Testcontainers with Spock

jim stafford

Fall 2024 v2020-08-14: Built: 2024-11-19 21:44 EST

Table of Contents

Chapter 1. Introduction

In several other lectures in this section I have individually covered the use of embedded resources, Docker, Docker Compose, and Testcontainers for the purpose of implementing integration tests using JUnit Jupiter.

In this lecture, I am going to cover using Docker Compose and Testcontainers with Spock to satisfy an additional audience. I am assuming the reader of this set of lecture notes may not have gone through the earlier material but is familiar with Docker and Spock (but not used together). I will be repeating some aspects of earlier lectures but provide only light detail. Please refer back to the earlier lecture notes if you need more details.

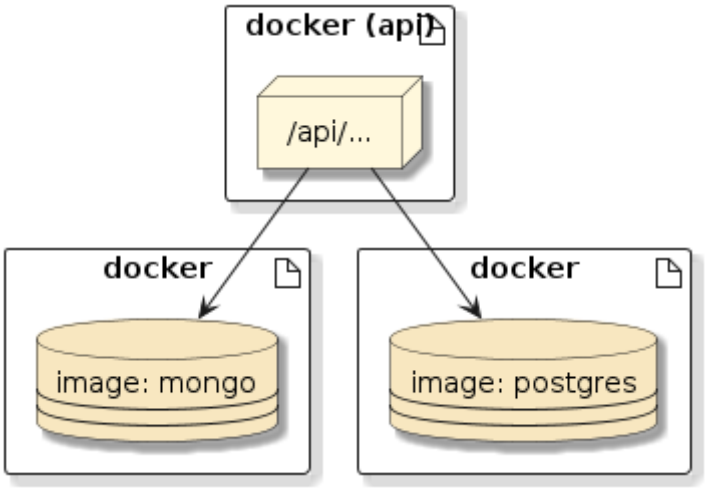

Figure 1. Target Integration Environment

Integration Unit Test terminology

I use the term "integration test" somewhat loosely but use the term "integration unit test" to specifically mean a test that uses the Spring context under the control of a simple unit test capable of being run inside of an IDE (without assistance) and executed during the Maven test phase. I use the term "unit test" to mean the same thing except with stubs or mocks and the lack of the overhead (and value) of the Spring context.

1.1. Goals

You will learn:

- to identify the capability of Docker Compose to define and implement a network of virtualized services running in Docker
- to identify the capability of Testcontainers to seamlessly integrate Docker and Docker Compose into unit test frameworks including Spock
- to author end-to-end, integration unit tests using Spock, Testcontainers, Docker Compose, and Docker
- to implement inspections of running Docker images
- to implement inspects of virtualized services during tests
- to instantiate virtualized services for use in development

1.2. Objectives

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

1. define a simple network of Docker-based services within Docker Compose
2. control the lifecycle of a Docker Compose network from the command line
3. implement a Docker Compose override file
4. control the lifecycle of a Docker Compose network using Testcontainers
5. implement an integration unit test within Spock, using Testcontainers and Docker Compose
6. implement a hierarchy of test classes to promote reuse

Chapter 2. Background

2.1. Application Background

The application we are implementing and looking to test is a set of voting services with back-end resources. Users cast votes using the Votes Service and obtain election results using the Elections Service. Casted votes are stored in MongoDB and election results are stored and queried in Postgres. The two services stay in sync through a JMS topic hosted on ActiveMQ.

Because of deployment constraints unrelated to testing — the two services have been hosted in the same JVM

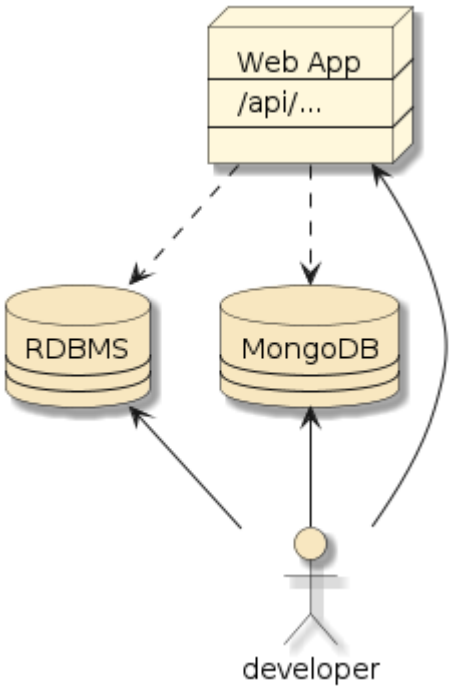

Figure 2. Voting and Election Services

2.2. Integration Testing Approach

The target of this lecture is the implementation of end-to-end integration tests. Integration tests do not replace fine-grain unit tests. In fact there are people with strong opinions (expressed) that believe any attention given to integration tests takes away from the critical role of unit tests when it comes to thorough testing. I will agree there is some truth to that — we should not get too distracted by this integration verification playground to the point that we end up placing tests that could be verified in pure, fast unit tests — inside of larger, slower integration tests. However, there has to be a point in the process where we need to verify some amount of useful end-to-end threads of our application in an **automated** manner — especially in today's world of microservices where critical supporting services have been broken out. Without the integration test — there is nothing that proves everything comes together during dynamic operation. Without the automation — there is no solid chance regression testing.

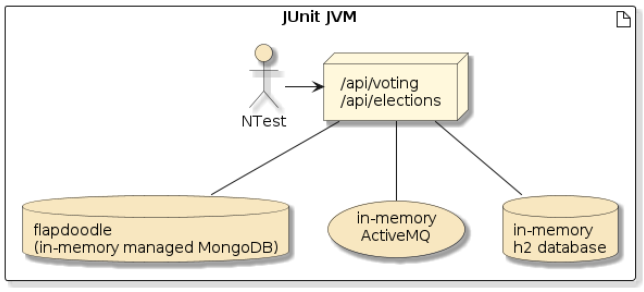

Figure 3. In-Memory/Simulated Integration Testing Environment

One way to begin addressing automated integration testing with back-end resources is through the use of in-memory configurations and simulation of dependencies — local to the unit test JVM. This addresses some of the integration need when it is something like a database or JMS server, but will miss the mark completely when we need particular versions of a full fledged application service.

