Docker Compose Integration Testing

jim stafford

Fall 2022 v2021-08-27: Built: 2022-12-07 06:19 EST

Table of Contents

Chapter 1. Introduction

In the last lecture we looked at a set of Voting Services and configured integration unit testing using simulated or other in-memory resources to implement an end-to-end integration thread in a single process.

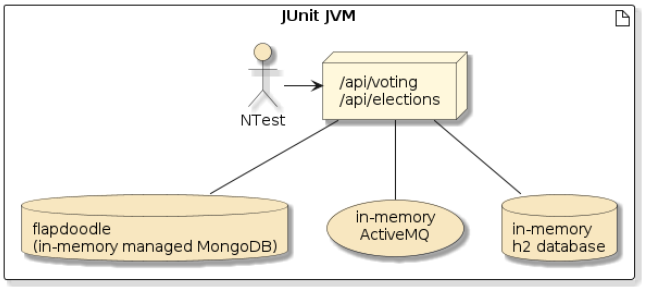

Figure 1. Integration Unit Test with In-Memory/Local Resources

But what if we wanted or needed to use real resources?

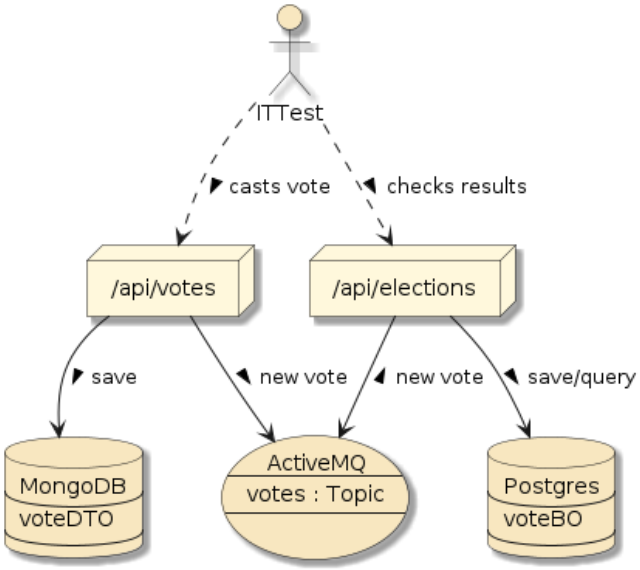

Figure 2. How Can We Test with Real Resources

What if we needed to test with a real or specific version of MongoDB, ActiveMQ, Postgres, or some other resource? What if some of those other resources were a supporting microservice?

We could implement an integration test — but how can we automate it?

In this lecture, we will explore using Docker for each of the integrated resources (services) and leverage Docker Compose to manage our individual services and the network.

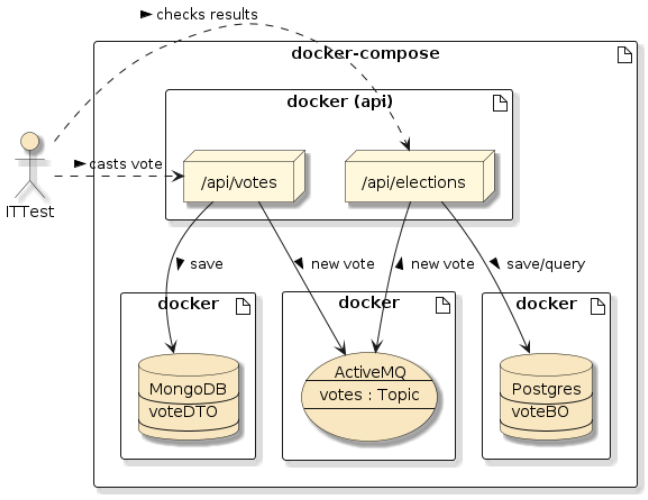

Figure 3. Integration Test with Docker and Docker Compose

1.1. Goals

You will learn:

- how to implement a network of services for development and testing using Docker Compose
- how to implement an integration test between real instances running in a virtualized environment
- how to interact with the running instances during testing

1.2. Objectives

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

1. create a Docker Compose file that defines a network of services and their dependencies
2. execute ad-hoc commands inside running images
3. integrate Docker Compose into a Maven integration test phase
4. author an integration test that uses real resource instances with dynamically assigned ports

Chapter 2. Integration Testing with Real Resources

We are in a situation were, we need to run integration tests against real components. These "real" components can be virtualized, but they primarily need to contain a specific feature of a specific version we are taking advantage of.

