Docker Compose Integration Testing

jim stafford

Fall 2024 v2021-08-27: Built: 2024-11-19 21:42 EST

Table of Contents

Chapter 1. Introduction

In the last lecture, we looked at a set of containerized services that we could conveniently launch and manage with Docker Compose.

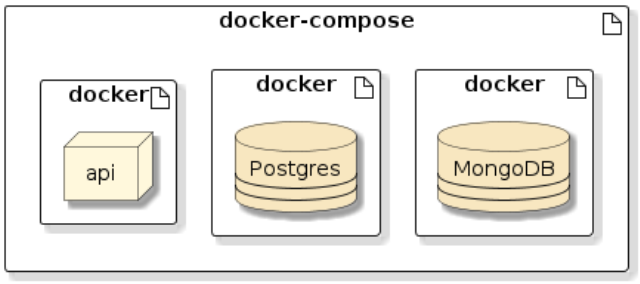


Figure 1. Our Services can be Managed using Docker Compose

How could we develop and test against those services?

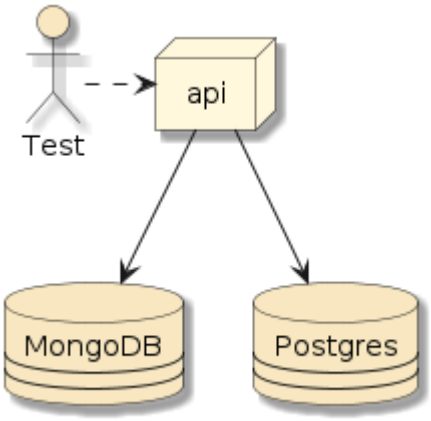


Figure 2. We want to Test API using Close-to-Real Services

We could use in-memory/fakes, but that would not take advantage of how close we could get our tests to represent reality using Docker.

Technically, Postgres and MongoDB do not have in-memory options. Postgres can be simulated with an SQL-compliant H2 database. There are some test fakes for MongoDB, but they are highly version and capability constrained.

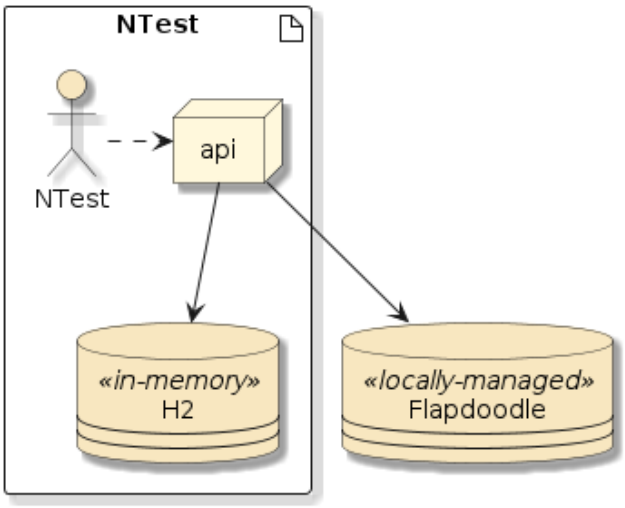


Figure 3. Testing with In-Memory/Local Resources

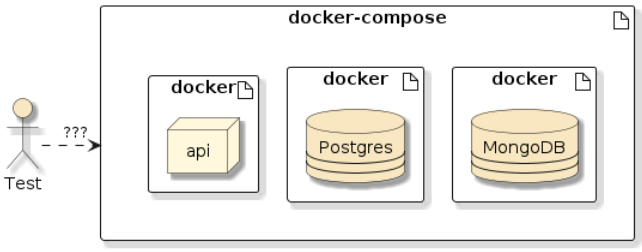
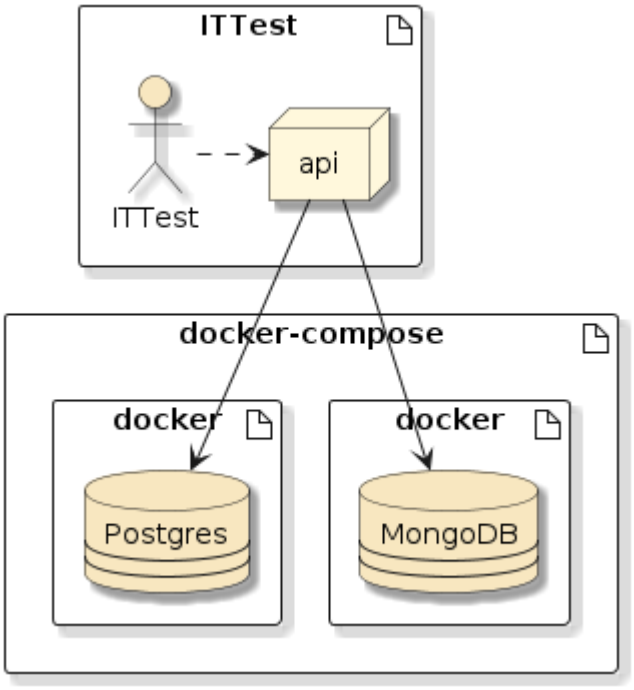


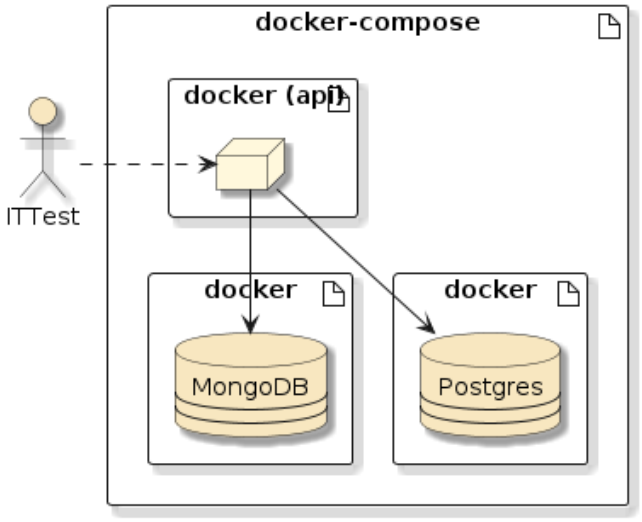
Figure 4. How Can We Test with Real Resources

What if we needed to test with a real or specific version of MongoDB, Postgres, or some other resource? We want to somehow integrate our development and testing with our capability to manage containers with Docker Compose and manage test lifecycles with Maven.

We want to develop and debug within the IDE and still use containerized dependencies.