We want to instead take advantage of the popularity of Docker and the ability to virtualize most back-end and many application services. We want/need this to be automated like our other tests so that they can be run as a part of any build or release. Because of their potential extended length of time and narrow focus — we will want to separate them into distinct modules to control when they are executed.

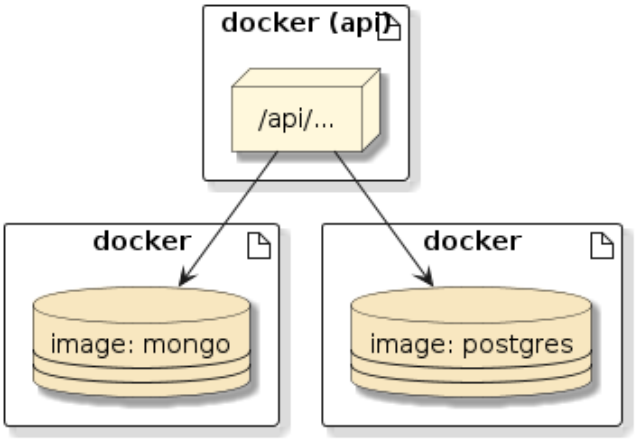

Figure 4. Virtualized Integration Testing Environment

2.3. Docker Compose

A network of services can be complex and managing many individual Docker images is clumsy. It would be best if we took advantage of a Docker network/service management layer called Docker Compose.

Docker Compose uses a YAML file to define the network, services, and even builds services with "source" build information. With that in place, we can issue a build, start and stop of the services as well as execute commands to run within the running images. All of this must be on the same machine.

Because Docker Compose is limited to a single machine and is primarily just a thin coordination layer around Docker — it is MUCH simpler to use than Kubernetes or MiniKube. For those familiar with Kubernetes — I like to refer to it as a "poor man's Helm Chart".

At a minimum, Docker Compose provides a convenient wrapper where we can place environment and runtime options for individual containers. These containers could be simple databases or JMS servers — eliminating the need to install software on the local development machine. The tool really begins to shine when we need to define dependencies and communication paths between services.

2.4. Testcontainers

Testcontainers provides a seamless integration of Docker and Docker Compose into unit test frameworks — including JUnit 4, JUnit 5, and Spock. Testcontainers manages a library of resource-specific containers that can provide access to properties that are specific to a particular type of image (e.g., databaseUrl for a Postgres container). Testcontainers also provide a generic container and a Docker Compose container — which provide all the necessary basics of running either a single image or a network of images.

Testcontainers provides features to

- parse the Docker Compose file to learn the configuration of the network

- assign optional variables used by the Docker Compose file
- expose specific container ports as random host ports
- identify the host port value of a mapped container port
- delay the start of tests while built-in and customizable "wait for" checks execute to make sure the network is up and ready for testing
- execute shell commands against the running containers
- share a running network between (possibly ordered) tests or restart a dirty network between tests

Chapter 3. Docker Compose

Before getting into testing, I will cover Docker Compose as a stand-alone capability. Docker Compose is very useful in standing up one or more Docker containers on a single machine, in a development or integration environment, without installing any software beyond Docker and Docker Compose (a simple binary).

3.1. Docker Compose File

Docker Compose uses one or more `YAML` Docker Compose files for configuration. The default primary file name is `docker-compose.yml`, but you can reference any file using the `-f` option.

The following is a Docker Compose File that defines a simple network of services. I reduced the version of the file in the example to `2` versus a current version of `3.8` since what I am demonstrating has existed for many (>5) years.

I have limited the service definitions to an image spec, environment variables, and dependencies. I have purposely not exposed any container ports at this time to avoid concurrent execution conflicts in the base file. I have also purposely left out any build information for the API image since that should have been built by an earlier module in the Maven dependencies. However, you will see a decoupled way to add port mappings and build information shortly when we get to the Docker Compose Override/Extend topic. For now — this is our core network definition.

Example docker-compose.yml File

```
version: '2'
services:
  mongo:
    image: mongo:4.4.0-bionic
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: secret
  postgres:
    image: postgres:12.3-alpine
    environment:
      POSTGRES_PASSWORD: secret
  activemq:
    image: rmohr/activemq:5.15.9
  api:
    image: dockercompose-votes-api:latest
    depends_on: ①
      - mongo
      - postgres
      - activemq
    environment:
      - spring.profiles.active=integration
      - MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin
      - DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
```

① defines a requirement as well as an `/etc/hostname` entry to dependent

3.2. Start Network

We can start the network using the `up` command. We can add a `-d` option to make all services run in the background. The runtime container names will have a project prefix and that value defaults to the name of the parent directory. It can be overridden using the `-p` option.

Starting Explicitly Named Network in Background

```
$ docker-compose -p foo up -d
Creating foo_activemq_1 ... done
Creating foo_postgres_1 ... done
Creating foo_mongo_1    ... done
Creating foo_api_1      ... done
```

The following shows the runtime Docker image name and port numbers for the running images. They all start with the project prefix "foo". This is important when trying to manage multiple instances of the network. Notice too that none of the ports have been mapped to a host port at this time. However, they are available on the internally defined "foo" network (i.e., accessible from the API service).

Partial Docker Status

```
$ docker ps                          ①
IMAGE                               PORTS                     NAMES
dockercompose-votes-api:latest                                foo_api_1
postgres:12.3-alpine                5432/tcp                  foo_postgres_1
rmohr/activemq:5.15.9               1883/tcp, 5672/tcp, ...   foo_activemq_1
mongo:4.4.0-bionic                  27017/tcp                 foo_mongo_1
```

① no internal container ports are being mapped to localhost ports at this time

3.3. Access Logs

You can access the logs of all running services or specific services running in the background using the `logs` command and by naming the services desired. You can also limit the historical size with `--tail` option and follow the log with `-f` option.

