construct our test as a normal Spring Boot integration unit test (NTest) except we have no core application to include in the Spring context — everything is provided through the test configuration. There is no need for a web server — we will use HTTP calls from the test's JVM to speak to the remote web server.

Docker images and Docker Compose networks of services take many seconds (~10-15secs) to completely startup. Thus we want to promote some level of efficiency between tests. We will instantiate and store the `DockerComposeContainer` in a static variable, initialize and shutdown once per test class, and reuse for each test method within that class. Since we are sharing the same network for each test method — I am also demonstrating the ability to control the order of the test methods.

Lastly — we can have the lifecycle of the network integrated with the JUnit test case by adding the `@Testcontainers` annotation to the class and the `@Container` annotation to the field holding the overall container. This takes care of automatically starting and stopping the network defined in the `env` variable.

Primary NTest Setup

```
import org.testcontainers.containers.DockerComposeContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;
...
@Testcontainers ⑤
@TestMethodOrder(MethodOrderer.OrderAnnotation.class) ④
@SpringBootTest(classes={ClientTestConfiguration.class}, ①
    webEnvironment = SpringBootTest.WebEnvironment.NONE) ②
public class ElectionCNTest {
    @Container ⑤
    private static DockerComposeContainer env = ③
                                ClientTestConfiguration.testEnvironment();

    @Test @Order(1)
    public void vote_counted_in_election() { //...
    @Test
    @Order(3) ④
    public void test3() { vote_counted_in_election(); }
    @Test @Order(2)
    public void test2() { vote_counted_in_election(); }
```

① Only test constructs in our application context — no application beans

② we do not need a web server — we are the client of a web server

③ sharing same network in all tests within this test case

④ controlling order of tests when using shared network

⑤ `@Testcontainers` and `@Container` annotations integrate the lifecycle of the network with the test case

5.2. Injecting Dynamically Assigned Port#s

We soon hit a chicken-and-the-egg problem when we attempt to inject the URLs int the test class.

```
@Autowired
private URI votesUrl;
@Autowired
private URI electionsUrl;
```

- The test class attempts to `@Autowire` URLs for the services

- the `@Bean` factories build the URLs from the host and port number

```
@Bean
public URI baseUrl() {
    return UriComponentsBuilder
.newInstance()
            .host(host)
            .port(port)//...
@Bean
public URI votesUrl(URI baseUrl) { //...
@Bean
public URI electionsUrl(URI baseUrl)
{//...
```

- the host and port number are injected into the configuration class using values from the Spring context

```
@SpringBootConfiguration()
public class ClientTestConfiguration {
  @Value("${it.server.host:localhost}")
  private String host;
  @Value("${it.server.port:9090}")
  private int port;
```

```
@Container
private static DockerComposeContainer
env =
 ClientTestConfiguration.testEnvironment
();
 // --
@Autowired
private URI votesUrl;
@Autowired
private URI electionsUrl;
```

- the port number information is not available until after the network is started and the network is not started until just before the first test

5.3. DynamicPropertySource

In what seemed like a special favor to Testcontainers — Spring added a `DynamicPropertySource` construct to the framework that allows for a property to be supplied late in the startup process.

- after starting the network but prior to injecting any URIs and running a test, Spring invokes the following annotated method in the JUnit test so that it may inject any late properties.

```
@DynamicPropertySource
private static void properties(DynamicPropertyRegistry registry) { ①
    ClientTestConfiguration.initProperties(registry, env);
}
```

① method is required to be static

- the callback method can then supply the missing property that will allow for the URI injections needed for the tests

```
public static void initProperties(DynamicPropertyRegistry registry,
DockerComposeContainer env){
    registry.add("it.server.port", ()->env.getServicePort("api", 8080));
    //...
}
```

Nice!

5.4. Injections Complete prior to Tests

With the injections in place, we can show that URLs with the dynamically assigned port numbers. We also have the opportunity to have the test wait for anything we can think of. Testcontainers waited for the container port to become active. The example below instructs Testcontainers to wait for our API calls to be available as well. This eliminates the need for that ugly `@BeforeEach` call in the last lecture where we needed to wait for the API server to be ready before running the tests.

Example @BeforeEach

```
@BeforeEach
public void init() throws IOException, InterruptedException {
    log.info("votesUrl={}", votesUrl); ①
    log.info("electionsUrl={}", votesUrl);

    /**
     * wait for various events relative to our containers
     */
    env.waitingFor("api", Wait.forHttp(votesUrl.toString())); ②
    env.waitingFor("api", Wait.forHttp(electionsUrl.toString()));
```

① logging injected URLs with dynamically assigned host port numbers

② instructing Testcontainers to also wait for the API to come available

Example URLs with Dynamically Assigned Port Numbers

```
ElectionCNTest#init:73 votesUrl=http://localhost:32989/api/votes
ElectionCNTest#init:74 electionsUrl=http://localhost:32989/api/votes
```

Chapter 6. Exec Commands

Testcontainers gives us the ability to execute commands against specific running containers. The following executes the database CLI interfaces, requests a dump of information, and then obtains the results from stdout.