We want to potentially package our API into a Docker image and then test it in semi-runtime conditions using containerized dependencies.



In this lecture, we will explore using Docker Compose as a development and Failsafe Integration Test aid.

1.1. Goals

You will learn:

- how to automate a full end-to-end heavyweight Failsafe integration test with Maven and Docker Compose
- how to again resolve networking issues when running in a CI/CD environment
- how to address environment-specific JVM setup

1.2. Objectives

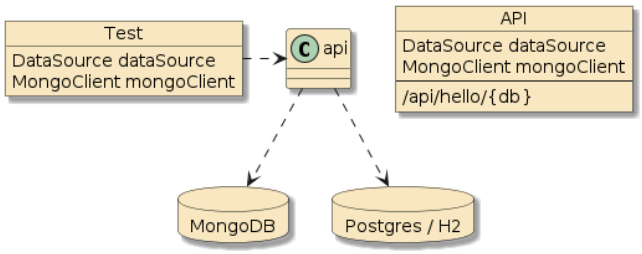
At the conclusion of this lecture and related exercises, you will be able to:

1. automate an end-to-end Failsafe integration test with Maven plugins and Docker Compose
2. setup host specifications relative to the host testing environment
3. insert a custom wrapper script to address environment-specific JVM processing options

Chapter 2. Starting Point

I will start this Docker Compose Integration Test lecture with an example set of tests and application that simply establishes a connection to the services. We will cover more about database and connection options in the next several weeks during the persistence topics, but right now we want to keep the topic focused on setting up the virtual environment and less about the use of the databases specifically.

The examples will use an API controller and tests injected with an RDBMS (`DataSource`) and MongoDB (`MongoClient`) components to verify the end-to-end communication is in place.



We will start with a Unit Integration Test (NTest) that configures the components with an in-memory H2 RDBMS — which is a common practice for this type of test.

- Postgres does not provide an embedded/in-memory option. Postgres and H2 are both SQL-compliant relational databases, and H2 will work fine for this level of test.
- MongoDB does not provide an embedded/in-memory option. I will cover some "embedded" MongoDB test options in the future persistence lectures. Until then, I will skip over the MongoDB testing option until we reach the Docker-based Maven Failsafe sections.

The real Postgres and MongoDB instances will be part of the follow-on sections of this lecture when we bring in Docker Compose.

2.1. Maven Dependencies

We add `starter-data-jpa` and `starter-data-mongo` dependencies to the module that will bootstrap the RDBMS and MongoDB database connections. These dependencies are added with `compile` scope since this will be directly used by our deployed production code and not limited to test.

For RDBMS, we will need a Postgres dependency for test/runtime and an H2 dependency for test. There is no `src/main` compile dependency on Postgres or H2. The additional dependencies supply DB-specific implementations of relational interfaces that our `src/main` code does not need to directly depend upon.

Required Maven Dependencies

```
<dependency> ①
    <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>
```

```
<dependency> ②
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>
```

① MongoDB and RDBMS interfaces our implementation code must depend upon

② RDBMS implementation code needed at test time or operational runtime

2.2. Unit Integration Test Setup

The H2 In-memory Unit Integration Test shows an injected `javax.sql.DataSource` that will be supplied by the JPA AutoConfiguration. There will be a similar MongoDB construct once we start using external databases.

H2 In-memory NTest

```
import javax.sql.DataSource;

@SpringBootTest(classes={DockerComposeHelloApp.class,
        ClientTestConfiguration.class},
        webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@ActiveProfiles({"test","inmemory"})
class HelloH2InMemoryNTest {
    @Value("${spring.datasource.url}")
    private String expectedJdbcUrl;
    @Autowired
    private DataSource dataSource;
```

The in-memory properties file supplies an H2 database that is suitable for running basic relational tests. We will inject that value directly into the test and have the `starter-data-jpa` module use that same URL to build the DataSource.

inmemory Profile Properties File

```
#DB to use during Unit Integration tests
spring.datasource.url=jdbc:h2:mem:dockercompose
```

2.3. Unit Integration Test

All we want to do during this example is to create a connection to the RDBMS database and verify that we have the expected connection. The snippet below shows a test that:

- verifies a DataSource was injected
- verifies the URL for connections obtained match expected

```
@Test
void can_get_connection() throws SQLException {
    //given
    then(dataSource).isNotNull(); ①
    String jdbcUrl;
    //when
    try(Connection conn=dataSource.getConnection()) { ②
        jdbcUrl=conn.getMetaData().getURL();
    }
    //then
    then(jdbcUrl).isEqualTo(expectedJdbcUrl); ③
    then(jdbcUrl).isEqualTo(expectedJdbcUrl);
    then(jdbcUrl).contains("jdbc:h2:mem"); ④
}
```

① container established `javax.sql.DataSource` starting point for SQL connections

② establish connection to obtain URL using Java's *try-with-resources* AutoCloseable feature for the connection

③ connection URL should match injected URL from property file

④ URL will start with standard `jdbc` URL for H2 in-memory DB

2.4. DataSource Service Injection

The example contains a very simple Spring MVC controller that will return the database URL for the injected `DataSource`.

Demo Controller to show Runtime Injection

```
@RestController
@RequiredArgsConstructor
public class HelloDBController {
    private final DataSource dataSource;

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

2.5. DataSource Service Injection Test

The test invokes the controller endpoint and verifies the database URL is what is expected.

Test to Verify DB URL used by API Service

```
@Test
void server_can_get_jdbc_connection() {
    //given
    URI url = helloDBUrl.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); ①
    then(jdbcUrl).contains("jdbc:h2:mem"); ②
}
```

① database URL should match URL injected from property file

② database URL should be a standard `jdbc` URL for an in-memory H2 database

2.6. DataSource Service Injection Test Result

The following snippet shows a portion of the in-memory RDBMS test results. The snippet below shows the server-side endpoint returning the JDBC URL used to establish a connection to the H2 database.

DataSource Service Injection Test Result

```
GET /api/hello/jdbc, headers=[accept:"text/plain, application/json, application/xml,
text/xml, application/*+json, application/*+xml, */*", user-agent:"masked",
host:"localhost:54539", connection:"keep-alive"]]

rcvd: [Content-Type:"text/plain;charset=UTF-8", Content-Length:"25", Date:"Tue, 06 Feb
2024 17:22:17 GMT", Keep-Alive:"timeout=60", Connection:"keep-alive"]
jdbc:h2:mem:dockercompose
```

At this point we can do development with the injected `DataSource` and work primarily within the IDE (no IT lifecycle test). A similar setup will be shown for MongoDB when we add Docker Compose. Let's start working on the Docker Compose aspects in the next section.

Chapter 3. Docker Compose Database Services

We will first cover what is required to use the database services during development. Our sole intention is to be able to connect to a live Postgres and MongoDB from JUnit as well as the API component(s) under test.

3.1. Docker Compose File

The snippet below shows a familiar set of Postgres and MongoDB database services from the previous Docker Compose lecture. For this lecture, I added the `ports` definition so that we could access the databases from our development environment in addition to later in the test network.

Both databases will be exposed to 0.0.0.0:port# on the Docker Host (which likely will be localhost during development). Postgres will be exposed using port 5432 and MongoDB using port 27017 on the Docker Host. We will have the opportunity to assign random available port numbers for use in automated Unit Integration Tests to keep tests from colliding.