Example Access to Logs

```
$ docker-compose -p foo logs --tail 2 -f mongo activemq
Attaching to foo_activemq_1, foo_mongo_1
mongo_1     | {"t":{"$date":"2020-08-15T14:10:20.757+00:00"},"s":"I",  ...
mongo_1     | {"t":{"$date":"2020-08-15T14:11:41.580+00:00"},"s":"I",  ...
activemq_1  |  INFO | No Spring WebApplicationInitializer types detected ...
activemq_1  |  INFO | jolokia-agent: Using policy access restrictor classpath:...
```

3.4. Execute Commands

You can execute commands inside a running container. The following shows an example of running the Postgres CLI (`psql`) against the `postgres` container to issue a SQL command against the `VOTE` table. This can be very useful during test debugging — where you can interactively inspect the state of the databases during a breakpoint in the automated test.

Example Exec Command

```
$ docker-compose -p foo exec postgres psql -U postgres -c "select * from VOTE"
 id | choice | date | source ①
----+--------+------+--------
(0 rows)
```

① executing command that runs inside the running container

3.5. Shutdown Network

We can shutdown the network using the `down` command or `<ctl>-C` if it was launched in the foreground. The project name is required if it is different from the parent directory name.

```
$ docker-compose -p foo down
Stopping foo_api_1      ... done
Stopping foo_activemq_1 ... done
Stopping foo_mongo_1    ... done
Stopping foo_postgres_1 ... done
Removing foo_api_1      ... done
Removing foo_activemq_1 ... done
Removing foo_mongo_1    ... done
Removing foo_postgres_1 ... done
Removing network foo_default
```

3.6. Override/Extend Docker Compose File

If CLI/shell access to the VMs is not enough, we can create an override file to specialize the base file. The following example maps key ports in each Docker container to a host port.

Example Docker Compose Override File

```
version: '2'
services:
  mongo: ①
    ports:
      - "27017:27017"
  postgres:
    ports:
      - "5432:5432"
  activemq:
```

```
    ports:
      - "61616:61616"
      - "8161:8161"
  api:
    build: ②
      context: ../dockercompose-votes-svc
      dockerfile: Dockerfile
    ports:
      - "${API_PORT}:8080"
```

① extending definitions of services from base file

② adding source module info to be able to rebuild image from this module

3.7. Using Mapped Host Ports

Mapping container ports to host ports is useful if you want to simply use Docker Compose to manage a development environment or you have a tool — like Mongo Compass — that requires a standard URL.

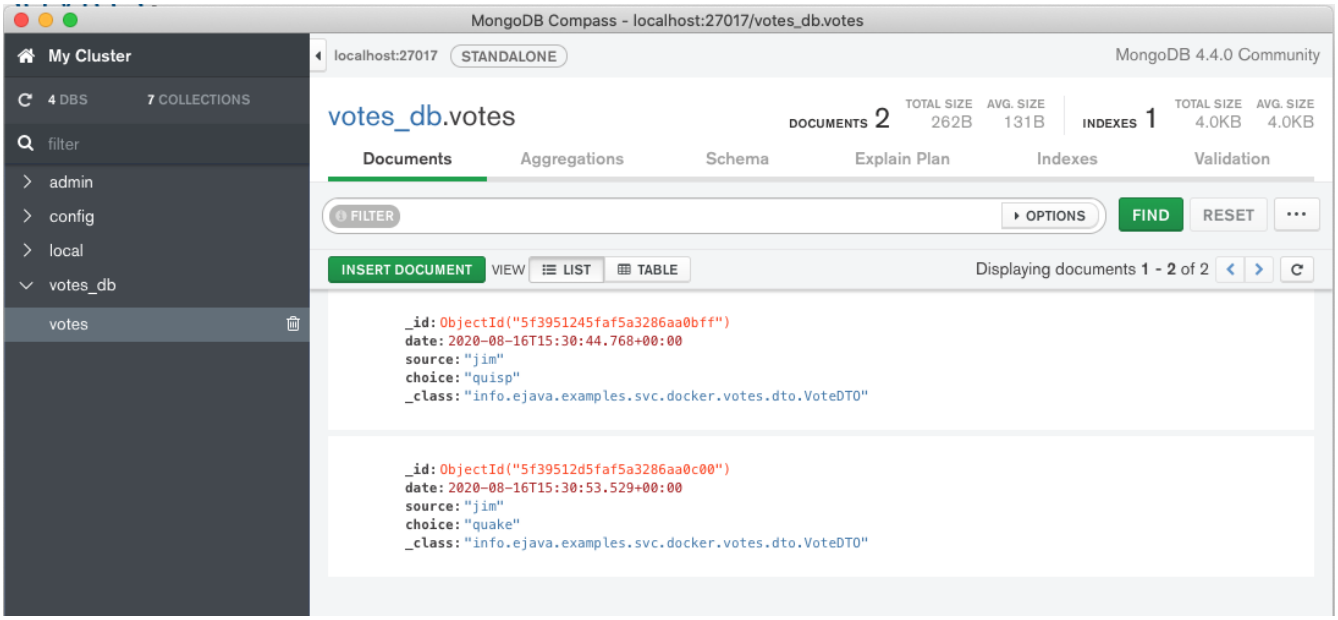

Figure 5. MongoDB Compass Connected to MongoDB in Docker Compose

3.8. Supplying Properties

Properties can be passed into the image by naming the variable. The value is derived from one of the following (in priority order):

1. `NAME: value` explicitly supplied in the Docker Compose File
2. `NAME=value` defined in environment variable
3. `NAME=value` defined in an environment file

The following are example environment files mapping `API_PORT` to either 9999 or 9090. We can activate an environment file using the `--env-file` option or have it automatically applied when

named `.env`.

