Testcontainers Unit Integration Testing

jim stafford

Fall 2024 v2024-06-14: Built: 2024-11-19 21:42 EST

Table of Contents

Chapter 1. Introduction

In previous sections we implemented "unit integration tests" (NTests) and with in-memory instances for back-end resources. This was lightweight, fast, and convenient but lacked the flexibility to work with more realistic implementations.

We later leveraged Docker, Docker Compose, and Maven Failsafe to implement heavyweight "integration tests" with real resources operating in a virtual environment. We self-integrated Docker and Docker Compose using several Maven plugins (`resources`, `build-helper-maven-plugin`, `spring-boot-maven-plugin`, `docker-maven-plugin`, `docker-compose-maven-plugin`, `maven-failsafe-plugin`, …) and Maven's integration testing phases. It was a mouthful, but it worked.

In this lecture I will introduce an easier, more seamless way to integrate Docker images 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 enabling certain use cases:

- Data access layer implementation testing
- External service integration an testing
- Lightweight development overhead to certain types of integration tests

We won't dive deep into database topics yet. We will continue to just focus on obtaining a database instance and connection.

1.1. Goals

You will learn:

- how to more easily integrate Docker images and Docker Compose into 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 a unit integration test using Docker and Testcontainers — specifically with the `PostgresSQLContainer`, `MongoDBContainer`, and `GenericContainer`
2. implement a Spring `@DynamicPropertySource` to obtain dynamically assigned properties in time for concrete component injections
3. implement a lightweight Maven Failsafe integration test using Docker Compose and Testcontainers — specifically with the `DockerComposeContainer`

Chapter 2. Testcontainers Overview

Testcontainers is a Java library that supports running Docker containers within JUnit tests and other test frameworks. We do not need extra Maven plugins to manage the containers. Testcontainers operates within the JVM and integrates with JUnit.

Maven Plugins Still Needed to Build Internal Artifacts

We still need Maven plugins to build the artifacts (e.g., executable JAR) to package into the Docker image if testing something constructed locally. We just don't need the web of plugins surrounding the `integration-test` phases.

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 (i.e., it can operate in local development and CI/CD Docker container environments).

Based on the changes made in Spring Boot 3, Spring and Spring Boot are very enthusiastic about Testcontainers. Spring Boot has also dropped its direct support of Flapdoodle. Flapdoodle is an embedded MongoDB test-time framework also geared for testing like Testcontainers — but specific to MongoDB and constrained to certain versions of MongoDB. By embracing Testcontainers, one is only constrained by what can be packaged into a Docker image.

Flapdoodle can still be integrated with Spring/Spring Boot. However, as we will cover in the MongoDB lectures (and specifically in the Spring Boot 3 porting notes), the specific dependencies and properties have changed. They are no longer directly associated with Spring/Spring Boot and now come from Flapdoodle sources.

Chapter 3. Base Example

I will start the Testcontainers example with some details using a database connection test that should be familiar to you from earlier testing lectures. This will allow you to see many familiar constructs before we move on to the Testcontainers topics. The goal of the database connection example(s) is to establish a database instance and connection to it. Database interaction will be part of later persistence lectures.

3.1. Maven Dependencies

We will be injecting components that will provide connections to the RDBMS and MongoDB databases. To support that, we start with the core JPA and MongoDB frameworks using their associated starter artifacts.

RDBMS and MongoDB Core Dependencies

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

3.2. Module Main Tree

The module's main tree contains a Spring MVC controller that will be injected with a RDBMS `DataSource` and `MongoClient`. The other source files are there to supply a generic Spring Boot application.

Module Main Tree

```
src/main/java
`-- info
    `-- ejava
        `-- examples
            `-- svc
                `-- tcontainers
                    `-- hello
                        |-- TestcontainersHelloApp.java
                        `-- controllers
                            |-- ExceptionAdvice.java
                            `-- HelloDBController.java ①
```

① controller injected with RDBMS `DataSource` and `MongoClient`

3.3. Injected RestController Component

We will use a familiar `@RestController` as an example server-side component to inject the lower-level `DataSource` and `MongoClient` components. The `DataSource` and `MongoClient` components encapsulate the database connections.

The database components are declared optional in the event that we are focusing on just RDBMS, MongoDB, or remote client in a specific test and do not need one or both database connections.

RestController Injected with DataSource and MongoClient Components

```
import com.mongodb.client.MongoClient;
import javax.sql.DataSource;
...
@RestController
@Slf4j
public class HelloDBController {
    private final DataSource dataSource;
    private final MongoClient mongoClient;
①
    public HelloDBController(@Autowired(required = false) DataSource dataSource,
                             @Autowired(required = false) MongoClient mongoClient) {
        this.dataSource = dataSource;
        this.mongoClient = mongoClient;
    }
```

① database components declared optional in event a specific test has not adequately configured the alternate

@RestControllers do not Perform Database Actions

`@RestController` components do not normally directly interact with a database. Their primary job is to interact with the web and accomplish tasks using injected service components. The example is thinned down to a single component — to do both/all — for simplicity.

3.3.1. RDBMS Connection Test Endpoint

The RDBMS path will make a connection to the database and return the resulting JDBC URL. A successful return of the expected URL will mean that the RDBMS connection was correctly established.

RDBMS Endpoint Tests Connection and Returns JDBC URL

```
@GetMapping(path="/api/hello/jdbc",
        produces = {MediaType.TEXT_PLAIN_VALUE})
public String helloDataSource() throws SQLException {
    try (Connection conn = dataSource.getConnection()) {
        return conn.getMetaData().getURL();
    }
```

```
}
```

3.3.2. MongoDB Connection Test Endpoint

The MongoDB path will return a cluster description that identifies the database connection injected. However, to actually test the connection — we are requesting a list of database names from MongoDB. The cluster description does not actively make a connection.

MongoDB Endpoint Tests Connection and Returns Cluster Description

```
@GetMapping(path="/api/hello/mongo",
        produces = {MediaType.TEXT_PLAIN_VALUE})
public String helloMongoClient() {
    log.info("dbNames: ", mongoClient.listDatabaseNames().first()); //test connection
    return mongoClient.getClusterDescription().getShortDescription();
}
```

3.4. In-Memory Example

I will start the demonstration with the RDBMS path and the H2 in-memory database to cover the basics.

3.4.1. Maven Dependencies

I have already shown the required JPA dependencies. For this test, we need to add the H2 database as a test dependency.

