Integration Unit Testing

jim stafford

Fall 2022 v2020-08-11: Built: 2022-12-07 06:19 EST

Table of Contents

Chapter 1. Introduction

In the testing lectures I made a specific point to separate the testing concepts of

- focusing on a single class with stubs and mocks
- integrating multiple classes through a Spring context
- having to manage separate processes using the Maven integration test phases and plugins

Having only a single class under test meets most definitions of **"unit testing"**. Having to manage multiple processes satisfies most definitions of **"integration testing"**. Having to integrate multiple classes within a single JVM using a single JUnit test is a bit of a middle ground because it takes less heroics (thanks to modern test frameworks) and can be moderately fast.

I have termed the middle ground **"integration unit testing"** in an earlier lecture and labeled them with the suffix "NTest" to signify that they should run within the surefire unit test Maven phase and will take more time than a mocked unit test. In this lecture, I am going to expand the scope of "integration unit test" to include simulated resources like databases and JMS servers. This will allow us to write tests that are moderately efficient but more fully test layers of classes and their underlying resources within the context of a thread that is more representative of an end-to-end usecase.

Given an application like the following with databases and a JMS server...

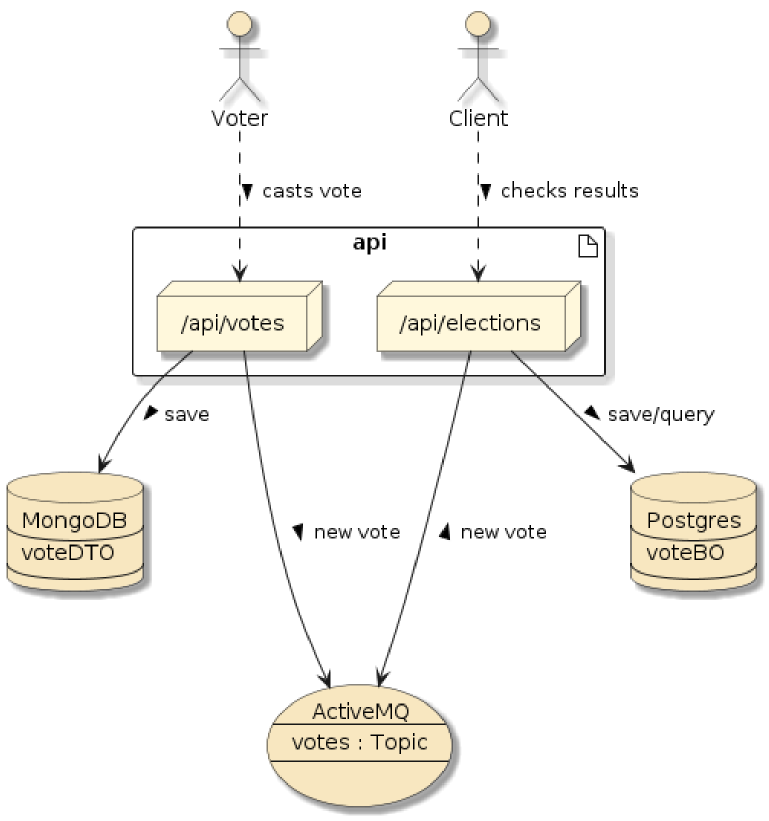


Figure 1. Votes Application

- how can we test application interaction with a real instance of the database?
- how can we test the integration between two processes communicating with JMS events?
- how can we test timing aspects between disparate user and application events?
- how can we test a **measured** amount of end-to-end tests with the scope of our unit integration tests?

1.1. Goals

You will learn:

- how to integrate MongoDB into a Spring Boot application
- how to integrate a Relational Database into a Spring Boot application
- how to integrate a JMS server into a Spring Boot application
- how to implement an integration unit test using embedded resources

1.2. Objectives

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

1. embed a simulated MongoDB within a JUnit test using Flapdoodle
2. embed an in-memory JMS server within a JUnit test using ActiveMQ
3. embed a relational database within a JUnit test using H2
4. verify an end-to-end test case using a unit integration test

Chapter 2. Votes and Elections Service

For this example, I have created two moderately parallel services — Votes and Elections — that follow a straight forward controller, service, repository, and database layering.