Example alt-env File

```
$ cat alt-env ①
API_PORT=9999
$ cat .env ②
API_PORT=9090
```

① used when `--env-file alt-env` supplied

② used by default

3.9. Specifying an Override File

You can specify an override file by specifying multiple Docker Compose files in priority order with the `-f` option. The following will use `docker-compose.yml` as a base and apply the augmentations from `development.yml`.

Example Explicit Override Specification

```
$ docker-compose -p foo -f ./docker-compose.yml -f ./development.yml up -d
Creating network "foo_default" with the default driver
```

You can have the additional file applied automatically if named `docker-compose.override.xml`. The example below uses the `docker-compose.xml` file as the primary and the `docker-compose.override.yml` file as the override.

Using Default Docker Compose and Docker Compose Override File Names

```
$ ls docker-compose*
docker-compose.override.yml docker-compose.yml
$ docker-compose -p foo up -d ①
```

① using default Docker Compose file with default override file

3.10. Override File Result

The following shows the new network configuration that shows the impact of the override file. Key communication ports of the back-end resources have been exposed on the localhost network.

Example Docker Compose Network Status with Override

```
$ docker ps                    ① ②
IMAGE                          PORTS                                       NAMES
dockercompose-votes-api:latest 0.0.0.0:9090->8080/tcp                      foo_api_1
mongo:4.4.0-bionic             0.0.0.0:27017->27017/tcp                    foo_mongo_1
rmohr/activemq:5.15.9          1883/tcp, ... 0.0.0.0:61616->61616/tcp      foo_activemq_1
postgres:12.3-alpine           0.0.0.0:5432->5432/tcp                      foo_postgres_1
```

① container ports are now mapped to (fixed) host ports

② API host port used the variable defined in `.env` file

Override files cannot reduce or eliminate collections

Override files can replace single elements but can only augment multiple elements. That means one cannot eliminate exposed ports from a base configuration file. Therefore it is best to keep from adding properties that may be needed in the base file versus adding to environment-specific files.

Chapter 4. Testcontainers and Spock

With an understanding of Docker Compose and a few Maven plugins — we could easily see how we could integrate our Docker images into an integration test using the Maven integration-test phases.

However, by using Testcontainers — we can integrate Docker Compose into our unit test framework much more seamlessly and launch tests in an ad-hoc manner right from within the IDE.

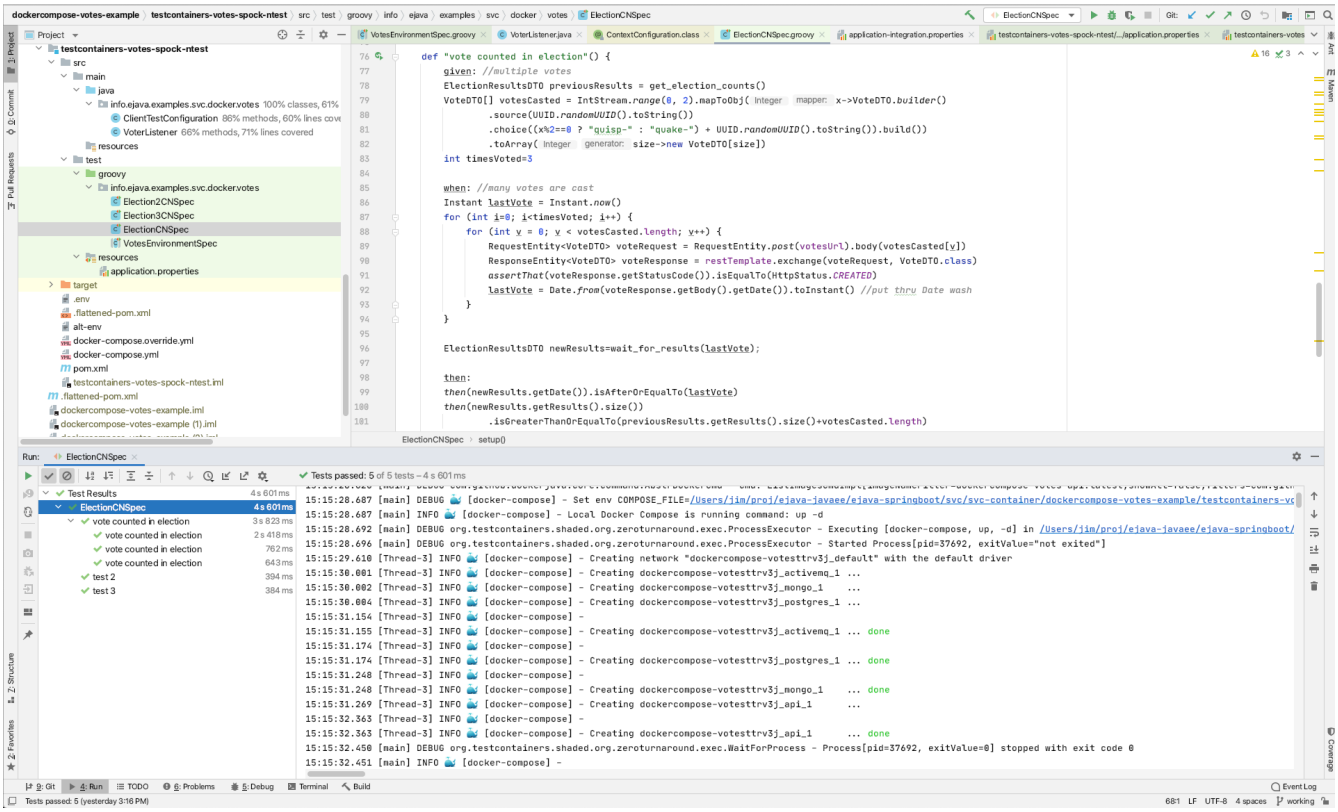

Figure 6. Example IDE Test Execution with Testcontainers and Docker Compose

4.1. Source Tree

The following shows the structure of the example integration module. We have already been working with the Docker Compose files at the root level in the previous section. Those files can be placed within the `src` directories if not being used interactively for developer commands — to keep the root less polluted.