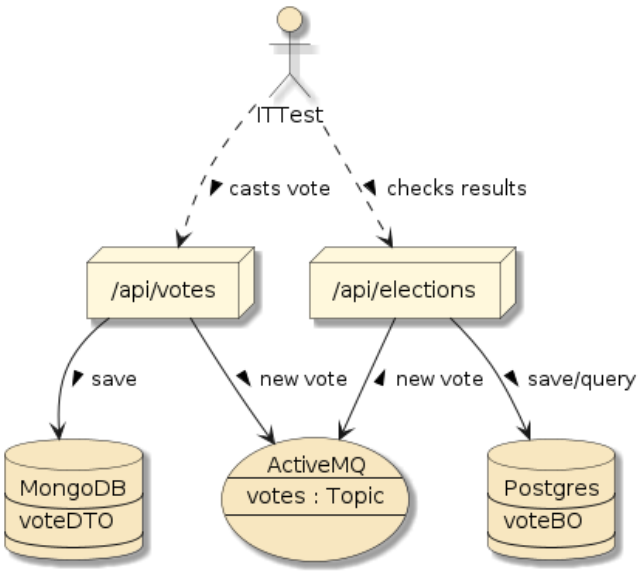

Figure 4. Need to Integrate with Specific Real Services

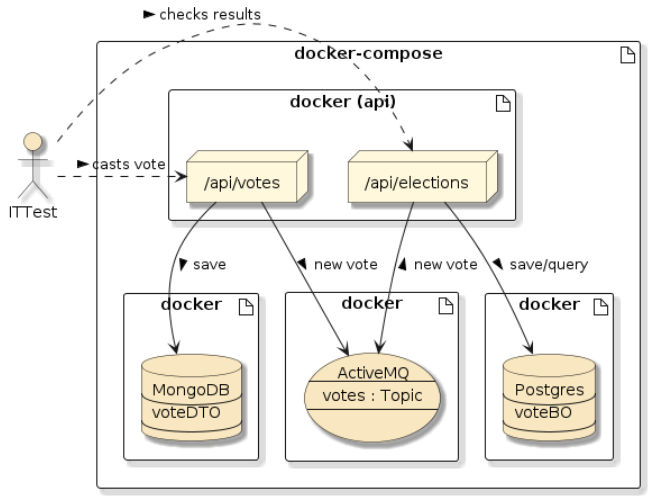

Figure 5. Virtualize Services with Docker

My example uses generic back-end resources as examples of what we need to integrate with. However, in the age of microservices — these examples could easily be lower-level applications offering necessary services for our client application to properly operate.

We need access to these resources in the development environment but would soon need them during automated integration tests running regression tests in the CI server as well.

Lets look to Docker for a solution ...

2.1. Managing Images

You know from our initial Docker lectures that we can easily download the images and run them individually (given some instructions) with the `docker run` command. Knowing that — we could try doing the following and almost get it to work.

Manually Starting Images

```
$ docker run --rm -p 27017:27017 \
-e MONGO_INITDB_ROOT_USERNAME=admin \
-e MONGO_INITDB_ROOT_PASSWORD=secret mongo:4.4.0-bionic

$ docker run --rm -p 5432:5432 \
-e POSTGRES_PASSWORD=secret postgres:12.3-alpine
```

```
$ docker run --rm -p 61616:61616 -p 8161:8161 \
rmohr/activemq:5.15.9

$ docker run --rm -p 9080:8080 \
-e
MONGODB_URI='mongodb://admin:secret@host.docker.internal:27017/votes_db?authSource=adm
in' \
-e DATABASE_URL='postgres://postgres:secret@host.docker.internal:5432/postgres' \
-e spring.profiles.active=integration dockercompose-votes-api:latest
```

However, this begins to get complicated when:

- we start integrating the API image with the individual resources through networking
- we want to make the test easily repeatable
- we want multiple instances of the test running concurrently on the same machine without interference with one another

Lets not mess with manual Docker commands for too long! There are better ways to do this with Docker Compose — covered earlier. I will review some of the aspects.