RDBMS and H2 Test Maven Dependencies

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>  ①
</dependency>
```

① H2 is used only in tests

3.4.2. Unit Integration Test Setup

The following snippet shows the unit integration test (NTest) setup using the in-memory H2 database in near full detail. The expected JDBC URL and actual DataSource is injected along with setup to communicate with the `@RestController`.

Unit Integration Test Core Setup

```
@SpringBootTest(classes={TestcontainersHelloApp.class,
        ClientNTestConfiguration.class},②
        properties={"spring.datasource.url=jdbc:h2:mem:testcontainers"},①
```

```
        webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@ActiveProfiles({"test"})
@Slf4j
class HelloH2InMemoryNTest {
    @Value("${spring.datasource.url}")
    private String expectedJdbcUrl; ③
    @Autowired
    private DataSource dataSource; ③
    @Autowired
    private RestTemplate anonymousUser;
    private UriBuilder dbUrl;

    @BeforeEach
    void init(@LocalServerPort int port) {
        dbUrl = UriComponentsBuilder.fromHttpUrl("http://localhost").port(port).path(
"/api/hello/{db}");
    }
```

① provides a concrete database URL for this test

② configuration declares the injected RestTemplate

③ DataSource will represent connection to provided URL property

Known, non-Dynamic Property values can be Declared Using Simple Properties

Notice the `spring.data.source.url` value is constant and was known well in advance of running the JUnit JVM. This allows it to be defined as a static property. That won't always be the case and will address my dynamic cases shortly, within this lecture.

3.4.3. Inject DataSource into Test

The following test verifies that the JDBC URL provided by the `DataSource` will be the same as the `spring.datasource.url` property injected. This verifies our test can connect to the database and satisfy the authentication requirements (none in this in-memory case).

Verify Connection using DataSource

```
@Test
void can_get_connection() throws SQLException {
    //given
    then(dataSource).isNotNull();
    String jdbcUrl;
    //when
    try(Connection conn=dataSource.getConnection()) {
        jdbcUrl=conn.getMetaData().getURL();
    }
    //then
    then(jdbcUrl).isEqualTo(expectedJdbcUrl);
}
```

3.4.4. Inject DataSource into Server-side

The next test verifies that the server-side component under test (`@RestController`) was injected with a `DataSource`, can complete a connection, and returns the expected JDBC URL.

Verify Server-side Connection

```
@Test
void server_can_get_jdbc_connection() {
    //given
    URI url = dbUrl.build("jdbc");
    RequestEntity<Void> request = RequestEntity.get(url).build();
    //when
    ResponseEntity<String> response = anonymousUser.exchange(request, String.class);
    //then
    String jdbcUrl=response.getBody();
    then(jdbcUrl).isEqualTo(expectedJdbcUrl);
```

At this point we have a pretty straight forward unit integration framework that is similar to ones we have used in past lectures. Lets move onto using Postgres and leverage Testcontainers to more easily integrate the Postgres Docker image.

Chapter 4. Postgres Container Example

At this point we want to take our RDBMS development and testing to the next level and work with a real RDBMS instance of our choosing - Postgres.

4.1. Maven Dependencies

The following snippet lists the primary Testcontainers Maven dependencies. Artifact versions are defined in the `testcontainers-bom`, which is automatically imported by the `spring-boot-dependencies` BOM. We add the `testcontainers` dependency for core library calls and `GenericContainer` management. We add the `junit-jupiter` artifact for the JUnit-specific integration. They are both defined with `scope=test` since there is no dependency between our production code and the Testcontainers.

Testcontainers Maven Dependencies

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

① core Testcontainers calls

② JUnit-specific calls

The Postgres database driver is added as a `scope=runtime` dependency so that we can communicate with the Postgres database instance during deployment and testing. The Postgres-specific Testcontainers artifact is added as a dependency to supply a convenient API for obtaining a running Postgres Docker image. Its `scope=test` restricts it only to testing.

Postgres/Testcontainers Dependencies

```
<dependency> ①
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>

<dependency> ②
    <groupId>org.testcontainers</groupId>
    <artifactId>postgresql</artifactId>
    <scope>test</scope>
</dependency>
```

① needed during deployment and tests

② needed during test only

4.2. Unit Integration Test Setup

The following shows the meat of how we setup our Postgres container and integrate it into our RDBMS unit integration test. There is a lot to conceptually unpack and will do so in the follow-on sections below.

PostgresSQLContainer Test Setup

```
@SpringBootTest ...
@Testcontainers //manages lifecycle of container
@Slf4j
class HelloPostgresTestcontainerNTest {
    @Container
    private static PostgreSQLContainer postgres = new PostgreSQLContainer<>(
"postgres:12.3-alpine");
    @DynamicPropertySource
    private static void addLateSpringContextProperties(DynamicPropertyRegistry
registry) {
        registry.add("spring.datasource.url",()->postgres.getJdbcUrl());
        registry.add("spring.datasource.username", ()->postgres.getUsername());
        registry.add("spring.datasource.password", ()->postgres.getPassword());
    }

    @Value("${spring.datasource.url}")
    private String expectedJdbcUrl;
    @Autowired
    private DataSource dataSource;
```

4.2.1. PostgresSQLContainer Setup

Testcontainers provides a `GenericContainer` that is what it sounds like. It generically manages the download, lifecycle, and communication of a Docker image. However, Testcontainers provides many container-specific "modules" that extend `GenericContainer` and encapsulate most of the module-specific setup work for you.

Since this lecture is trying to show you the "easy street" to integration, we'll go the module-specific `PostgreSQLContainer` route. All we need to do is

- specify a specific image name if we do not want to accept the default.
- declare the container using a static variable so that it is instantiated before the tests
- annotate the container with `@Container` to identify it to be managed
- annotate the test case with `@Testcontainers` to have the library automatically manage the lifecycle of the container(s).

Container Instantiated during Static Initialization

```
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;
import org.testcontainers.containers.PostgreSQLContainer;
...
@Testcontainers //manages lifecycle of container ③
class HelloPostgresTestcontainerNTest {
    @Container ②
    private static PostgreSQLContainer postgres = new PostgreSQLContainer<>(
"postgres:12.3-alpine"); ①
```

① declare container using a static variable

② `@Container` identifies container(s) for Testcontainers to manage

③ `@Testcontainers` annotation activates management that automatically starts/stops annotated containers

4.2.2. PostgresSQLContainer Startup

With just that in place (and the Docker daemon running), we can attempt to run the tests and see the container lifecycle in action.

- Testcontainers first starts a separate `Ryuk` container that will help manage the containers externally. The `Ryuk` container is primarily used to perform cleanup at the end of the tests.
- if required, the `postgres` Docker image is automatically downloaded
- the `postgres` Docker container is started
- a connection to the running `postgres` Docker container is made available to the host JVM (`jdbc:postgresql://localhost:55747/test`)