```
services:
  postgres:
    image: postgres:12.3-alpine
    environment:
      POSTGRES_PASSWORD: secret
    ports: #only needed when IT tests are directly using DB
      - "${HOST_POSTGRES_PORT:-5432}:5432" ① ③
  mongodb:
    image: mongo:4.4.0-bionic
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: secret
    ports: #only needed when IT tests are directly using DB
      - "${HOST_MONGO_PORT:-27017}:27017" ② ③
```

① Postgres will be available on the Docker Host: `HOST_POSTGRES_PORT` (possibly localhost:5432 by default)

② MongoDB will be available on the Docker Host: `HOST_MONGO_PORT` (possibly localhost:27017 by default)

③ if only port# is specified, the port# will listen on all network connections (expressed as `0.0.0.0`)

3.2. Manually Starting Database Containers

With that definition in place, we can launch our two database containers exposed as 0.0.0.0:27017 and 0.0.0.0:5432 on the localhost. `0.0.0.0` is a value that means "all network interfaces" — internal-only (localhost) and externally exposed.

```
$ docker-compose -f src/main/docker/docker-compose.yml up mongodb postgres
...
[+] Running 2/0
 ✔ Container docker-postgres-1  Created    0.0s
 ✔ Container docker-mongodb-1   Created    0.0s
Attaching to mongodb-1, postgres-1
...
--
$ docker ps
CONTAINER ID   IMAGE                 PORTS                      NAMES
9856be1d8504   mongo:4.4.0-bionic    0.0.0.0:27017->27017/tcp   docker-mongodb-1
47d8346e2a78   postgres:12.3-alpine  0.0.0.0:5432->5432/tcp     docker-postgres-1
```

3.3. Postgres Container IT Test

Knowing we will be able to have a live set of databases at test time, we can safely author an IT test that looks very similar to the earlier NTest except that is uses 2 new profile property files.

- containerdb - is a profile that sets the base properties for remote Postgres (versus in-memory H2) and MongoDB instances. The host and port numbers will be dynamically assigned during that time by Maven plugins and a property filter.
- containerdb-dev - is a profile that overrides test-time settings during development. The host and port numbers will be fixed values.

```
@SpringBootTest(classes={DockerComposeHelloApp.class,
        ClientTestConfiguration.class},
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
//uncomment containerdb-dev when developing/running against a fixed instance
@ActiveProfiles({"test","containerdb","containerdb-dev"})
@Slf4j
class HelloPostgresContainerIT {
    @Value("${spring.datasource.url}")
    private String expectedJdbcUrl;
    @Autowired
    private DataSource dataSource;
```

@ActiveProfile can be programmatically defined

We will cover the use of an `ActiveProfilesResolver` capability later in the course within the MongoDB section — that will allow the `@ActiveProfile` to be programmatically defined without source code change.

The primary `containerdb` property file is always active and will reference the Postgres and MongoDB databases using ${placeholders} for the Docker Host and port numbers.

application-containerdb.properties

```
#used when running local application Test against remote DB
spring.datasource.url=jdbc:postgresql://${ejava-
parent.docker.hostname}:${host.postgres.port}/postgres ①
spring.datasource.username=postgres
spring.datasource.password=secret

spring.data.mongodb.uri=mongodb://admin:secret@${ejava-
parent.docker.hostname}:${host.mongodb.port}/test?authSource=admin ①
```

① placeholders will be replaced by literal values in pom when project builds

The `containerdb-dev` property file removes the ${placeholders} and uses localhost:fixed-port# to reference the running Postgres and MongoDB databases during development. This must get commented out before running the automated tests using Maven and Failsafe during the build.

application-containerdb-dev.properties

```
#used when running local application Test against remote DB
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres ①
spring.data.mongodb.uri=mongodb://admin:secret@localhost:27017/test?authSource=admin
①
```

① this assumes Docker Host is `localhost` during active IDE development

The net result of the layered set of property files is that the Postgres username and password will be re-used while the URLs are being wholesale replaced.

Placeholders are Replaced Before Runtime

Since we are using Maven resource filtering, the application-containerdb.properties placeholders will be replaced during the build. That means the placeholders will contain literal values during test execution and cannot be individually overridden.

3.4. Postgres Container IT Result

The following snippet shows the example results of running the IT test within the IDE, with the "-dev" profile activated, and running against the Postgres database container manually started using Docker Compose.

```
GET http://localhost:54204/api/hello/jdbc, returned 200 OK/200
sent: [Accept:"text/plain, application/json, application/xml, text/xml,
application/*+json, application/*+xml, */*", Content-Length:"0"]

rcvd: [Content-Type:"text/plain;charset=UTF-8", Content-Length:"41", Date:"Tue, 06 Feb
2024 16:58:58 GMT", Keep-Alive:"timeout=60", Connection:"keep-alive"]
jdbc:postgresql://localhost:5432/postgres ①
```

① last line comes from `conn.getMetaData().getURL()` in the controller

From here, we can develop whatever RDBMS code we need to develop, set breakpoints, write/execute tests, etc. before we commit/push for automated testing.

3.5. MongoDB Container IT Test

We have a similar job to complete with MongoDB. Much will look familiar. In this case, we will inject the source database URL and a `MongoClient` that has been initialized with that same database URL.

MongoDB Container IT Test