This is an integration test-only module, so there will be no application code in the `src/main` tree. I took the opportunity to place common network helper code in the `src/main` tree to mimic what might be packaged up into test module support JAR if we need this type of setup in multiple test modules.

The `src/test` tree contains files that are specific to the specific integration tests performed. I also went a step further and factored out a base test class and then copied the initial `ElectionCNSpec` test case to demonstrate reuse within a test case and shutdown/startup in between test cases.

Example Integration Unit Test Tree

```
|-- alt-env
```

```
|-- docker-compose.override.yml
|-- docker-compose.yml
|-- pom.xml
`-- src
   |-- main
   |   |-- java
   |   |   `-- info
...
   |   |           `-- votes
   |   |               |-- ClientTestConfiguration.java
   |   |               `-- VoterListener.java
   |   `-- resources
   `-- test
       |-- groovy
       |   `-- info
...
       |           `-- votes
       |               |-- VotesEnvironmentSpec.groovy
       |               |-- ElectionCNSpec.groovy
       |               |-- Election2CNSpec.groovy
       |               `-- Election3CNSpec.groovy
       `-- resources
           `-- application.properties
```

4.2. @SpringBootConfiguration

Configuration is being supplied to the tests by the `ClientTestConfiguration` class. The following shows some traditional `@Value` property value injections that could have also been supplied through a `@ConfigurationProperties` class. We want these values set to the assigned host information at runtime.

Traditional @SpringBootConfiguration

```
@SpringBootConfiguration()
@EnableAutoConfiguration
public class ClientTestConfiguration {
    @Value("${it.server.host:localhost}")
    private String host; ①
    @Value("${it.server.port:9090}")
    private int port; ②
...
```

① value is commonly `localhost`

② value is **dynamically** generated at runtime

4.3. Traditional @Bean Factories

The configuration class supplies a traditional set of `@Bean` factories with base URLs to the two

services. We want the later two URIs injected into our test. So far so good.

@Bean Factories

```
//public class ClientTestConfiguration { ...
@Bean
public URI baseUrl() {
    return UriComponentsBuilder.newInstance()
            .scheme("http").host(host).port(port).build().toUri();
}
@Bean
public URI votesUrl(URI baseUrl) {
    return UriComponentsBuilder.fromUri(baseUrl).path("api/votes")
            .build().toUri();
}
@Bean
public URI electionsUrl(URI baseUrl) {
    return UriComponentsBuilder.fromUri(baseUrl).path("api/elections")
            .build().toUri();
}
@Bean
public RestTemplate anonymousUser(RestTemplateBuilder builder) {
    RestTemplate restTemplate = builder.build();
    return restTemplate;
}
```

4.4. DockerComposeContainer

In order to obtain the assigned port information required by the URI injections, we first need to define our network container. The following shows a set of static helper methods that locates the Docker Compose file, instantiates the Docker Compose network container, assigns it a project name, and exposes container port `8080` from the API to a random available host port.

During network startup, Testcontainers will also wait for network activity on that port before returning control back to the test.

Creating the DockerComposeContainer

```
    public static File composeFile() {
        File composeFile = new File("./docker-compose.yml"); ①
        Assertions.assertThat(composeFile.exists()).isTrue();
        return composeFile;
    }

    public static DockerComposeContainer testEnvironment() {
        DockerComposeContainer env =
            new DockerComposeContainer("dockercompose-votes", composeFile())
                .withExposedService("api", 8080);
        return env;
```

```
    }
```

① Testcontainers will fail if Docker Compose file reference does not include an explicit parent directory (i.e., `./` is required)

Mapped Volumes may require additional settings

Testcontainers automatically detects whether the test is being launched from within or outside a Docker image (outside in this example). Some additional tweaks to the Docker Compose file are required only if disk volumes are being mapped. These tweaks are called forming a "wormhole" to have Docker spawn sibling containers and share resources. We are not using volumes and will not be covering the wormhole pattern here.

4.5. @SpringBootTest

The following shows an example `@SpringBootTest` declaration. The test is a pure client to the server-side and contains no service web tier. The configuration was primarily what I just showed you — being primarily based on the URIs.

The test uses an optional `@Stepwise` orchestration for tests in case there is an issue sharing the dirty service state that a known sequence can solve. This should also allow for a lengthy end-to-end scenario to be broken into ordered steps along test method boundaries.

Here is also where the URIs are being injected — but we need our network started before we can derive the ports for the URIs.

Example @SpringBootTest Declaration

```
@SpringBootTest(classes = [ClientTestConfiguration.class],
    webEnvironment = SpringBootTest.WebEnvironment.NONE)
@Stepwise
@Slf4j
@DirtiesContext
abstract class VotesEnvironmentSpec extends Specification {
    @Autowired
    protected RestTemplate restTemplate
    @Autowired
    protected URI votesUrl
    @Autowired
    protected URI electionsUrl

    def setup() {
        log.info("votesUrl={}", votesUrl) ①
        log.info("electionsUrl={}", electionsUrl)
    }
```

① URI injections — based on dynamic values — must occur before tests

4.6. Spock Network Management

Testcontainers management within Spock is more manual that with JUnit — mostly because Spock does not provide first-class framework support for static variables. No problem, we can find many ways to get this to work. The following shows the network container being placed in a `@Shared` property and started/stopped at the Spec level.

Set System Property in setupSpec()

```
@Shared ①
protected DockerComposeContainer env = ClientTestConfiguration.testEnvironment()

def setupSpec() {
    env.start() ②
}
def cleanupSpec() {
    env.stop() ③
}
```

① network is instantiated and stored in a `@Shared` variable accessible to all tests

② test case initialization starts the network

③ test case cleanup stops the network

But what about the dynamically assigned port numbers? We have three ways that can be used to resolve them.

4.7. Set System Property

During `setupSpec`, we can set System Properties to be used when forming the Spring Context for each test.

Set System Property in setupSpec() Option

```
def setupSpec() {
    env.start() ①
    System.setProperty("it.server.port", ""+env.getServicePort("api", 8080));
}
```

① after starting network, dynamically assigned port number obtained and set as a System Property for individual test cases

In hindsight, this looks like a very concise way to go. However, there were two other options available that might be of interest in case they solve other issues that arise elsewhere.