```
DockerClientProviderStrategy -- Loaded
org.testcontainers.dockerclient.UnixSocketClientProviderStrategy from
~/.testcontainers.properties, will try it first
DockerClientProviderStrategy -- Found Docker environment with local Unix socket
(unix:///var/run/docker.sock)

DockerClientFactory -- Docker host IP address is localhost
DockerClientFactory -- Connected to docker:
...①
ryuk:0.5.1 -- Creating container for image: testcontainers/ryuk:0.5.1
ryuk:0.5.1 -- Container testcontainers/ryuk:0.5.1 is starting:
b8a3a26580339bb98445f160610db339834216dfc1403f5e1a2b99800feb1a43
ryuk:0.5.1 -- Container testcontainers/ryuk:0.5.1 started in PT5.450402S
utility.RyukResourceReaper -- Ryuk started - will monitor and terminate Testcontainers
containers on JVM exit
...②
tc.postgres:12.3-alpine -- Pulling docker image: postgres:12.3-alpine. Please be
patient; this may take some time but only needs to be done once.
```

```
tc.postgres:12.3-alpine -- Starting to pull image
tc.postgres:12.3-alpine -- Pulling image layers:  0 pending,  0 downloaded,  0
extracted, (0 bytes/0 bytes)
tc.postgres:12.3-alpine -- Pulling image layers:  7 pending,  1 downloaded,  0
extracted, (1 KB/? MB)
...
tc.postgres:12.3-alpine -- Pull complete. 8 layers, pulled in 5s (downloaded 55 MB at
11 MB/s)

③
tc.postgres:12.3-alpine -- Creating container for image: postgres:12.3-alpine
tc.postgres:12.3-alpine -- Container postgres:12.3-alpine is starting:
78c1eda9cd432bb1fa434d65db1c4455977e0a0975de313eea9af30d09795caa
tc.postgres:12.3-alpine -- Container postgres:12.3-alpine started in PT1.913888S
tc.postgres:12.3-alpine -- Container is started (JDBC URL:
jdbc:postgresql://localhost:55747/test?loggerLevel=OFF) ④

Starting HelloPostgresTestcontainerNTest using Java 17.0.3 with PID 1853 (started by
jim in testcontainers-ntest-example) ⑤
...
```

① Testcontainers starts separate Ryuk container to manage container cleanup

② if required, the Docker image is automatically downloaded

③ the Docker container is started

④ connection(s) to the running Docker container made available to host JVM

⑤ test(s) start

If that is all we provided, we would see traces of the error shown in the following snippet — indicating we are missing configuration properties providing connection information needed to construct the `DataSource`.

Spring Requires DataSource Properties

```
Error creating bean with name
'info.ejava.examples.svc.tcontainers.hello.HelloPostgresTestcontainerNTest': Injection
of autowired dependencies failed
...
Could not resolve placeholder 'spring.datasource.url' in value
"${spring.datasource.url}" ①
```

① we do not know this value at development or JVM startup time

4.2.3. DynamicPropertyRegistry

Commonly we will have the connection information known at the start of heavyweight Maven Failsafe integration tests that is passed in as a concrete property. That would have been established during the `pre-integration-test` phase. In this case, the JVM and JUnit have already started and it is too late to have a property file in place with these dynamic properties.

A clean solution is to leverage Spring's DynamicPropertyRegistry test construct and define them at runtime.

- declare a static method that accepts a single DynamicPropertyRegistry
- annotate the static method with @DynamicPropertySource
- populate the DynamicPropertyRegistry with properties using a Supplier<Object> lambda function

Dynamically Define DataSource Properties from Container at Runtime

```
import org.springframework.test.context.DynamicPropertyRegistry;
import org.springframework.test.context.DynamicPropertySource;
...
class HelloPostgresTestcontainerNTest {
    @Container
    private static PostgreSQLContainer postgres = ...
    @DynamicPropertySource ①
    private static void addLateSpringContextProperties(DynamicPropertyRegistry
registry) {
        registry.add("spring.datasource.url",()->postgres.getJdbcUrl()); ②
        registry.add("spring.datasource.username", ()->postgres.getUsername());
        registry.add("spring.datasource.password", ()->postgres.getPassword());
    }
```

① `@DynamicPropertySource` identifies static methods that work with the `DynamicPropertyRegistry`

② defines property supplier methods that can produce the property value at runtime

Spring added DynamicPropertySource to Support Testcontainers

The DynamicPropertySource Javadoc states this test construct was originally added in Spring 5.2.5 to explicitly support testing with Testcontainers (but can be used for anything).

With the `PostgresSQLContainer` instance started and `@DynamicPropertySource` in place, the properties are successfully injected into the test.

Dynamic Property Injection

```
@Value("${spring.datasource.url}")
private String expectedJdbcUrl;

@Test
void can_populate_spring_context_with_dynamic_properties() {
    then(expectedJdbcUrl).matches(
"jdbc:postgresql://(?:localhost|host.docker.internal):[0-9]+/test.*");
}
```

4.3. Inject Postgres Container DataSource into Test

Like with the in-memory H2 database test, we are able to confirm that the injected `DataSource` is able to establish a connection to the Postgres database running within the container managed by Testcontainers.

Verify Connection using DataSource

```
@Test
void can_get_connection() throws SQLException {
    //given
    then(dataSource).isNotNull();
    Connection conn=dataSource.getConnection();
    //when
    String jdbcUrl;
    try (conn) {
        jdbcUrl=conn.getMetaData().getURL();
    }
    //then
    then(jdbcUrl).isEqualTo(expectedJdbcUrl); ①
}
```

① `expectedJdbcUrl` already matches `jdbc:postgresql://(?:localhost|host.docker.internal):[0-9]+/test.*`

4.4. Inject Postgres Container DataSource into Server-side

We have also successfully injected the `DataSource` into the server-side component (which is running in the same JVM as the JUnit test).

Verify Server-side Connection

```
@Test
void server_can_get_jdbc_connection() {
    //given
    URI url = dbUrl.build("jdbc");
    RequestEntity<Void> request = RequestEntity.get(url).build();
    //when
    ResponseEntity<String> response = anonymousUser.exchange(request, String.class);
    //then
    then(response.getStatusCode()).isEqualTo(HttpStatus.OK);
    String jdbcUrl=response.getBody();
    then(jdbcUrl).isEqualTo(expectedJdbcUrl);
}
```

4.5. Runtime Docker Containers

The following snippet of a `docker ps` command during the test shows the Testcontainers `Ryuk` container running along side our `postgres` container. The ports for both are dynamically assigned.

docker ps Output

```
IMAGE                         PORTS                     NAMES
postgres:12.3-alpine          0.0.0.0:55747->5432/tcp   nervous_benz ①
testcontainers/ryuk:0.5.1     0.0.0.0:55742->8080/tcp   testcontainers-ryuk-6d7a4199-
9471-4269-...
```

① postgres 5432 SQL port is exposed to localhost using a dynamically assigned port number

Recall that Testcontainers supplied the `postgres` container port to our test using the module-specific `getJdbcUrl()` method.

Module-specific Containers Provide Convenience Methods