2.1. Main Application Flows

Table 1. Service Controller/Database Dependencies

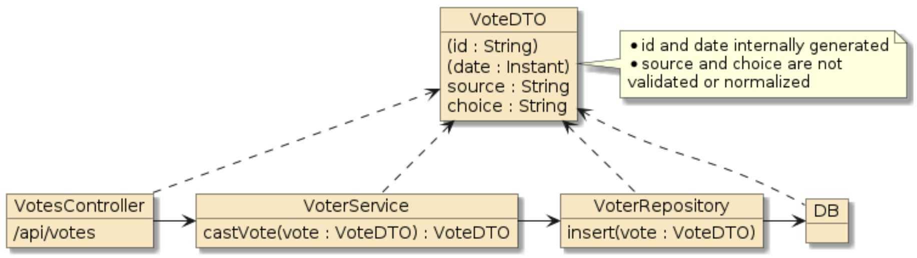


Figure 2. VotesService

The Votes Service accepts a vote (VoteDTO) from a caller and stores that directly in a database (MongoDB).

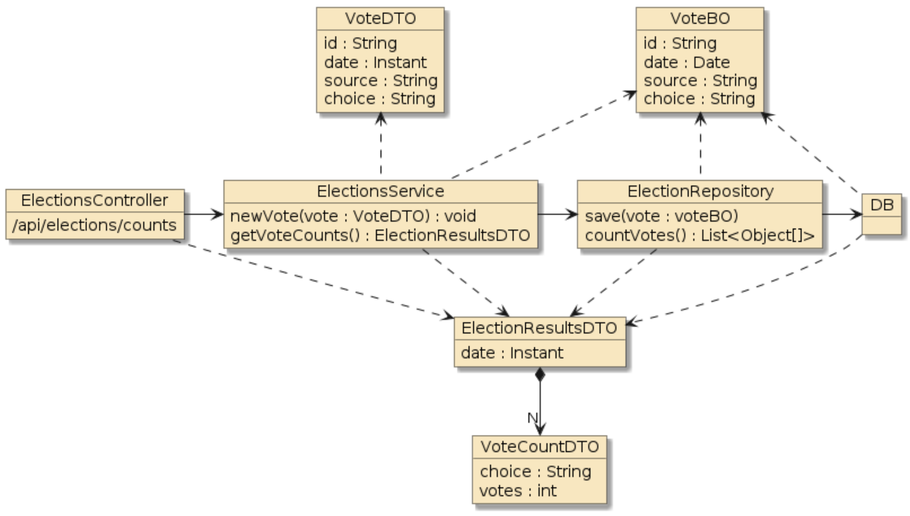


Figure 3. ElectionsService

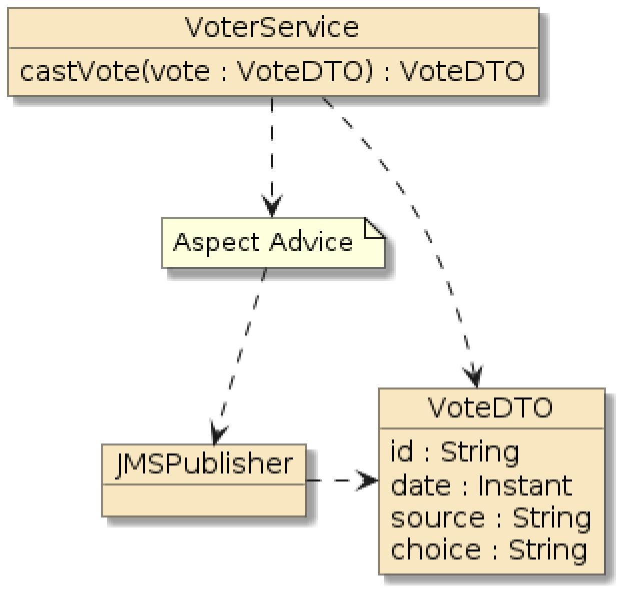
The Elections service transforms received votes (VoteDTO) into database entity instances (VoteBO) and stores them in a separate database (Postgres) using Java Persistence API (JPA). The service uses that persisted information to provide election results from aggregated queries of the database.