Chapter 3. Docker Compose Configuration File

The Docker Compose (configuration) file is based on YAML — which uses a concise way to express information based on indentation and firm symbol rules. Assuming we have a simple network of four (4) nodes, we can limit our definition to a `version` and `services`.

docker-compose.yml Shell

```
version: '3.8'
services:
  mongo:
    ...
  postgres:
    ...
  activemq:
    ...
  api:
    ...
```

Refer to the Compose File Reference for more details.

3.1. mongo Service Definition

The `mongo` service defines our instance of MongoDB.

mongo Service Definition

```
  mongo:
    image: mongo:4.4.0-bionic
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: secret
#    ports:
#      - "27017:27017"
```

3.2. postgres Service Definition

The `postgres` service defines our instance of Postgres.

postgres Service Definition

```
  postgres:
    image: postgres:12.3-alpine
#    ports:
#      - "5432:5432"
    environment:
```

```
      POSTGRES_PASSWORD: secret
```

3.3. activemq Service Definition

The `activemq` service defines our instance of ActiveMQ.

activemq Service Definition

```
  activemq:
    image: rmohr/activemq:5.15.9
#    ports:
#      - "61616:61616"
#      - "8161:8161"
```

- port 61616 is used for JMS communication
- port 8161 is an HTTP server that can be used for HTML status

3.4. api Service Definition

The `api` service defines our API server with the Votes and Elections Services. This service will become a client of the other three services.

api Service Definition

```
  api:
    build:
      context: ../dockercompose-votes-svc
      dockerfile: Dockerfile
    image: dockercompose-votes-api:latest
    ports:
      - "${API_PORT}:8080"
    depends_on:
      - mongo
      - postgres
      - activemq
    environment:
      - spring.profiles.active=integration
      - MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin
      - DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
```

3.5. Compose Override Files

I left off port definitions from the primary file on purpose. That will become more evident in the Testcontainers topic in the next lecture when we need dynamically assigned port numbers. However, for purposes here we need well-known port numbers and can do so easily with an additional configuration file — `docker-compose.override.yml`.

Docker Compose files can be layered from base (shown above) to specialized. The following example shows the previous definitions being extended to include mapped host port# mappings. We might add this override in the development environment to make it easy to access the service ports on the host's local network.

Example Compose Override File

```
version: '3.8'
services:
  mongo:
    ports:
      - "27017:27017"
  postgres:
    ports:
      - "5432:5432"
  activemq:
    ports:
      - "61616:61616"
      - "8161:8161"
```

When started — notice how the container port# is mapped according to how the override file has specified.

Port Mappings with Compose Override File Used

```
$ docker ps
IMAGE                            PORTS
dockercompose-votes-api:latest   0.0.0.0:9090->8080/tcp
postgres:12.3-alpine             0.0.0.0:5432->5432/tcp, 0.0.0.0:32812->5432/tcp
mongo:4.4.0-bionic               0.0.0.0:27017->27017/tcp, 0.0.0.0:32813->27017/tcp
rmohr/activemq:5.15.9            1883/tcp, 5672/tcp, 0.0.0.0:8161->8161/tcp, 61613-
61614/tcp, 0.0.0.0:61616->61616/tcp ①
```

① notice that only the ports we mapped are exposed

Override Limitations May Cause Compose File Refactoring

There is a limit to what you can override versus augment. Single values can replace single values. However, lists of values can only contribute to a larger list. That means we cannot create a base file with ports mapped and then a build system override with the port mappings taken away.

Chapter 4. Test Drive

Lets test out our services before demonstrating a few more commands. Everything is up and running and only the API port is exposed to the local host network using port# 9090.

Running Network Port Mapping

```
$ docker ps
IMAGE                            PORTS
dockercompose-votes-api:latest   0.0.0.0:9090->8080/tcp ①
postgres:12.3-alpine             5432/tcp
mongo:4.4.0-bionic               27017/tcp
rmohr/activemq:5.15.9            1883/tcp, 5672/tcp, 8161/tcp, 61613-61614/tcp,
61616/tcp
```

① only the API has its container port# (`8080`) mapped to a host port# (`9090`)

4.1. Clean Starting State

We start off with nothing in the Vote or Election databases.

Clean Starting State

```
$ curl http://localhost:9090/api/votes/total
0

$ curl http://localhost:9090/api/elections/counts
{
  "date" : "1970-01-01T00:00:00Z",
  "results" : [ ]
}
```

4.2. Cast Two Votes

We can then cast votes for different choices and have them added to MongoDB and have a JMS message published.