```
import com.mongodb.client.MongoClient;
...
@SpringBootTest(classes={DockerComposeHelloApp.class,
        ClientTestConfiguration.class},
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
//uncomment containerdb-dev when developing/running against a fixed instance
@ActiveProfiles({"test","containerdb","containerdb-dev"})
class HelloMongoContainerIT {
    @Value("${spring.data.mongodb.uri}")
    private String expectedMongoUrl;
    @Autowired
    private MongoClient mongoClient;
```

We will verify the `MongoClient` is using the intended URL information. Unfortunately, that is not a simple getter() — so we need to do some regular expression evaluations within a block of cluster description text.

MongoClient Injection Test

```
@Test
void can_get_connection() throws SQLException, JsonProcessingException {
    //given
    then(mongoClient).isNotNull();
    //when
    String shortDescription = mongoClient.getClusterDescription().getShortDescription
();
    //then
    new MongoVerifyTest().actual_hostport_matches_expected(expectedMongoUrl,
shortDescription);
}
```

The following snippet shows two (2) example cluster shortDescription Strings that can be returned.

Example Cluster Short Descriptions

```
{type=STANDALONE, servers=[{address=localhost:56295, type=STANDALONE,
```

```
roundTripTime=20.3 ms, state=CONNECTED}]} ①

{type=STANDALONE, servers=[{address=host.docker.internal:56295, type=STANDALONE,
roundTripTime=20.3 ms, state=CONNECTED}]} ②
```

① example expected shortDescription() when Docker Host is localhost

② example expected shortDescription() during automated testing within CI/CD Docker container when localhost is not the Docker Host

The following snippet shows some of the details to extract the host:portNumber from the shortDescription and mongoDB URL to see if they match.

MongoDB URL Validation Utility

```
public class MongoDBVerifyTest {
        //..., servers=[{address=localhost:56295, type=STANDALONE... ①
    private static final Pattern DESCR_ADDRESS_PATTERN = Pattern.compile("address=([A-
Za-z\\.:0-9]+),");
        //mongodb://admin:secret@localhost:27017/test?authSource=admin ②
    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
());

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

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

① m1 is targeting the runtime result

② m2 is targeting the configuration property setting

3.6. MongoClient Service Injection

The following snippet shows a server-side component that has been injected with the MongoClient and will return the cluster short description when called.

MongoClient Service Injection

```
@RestController
@RequiredArgsConstructor
@Slf4j
public class HelloDBController {
```

```
    private final MongoClient mongoClient;
...
    @GetMapping(path="/api/hello/mongo",
            produces = {MediaType.TEXT_PLAIN_VALUE})
    public String helloMongoClient() throws SQLException {
        return mongoClient.getClusterDescription().getShortDescription();
    }
```

The following snippet shows an API test — very similar to the Postgres "container DB" test — that verifies we have MongoDB client injected on the server-side and will be able to communicate with the intended DB server when needed.

MongoClient Service Injection Test

```
@Test
void server_can_get_mongo_connection() {
    //given
    URI url = helloDBUrl.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);
}
```

3.7. MongoDB Container IT Result

The following snippet shows an exchange between the IT test client and the server-side components. The endpoint returns the cluster short description text with the host:portNumber value from the injected `MongoClient`.

MongoDB Container IT Result

```
GET http://localhost:54507/api/hello/mongo, returned 200 OK/200
sent: [Accept:"text/plain, application/json, application/xml, text/xml,
application/*+json, application/*+xml, */*", Content-Length:"0"]

rcvd: [Content-Type:"text/plain;charset=UTF-8", Content-Length:"110", Date:"Tue, 06
Feb 2024 17:19:09 GMT", Keep-Alive:"timeout=60", Connection:"keep-alive"]
{type=STANDALONE, servers=[{address=localhost:27017, type=STANDALONE,
roundTripTime=70.2 ms, state=CONNECTED}] ①
```

① last line comes from `mongoClient.getClusterDescription().getShortDescription()` call in the controller

Chapter 4. Automating Container DB IT Tests

We have reached a point where:

- the server-side is injected with a database connection,
- the integration tests inquire about that connection, and
- the test results confirm whether the connection injected into the server has the correct properties

We can do that within the IDE using the `-dev` profile. To complete the module, we need wire the test to be automated within the Maven build.

4.1. Generate Random Available Port Numbers

We already have the `build-helper-maven-plugin` automatically activated because of the `application-it.properties` activation file being in place. The default plugin definition from the `ejava-build-parent` will generate a random available port# for `server.http.port`. We will continue to use that for the API and now generate two additional port numbers for database communications.

The following snippet shows an extension of the parent plugin declaration to include ports to expose Postgres and MongoDB on the Docker Host.

Generate Random Database Ports

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>build-helper-maven-plugin</artifactId>
    <executions>
        <execution>
            <id>reserve-network-port</id>
            <configuration>
                <portNames combine.children="append"> ①
                    <portName>host.postgres.port</portName>
                    <portName>host.mongodb.port</portName>
                </portNames>
            </configuration>
        </execution>
    </executions>
</plugin>
```

① two (2) additional `portName` s will be generated during the parent-defined `process-resources` phase

There are at least two ways to extend a parent definition for collections like `portNames`:

- combine.children="append" — adds child elements to parent elements
- combine.parent="override" — replaces the parent elements with child

elements

The following snippet shows the impact of the above definition when executing the `process-resources` page.

Port Allocation Example Output

```
$ mvn process-resources
...
[INFO] --- build-helper-maven-plugin:3.4.0:reserve-network-port (reserve-network-port)
@ dockercompose-it-example ---
[INFO] Reserved port 55225 for server.http.port
[INFO] Reserved port 55226 for host.postgres.port
[INFO] Reserved port 55227 for host.mongodb.port
```

We will use these variables to impact the containers and integration tests.

4.2. Launch Docker Compose with Variables

With the port number variables defined, we can start the network of containers using Docker Compose. The following snippet shows using the io.brachu:docker-compose-maven-plugin to implement the execution.

Launch Docker Compose

```
<plugin>
    <groupId>io.brachu</groupId>
    <artifactId>docker-compose-maven-plugin</artifactId>
    <configuration>
        <projectName>${project.artifactId}</projectName> ①
        <workDir>${project.basedir}/src/main/docker</workDir> ②
        <file>${project.basedir}/src/main/docker/docker-compose.yml</file> ③
        <skip>${skipITs}</skip> ④
        <env> ⑤
            <HOST_API_PORT>${server.http.port}</HOST_API_PORT>
            <HOST_POSTGRES_PORT>${host.postgres.port}</HOST_POSTGRES_PORT>
            <HOST_MONGODB_PORT>${host.mongodb.port}</HOST_MONGODB_PORT>
        </env>
        <wait> ⑥
            <value>${it.up.timeout.secs}</value>
            <unit>SECONDS</unit>
        </wait>
        <forceBuild>true</forceBuild> ⑦
        <removeOrphans>true</removeOrphans> ⑧
    </configuration>
    <executions>
        <execution>
            <goals> ⑨
                <goal>up</goal>
                <goal>down</goal>
```