4.8. ApplicationContextInitializer

A more verbose and likely legacy Spring way of adding the port values is through a Spring `ApplicationContextInitializer` that can get added to the Spring application context using the

`@ContextConfiguration` annotation and some static constructs within the Spock test.

The network container gets initialized — like usual — except a reference to the container gets assigned to a static variable where the running container can be inspected for dynamic values during an `initialize()` callback.

ApplicationContextInitializer Option

```
...
import org.springframework.context.ApplicationContextInitializer
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.boot.test.util.TestPropertyValues;
...
@SpringBootTest(...
@ContextConfiguration(initializers = Initializer.class) ④
...
abstract class VotesEnvironmentSpec extends Specification {
    private static DockerComposeContainer staticEnv ①
    static class Initializer ③
        implements ApplicationContextInitializer<ConfigurableApplicationContext> {
        @Override
        void initialize(ConfigurableApplicationContext ctx) {
            TestPropertyValues values = TestPropertyValues.of(
                    "it.server.port=" + staticEnv.getServicePort("api", 8080))
            values.applyTo(ctx)
        }
    }

@Shared
protected DockerComposeContainer env = ClientTestConfiguration.testEnvironment()
def setupSpec() {
    staticEnv = env ②
    env.start()
   ...
```

① static variable declared to hold reference to singleton network

② `@Shared` network assigned to static variable

③ `Initializer` class defined to obtain network information from network and inject into test properties

④ `Initializer` class registered with Spring application context

4.9. DynamicPropertySource

A similar, but more concise way to leverage the callback approach is to leverage the newer Spring `@DynamicPropertySource` construct. At a high level — nothing has changed with the management of the network container. Spring simply eliminated the need to create the boilerplate class, etc. when supplying properties dynamically.

DynamicPropertySource Option

```
import org.springframework.test.context.DynamicPropertyRegistry
import org.springframework.test.context.DynamicPropertySource
...
private static DockerComposeContainer staticEnv ①
@DynamicPropertySource ③
static void properties(DynamicPropertyRegistry registry) {
    registry.add("it.server.port", ()->staticEnv.getServicePort("api", 8080));
}

@Shared
protected DockerComposeContainer env = ClientTestConfiguration.testEnvironment()
def setupSpec() {
    staticEnv = env ②
    env.start()
}
```

① static variable declared to hold reference to singleton network

② `@Shared` network assigned to static variable

③ `@DynamicPropertySource` defined on a **static** method to obtain network information from network and inject into test properties

4.10. Resulting Test Initialization Output

The following shows an example startup prior to executing the first test. You will see TestContainers start Docker Compose in the background and then wait close to ~12 seconds for the API port 8080 to become active.

Maven/Spock Test Startup

```
13:52:28.467 DEBUG  [docker-compose] - Set env COMPOSE_FILE=
  .../dockercompose-votes-example/testcontainers-votes-spock-ntest/./docker-
compose.yml
13:52:28.467 INFO  [docker-compose] - Local Docker Compose is running command: up -d
13:52:28.472 DEBUG org.testcontainers.shaded.org.zeroturnaround.exec.ProcessExecutor -
  Executing [docker-compose, up, -d]
...
13:52:28.996 INFO  [docker-compose] - Creating network "dockercompose-
votesdkakfi_default" with the default driver
INFO  [docker-compose] - Creating dockercompose-votesdkakfi_mongo_1 ...
INFO  [docker-compose] - Creating dockercompose-votesdkakfi_postgres_1 ...
INFO  [docker-compose] - Creating dockercompose-votesdkakfi_activemq_1 ...
INFO  [docker-compose] - Creating dockercompose-votesdkakfi_activemq_1 ... done
INFO  [docker-compose] - Creating dockercompose-votesdkakfi_mongo_1    ... done
INFO  [docker-compose] - Creating dockercompose-votesdkakfi_postgres_1 ... done
INFO  [docker-compose] - Creating dockercompose-votesdkakfi_api_1      ...
INFO  [docker-compose] - Creating dockercompose-votesdkakfi_api_1      ... done
13:52:30.803 DEBUG org.testcontainers.shaded.org.zeroturnaround.exec.WaitForProcess -
```

```
Process...
13:52:30.804 INFO  [docker-compose] - Docker Compose has finished running
... (waiting for containers to start)
13:52:45.100 DEBUG
org.springframework.test.context.support.DependencyInjectionTestExecutionListener -
 :: Spring Boot ::        (v2.3.2.RELEASE)
...
---

At this point, we are ready to use normal `restTemplate` or `WebClient` calls to
test our interface to the overall application.

---
13:52:48.031 VotesEnvironmentSpec votesUrl=http://localhost:32838/api/votes
13:52:48.032 VotesEnvironmentSpec electionsUrl=http://localhost:32838/api/elections
```

Chapter 5. Additional Waiting

Testcontainers will wait for the exposed port to become active. We can add additional wait tests to be sure the network is in a ready state to be tested. The following adds a check for the two URLs to return a successful response.

Example Wait For URL

```
def setup() {
    /**
     * wait for various events relative to our containers
     */
    env.waitingFor("api", Wait.forHttp(votesUrl.toString())) ①
    env.waitingFor("api", Wait.forHttp(electionsUrl.toString()))
```

① test setup holding up start of test for two API URL calls to be successful

Chapter 6. Executing Commands

If useful, we can also invoke commands within the running network containers at points in the test. The following shows a CLI command invoked against each database container that will output the current state at this point in the test.