The fact that the applications use MongoDB, Postgres Relational DB, and JPA will only be a very small part of the lecture material. However, it will serve as a basic template of how to integrate these resources for much more complicated unit integration tests and deployment scenarios.

2.2. Service Event Integration

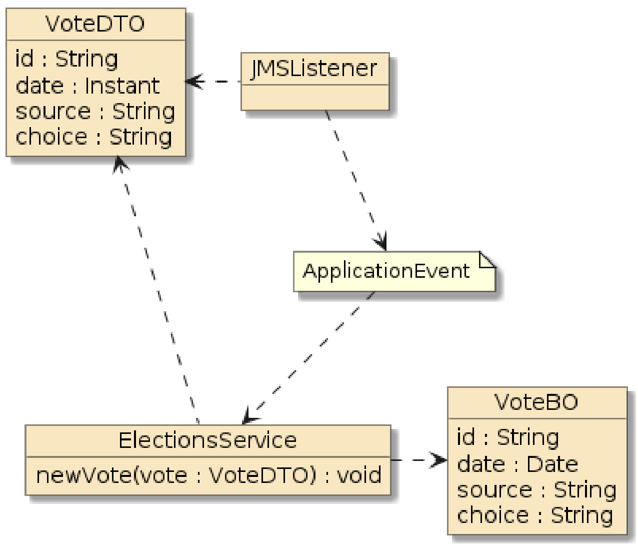
The two services are integrated through a set of Aspects, ApplicationEvent, and JMS logic that allow the two services to be decoupled from one another.

Table 2. Async Dependencies



The Votes Service events layer defines a pointcut on the successful return of the `VotesService.castVote()` and publishes the resulting vote (VoteDTO) — with newly assigned ID and date — to a JMS destination.

Figure 4. Votes Service



The Elections Service eventing layer subscribes to the `votes` destination and issues an internal `NewVote` POJO ApplicationEvent — which is relayed to the `ElectionsService` for mapping to an entity class (VoteBO) and storage in the DB for later query.

Figure 5. Elections Service

The fact that the applications use JMS will only be a small part of the lecture material. However, it too will serve as a basic template of how to integrate another very pertinent resource for distributed systems.

Chapter 3. Physical Architecture

I described five (5) functional services in the previous section: Votes, Elections, MongoDB, Postgres, and ActiveMQ (for JMS).

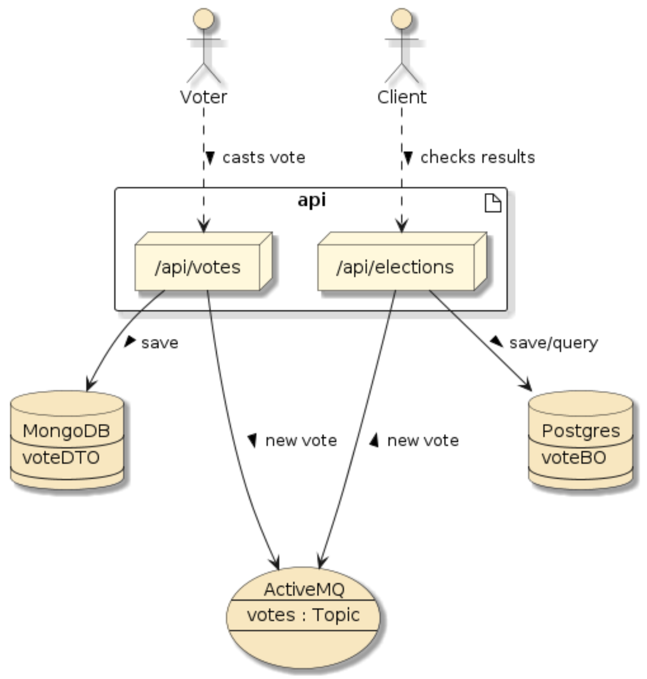


Figure 6. Physical Architecture

I will eventually mapped them to four (4) physical nodes: api, mongo, postgres, and activemq. Both Votes and Elections have been co-located in the same Spring Boot application because the Internet deployment platform may not have a JMS server available for our use.