Cast Two Votes

```
$ curl -X POST http://localhost:9090/api/votes -H "Content-Type: application/json" -d
'{"source":"jim","choice":"quisp"}'
{
  "id" : "5f31eed580cfe474aeaa1536",
  "date" : "2020-08-11T01:05:25.168505Z",
  "source" : "jim",
  "choice" : "quisp"
}
$ curl -X POST http://localhost:9090/api/votes -H "Content-Type: application/json" -d
```

```
'{"source":"jim","choice":"quake"}'
{
  "id" : "5f31eee080cfe474aeaa1537",
  "date" : "2020-08-11T01:05:36.374043Z",
  "source" : "jim",
  "choice" : "quake"
}
```

4.3. Observe Updated State

At this point we can locate some election results in Postgres using API calls.

Updated State

```
$ curl http://localhost:9090/api/elections/counts
{
  "date" : "2020-08-11T01:05:36.374Z",
  "results" : [ {
    "choice" : "quake",
    "votes" : 1
  }, {
    "choice" : "quisp",
    "votes" : 1
  } ]
}
```

Chapter 5. Inspect Images

This is a part that I think is really useful and easy. Docker Compose provides an easy interface for running commands within the images.

5.1. Exec Mongo CLI

In the following example, I am running the `mongo` command line interface (CLI) command against the running `mongo` service and passing in credentials as command line arguments. Once inside, I can locate our `votes_db` database, `votes` collection, and two documents that represent the votes I was able to cast earlier.

Exec Command Against Running mongo Image

```
$ docker-compose exec mongo mongo -u admin -p secret --authenticationDatabase admin ①
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("1fbd09ab-73e3-459f-b5f5-5d23903f672c") }
MongoDB server version: 4.4.0

> show dbs ②
admin     0.000GB
config    0.000GB
local     0.000GB
votes_db  0.000GB
> use votes_db
switched to db votes_db
> show collections
votes
> db.votes.find({},{"choice":1}) ③
{ "_id" : ObjectId("5f31eed580cfe474aeaa1536"), "choice" : "quisp" }
{ "_id" : ObjectId("5f31eee080cfe474aeaa1537"), "choice" : "quake" }
> exit ④
bye
```

① running `mongo` CLI command inside running `mongo` image with command line args expressing credentials

② running CLI commands to inspect database

③ listing documents in the votes database

④ exiting CLI and returning to host shell

5.2. Exec Postgres CLI

In the following example, I am running the `psql` CLI command against the running `postgres` service and passing in credentials as command line arguments. Once inside, I can locate our Flyway

migration and VOTE table and list some of the votes that are in the election.

Exec Command Against Running postgres Image

```
$ docker-compose exec postgres psql -U postgres ①
psql (12.3)
Type "help" for help.

postgres=# \d+ ②
                                    List of relations
 Schema |         Name          | Type  |  Owner   |    Size    |         Description
--------+-----------------------+-------+----------+------------+
-----------------------------
 public | flyway_schema_history | table | postgres | 16 kB      |
 public | vote                  | table | postgres | 8192 bytes | countable votes for
election
(2 rows)

postgres=# select * from vote; ③
            id            | choice |          date           | source
--------------------------+--------+-------------------------+--------
 5f31eed580cfe474aeaa1536 | quisp  | 2020-08-11 01:05:25.168 | jim
 5f31eee080cfe474aeaa1537 | quake  | 2020-08-11 01:05:36.374 | jim
(2 rows)

postgres=# \q ④
```

① running `psql` CLI command inside running `postgres` image with command line args expressing credentials

② running CLI commands to inspect database

③ listing table rows in the vote table

④ exiting CLI and returning to host shell

5.3. Exec Impact

With the capability to exec a command inside the running containers, we can gain access to a significant amount of state of our application and databases without having to install any software beyond Docker.

Chapter 6. Integration Test Setup

At this point we should understand what Docker Compose is and how to configure it for use with our specific integration test. I now want to demonstrate it being used in an automated "integration test" where it will get executed as part of the Maven integration-test phases.

6.1. Integration Properties

We will be launching our API image with the following Docker environment expressed in the Docker Compose file.

API Docker Environment

```
environment:
  - spring.profiles.active=integration
  - MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin
  - DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
```

That will get digested by the run_env.sh script to produce the following.