```
@Container
private static PostgreSQLContainer postgres = ...
@DynamicPropertySource
private static void addLateSpringContextProperties(DynamicPropertyRegistry registry) {
    registry.add("spring.datasource.url",()->postgres.getJdbcUrl()); ①
...
```

① contained dynamically assigned port number (`55747`)

Chapter 5. MongoDB Container Example

We next move on to establish an integration test using MongoDB and Testcontainers. Much of the `MongoDBContainer` concepts are the same as with `PostgresSQLContainer`, except the expected property differences and a configuration gotcha.

5.1. Maven Dependencies

The following snippet shows the additional MongoDB module dependency we need to add on top of the `spring-boot-starter-data-mongodb`, `testcontainers`, and `junit-jupiter` dependencies we added earlier.

Additional Testcontainers MongoDB Module Dependency

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>mongodb</artifactId>
    <scope>test</scope>
</dependency>
```

5.2. Unit Integration Test Setup

The following snippet shows the meat of the overall `MongoDBContainer` setup. As with `PostgresSQLContainer`, we have the choice to use the `GenericContainer` or the module-specific `MongoDBContainer`. However, with this database I ran into a time-consuming gotcha that might influence choices.

MongoDB started supporting multi-document transactions with version 4.0. However, to use transactions — one must define a "replica set" (just consider that a "setting"). Wanting to be full featured for testing, `MongoDBContainer` configures MongoDB for a (single-node) replica set. However, that is performed using an admin-level command issued to the just started MongoDB database instance. That command fails if authentication is enabled because the replica set admin command from the `MongoDBContainer` is issued without credentials. If that command fails, `MongoDBContainer` considers the Docker container startup as failed and terminates.

I will demonstrate how to run the `MongoDBContainer`, enabling transactions by disabling authentication. I will also demonstrate how to run MongoDB image within the `GenericContainer`, enabling security and no support for transactions. Both, solely have the goal to complete a connection as part of this lecture. I am not demonstrating any database-specific capabilities beyond that.

MongoDBContainer Test Setup

```
@SpringBootTest ...
@Testcontainers //manages lifecycle of container
@Slf4j
class HelloMongoDBTestcontainerNTest {
    @Container
```

```
    private static MongoDBContainer mongoDB = new MongoDBContainer("mongo:4.4.0-
bionic");
    @DynamicPropertySource
    private static void addLateSpringContextProperties(DynamicPropertyRegistry
registry) {
        registry.add("spring.data.mongodb.uri",()-> mongoDB.getReplicaSetUrl(
"testcontainers"));
    }

    @Value("${spring.data.mongodb.uri}")
    private String expectedMongoUrl;
    @Autowired
    private MongoClient mongoClient;
```

① do not enable authentication with `MongoDBContainer`

5.2.1. MongoDBContainer Setup

Like with the `PostgresSQLContainer`, we

- specify the name of the MongoDB image
- instantiate the `MongoDBContainer` during static initialization
- annotate the container with `@Container` to identify it to be managed
- annotate the test case with `@Testcontainers` to have the library automatically manage the lifecycle of the container(s).

Also of important note — `MongoDBContainer` does not enable authentication and we **must not** enable it by setting environment variables

- `MONGO_INITDB_ROOT_USERNAME`
- `MONGO_INITDB_ROOT_PASSWORD`

MongoDBContainer does not support Authentication

```
import org.testcontainers.containers.MongoDBContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

@Testcontainers //manages lifecycle of container
class HelloMongoDBTestcontainerNTest {
    @Container
    private static MongoDBContainer mongoDB = new MongoDBContainer("mongo:4.4.0-
bionic")
//            .withEnv("MONGO_INITDB_ROOT_USERNAME", "admin") ①
//            .withEnv("MONGO_INITDB_ROOT_PASSWORD", "secret")
            ;
```

① setting username/password enables MongoDB authentication and causes `MongoDBContainer` ReplicaSet configuration commands to fail

5.2.2. GenericContainer Setup

We can alternatively use the `GenericContainer` and push all the levers we want. In the following example, we will instantiate a MongoDB Docker container that supports authentication but has not been setup with a replica set to support transactions.

Can enable MongoDB Authentication with GenericContainer

```
import org.testcontainers.containers.GenericContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

@Testcontainers //manages lifecycle of container
class HelloMongoDBGenericContainerNTest {
    @Container
    private static GenericContainer mongoDB = new GenericContainer("mongo:4.4.0-
bionic")
            .withEnv("MONGO_INITDB_ROOT_USERNAME", "admin")
            .withEnv("MONGO_INITDB_ROOT_PASSWORD", "secret")
            .withExposedPorts(27017);
```

5.2.3. MongoDB Dynamic Properties

At this point, we need to locate and express the MongoDB URI. The technique will be slightly different for `MongoDBContainer` and `GenericContainer` because only the `MongoDBContainer` knows what a MongoDB URI is.

The snippet below shows how we can directly define the MongoDB URI property using the `MongoDBContainer.getReplicaSetUrl()`. The resulting MongoDB URI will be `mongodb://localhost:63216/testcontainers`;

Obtain MongoDB URI from MongoDBContainer

```
@DynamicPropertySource
private static void addLateSpringContextProperties(DynamicPropertyRegistry registry) {
    registry.add("spring.data.mongodb.uri",()-> mongoDB.getReplicaSetUrl(
"testcontainers"));
}
```

The snippets below show how we can assemble the MongoDB URI using properties that the `GenericContainer` knows about (e.g., hostname, port) and what we know we provided to the `GenericContainer`.

Obtain MongoDB URI from GenericContainer

```
@DynamicPropertySource
private static void addLateSpringContextProperties(DynamicPropertyRegistry registry) {
    String userName = (String)mongoDB.getEnvMap().get("MONGO_INITDB_ROOT_USERNAME");
①
```

```
    String password = (String)mongoDB.getEnvMap().get("MONGO_INITDB_ROOT_PASSWORD");
    registry.add("spring.data.mongodb.uri",()->
        ClientNTestConfiguration.mongoUrl(userName, password,
                mongoDB.getHost(), mongoDB.getMappedPort(27017), "testcontainers"));
}
```

① userName and password were supplied as environment variables when `GenericContainer` created

With userName, password, host, port, and database name — we can manually create a MongoDB URI.