3.1. Integration Unit Test Physical Architecture

For integration unit tests, we will use a single JUnit JVM with the Spring Boot Services and the three resources embedded using the following options:

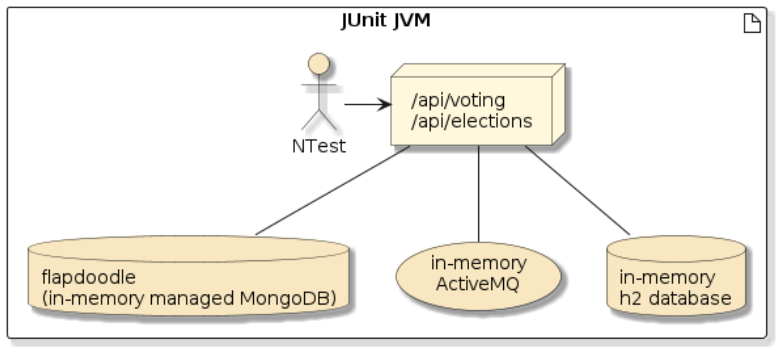


Figure 7. Integration Unit Testing Physical Architecture

- Flapdoodle - an open source initiative that markets itself to implementing an embedded MongoDB. It is incorrect to call the entirety of Flapdoodle "embedded". The management of MongoDB is "embedded" but a real server image is being downloaded and executed behind the scenes.
  - another choice is fongo. Neither are truly embedded and neither had much activity in the past 2 years, but flapdoodle has twice as many stars on github and has been active more recently (as of Aug 2020).

- H2 Database in memory RDBMS we used for user management during the later security topics
- ActiveMQ (Classic) used in embedded mode

Chapter 4. Mongo Integration

In this section we will go through the steps of adding the necessary MongoDB dependencies to implement a MongoDB repository and simulate that with an in-memory DB during unit integration testing.

4.1. MongoDB Maven Dependencies

As with most starting points with Spring Boot — we can bootstrap our application to implement a MongoDB repository by forming an dependency on `spring-boot-starter-data-mongodb`.

Primary MongoDB Maven Dependency

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

That brings in a few driver dependencies that will also activate the `MongoAutoConfiguration` to establish a default `MongoClient` from properties.

MongoDB Starter Dependencies

```
[INFO] +- org.springframework.boot:spring-boot-starter-data-
mongodb:jar:2.3.2.RELEASE:compile
[INFO] |  +- org.mongodb:mongodb-driver-sync:jar:4.0.5:compile
[INFO] |  |  +- org.mongodb:bson:jar:4.0.5:compile
[INFO] |  |  \- org.mongodb:mongodb-driver-core:jar:4.0.5:compile
[INFO] |  \- org.springframework.data:spring-data-mongodb:jar:3.0.2.RELEASE:compile
```

4.2. Test MongoDB Maven Dependency

For testing, we add a dependency on `de.flapdoodle.embed.mongo`. By setting scope to `test`, we avoid deploying that with our application outside of our module testing.

Test MongoDB Maven Dependency

```
<dependency>
    <groupId>de.flapdoodle.embed</groupId>
    <artifactId>de.flapdoodle.embed.mongo</artifactId>
    <scope>test</scope>
</dependency>
```

The `flapdoodle` dependency brings in the following artifacts.

Flapdoodle Dependencies

```
[INFO] +- de.flapdoodle.embed:de.flapdoodle.embed.mongo:jar:2.2.0:test
[INFO] |  \- de.flapdoodle.embed:de.flapdoodle.embed.process:jar:2.1.2:test
[INFO] |     +- org.apache.commons:commons-lang3:jar:3.10:compile
[INFO] |     +- net.java.dev.jna:jna:jar:4.0.0:test
[INFO] |     +- net.java.dev.jna:jna-platform:jar:4.0.0:test
[INFO] |     \- org.apache.commons:commons-compress:jar:1.18:test
```

4.3. MongoDB Properties

The following lists a core set of MongoDB properties we will use no matter whether we are in test or production. If we implement the most common scenario of a single single database — things get pretty easy to work through properties. Otherwise we would have to provide our own `MongoClient` `@Bean` factories to target specific instances.