Example Commands Issued to Running Containers

```
import org.testcontainers.containers.Container.ExecResult;
import org.testcontainers.containers.ContainerState;
...
ContainerState mongo = (ContainerState) env.getContainerByServiceName("mongo_1")
        .orElseThrow();
ExecResult result = mongo.execInContainer("mongo",
        "-u", "admin", "-p", "secret", "--authenticationDatabase", "admin",
        "--eval", "db.getSiblingDB('votes_db').votes.find()");
log.info("voter votes = {}", result.getStdout());

ContainerState postgres = (ContainerState)env.getContainerByServiceName("postgres_1")
        .orElseThrow();
result = postgres.execInContainer("psql",
        "-U", "postgres",
        "-c", "select * from vote");
log.info("election votes = {}", result.getStdout());
```

That is a bit unwieldy, but demonstrates what we can do from a shell perspective and we will improve on this in a moment by using the API.

6.1. Exec MongoDB Command Output

The following shows the stdout obtained from the MongoDB container after executing the login and query of the `votes` collection.

Exec MongoDB Command Output

```
ElectionCNTest#init:105 voter votes = MongoDB shell version v4.4.0
connecting to:
mongodb://127.0.0.1:27017/?authSource=admin&compressors=disabled&gssapiServiceName=mon
godb
Implicit session: session { "id" : UUID("5f903fe7-b43c-4ce8-b6ae-7ef53fcbf434") }
MongoDB server version: 4.4.0
{ "_id" : ObjectId("5f357fef01737362e202a96d"), "date" : ISODate("2020-08-
13T18:01:19.872Z"), "source" : "b67e012e-3e2f-4a66-b24b-b64d06d9b4c2", "choice" :
"quisp-de5fd4f2-8ab8-4997-852e-2bfb97862c87", "_class" :
"info.ejava.examples.svc.docker.votes.dto.VoteDTO" }
{ "_id" : ObjectId("5f357ff001737362e202a96e"), "date" : ISODate("2020-08-
13T18:01:20.515Z"), "source" : "af366d9b-53cb-4487-8f21-e634eca08d67", "choice" :
"quake-784f3df6-c6c4-4c3b-8d45-58636b335096", "_class" :
"info.ejava.examples.svc.docker.votes.dto.VoteDTO" }
```

```
...
```

6.2. Exec Postgres Command Output

The following shows the stdout from the Postgres container after executing the login and query of the `VOTE` table.

Exec Postgres Command Output

```
ElectionCNTest#init:99 election votes =
            id            |                  choice                  |          date
|              source
--------------------------+--------------------------------------------+
-------------------------+-------------------------------------
 5f357fef01737362e202a96d | quisp-de5fd4f2-8ab8-4997-852e-2bfb97862c87 | 2020-08-13
18:01:19.872 | b67e012e-3e2f-4a66-b24b-b64d06d9b4c2
 5f357ff001737362e202a96e | quake-784f3df6-c6c4-4c3b-8d45-58636b335096 | 2020-08-13
18:01:20.515 | af366d9b-53cb-4487-8f21-e634eca08d67
 ...
(6 rows)
```

Chapter 7. Connect to Resources

Executing a command against a running service may be useful for interactive work. In fact, we could create a breakpoint in the test and then manually go out to inspect the back-end resources (using `docker ps` to locate the container and `docker exec` to run a shell within the container) if we have access to the host network.

However, it can be clumsy to make any sense of the stdout result when writing an automated test. If we actually need to get state from the resource — it will be much simpler to use a first-class resource API to obtain results.

Lets do that now.

7.1. Maven Dependencies

To add resource clients for our three back-end resources we just need to add the following familiar dependencies. We first introduced them in the API module's dependencies in an earlier lecture.