Format MongoDB URI from Properties

```
public static String mongoUrl(String userName, String password, String host, int port,
String database) {
    return String.format("mongodb://%s:%s@%s:%d/%s?authSource=admin", userName,
password, host, port, database);
}
```

The resulting MongoDB URI will be `mongodb://admin:secret@localhost:54568/testcontainers?authSource=admin`;

5.3. Inject MongoDB Container Client into Test

With our first test, we are able to verify that we can inject a `MongoClient` into the test case and that it matches the expected MongoDB URI. We also need to execute additional command(s) to verify the connection because just describing the cluster/connection does not establish any communication between the MongoClient and MongoDB.

Verify Connection using MongoClient

```
@Test
void can_get_connection() {
    //given
    then(mongoClient).isNotNull();
    //when
    String shortDescription = mongoClient.getClusterDescription().getShortDescription
();
    //then
    new MongoVerifyTest().actual_hostport_matches_expected(expectedMongoUrl,
shortDescription); ①
    then(mongoClient.listDatabaseNames()).contains("admin"); ②
}
```

① host:port will get extracted from the Mongo URI and cluster description and compared

② databaseNames() called to test connection

The following snippet shows an extraction and comparison of the host:port values from the Mongo

URI and cluster description values.

Verify Connection Matches Injected Value

```
//..., servers=[{address=localhost:56295, type=STANDALONE...
private static final Pattern DESCR_ADDRESS_PATTERN = Pattern.compile("address=([A-Za-
z\\.:0-9]+),");
//mongodb://admin:secret@localhost:27017/testcontainers
private static final Pattern URL_HOSTPORT_PATTERN = Pattern.compile("[@/]([A-Za-z
\\.:0-9]+)/");

void actual_hostport_matches_expected(String expectedMongoUrl, String description) {
    Matcher m1 = DESCR_ADDRESS_PATTERN.matcher(description);
    then(expectedMongoUrl).matches(url->m1.find(), DESCR_ADDRESS_PATTERN.toString())
;①

    Matcher m2 = URL_HOSTPORT_PATTERN.matcher(expectedMongoUrl);
    then(expectedMongoUrl).matches(url->m2.find(), URL_HOSTPORT_PATTERN.toString());①

    then(m1.group(1)).isEqualTo(m2.group(1)); ②
}
```

① extracting host:port values from sources

② comparing extracted host:port values

5.4. Inject MongoDB Container Client into Server-side

This next snippet shows we can verify the server-side was injected with a `MongoClient` that can communicate with the expected database.

Verify Server-side Connection

```
@Test
void server_can_get_mongo_connection() {
    //given
    URI url = dbUrl.build("mongo");
    RequestEntity<Void> request = RequestEntity.get(url).build();
    //when
    String shortDescription = anonymousUser.exchange(request, String.class).getBody();
    //then
    new MongoVerifyTest().actual_hostport_matches_expected(expectedMongoUrl,
shortDescription);
}
```

5.5. Runtime Docker Containers

The following snippet of a docker ps command during the test shows the Testcontainers Ryuk container running along side our `mongo` container. The ports for both are dynamically assigned.

docker ps Output

```
IMAGE                         CREATED         PORTS                      NAMES
mongo:4.4.0-bionic            1 second ago    0.0.0.0:59627->27017/tcp   gifted_hoover
testcontainers/ryuk:0.5.1     1 second ago    0.0.0.0:59620->8080/tcp    testcontainers-
ryuk-a6c35bc3-66d7-...
```

Chapter 6. Docker Compose Example

The previous two examples represent the heart of what Testcontainers helps us achieve — easy integration with back-end resources for testing. Testcontainers can help automate many more tasks, including interfacing with Docker Compose. In the following example, I will use a familiar Docker Compose setup to demonstrate:

- Testcontainers interface with Docker Compose
- obtain container connections started using Docker Compose
- build and test the server-side application under test using Testcontainers and Docker Compose

6.1. Maven Aspects

The `DockerComposeContainer` is housed within the primary `org.testcontainers:testcontainers` dependency. There are no additional Maven dependencies required.

However, our JUnit test(s) will need to run after the Spring Boot Executable JAR has been created, so we need to declare the test with an `IT` suffix and add the Maven Failsafe plugin to run the test during the `integration-test` phase. Pretty simple.

Run Test(s) During integration-test Phase using Maven Failsafe Plugin

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-failsafe-plugin</artifactId>
            <executions>
                <execution>
                    <id>integration-test</id>
                    <goals>
                        <goal>integration-test</goal>
                    </goals>
                </execution>
                <execution>
                    <id>verify</id>
                    <goals>
                        <goal>verify</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
...
```

6.2. Example Files

The following snippet shows our basic module structure where we have

- Docker/Docker Compose resource files intended for production
- Docker/Docker Compose resource files used for testing
- an integration test

The dual Docker Compose files is not required. This is only an example of using multiple files and if you need something unique for testing. Our JUnit test is defined as a Maven Failsafe IT test so that the Spring Boot Executable JAR can be built in time to run the test. We will not need the pre/post-integration-test phases.

```
src/main
|-- docker ①
|   |-- Dockerfile
|   |-- docker-compose.yml
|   `-- run_env.sh
...
src/test
|-- java
|   `-- info
|       `-- ejava
|           `-- examples
|               `-- svc
|                   `-- tcontainers
|                       `-- hello
|                           |-- ClientNTestConfiguration.java
|                           |-- HelloApiContainerIT.java
|
...
`-- resources
    |-- docker-compose-test.yml ②
    ...
```

① production Docker/Docker Compose configuration

② Docker Compose configuration settings for test

If we test within the IDE, the Spring Boot Executable JAR must be available to build the API's Docker image. The Spring Boot Executable JAR can be re-built using the following commands

Rebuild the Spring Boot Executable JAR for API image

```
$ mvn clean package -DskipTests && ls target/*bootexec*
...
target/testcontainers-ntest-example-6.1.0-SNAPSHOT-bootexec.jar
```

6.2.1. Production Docker Compose File

The following snippet shows an example Docker Compose file playing the role of what we might run in production (minus the plaintext credentials). Notice that the

- Postgres and MongoDB ports are not externally defined. The Postgres and MongoDB ports are already defined by their Dockerfile definition. The API will be able to communicate with the databases using the internal Docker network created by Docker Compose.
- only the image identification is supplied for the API. The build commands will not be necessary or desired in production.

src/main/docker/docker-compose.yml