```
            </goals>
        </execution>
    </executions>
    </executions>
</plugin>
```

① project name identifies a common alias each of the services are scoped under

② directory to resolve relative paths in `docker-compose.yml` file

③ we have placed the docker-compose.yml file in a sub-directory

④ we do not want to run this plugin if `install` is invoked with `-DskipITs`

⑤ these variables are expanded by Maven and passed into Docker Compose as environment variables

⑥ maximum time to wait for container to successfully start before failing

⑦ for source images, will force a build before running (e.g., run --build)

⑧ containers will be removed after being stopped (e.g., run --rm)

⑨ `up` already assigned to `pre-integration-test` and `down` assigned to `post-integration-test`

There are several uses of a directory context at play. They each are used to resolve relative path references.

- Docker build making reference to files copied
- Docker Compose build command making reference to the Dockerfile

The simplest setup is when the Dockerfile and docker-compose.yml file(s) are at the root of the module. In that case, the context defaults to `${project.basedir}` and all module artifacts are in sight.

If the `docker-compose.yml` and `Dockerfile` are placed within the `src` tree (e.g., `src/main/docker`), docker-compose will default the context to the `docker-compose.yml` directory. That can lead to some confusing relative references.

This example tries to make the command line simple, while still automating by:

- setting the `docker-compose-maven-plugin` `workDir` property to the directory of the `docker-compose.yml` file.
- sets `docker-compose.yml` `context` property to `../../..` (a.k.a. `${project.basedir}`)
- has all other file references relative to `${project.basedir}`

4.3. Filtering Test Resources

We also need to filter the test property files with our generated Maven properties:

- `ejava-parent.docker.hostname` — hostname of the Docker Host running images. This is being determined in the `ejava-build-parent` parent pom.xml.

- `host.postgres.port` — Docker Host port number to access Postgres
- `host.mongodb.port` — Docker Host port number to access MongoDB

The following snippet shows the source version of our property file — containing placeholders that match the generated Maven properties

src/test/resources/application-containerdb.properties

```
spring.datasource.url=jdbc:postgresql://${ejava-
parent.docker.hostname}:${host.postgres.port}/postgres ① ②
spring.datasource.username=postgres
spring.datasource.password=secret

spring.data.mongodb.uri=mongodb://admin:secret@${ejava-
parent.docker.hostname}:${host.mongodb.port}/test?authSource=admin ① ③
```

① `${ejava-parent.docker.hostname}` is being defined in `ejava-build-parent` pom.xml

② `${host.postgres.port}` is being defined in this project's pom.xml using `build-helper-maven-plugin`

③ `${host.mongodb.port}` is being defined in this project's pom.xml using `build-helper-maven-plugin`

These are being defined in a file. Therefore we need to update the file's values at test time. Maven provides first-class support for filtering placeholders in resources files. So much so, that one can define the filtering rules outside the scope of the resources plugin. The following (overly verbose) snippet below identifies two testResource actions — one with and one without filtering:

- filtering=true — containerdb.properties will be expanded when copied
- filtering=false — other resource files will be copied without filtering

Test Resource Filtering

```
<build>
    <testResources>
        <testResource>
            <directory>src/test/resources</directory>
            <filtering>true</filtering> ①
            <includes>
                <include>application-containerdb.properties</include>
            </includes>
        </testResource>
        <testResource>
            <directory>src/test/resources</directory>
            <filtering>false</filtering> ②
            <excludes>
                <exclude>application-containerdb.properties</exclude>
            </excludes>
        </testResource>
    </testResources>
```

① all resources matching this specification will have placeholders subject to filtering

② all resources subject to this specification will be copied as provided

Target Specific Files when Filtering

When adding resource filtering, it is good to identify specific files to filter versus all files. Doing it that way helps avoid unexpected and/or unwanted expansion to other resource files.

Resource File Copied to target Tree is modified

Maven is copying the resource file(s) from the `src/test/resources` tree to `target/test-classes`. `target/test-classes` will be part of the test classpath. Our configuration is adding an optional filtering to the copy process. This does not change the original in `src/test/resources`.

You cannot just specify the `filter=true` specification. If you do — the files that do not match the specification will not be copied at all.

The following snippet shows the result of processing the test resource files. One file was copied (filtered-true) followed by five (5) files (filtered=false). The test resource files are being copied to the `target/test-classes` directory.

Resource Filtering Output

```
$ mvn process-test-resources
...
[INFO] --- build-helper-maven-plugin:3.4.0:reserve-network-port (reserve-network-port)
@ dockercompose-it-example ---
[INFO] Reserved port 56026 for server.http.port
[INFO] Reserved port 56027 for host.postgres.port
[INFO] Reserved port 56028 for host.mongodb.port
...
[INFO] --- maven-resources-plugin:3.3.1:testResources (default-testResources) @
dockercompose-it-example ---
[INFO] Copying 1 resource from src/test/resources to target/test-classes ①
[INFO] Copying 5 resources from src/test/resources to target/test-classes ②
```

① 1 file copied with property filtering applied

② remaining files copied without property filtering

The snippet below shows the output result of the filtering. The `target/test-classes` copy of the file has the known placeholders expanded to their build-time values.

Resource Filtering Result

```
$ cat target/test-classes/application-containerdb.properties
...
spring.datasource.url=jdbc:postgresql://localhost:56027/postgres ①
spring.datasource.username=postgres
spring.datasource.password=secret
```

```
spring.data.mongodb.uri=mongodb://admin:secret@localhost:56028d/test?authSource=admin
②
```

① `${ejava-parent.docker.hostname}` was expanded to `localhost` and `${host.postgres.port}` was expanded to `56027`

② `${ejava-parent.docker.hostname}` was expanded to `localhost` and `${host.mongodb.port}` was expanded to `56028`

4.4. Failsafe

There are no failsafe changes required. All dynamic changes so far are being handled by the test property filtering and Docker Compose.

4.5. Test Execution

We start out test with a `mvn clean verify` to run all necessary phases. The snippet below shows a high-level view of the overall set of module tests.

Test Execution