Example Execute Commands

```
/**
 * run sample commands directly against containers
 */
ContainerState mongo = (ContainerState) env.getContainerByServiceName("mongo_1")
    .orElseThrow()
ExecResult result = mongo.execInContainer("mongo", ①
        "-u", "admin", "-p", "secret", "--authenticationDatabase", "admin",
        "--eval", "db.getSiblingDB('votes_db').votes.find()");
log.info("voter votes = {}", result.getStdout()) ②

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

① executing shell command inside running container in network

② obtaining results in stdout

6.1. Example Command Output

The following shows the output of the standard output obtained from the two containers after running the CLI query commands.

Example Command Output

```
14:32:15.075 ElectionCNSpec#setup:67 voter votes = MongoDB shell version v4.4.0
connecting to:
mongodb://127.0.0.1:27017/?authSource=admin&compressors=disabled&gssapiServiceName=mon
godb
Implicit session: session { "id" : UUID("a824b7b8-634a-426b-8d21-24c5680864f6") }
MongoDB server version: 4.4.0

{ "_id" : ObjectId("5f382a2c62cb0d4f36d96cfa"),
  "date" : ISODate("2020-08-15T18:32:12.706Z"),
  "source" : "684c586f...",
  "choice" : "quisp-82...",
  "_class" : "info.ejava.examples.svc.docker.votes.dto.VoteDTO" }
{ "_id" : ObjectId("5f382a2d62cb0d4f36d96cfb"),
  "date" : ISODate("2020-08-15T18:32:13.511Z"),
```

```
  "source" : "df3a973a...",
  "choice" : "quake-5e...",
  "_class" : "info.ejava.examples.svc.docker.votes.dto.VoteDTO" }
...

14:32:15.263 main  INFO     i.e.e.svc.docker.votes.ElectionCNSpec#setup:73 election
votes =
           id            | choice      |          date           | source
-------------------------+-------------+-------------------------+------------
5f382a2c62cb0d4f36d96cfa | quisp-82... | 2020-08-15 18:32:12.706 | 684c586f...
5f382a2d62cb0d4f36d96cfb | quake-5e... | 2020-08-15 18:32:13.511 | df3a973a...
...
(6 rows)
```

Chapter 7. Client Connections

Although an interesting and potentially useful feature to be able to execute a random shell command against a running container under test — it can be very clumsy to interpret the output when there is another way. We can — instead — establish a resource client to any of the services we need additional state from.

The following will show adding resource client capabilities that were originally added to the API server. If necessary, we can use this low-level access to trigger specific test conditions or evaluate something performed.

7.1. Maven Dependencies

The following familiar Maven dependencies can be added to the pom.xml to add the resources necessary to establish a client connection to each of the three back-end resources.

Client Connection Maven Dependencies

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

7.2. Hard Coded Application Properties

We can simply add the following hard-coded resource properties to a property file since this is static information necessary to complete the connections.

Hard Coded Properties

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

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

```
spring.datasource.password=secret
```

However, we still will need the following properties added that consist of dynamically assigned values.

Dynamic Properties Needed

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

7.3. Dynamic URL Helper Methods

The following helper methods are used to form a valid URL String once the hostname and port number are known.

Dynamic URL Helper Methods

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

7.4. Adding Dynamic Properties

The hostname and port number(s) can be obtained from the running network and supplied to the Spring context using one of the three techniques shown earlier (`System.setProperty`, `ConfigurableApplicationContext`, or `DynamicPropertyRegistry`). The following shows the `DynamicPropertyRegistry` technique.

Adding Dynamic Properties

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

```
    registry.add("spring.datasource.url",()->jdbcUrl(
            env.getServiceHost("postgres", null),
            env.getServicePort("postgres", 5432)
        ));
}
```

① helper method called from `@DynamicPropertySource` callback in unit test

7.5. Adding JMS Listener

We can add a class to subscribe and listen to the `votes` topic by declaring a `@Component` with a method accepting a JMS `TextMessage` and annotated with `@JmsListener`. The following example just prints debug messages of the events and counts the number of messages received.

Example JMS Listener

```
...
import org.springframework.jms.annotation.JmsListener;
import jakarta.jms.TextMessage;

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

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

We also need to add the JMS Listener `@Component` to the Spring application context using the `@SpringBootTest.classes` property

Adding JMS Listener to Application Context

```
@SpringBootTest(classes = [ClientTestConfiguration.class, VoterListener.class],
```

7.6. Injecting Resource Clients

The following shows injections for the resource clients. I have already showed the details behind the `VoterLister`. That is ultimately supported by the JMS AutoConfiguration and the `spring.activemq.broker-url` property.

The `MongoClient` and `JdbcClient` are directly provided by the Mongo and JPA AutoConfiguration and the `spring.data.mongodb.uri` and `spring.datasource.url` properties.

Inject Resource Clients

```
@Autowired
protected MongoClient mongoClient
@Autowired
protected VoterListener listener
@Autowired
protected JdbcTemplate jdbcTemplate
```

7.7. Resource Client Calls

The following shows an example set of calls that simply obtains document/message/row counts. However, with that capability demonstrated — much more is easily possible.

Example Resource Client Calls

```
/**
 * connect directly to explosed port# of images to obtain sample status
 */
log.info("mongo client vote count={}",
    mongoClient.getDatabase("votes_db").getCollection("votes").countDocuments())
log.info("activemq msg={}", listener.getMsgCount().get())
log.info("postgres client vote count={}",
    jdbcTemplate.queryForObject("select count (*) from vote", Long.class))
```

The following shows the output from the example resource client calls

Example Resource Call Output

```
ElectionCNSpec#setup:54 mongo client vote count=18
ElectionCNSpec#setup:55 activemq msg=18
ElectionCNSpec#setup:57 postgres client vote count=18
```

Chapter 8. Test Hierarchy

Much of what I have covered can easily go into a helper class or test base class and potentially be part of a test dependency library if the amount of integration testing significantly increases and must be broken out.

8.1. Network Helper Class

The following summarizes the helper class that can encapsulate the integration between Testcontainers and Docker Compose. This class is not specific to running in any one test framework.