```
services:
  postgres:
    image: postgres:12.3-alpine
    environment:
      POSTGRES_PASSWORD: secret
  mongodb:
    image: mongo:4.4.0-bionic
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: secret
  api: ①
    image: testcontainers-ntest-example:latest
    ports:
      - "${HOST_API_PORT:-8080}:8080" ②
    depends_on:
      - postgres
      - mongodb
    environment:
      - DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
      -
MONGODB_URI=mongodb://admin:secret@mongodb:27017/testcontainers?authSource=admin
```

① only runtime commands supplied, build commands not desired here

② API defaults to exposed pre-defined port here

6.2.2. Test Docker Compose File

The snippet below shows a second Docker Compose file playing the role of the test overrides. Here we are defining the build commands the API image. Note that we need to set the Docker build for context to a high enough level to include the target tree with the Spring Boot Executable JAR.

src/test/resources/docker-compose-test.yml

```
services:
  api:
    build: #set build context to module root so that target/*bootexec.jar is within
view
```

```
      context: ../../..
      dockerfile: src/main/docker/Dockerfile
    ports:
      - "8080"  #API port exposed on random host port
```

6.2.3. Dockerfile

The following snippet shows a familiar Dockerfile that will establish a layered Docker image. I have assigned a few extra variables (e.g., `BUILD_ROOT`, `RESOURCE_DIR`) that might be helpful building the image under different circumstances but were not needed here.

I have also included the use of a `run.sh` bash script in the event we need to add some customization to the API JVM. It is used in this example to convert provided environment variables into `DataSource` and `MongoClient` properties required by the server-side API.

src/main/docker/Dockerfile

```
FROM eclipse-temurin:17-jre AS builder
WORKDIR /builder
ARG BUILD_ROOT=.
ARG JAR_FILE=${BUILD_ROOT}/target/*-bootexec.jar
ARG RESOURCE_DIR=${BUILD_ROOT}/src/main/docker
COPY ${JAR_FILE} application.jar
RUN java -Djarmode=tools -jar application.jar extract --layers --launcher
--destination extracted
COPY ${RESOURCE_DIR}/run_env.sh .

FROM eclipse-temurin:17-jre
WORKDIR /application
COPY --from=builder /builder/extracted/dependencies/ ./
COPY --from=builder /builder/extracted/spring-boot-loader/ ./
COPY --from=builder /builder/extracted/snapshot-dependencies/ ./
COPY --from=builder /builder/extracted/application/ ./
COPY --chmod=555 --from=builder /builder/run_env.sh ./
#https://github.com/spring-projects/spring-boot/issues/37667
ENTRYPOINT ["./run_env.sh",
"java","org.springframework.boot.loader.launch.JarLauncher"]
```

6.3. Integration Test Setup

The snippet below shows the JUnit setup for the `DockerComposeContainer`.

- the test is named as a Maven Failsafe IT test. It requires no special input properties
- there is no web/server-side environment hosted within this JVM. The server-side will run within the API Docker image.
- details of the `DockerComposeContainer` and `DynamicPropertyRegistry` configuration are contained within the `ClientNTestConfiguration` class

- the test case is annotated with `@Testcontainers` to trigger the lifecycle management of containers
- the `DockerComposeContainer` is declared during static initialization and annotated with `@Container` so that its lifecycle gets automatically managed
- `@BeforeEach` initialization is injected with the host and port of the remote API running in Docker, which will not be known until the `DockerComposeContainer` started and the API Docker container is launched

```
import org.testcontainers.containers.DockerComposeContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

@SpringBootTest(classes= ClientNTestConfiguration.class,
    webEnvironment = SpringBootTest.WebEnvironment.NONE)
@EnableAutoConfiguration
@Testcontainers //manages lifecycle for @Containers ①
class HelloApiContainerIT {
    @Container ①
    private static DockerComposeContainer network = ClientNTestConfiguration
.testEnvironment();
    @DynamicPropertySource ②
    private static void addLateSpringContextProperties(DynamicPropertyRegistry
registry) {
        ClientNTestConfiguration.initProperties(registry, network);
    }

    @BeforeEach
    void init( ③
            @Value("${it.server.host:localhost}") String remoteApiContainerHost,
            @Value("${it.server.port:9090}") int remoteApiContainerPort) {
        dbUrl=UriComponentsBuilder.fromHttpUrl("http://localhost")
                .host(remoteApiContainerHost)
                .port(remoteApiContainerPort)
                .path("/api/hello/{db}");
    }
```

① `@Testcontainers` annotation will start/stop all Testcontainers marked with `@Container`

② `DynamicPropertyRegistry` will be used to supply required properties not known until Docker containers start

③ host and port of remote API will not be known until Docker containers start, thus come from `DynamicPropertyRegistry`

6.3.1. DockerComposeContainer Setup

With the added complexity of the DockerComposeContainer setup, I chose to move that to a sharable static helper method. There is again a lot to unpack here.

DockerComposeContainer Construction

```
public static DockerComposeContainer testEnvironment() {
    return new DockerComposeContainer("testcontainers-ntest",①
      List.of(new File("src/main/docker/docker-compose.yml"), //core wiring ②
             new File("target/test-classes/docker-compose-test.yml"))) //build & port
info
        .withBuild(true) ③
        .withExposedService("api", 8080) ④
        .withExposedService("postgres", 5432)
        .withExposedService("mongodb", 27017)
        .withLocalCompose(true) ⑤
        //https://github.com/testcontainers/testcontainers-java/pull/5608
        .withOptions("--compatibility") //change dashes to underscores ⑥
                                        //when using local=true
                                        //so testcontainers can find the
container_name
        .withStartupTimeout(Duration.ofSeconds(100));
}
```

① identifier

② configuration files

③ force image build

④ Identify Exposed Ports

⑤ Local or container-based Docker Compose

⑥ V1 or V2 Docker Compose container naming syntax

Identifier

Like with the command line, we can supply an identifier (e.g., `testcontainers-ntest`) that will prefix each container name created (e.g., `testcontainers-ntestkxzcpt_mongodb_1`). This is not unique by itself, but it does help describe where the container came from.

Example Supplying Identifier via Command Line

```
$ docker-compose ... -p testcontainers-ntest up
#docker ps --format '{{.Names}}'
testcontainers-ntest-api-1
testcontainers-ntest-postgres-1
testcontainers-ntest-mongodb-1
```

Configuration Files

We can express the configuration in one or more files. This example uses two.

Example Supplying Multiple Files via Command Line

```
$ docker-compose -f src/main/docker/docker-compose.yml -f src/test/resources/docker-
```

```
compose-test.yml ...
```

Force Image Build

The image build is being forced each time the test is run.

Example Forcing Build via Command Line

```
$ docker-compose ... -p testcontainers-ntest up --build
```

This, of course, depends on the Spring Boot Executable JAR to be in place in the `target` directory. This is why we named in as a Failsafe IT test, so that it will run after the Maven `package` phase.

Quick Build/Re-build API and Spring Boot Executable JAR

If you are running the IT test in the IDE and need to package or update the Spring Boot Executable JAR, you can use the following command.