```
$ mvn clean verify
...
[INFO] --- build-helper-maven-plugin:3.4.0:reserve-network-port (reserve-network-port)
@ dockercompose-it-example ---
[INFO] Reserved port 56176 for server.http.port
[INFO] Reserved port 56177 for host.postgres.port
[INFO] Reserved port 56178 for host.mongodb.port
...
[INFO] --- maven-resources-plugin:3.3.1:testResources (default-testResources) @
dockercompose-it-example ---
[INFO] Copying 1 resource from src/test/resources to target/test-classes
[INFO] Copying 5 resources from src/test/resources to target/test-classes
...
[INFO] --- maven-surefire-plugin:3.1.2:test (default-test) @ dockercompose-it-example
---
... ①
[INFO] --- docker-compose-maven-plugin:1.0.0:up (default) @ dockercompose-it-example
---
...
[INFO] --- maven-failsafe-plugin:3.1.2:integration-test (integration-test) @
dockercompose-it-example ---
... ②
[INFO] --- docker-compose-maven-plugin:1.0.0:down (default) @ dockercompose-it-example
---
...
[INFO] --- maven-failsafe-plugin:3.1.2:verify (verify) @ dockercompose-it-example ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

```
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:55 min
```

① (*Test) Unit/Unit Integration Tests are run here

② (*IT) Integration Tests are run here

4.5.1. Test Phase

During the `test` phase, our tests/API run with the in-memory version of the RDBMS. SpringBootTest is responsible for the random available port number without any Maven help.

Test Phase

```
[INFO] Running info.ejava.examples.svc.docker.hello.HelloH2InMemoryNTest
...
GET /api/hello/jdbc, headers=[accept:"text/plain, application/json, application/xml,
text/xml, application/*+json, application/*+xml, */*", user-agent:"masked",
host:"localhost:56181", connection:"keep-alive"]]
13:46:58.767 main DEBUG i.e.e.c.web.RestTemplateLoggingFilter#intercept:37
GET http://localhost:56181/api/hello/jdbc, returned 200 OK/200
...
jdbc:h2:mem:dockercompose
```

4.5.2. Docker Compose Up

The Maven Docker Compose Plugin starts the identified services within the `docker-compose.yml` file during the `pre-integration-test` phase.

Docker Compose Up

```
 [INFO] --- docker-compose-maven-plugin:1.0.0:up (default) @ dockercompose-it-example
---
...
 Network dockercompose-it-example_default  Creating
 Network dockercompose-it-example_default  Created
 Container dockercompose-it-example-mongodb-1  Creating
 Container dockercompose-it-example-postgres-1  Creating
 Container dockercompose-it-example-postgres-1  Created
 Container dockercompose-it-example-mongodb-1  Created
 Container dockercompose-it-example-postgres-1  Starting
 Container dockercompose-it-example-mongodb-1  Starting
 Container dockercompose-it-example-mongodb-1  Started
 Container dockercompose-it-example-postgres-1  Started
```

4.5.3. MongoDB Container Test Output

The following snippet shows the output of the automated IT test where the client and API are using generated port numbers to reach the API and MongoDB database. The API is running local to the IT

test, but the MongoDB database is running on the Docker Host — which happens to be localhost in this case.

MongoDB Container Test Output

```
[INFO] Running info.ejava.examples.svc.docker.hello.HelloMongoDBContainerIT
...
GET http://localhost:56216/api/hello/mongo, returned 200 OK/200
...
{type=STANDALONE, servers=[{address=localhost:56178, type=STANDALONE,
roundTripTime=25.2 ms, state=CONNECTED}]
```

4.5.4. Postgres Container Test Output

The following snippet shows the output of the automated IT test where the client and API are using generated port numbers to reach the API and Postgres database. The API is running local to the IT test, but the Postgres database is running on the Docker Host — which also happens to be localhost in this case.

Postgres Container Test Output

```
[INFO] Running info.ejava.examples.svc.docker.hello.HelloPostgresContainerIT
...
GET http://localhost:56216/api/hello/jdbc, returned 200 OK/200
...
jdbc:postgresql://localhost:56177/postgres
```

Chapter 5. CI/CD Test Execution

During CI/CD execution, the `${ejava-parent.docker.hostname}` placeholder gets replaced with `host.docker.internal` and mapped to `host-gateway` inside the container to represent the address of the Docker Host.

The following snippet shows an example of the alternate host mapping. Note that the API is still running local to the IT test — therefore it is still using `localhost` to reach the API.

CI/CD Test Execution

```
$ docker-compose run mvn mvn clean verify
...
[INFO] --- build-helper-maven-plugin:3.4.0:reserve-network-port (reserve-network-port)
@ dockercompose-it-example ---
[INFO] Reserved port 40475 for server.http.port
[INFO] Reserved port 41767 for host.postgres.port
[INFO] Reserved port 35869 for host.mongodb.port
...
[INFO] Running info.ejava.examples.svc.docker.hello.HelloMongoDBContainerIT
...
GET http://localhost:40475/api/hello/mongo, returned 200 OK/200 ①
...
{type=STANDALONE, servers=[{address=host.docker.internal:35869, type=STANDALONE,
roundTripTime=43.5 ms, state=CONNECTED}] ②
...
[INFO] Running info.ejava.examples.svc.docker.hello.HelloPostgresContainerIT
...
GET http://localhost:40475/api/hello/jdbc, returned 200 OK/200 ①
...
jdbc:postgresql://host.docker.internal:41767/postgres ②
```

① API is running local to IT tests, thus they use `localhost` to reach

② Postgres and MongoDB containers are running on Docker Host, thus use `host.docker.internal` to reach

Chapter 6. Adding API Container

At this point in the lecture, we have hit most of the primary learning objectives I wanted to make except one: building a Docker image under test using Docker Compose.

6.1. API in Docker Compose

We can add the API to the Docker Compose file — the same as with Postgres and MongoDB. However, there are at least two differences

1. the API is dependent on the `postgres` and `mongodb` services
2. the API is under development — possibly changing — thus should be built

Therefore, there is a separate `build.dockerfile` property to point to the source of the image and a `depends_on` property to identify the dependency/linkage to the other services.

API in Docker Compose

```
  api:
    build:
      context: ../../.. ①
      dockerfile: src/main/docker/Dockerfile ②
    image: dockercompose-hello-example:latest ③
    ports:
      - "${HOST_API_PORT:-8080}:8080"
    depends_on: ④
      - postgres
      - mongodb
    environment: ⑤
      - DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
      - MONGODB_URI=mongodb://admin:secret@mongodb:27017/test?authSource=admin
```

① directory where all relative paths will be based (`${project.basedir}`)

② source Dockerfile to build image

③ name and tag for built and executed image

④ services to wait for and to add references in `/etc/hosts`

⑤ environment variables to set at runtime

Adding this service to the Docker Compose file will cause the API image to build and container to start along with the Postgres and MongoDB containers.