Core MongoDB Properties

```
#mongo
spring.data.mongodb.authentication-database=admin ①
spring.data.mongodb.database=votes_db ②
```

① identifies the mongo database with user credentials

② identifies the mongo database for our document collections

4.4. MongoDB Repository

Spring Data provides a very nice repository layer that can handle basic CRUD and query capabilities with a simple interface definition that extends `MongoRepository<T,ID>`. The following shows an example declaration for a `VoteDTO` POJO class that uses a String for a primary key value.

MongoDB VoterRepository Declaration

```java
import info.ejava.examples.svc.docker.votes.dto.VoteDTO;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface VoterRepository extends MongoRepository<VoteDTO, String> {
}
```

4.5. VoteDTO MongoDB Document Class

The following shows the MongoDB document class that doubles as a Data Transfer Object (DTO) in the controller and JMS messages.

Example VoteDTO MongoDB Document Class

```java
import lombok.*;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import java.time.Instant;

@Document("votes") ①
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class VoteDTO {
    @Id
    private String id; ②
    private Instant date;
    private String source;
    private String choice;
}
```

① MongoDB `Document` class mapped to the `votes` collection

② VoteDTO.id property mapped to `_id` field of MongoDB collection

Example Stored VoteDTO Document

```
{
"_id":{"$oid":"5f3204056ac44446600b57ff"},
"date":{"$date":{"$numberLong":"1597113349837"}},
"source":"jim",
"choice":"quisp",
"_class":"info.ejava.examples.svc.docker.votes.dto.VoteDTO"
}
```

4.6. Sample MongoDB/VoterRepository Calls

The following snippet shows the injection of the repository into the service class and two sample calls. At this point in time, it is only important to notice that our simple repository definition gives us the ability to insert and count documents (and more!!!).

Sample MongoDB/VoterRepository Calls

```
@Service
@RequiredArgsConstructor ①
public class VoterServiceImpl implements VoterService {
    private final VoterRepository voterRepository; ①

    @Override
    public VoteDTO castVote(VoteDTO newVote) {
        newVote.setId(null);
        newVote.setDate(Instant.now());
        return voterRepository.insert(newVote); ②
    }
    @Override
    public long getTotalVotes() {
        return voterRepository.count(); ③
    }
```

① using constructor injection to initialize service with repository

② repository inherits ability to insert new documents

③ repository inherits ability to get count of documents

This service is then injected into the controller and accessed through the `/api/votes` URI. At this point we are ready to start looking at the details of how to report the new votes to the `ElectionsService`.

Chapter 5. ActiveMQ Integration

In this section we will go through the steps of adding the necessary ActiveMQ dependencies to implement a JMS publish/subscribe and simulate that with an in-memory JMS server during unit integration testing.

5.1. ActiveMQ Maven Dependencies

The following lists the dependencies we need to implement the Aspects and JMS capability within the application.

ActiveMQ Primary Maven Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
```

The ActiveMQ starter brings in the following dependencies and actives the `ActiveMQAutoConfiguration` class that will setup a JMS connection based on properties.

ActiveMQ Starter Dependencies

```
[INFO] +- org.springframework.boot:spring-boot-starter-
activemq:jar:2.3.2.RELEASE:compile
[INFO] |  +- org.springframework:spring-jms:jar:5.2.8.RELEASE:compile
[INFO] |  |  +- org.springframework:spring-messaging:jar:5.2.8.RELEASE:compile
[INFO] |  |  \- org.springframework:spring-tx:jar:5.2.8.RELEASE:compile
```

5.2. ActiveMQ Integration Unit Test Properties

The following lists the core property required by ActiveMQ in all environments. Without the `pub-sub-domain` property defined, ActiveMQ defaults to a queue model — which will not allow our integration tests to observe the traffic flow if we care to.

ActiveMQ Core Properties

```
#activemq
spring.jms.pub-sub-domain=true ①
```

① tells ActiveMQ to use topics versus queues

The following lists the properties that are unique to the local integration unit tests.

ActiveMQ Test Properties