Spring Boot Properties

```
--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/votes_db?authSource=admin
```

That will be integrated with the following properties from the `integration` profile.

application-integration.properties

```
#activemq
spring.activemq.broker-url=tcp://activemq:61616

#rdbms
spring.jpa.show-sql=true
spring.jpa.generate-ddl=false
spring.jpa.hibernate.ddl-auto=validate
spring.flyway.enabled=true
```

I have chosen to hard-code the integration URL for ActiveMQ into the properties file since we won't be passing in an ActiveMQ URL in production. The MongoDB and Postgres properties will originate from environment variables versus hard coding them into the integration properties file to better match the production environment and further test the run_env.sh launch script.

6.2. Maven Build Helper Plugin

We will want a random, not in use port# assigned when we run the integration tests so that multiple instances of the test can be run concurrently on the same build server without colliding. We can leverage the `build-helper-maven-plugin` to identify a port# and have it assigned the value to a Maven property. I am assigning it to a `docker.http.port` property that I made up.

Generate Random Port# for Integration Test

```
<!-- assigns a random port# to property server.http.port -->
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>build-helper-maven-plugin</artifactId>
    <executions>
        <execution>
            <id>reserve-network-port</id>
            <goals>
                <goal>reserve-network-port</goal>
            </goals>
            <phase>pre-integration-test</phase>
            <configuration>
                <portNames>
                    <portName>docker.http.port</portName> ①
                </portNames>
            </configuration>
        </execution>
    </executions>
</plugin>
```

① a dynamically obtained network port# is assigned to the `docker.http.port` Maven property

The following is an example output of the `build-helper-maven-plugin` during the build.

Example Maven Build Helper Plugin Output

```
[INFO] --- build-helper-maven-plugin:3.1.0:reserve-network-port (reserve-network-port)
@ dockercompose-votes-it ---
[INFO] Reserved port 60616 for docker.http.port
```

6.3. Maven Docker Compose Plugin

After generating a random port#, we can start our Docker Compose network. I am using the https://github.com/br4chu/docker-compose-maven-plugin `docker-compose-maven-plugin`] to perform that role. It automatically hooks into the `pre-integration-test` phase to issue the `up` command and the `post-integration-test` phase to issue the `down` command when we configure it the following way. It also allows us to name and pass variables into the Docker Compose file.

```
<plugin>
```

```
    <groupId>io.brachu</groupId>
    <artifactId>docker-compose-maven-plugin</artifactId>
    <configuration>
        <projectName>${project.artifactId}</projectName>
        <file>${project.basedir}/docker-compose.yml</file>
        <env>
            <API_PORT>${docker.http.port}</API_PORT> ①
        </env>
    </configuration>
    <executions>
        <execution>
            <goals>
                <goal>up</goal>
                <goal>down</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

① dynamically obtained network port# is assigned to Docker Compose file's `API_PORT` variable, which controls the port mapping of the API server

6.4. Maven Docker Compose Plugin Output

The following shows example plugin output during the `pre-integration-test` phase that is starting the services prior to running the tests.

Example Maven Docker Compose Plugin pre-integration-test Output

```
[INFO] --- docker-compose-maven-plugin:0.8.0:up (default) @ dockercompose-votes-it ---
Creating network "dockercompose-votes-it_default" with the default driver
...
Creating dockercompose-votes-it_mongo_1    ... done
Creating dockercompose-votes-it_api_1      ... done
```

The following shows example plugin output during the `post-integration-test` phase that is shutting down the services after running the tests.

Example Maven Docker Compose Plugin post-integration-test Output

```
[INFO] --- docker-compose-maven-plugin:0.8.0:down (default) @ dockercompose-votes-it
---
Killing dockercompose-votes-it_api_1      ...
Killing dockercompose-votes-it_api_1      ... done
Killing dockercompose-votes-it_postgres_1 ... done
Removing dockercompose-votes-it_mongo_1    ... done
Removing dockercompose-votes-it_postgres_1 ... done
Removing network dockercompose-votes-it_default
```

6.5. Maven Failsafe Plugin

The following shows the configuration of the `maven-failsafe-plugin`. Generically, it runs in the `integration-test` phase, matches/runs the `IT` tests, and adds test classes to the classpath. More specific to Docker Compose — it accepts the dynamically assigned port# and passes it to JUnit using the `it.server.port` property.

Example Failsafe Plugin Configuration

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <executions>
        <execution>
            <id>integration-test</id>
            <goals>
                <goal>integration-test</goal>
            </goals>
            <configuration>
                <includes>
                    <include>**/*IT.java</include>
                </includes>
                <systemPropertyVariables>
                    <it.server.port>${docker.http.port}</it.server.port> ①
                </systemPropertyVariables>
                <additionalClasspathElements>
<additionalClasspathElement>${basedir}/target/classes</additionalClasspathElement>
                </additionalClasspathElements>
            </configuration>
        </execution>
    </executions>
</plugin>
```