6.2. API Dockerfile

The following API Dockerfile should be familiar to you — with the addition of the `run_env.sh` file. It is a two stage Dockerfile where:

1. the elements of the final Docker image are first processed and staged

2. the elements are put into place and runtime aspects are addressed

```
FROM eclipse-temurin:17-jre AS builder ①
WORKDIR /builder
COPY src/main/docker/run_env.sh .
ARG JAR_FILE=target/*-bootexec.jar
COPY ${JAR_FILE} application.jar
RUN java -Djarmode=tools -jar application.jar extract --layers --launcher
--destination extracted

FROM eclipse-temurin:17-jre ②
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"]
```

① process/stage elements of the final image

② final layout/format of elements and runtime aspects of image

6.3. run_env.sh

I have added the run_env.sh to represent a wrapper around the Java executable. It is very common to do this when configuring the JVM execution. In this example, I am simply mapping a single string database URL to standard Spring Data properties.

Environment variables with the following format ...

Compact DB URL Formats

```
DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
MONGODB_URI=mongodb://admin:secret@mongodb:27017/test?authSource=admin
```

... will be transformed into command line parameters with the following form.

Standard Spring Data Properties

```
--spring.data.datasource.url=jdbc:postgresql://host.docker.internal:5432/postgres
--spring.data.datasource.username=postgres
--spring.data.datasource.password=secret
--spring.data.mongodb.uri=mongodb://admin:secret@mongodb:27017/test?authSource=admin
```

The MongoDB URL is very trivial. Postgres will require a little work with the bash shell.

6.3.1. run_env.sh High Level Skeleton

The following snippet shows the high level skeleton of the `run_env.sh` script we will put in place to address all types of environment variables we will see in our environments. The shell will launch whatever command was passed to it ("$@") and append the `OPTIONS` that it was able to construct from environment variables. We will place this shell script in the `src/main/docker` directory to be picked up by the Dockerfile.

The resulting script was based upon the much more complicated example. I was able to simplify it by assuming the database properties would only be for Postgres and MongoDB.

run_env.sh Environment Variable Script

```
#!/bin/bash

OPTIONS=""

#ref: https://raw.githubusercontent.com/heroku/heroku-buildpack-jvm-
common/main/opt/jdbc.sh
if [[ -n "${DATABASE_URL:-}" ]]; then
  # ...
fi

if [[ -n "${MONGODB_URI:-}" ]]; then
  # ...
fi

exec $@ ${OPTIONS} ①
```

① executing the passed in arguments as the command line, in addition to the contents of `${OPTIONS}`

6.3.2. Script Output

When complete, a `DATABASE_URL` will be transformed into separate Spring Data JPA `url`, `username`, and `password` properties.

```
$ export DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres && bash
src/main/docker/run_env.sh echo ① ②
--spring.datasource.url=jdbc:postgresql://postgres:5432/postgres
--spring.datasource.username=postgres --spring.datasource.password=secret ③

$ unset DATABASE_URL ④
```

① `DATABASE_URL` environment variable set with compound value

② `echo` command executed with contents of `${OPTIONS}` containing results of processing `DATABASE_URL` value

③ the contents of `${OPTIONS}`; output by `echo` command

④ clearing the `DATABASE_URL` property from the current shell environment

A `MONGODB_URI` will be pass thru as a single URL using the Spring Data Mongo `url` property.

```
export MONGODB_URI=mongodb://admin:secret@mongo:27017/test?authSource=admin && bash
src/main/docker/run_env.sh echo
--spring.data.mongodb.uri=mongodb://admin:secret@mongo:27017/test?authSource=admin

$ unset MONGODB_URI
```

Running that within Docker, results in the following.

Demonstrating run_env.sh within Docker

```
docker run \
-e DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres \ ①
-e MONGODB_URI=mongodb://admin:secret@mongo:27017/test?authSource=admin \
-v `pwd`/src/main/docker/run_env.sh:/tmp/run_env.sh \ ②
openjdk:17.0.2 \ ③
/tmp/run_env.sh echo ④

--spring.datasource.url=jdbc:postgresql://postgres:5432/postgres
--spring.datasource.username=postgres --spring.datasource.password=secret
--spring.data.mongodb.uri=mongodb://admin:secret@mongo:27017/test?authSource=admin ⑤
```

① `-e` sets environment variable for running Docker container

② mapping `run_env.sh` in source tree to location within Docker container

③ using a somewhat vanilla Docker image

④ executing `echo` command using `run_env.sh` within Docker container

⑤ output of `run_env.sh` after processing the `-e` environment variables

Lets break down the details of `run_env.sh`.

6.3.3. DataSource Properties

The following script will breakout URL, username, and password using bash regular expression matching and turn them into Spring Data properties on the command line.

DataSource Properties

```
#DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres

if [[ -n "${DATABASE_URL:-}" ]]; then
  pattern="^postgres://(.+):(.+)@(.+)$"  ①
  if [[ "${DATABASE_URL}" =~ $pattern ]]; then ②
    JDBC_DATABASE_USERNAME="${BASH_REMATCH[1]}"
    JDBC_DATABASE_PASSWORD="${BASH_REMATCH[2]}"
    JDBC_DATABASE_URL="jdbc:postgresql://${BASH_REMATCH[3]}"
```

```
    OPTIONS="${OPTIONS} --spring.datasource.url=${JDBC_DATABASE_URL} "
    OPTIONS="${OPTIONS} --spring.datasource.username=${JDBC_DATABASE_USERNAME}"
    OPTIONS="${OPTIONS} --spring.datasource.password=${JDBC_DATABASE_PASSWORD}"
  else
    OPTIONS="${OPTIONS} --no.match=${DATABASE_URL}" ③
  fi
fi
```

① regular expression defining three (3) extraction variables: username, password, and URI snippet

② if the regular expression finds a match, we will pull that apart and assemble the properties using `BASH_REMATCH`

③ if no match is found, `--no.match` is populated with the DATABASE_URL to be printed for debug reasons

6.3.4. MongoDB Properties

The Mongo URL we are using can be passed in as a single URL property. If Postgres was this straight forward, we could have stuck with the `CMD` option.