Back-end Resource Connection Dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
```

7.2. Injected Clients

The following resource clients will be injected into the test class. These are made available by the individual AutoConfiguration class for the resource types.

Resource Clients to be Injected

```
@Autowired
private MongoClient mongoClient;
@Autowired
private JmsTemplate jmsTemplate;
@Autowired
private JdbcTemplate jdbcTemplate;
```

The AutoConfiguration classes will require the following properties defined

Required URL properties

```
spring.data.mongodb.uri
spring.activemq.broker-url
spring.datasource.url
```

7.3. URL Templates

The URLs can be built using the following hard-coded helper methods as long as we know the host and port number of each service.

URL Template Helper Methods

```
public static String mongoUrl(String host, int port) {
    return String.format("mongodb://admin:secret@%s:%d/votes_db?authSource=admin",
host, port);
}
public static String jmsUrl(String host, int port) {
    return String.format("tcp://%s:%s", host, port);
}
public static String jdbcUrl(String host, int port) {
    return String.format("jdbc:postgresql://%s:%d/postgres", host, port);
}
```

7.4. Providing Dynamic Resource URL Declarations

The host and port numbers can be supplied from the network — just like we did with the API. Therefore, we can expand the dynamic property definition to include the three other properties.

Dynamic Property Definitions

```
public static void initProperties(DynamicPropertyRegistry registry,
DockerComposeContainer env) {
    registry.add("it.server.port", ()->env.getServicePort("api", 8080));
    registry.add("spring.data.mongodb.uri",()-> mongoUrl( ②
            env.getServiceHost("mongo", null),
            env.getServicePort("mongo", 27017)));  ①
    registry.add("spring.activemq.broker-url", ()->jmsUrl(
            env.getServiceHost("activemq", null),
            env.getServicePort("activemq", 61616)));
    registry.add("spring.datasource.url",()->jdbcUrl(
            env.getServiceHost("postgres", null),
            env.getServicePort("postgres", 5432)));
}
```

① dynamically assigned host port numbers are made available from running network

② properties are provided to Spring late in the startup process — but in time to inject before the tests

7.5. Application Properties

The dynamically created URLs properties will be joined up with the following hard-coded application properties to complete and connection information.

Hard-coded Application Properties

```
#activemq
spring.jms.pub-sub-domain=true

#postgres
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.username=postgres
spring.datasource.password=secret
```

7.6. JMS Listener

To obtain the published JMS messages — we add the following component with a JMS Listener method. This will print a debug of the message and increment a counter.

```
//...
import org.springframework.jms.annotation.JmsListener;
import jakarta.jms.JMSException;
import jakarta.jms.Message;
import jakarta.jms.TextMessage;

@Component
@Slf4j
public class VoterListener {
    @Getter
    private AtomicInteger msgCount=new AtomicInteger(0);

    @JmsListener(destination = "votes")
    public void receive(Message msg) throws JMSException {
        log.info("jmsMsg={}, {}", msgCount.incrementAndGet(), ((TextMessage) msg)
.getText());
    }
}
```

We must add the JMS listener class to the Spring application context of the test. The following example shows that being explicitly done in the `@SpringBootTest.classes` annotation.

Add Component to Test Application Context

```
@SpringBootTest(classes={ClientTestConfiguration.class, VoterListener.class}, ①
    webEnvironment = SpringBootTest.WebEnvironment.NONE)
```

```
//...
public class ElectionCNTest {
```

① adding VoterListener component class to Spring context

7.7. Obtain Client Status

The following shows a set calls to the client interfaces to show the basic capability to communicate with the network services. This gives us the ability to add debug or obscure test verification.

Example Network Service Client Calls

```
@BeforeEach
public void init() throws IOException, InterruptedException {
    ...
    /**
     * connect directly to explosed port# of images to obtain sample status
     */
    log.info("mongo client vote count={}", ①
        mongoClient.getDatabase("votes_db").getCollection("votes").countDocuments());
    log.info("activemq msg={}", listener.getMsgCount().get()); ②
    log.info("postgres client vote count={}", ③
        jdbcTemplate.queryForObject("select count (*) from vote", Long.class));
```

① getting the count of vote documents from MongoDB client

② getting number of messages received from JMS listener

③ getting the number of vote rows from Postgres client

7.8. Client Status Output

The following shows an example of the client output in the `@BeforeEach` method, captured after the first test and before the second test.

Example Client Status Output

```
ElectionCNTest#init:85 mongo client vote count=6
ElectionCNTest#init:87 activemq msg=6
ElectionCNTest#init:88 postgres client vote count=6
```

Very complete!

Chapter 8. Summary

In this module, we learned:

- how to more seamlessly integrate Docker and DockerCompose into unit tests using Testcontainers library
- how to inject dynamically assigned values into the application context to allow them to be injected into components at startup
- to execute shell commands from a JUnit test into a running container using Testcontainers library
- to establish client connection to back-end resources from our JUnit JVM operating the unit test
  - in the event that we need this information to verify test success or simply perform some debug of the scenario

Although integration tests should never replace unit tests, the capability demonstrated in this lecture shows how we can create very capable end-to-end tests to verify the parts will come together correctly. For example, it was not until I wrote and executed the integration tests in this lecture that I discovered I was accidentally using JMS queuing semantics versus topic semantics between the two services. When I added the extra JMS listener — the Elections Service suddenly started loosing messages. Good find!!