① passing in generated `docker.http.port` value into `it.server.port` property

At this point, both Docker Compose and Failsafe/JUnit have been given the same dynamically assigned port#.

6.6. IT Test Client Configuration

The following shows the IT test configuration class that maps the `it.server.port` property to the baseUrl for the tests.

ClientTestConfiguration Mapping it.server.port to baseUrl

```
@SpringBootConfiguration()
@EnableAutoConfiguration       //needed to setup logging
public class ClientTestConfiguration {
    @Value("${it.server.host:localhost}")
    private String host;
```

```
    @Value("${it.server.port:9090}") ①
    private int port;

    @Bean
    public URI baseUrl() {
        return UriComponentsBuilder.newInstance()
                .scheme("http")
                .host(host)
                .port(port)
                .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;
    }
```

① API port# property injected through Failsafe plugin configuration

6.7. Example Failsafe Output

The following shows the Failsafe and JUnit output that runs during the `integration-test`.

Example Failsafe Output

```
[INFO] --- maven-failsafe-plugin:3.0.0-M4:integration-test (integration-test) @
dockercompose-votes-it ---
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running info.ejava.examples.svc.docker.votes.ElectionIT
...
...ElectionIT#init:46 votesUrl=http://localhost:60616/api/votes ①
...ElectionIT#init:47 electionsUrl=http://localhost:60616/api/elections
...
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 10.372 s - in
info.ejava.examples.svc.docker.votes.ElectionIT
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

① URLs with dynamic host port# assigned for API

6.8. IT Test Setup

The following shows the common IT test setup where the various URLs are being constructed around the injected.

IT Test Setup

```
@SpringBootTest(classes={ClientTestConfiguration.class},
    webEnvironment = SpringBootTest.WebEnvironment.NONE)
@Slf4j
public class ElectionIT {
    @Autowired
    private RestTemplate restTemplate;
    @Autowired
    private URI votesUrl;
    @Autowired
    private URI electionsUrl;
    private static Boolean serviceAvailable;

    @PostConstruct
    public void init() {
        log.info("votesUrl={}", votesUrl);
        log.info("electionsUrl={}", electionsUrl);
    }
```

6.9. Wait For Services Startup

We have at least one more job to do before our tests — we have to wait for the API server to finish starting up. We can add that logic to a @BeforeEach and remember the answer from the first attempt in all following attempts.

Example Wait For Services Startup

```
    @BeforeEach
    public void serverRunning() {
        List<URI> urls = new ArrayList<>(Arrays.asList(
                UriComponentsBuilder.fromUri(votesUrl).path("/total").build().toUri(),

UriComponentsBuilder.fromUri(electionsUrl).path("/counts").build().toUri()
        ));

        if (serviceAvailable!=null) { assumeTrue(serviceAvailable);}
        else {
            assumeTrue(() -> { ①
                for (int i=0; i<10; i++) {
                    try {
                        for (Iterator<URI> itr = urls.iterator(); itr.hasNext();) {
                            URI url = itr.next();
                            restTemplate.getForObject(url, String.class); ②
```

```
                        itr.remove(); ③
                    }
                    return serviceAvailable = true; ④
                } catch (Exception ex) {
                    //...
                }
            }
            return serviceAvailable=false;
        });
    }
}
```

① Assume.assumeTrue will not run the tests if evaluates false

② checking for a non-exception result

③ removing criteria once satisfied

④ evaluate true if all criteria satisfied

At this point our tests are the same as most other Web API test where we invoke the server using HTTP calls using the assembled URLs.

Chapter 7. Summary

In this module we learned:

- to create a Docker Compose file that defines a network of services and their dependencies
- to integrate Docker Compose into a Maven integration test phase
- to implement an integration test that uses dynamically assigned ports
- execute ad-hoc commands inside running images