MongoDB Property

```
if [[ -n "${MONGODB_URI:-}" ]]; then
  OPTIONS="${OPTIONS} --spring.data.mongodb.uri=${MONGODB_URI}"
fi
```

6.4. API Container Integration Test

The API Container IT test is very similar to the others except that it is client-side only. That means there will be no `DataSource` or `MongoClient` to inject and the responses from the API calls should return URLs that reference the `postgres:5432` and `mongodb:27017` host and port numbers — matching the Docker Compose configuration.

API Container IT Test

```
@SpringBootTest(classes=ClientTestConfiguration.class, ①
    webEnvironment = SpringBootTest.WebEnvironment.NONE) ②
@ActiveProfiles({"test","it"})
//we have project Mongo dependency but don't have or need Mongo for this remote client
@EnableAutoConfiguration(exclude = { ③
        MongoAutoConfiguration.class,
        MongoDataAutoConfiguration.class
})
@Slf4j
class HelloApiContainerIT {
    @Autowired
    private RestTemplate anonymousUser;
```

① not using @SpringBootApplication class

② not running a local web server

③ helps eliminate meaningless MongoDB errors in the client

I am going to skip over some testing setup details that I am pretty sure you can either guess or find by inspecting the source. The following test contacts the API running within a container within the Docker Compose network and verifies it is using a database connection to a host named `postgres` and using port `5432`.

Postgres DataSource API Test

```
@Test
void server_can_get_jdbc_connection() {
    //given
    String name="jim";
    URI url = helloDBUrl.build("jdbc");
    RequestEntity<Void> request = RequestEntity.get(url).build();
    //when
    String jdbcUrl = anonymousUser.exchange(request, String.class).getBody();
    //then
    then(jdbcUrl).contains("jdbc:postgresql");
}
```

The following test contacts the API as well to verify it is using a database connection to a host named `mongodb` and using port `27017`.

MongoDB Client API Test

```
@Test
void server_can_get_mongo_connection() {
    //given
    URI url = helloDBUrl.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);
}
```

6.4.1. Building and Running with API Service

The primary difference in the build — after adding the API service — is the building of the API according to the Dockerfile instructions and then starting all the images during the `pre-integration-test` phase.

Building and Running with API Service

```
[INFO] --- docker-compose-maven-plugin:1.0.0:up (default) @ dockercompose-it-example
```

```
---
#0 building with "desktop-linux" instance using docker driver

#1 [api internal] load build definition from Dockerfile
#1 transferring dockerfile: 788B 0.0s done
#1 DONE 0.0s

#2 [api internal] load metadata for docker.io/library/eclipse-temurin:17-jre
#2 DONE 0.0s

#3 [api internal] load .dockerignore
#3 transferring context: 2B done
#3 DONE 0.0s

#4 [api builder 1/5] FROM docker.io/library/eclipse-temurin:17-jre
#4 DONE 0.0s

#5 [api internal] load build context
#5 transferring context: 63.78MB 1.2s done
#5 DONE 1.2s

#6 [api builder 2/5] WORKDIR /builder
#6 CACHED

#7 [api builder 3/5] COPY src/main/docker/run_env.sh .
#7 CACHED

#8 [api builder 4/5] COPY target/*-bootexec.jar application.jar
#8 DONE 0.6s

#9 [api builder 5/5] RUN java -Djarmode=tools -jar application.jar extract --layers
--launcher --destination extracted
#9 DONE 1.6s

#10 [api stage-1 3/7] COPY --from=builder /builder/extracted/dependencies/ ./
#10 CACHED

#11 [api stage-1 4/7] COPY --from=builder /builder/extracted/spring-boot-loader/ ./
#11 CACHED

#12 [api stage-1 5/7] COPY --from=builder /builder/extracted/snapshot-dependencies/ ./
#12 CACHED

#13 [api stage-1 6/7] COPY --from=builder /builder/extracted/application/ ./
#13 CACHED

#14 [api stage-1 2/7] WORKDIR /application
#14 CACHED

#15 [api stage-1 7/7] COPY --chmod=555 --from=builder /builder/run_env.sh ./
#15 CACHED
```

```
#16 [api] exporting to image
#16 exporting layers done
#16 writing image
sha256:cadb45a28f3e8c2d7b521ec4ffda5276e72cbb147225eb1527ee4356f6ba44cf done
#16 naming to docker.io/library/dockercompose-hello-example:latest done
#16 DONE 0.0s

#17 [api] resolving provenance for metadata file
#17 DONE 0.0s
 Network dockercompose-it-example_default  Creating
 Network dockercompose-it-example_default  Created
 Container dockercompose-it-example-postgres-1  Creating
 Container dockercompose-it-example-mongodb-1  Creating
 Container dockercompose-it-example-postgres-1  Created
 Container dockercompose-it-example-mongodb-1  Created
 Container dockercompose-it-example-api-1  Creating
 Container dockercompose-it-example-api-1  Created
 Container dockercompose-it-example-mongodb-1  Starting
 Container dockercompose-it-example-postgres-1  Starting
 Container dockercompose-it-example-mongodb-1  Started
 Container dockercompose-it-example-postgres-1  Started
 Container dockercompose-it-example-api-1  Starting
 Container dockercompose-it-example-api-1  Started
```

6.4.2. API Container IT CI/CD Test Output

The following snippet shows key parts of the API Container IT test output when running within the CI/CD environment.

API Container IT CI/CD Test Output

```
[INFO] Running info.ejava.examples.svc.docker.hello.HelloApiContainerIT
...
GET http://host.docker.internal:37875/api/hello/jdbc, returned 200 OK/200 ①
...
jdbc:postgresql://postgres:5432/postgres ②
...
GET http://host.docker.internal:37875/api/hello/mongo, returned 200 OK/200 ①
...
{type=STANDALONE, servers=[{address=mongodb:27017, type=STANDALONE, roundTripTime=79.6
ms, state=CONNECTED}] ③
```

① API is running in remote Docker image on Docker Host — which is not localhost

② API is using a Postgres database for its DataSource. Postgres is running on a host known to the API as `postgres` because of the Docker Compose file dependency mapping

③ API is using a MongoDB database for its MongoClient. MongoDB is running on a host known to the API as `mongodb` because of the Docker Compose file dependency mapping

Chapter 7. Summary

In this module, we learned:

- to integrate Docker Compose into a Maven integration test phase
- to wrap the JVM execution in a wrapper script to process custom runtime options
- to address development, build, and CI/CD environment impact to hostnames and port numbers.