```
$ mvn clean package -DskipTests
```

Identify Exposed Ports

We identify which ports in each service to expose to this JVM. Testcontainers will start an "ambassador" container (alpine/socat) to proxy commands between the JUnit JVM and the running containers. There will not be a direct connection between the JUnit JVM and Docker containers launched. There is no requirement to expose the service ports the host network. The "ambassador" container will proxy all communications between host and Docker network.

Docker Containers with "Ambassador"/Proxy and "Cleanup"

```
IMAGE                          PORTS
testcontainers/ryuk:0.5.1      0.0.0.0:54186->8080/tcp
alpine/socat:1.7.4.3-r0        0.0.0.0:54193->2000/tcp, 0.0.0.0:54194->2001/tcp,
                                                        0.0.0.0:54195->2002/tcp ①
mongo:4.4.0-bionic             27017/tcp ②
postgres:12.3-alpine           5432/tcp  ②
testcontainers-ntest-example:latest   0.0.0.0:8080->8080/tcp ③
```

① `alpine/socat` "ambassador" container establishes proxy connections to Docker Compose-launched services

② services are not exposed to host network by default — they are proxied by "ambassador"

③ API service was exposed to host network because of contents in configuration

Local or Container-based Docker Compose

The localCompose flag determines whether to use the binaries from the local host (true) or from a Testcontainers-supplied container implementing Compose V2 (false). I have set this to

`localCompose=true` because I found the launched Docker Compose container needs to successfully mount the source filesystem to read resources like the docker-compose.yml files. This does require Docker Compose binaries to be present in all development and CI/CD environments, but it is a much more flexible option to work with.

V1 or V2 Docker Compose Container Naming

I am launching Docker Compose with the --compatibility mode, which impacts Testcontainers' ability to locate Docker images once launched within a CI/CD environment. Docker Compose V1 used underscores (_) to separate word tokens in the image name (e.g., `testcontainers-ntestkxzcpt_mongodb_1`). Underscore (_) is not a valid character in DNS names [1] so Docker Compose changed the delimiter to dashes (-) in V2 (e.g., `testcontainers-ntestgxxui6-mongodb-1`) so that the Docker name could be used as a DNS name.

Docker Compose V2 uses DNS-legal Dashes (-) for Word Token Separators

```
$ docker-compose -f src/main/docker/docker-compose.yml -f src/test/resources/docker-
compose-test.yml up

#docker ps --format {{.Names}}
docker-api-1 ①
docker-mongodb-1
docker-postgres-1
```

① V2 mode uses DNS-legal dashes in names to make name DNS-usable

The problem arises when the Testcontainers uses the V1 formatting strategy to locate a name formatted using V2.

HelloApiContainerIT Failure Because Present Name Not Found

```
ERROR tc.alpine/socat:1.7.4.3-r0 -- Could not start container
org.testcontainers.containers.ContainerLaunchException: Aborting attempt to link to
container testcontainers-ntesthz5z8g_mongodb_1 as it is not running ①
```

① looking for V1 `testcontainers-ntesthz5z8g_mongodb_1` vs V2 `testcontainers-ntesthz5z8g-mongodb-1`

We can compensate for that using the Docker Compose V2 `--compatibility` flag.

Docker Compose V2 --compatibility Returns Word Token Separators to V1 Underscore(_)

```
$ docker-compose -f src/main/docker/docker-compose.yml -f src/test/resources/docker-
compose-test.yml --compatibility up

#docker ps --format {{.Names}}
docker_api_1 ①
docker_mongodb_1
docker_postgres_1
```

① --compatibility mode uses V1 underscore (_) characters

Manual Commands were just Demonstrations for Clarity

The manual docker-compose commands shown above were to demonstrate how Testcontainers will translate your options to a Docker Compose command. You will not need to manually execute a Docker Compose command with Testcontainers.

6.3.2. Dynamic Properties

Once the Docker Compose services are running, we need to obtain URI information to the services. If you recall the JUnit test setup, we initially need the `it.server.host` and `it.server.port` properties to inject into `@BeforeEach` initialization method.

JUnit Test Setup Requires API host:port Properties

```
@BeforeEach
void init(@Value("${it.server.host:localhost}") String remoteApiContainerHost,
          @Value("${it.server.port:9090}") int remoteApiContainerPort) {
    dbUrl=UriComponentsBuilder.fromHttpUrl("http://localhost")
            .host(remoteApiContainerHost)
            .port(remoteApiContainerPort)
            .path("/api/hello/{db}");
}
```

As with the previous examples, we can obtain the runtime container properties and form/supply the application properties using a static method annotated with `@DynamicPropertySource` and taking a DynamicPropertyRegistry` argument.

Property Initialization was Delegated to Helper Method

```
@DynamicPropertySource
private static void addLateSpringContextProperties(DynamicPropertyRegistry registry) {
    ClientNTestConfiguration.initProperties(registry, network); ①
}
```

① calls static helper method within configuration class

The example does the work within a static helper method in the configuration class. The container properties are obtained from the running `DockerComposeContainer` using accessor methods that accept the service name and targeted port number as identifiers. Notice that the returned properties point to the "ambassador"/socat proxy host/ports and not directly to the targeted containers.

getServiceHost() Ignores Port# Under the Hood

I noticed that port number was ignored by the underlying implementation for `getServiceHost()`, so I am simply passing a null for port.

Obtain Dynamic Properties from DockerComposeContainer

```
public static void initProperties(DynamicPropertyRegistry registry,
DockerComposeContainer network) {
    //needed for @Tests to locate API Server
    registry.add("it.server.port", ()->network.getServicePort("api", 8080)); //60010
②
    registry.add("it.server.host", ()->network.getServiceHost("api", null));
//localhost
...

//docker ps --format '{{.Names}}\t{{.Ports}}'
//testcontainers-socat...             0.0.0.0:60010->2000/tcp, ... ①
//testcontainers-ntesttn8uks_api_1    0.0.0.0:59998->8080/tcp ①
```

① socat port `60010` proxies API port `8080`

② socat proxy port `60010` supplied for `getServicePort()`

In the event we want connections to the back-end databases, we will need to provide the standard `spring.datasource.*` and `spring.data.mongodb.uri` properties.

Injecting Back-end Database Services

```
@Autowired //optional -- just demonstrating we have access to DB
private DataSource dataSource;
@Autowired //optional -- just demonstrating we have access to DB
private MongoClient mongoClient;
```

We can obtain those properties from the running `DockerComposeContainer` and supply values we know from the Docker Compose files.

Obtain Dynamic Properties for Back-end Databases