ClientTestConfiguration Helper Class

```
public class ClientTestConfiguration { ①
    public static File composeFile() { ...
    public static DockerComposeContainer testEnvironment() { ...
    public static void initProperties(DynamicPropertyRegistry registry,
DockerComposeContainer env) { ...
    public static void initProperties(DockerComposeContainer env) { ...
    public static void initProperties(ConfigurableApplicationContext ctx,
DockerComposeContainer env) { ...
    public static String mongoUrl(String host, int port) { ...
    public static String jmsUrl(String host, int port) { ...
    public static String jdbcUrl(String host, int port) { ...
```

① Helper class can encapsulate details of network without ties to actual test framework

8.2. Integration Spec Base Class

The following summarizes the base class that encapsulates starting/stopping the network and any helper methods used by tests. This class is specific to operating tests within Spock.

VotesEnvironmentSpec Test Base Class

```
abstract class VotesEnvironmentSpec extends Specification { ①
    def setupSpec() {
        configureEnv(env)
        ...
    void configureEnv(DockerComposeContainer env) {} ②
    def cleanupSpec() { ...
    def setup() { ...
    public ElectionResultsDTO wait_for_results(Instant resultTime) { ...
    public ElectionResultsDTO get_election_counts() { ...
```

① test base class integrates helper methods in with test framework

② extra environment setup call added to allow subclass to configure network before started

8.3. Specialized Integration Test Classes

The specific test cases can inherit all the setup and focus on their individual tests. Note that the example I provided uses the same running network within a test case class (i.e., all test methods in a test class share the same network state). Separate test cases use fresh network state (i.e., the network is shutdown, removed, and restarted between test classes).

Example Test Case

```
class ElectionCNSpec extends VotesEnvironmentSpec { ①
    @Override
    def void configureEnv(DockerComposeContainer dc) { ...
    def cleanup() { ...
    def setup() { ...
    def "vote counted in election"() { ...
    def "test 2"() { ...
    def "test 3"() { ...
```

① concrete test cases provide specific tests and extra configuration, setup, and cleanup specific to the tests

Example Test Case 2

```
class Election2CNSpec extends VotesEnvironmentSpec {
    def "vote counted in election"() { ...
    def "test 2"() { ...
    def "test 3"() { ...
```

8.4. Test Execution Results

The following image shows the completion results of the integration tests. Once thing to note with Spock is that it only seems to attribute time to a test setup/execution/cleanup and not to the test case's setup and cleanup. Active MQ is very slow to shutdown and there is easily 10-20 seconds in between test cases that is not depicted in the timing results.

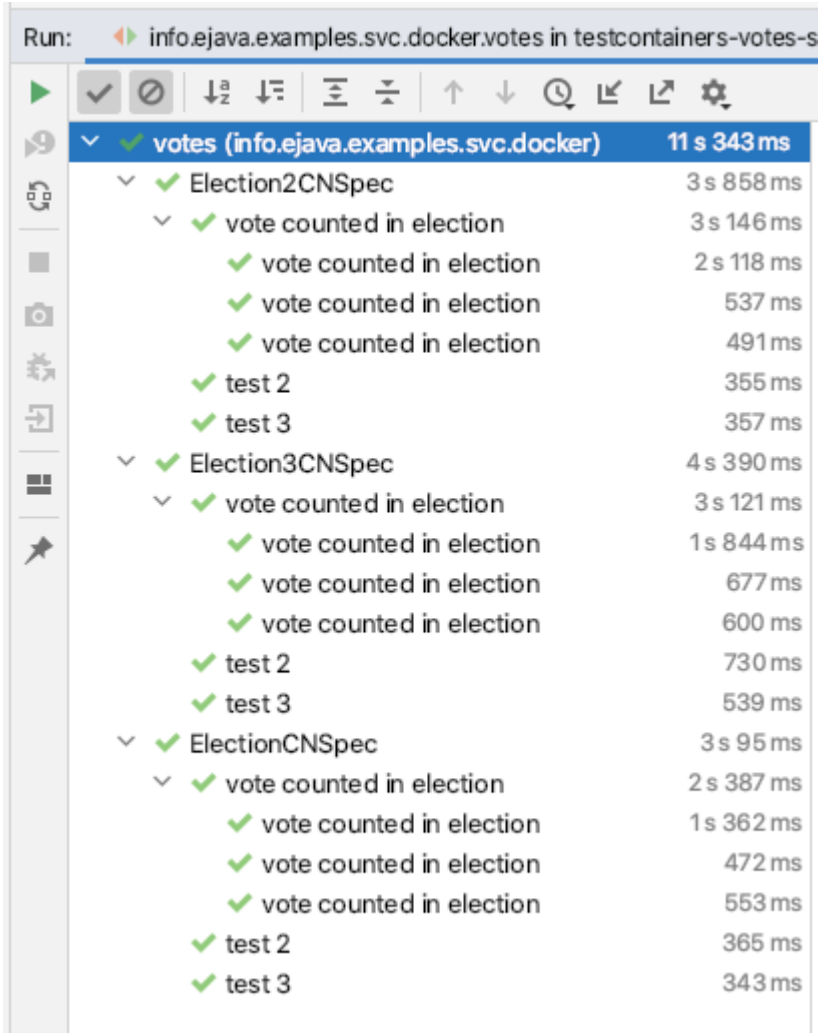

Figure 7. Test Execution Results

Chapter 9. Summary

This lecture covered a summary of capability for Docker Compose and Testcontainers integrated into Spock to implement integrated unit tests. The net result is a seamless test environment that can verify that a network of components — further tested in unit tests — integrate together to successfully satisfy one or more end-to-end scenarios. For example, it was not until integration testing that I realized my JMS communications was using a queue versus a topic.

In this module, we learned:

- to identify the capability of Docker Compose to define and implement a network of virtualized services running in Docker
- to identify the capability of Testcontainers to seamlessly integrate Docker and Docker Compose into unit test frameworks including Spock
- to author end-to-end, unit integration tests using Spock, Testcontainers, Docker Compose, and Docker
- to implement inspections of running Docker images
- to implement inspects of virtualized services during tests
- to instantiate virtualized services for use in development
- to implement a hierarchy of test classes to promote reuse