```
//optional -- only if @Tests directly access the DB
registry.add("spring.data.mongodb.uri",()-> mongoUrl("admin", "secret",
        network.getServiceHost("mongodb", null),
        network.getServicePort("mongodb", 27017), //60009 ①
        "testcontainers"
));
registry.add("spring.datasource.url",()->jdbcUrl(
        network.getServiceHost("postgres", null),
        network.getServicePort("postgres", 5432) //60011 ②
));
registry.add("spring.datasource.driver-class-name",()->"org.postgresql.Driver");
registry.add("spring.datasource.username",()->"postgres");
registry.add("spring.datasource.password",()->"secret");

//docker ps --format '{{.Names}}\t{{.Ports}}'
//testcontainers-socat...             ..., 0.0.0.0:60011->2001/tcp, 0.0.0.0:60009-
>2002/tcp
```

```
//testcontainers-ntesttn8uks_mongodb_1    27017/tcp ①
//testcontainers-ntesttn8uks_postgres_1   5432/tcp ②
```

① socat port `60009` proxies mongodb port `27017`

② socat port `60011` proxies postgres port `5432`

One last set of helper methods assemble the runtime/dynamic container properties and form URLs to be injected as mandatory database component properties.

Database URI Helper Methods

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

With the test setup complete, we are ready to verify our

- test can communicate with the databases
- test can communicate with the server-side API
- the server-side API can communicate with the databases

6.4. Inject Postgres Connection into Test

The test below shows that the JUnit test can obtain a connection to the database via the locally-injected `DataSource` and verifies it is a Postgres instance. The port will not be `5432` in this case because the JUnit test is remote from the Docker Compose containers and communicates to each of them using the `socat` proxy.

Inject Postgres Connection into Test

```
@Test
void dataSource_can_provide_connection() throws SQLException {
    //given
    then(dataSource).isNotNull();
    Connection conn=dataSource.getConnection();
    //when
    String jdbcUrl;
    try (conn) {
        jdbcUrl=conn.getMetaData().getURL();
    }
    //then
    then(jdbcUrl).contains("jdbc:postgresql")
                 .doesNotContain("5432"); //test uses socat proxy;
```

```
}
//testcontainers-ntestvgbpgg_postgres_1         5432/tcp
//testcontainers-socat-dzm9Rn9w                 0.0.0.0:65149->2001/tcp, ...
//jdbc:postgresql://localhost:65149/postgres ①
```

① test obtains `DataSource` connection via `socat` proxy port

6.5. Inject Postgres Connection into Server-Side

The next test contacts the server-side API to obtain the JDBC URL of its `DataSource`. The server-side API will use port `5432` because it is a member of the Docker network setup by Docker Compose and configured to use `postgres:5432` via the `docker-compose.yml` environment variable (`DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres`).

Inject Postgres Connection into Server-Side

```
@Test
void server_can_get_jdbc_connection() {
    //given
    URI url = dbUrl.build("jdbc");
    RequestEntity<Void> request = RequestEntity.get(url).build();
    //when
    String jdbcUrl = anonymousUser.exchange(request, String.class).getBody();
    //then
    //hostname will be postgres and port will be default internal 5432
    then(jdbcUrl).contains("jdbc:postgresql","postgres:5432");
}
//testcontainers-ntestvgbpgg_api_1              0.0.0.0:65145->8080/tcp
//testcontainers-ntestvgbpgg_postgres_1         5432/tcp
//jdbc:postgresql://postgres:5432/postgres ①
```

① server-side API uses local Docker network for `DataSource` connection

The specific URL was provided as an environment variable in the Docker Compose configuration.

docker-compose.yml Server-side Environment Variables

```
services:
  api:
    environment:
      - DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
      - ...
```

6.6. Inject MongoClient into Test

The snippet below shows that the JUnit test can obtain a connection to MongoDB via the locally-injected `MongoClient`. The port will not be `27017` in this case either because the JUnit test is remote from the Docker Compose containers and communicates to each of them using the `socat` proxy.

Inject MongoClient into Test

```
@Test
void mongoClient_can_get_connection() {
    //given
    then(mongoClient).isNotNull();
    //then
    then(mongoClient.getClusterDescription().getShortDescription())
                .doesNotContain("27017");
    then(mongoClient.listDatabaseNames()).contains("admin");
}
//{type=STANDALONE, servers=[{address=localhost:65148, ...
//testcontainers-socat-dzm9Rn9w           0.0.0.0:65148->2000/tcp
//testcontainers-ntestvgbpgg_mongodb_1    27017/tcp
```

6.7. Inject MongoClient into Server-side

The snippet below shows that the server-side API can obtain a connection to MongoDB via its injected `MongoClient` and will use hostname `mongodb` and port `27017` because it is on the same local Docker network as MongoDB and was configured to do so using the `docker-compose.yml` environment variable (`MONGODB_URI=mongodb://admin:secret@mongodb:27017/testcontainers?authSource=admin`).

Inject MongoClient into Server-side

```
@Test
void server_can_get_mongo_connection() {
    //given
    URI url = dbUrl.build("mongo");
    RequestEntity<Void> request = RequestEntity.get(url).build();
    //when
    String shortDescription = anonymousUser.exchange(request, String.class).getBody();
    //then
    //hostname will be mongo and port will be default internal 27017
    then(shortDescription).contains("address=mongodb:27017");
}
//{type=STANDALONE, servers=[{address=mongodb:27017, ...
//testcontainers-ntestvgbpgg_api_1              0.0.0.0:65145->8080/tcp
//testcontainers-ntestvgbpgg_mongodb_1          27017/tcp
```

docker-compose.yml Server-side Environment Variables

```
services:
  api:
    environment:
      - ...
      -
MONGODB_URI=mongodb://admin:secret@mongodb:27017/testcontainers?authSource=admin
```

[1] *"Migrate to Compose V2"*, docker.com

Chapter 7. Summary

In this module, we learned:

- how to more seamlessly integrate back-end resources into our tests using Docker, Docker Compose, and Testcontainers using Testcontainers library
- how to inject dynamically assigned properties into the application context of a test to allow them to be injected into components at startup
- to establish client connection to back-end resources from our JUnit JVM operating the unit test

Although integration tests should never fully 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. More features exist within Testcontainers than were covered here. Some examples include

- waitingFor strategies that help determine when the container is ready for use
- exec commands into container that could allow us to issue CLI database commands if helpful. Remember the `MongoDBContainer` issues the replicaSet commands using the Mongo CLI interface. It is an example of a container exec command.

```
log.debug("Initializing a single node node replica set...");
Container.ExecResult execResultInitRs = this.execInContainer(this
.buildMongoEvalCommand("rs.initiate();"));
...
private String[] buildMongoEvalCommand(String command) {
    return new String[]{"sh", "-c", "mongosh mongo --eval \"" + command + "\"  ||
mongo --eval \"" + command + "\""};
}
```