```
#activemq
spring.activemq.in-memory=true ①
spring.activemq.pool.enabled=false
```

① activemq will establish in-memory destinations

5.3. Service Joinpoint Advice

I used Aspects to keep the Votes Service flow clean of external integration and performed that by enabling Aspects using the `@EnableAspectJAutoProxy` annotation and defining the following `@Aspect` class, joinpoint, and advice.

Example Service Joinpoint Advice

```
@Aspect
@Component
@RequiredArgsConstructor
public class VoterAspects {
    private final VoterJMS votePublisher;

    @Pointcut("within(info.ejava.examples.svc.docker.votes.services.VoterService+)")
    public void voterService(){} ①

    @Pointcut("execution(*..VoteDTO castVote(..))")
    public void castVote(){} ②

    @AfterReturning(value = "voterService() && castVote()", returning = "vote")
    public void afterVoteCast(VoteDTO vote) { ③
        try {
            votePublisher.publish(vote);
        } catch (IOException ex) {
            ...
        }
    }
}
```

① matches all calls implementing the `VoterService` interface

② matches all calls called `castVote` that return a `VoteDTO`

③ injects returned VoteDTO from matching calls and calls publish to report event

5.4. JMS Publish

The publishing of the new vote event using JMS is done within the `VoterJMS` class using an injected `jmsTemplate` and `ObjectMapper`. Essentially, the method marshals the `VoteDTO` object into a JSON text string and publishes that in a `TextMessage` to the "votes" topic.

JMS Publisher Code

```
@Component
@RequiredArgsConstructor
public class VoterJMS {
    private final JmsTemplate jmsTemplate; ①
    private final ObjectMapper mapper; ②
...

    public void publish(VoteDTO vote) throws JsonProcessingException {
        final String json = mapper.writeValueAsString(vote); ③

        jmsTemplate.send("votes", new MessageCreator() { ④
            @Override
            public Message createMessage(Session session) throws JMSException {
                return session.createTextMessage(json); ⑤
            }
        });
    }
}
```

① inject a jmsTemplate supplied by ActiveMQ starter dependency

② inject ObjectMapper that will marshal objects to JSON

③ marshal vote to JSON string

④ publish the JMS message to the "votes" topic

⑤ publish vote JSON string using a JMS `TextMessage`

5.5. ObjectMapper

The `ObjectMapper` that was injected in the `VoterJMS` class was built using a custom factory that configured it to use formatting and write timestamps in ISO format versus binary values.

ObjectMapper Factory

```
@Bean
public Jackson2ObjectMapperBuilder jacksonBuilder() {
    Jackson2ObjectMapperBuilder builder = new Jackson2ObjectMapperBuilder()
            .indentOutput(true)
            .featuresToDisable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);
    return builder;
}

@Bean
public ObjectMapper jsonMapper(Jackson2ObjectMapperBuilder builder) {
    return builder.createXmlMapper(false).build();
}
```

5.6. JMS Receive

The JMS receive capability is performed within the same `VoterJMS` class to keep JMS implementation encapsulated. The class implements a method accepting a JMS `TextMessage` annotated with `@JmsListener`. At this point we could have directly called the `ElectionsService` but I chose to go another level of indirection and simply issue an `ApplicationEvent`.

JMS Receive Code

```
@Component
@RequiredArgsConstructor
public class VoterJMS {
...
    private final ApplicationEventPublisher eventPublisher;
    private final ObjectMapper mapper;

    @JmsListener(destination = "votes") ②
    public void receive(TextMessage message) throws JMSException { ①
        String json = message.getText();
        try {
            VoteDTO vote = mapper.readValue(json, VoteDTO.class); ③
            eventPublisher.publishEvent(new NewVoteEvent(vote)); ④
        } catch (JsonProcessingException ex) {
            //...
        }
    }
}
```

① implements a method receiving a JMS `TextMessage`

② method annotated with `@JmsListener` against the `votes` topic

③ JSON string unmarshaled into a `VoteDTO` instance

④ Simple `NewVote` POJO event created and issued internal

5.7. EventListener

An `EventListener` `@Component` is supplied to listen for the application event and relay that to the `ElectionsService`.

Example Application Event Listener

```
import org.springframework.context.event.EventListener;

@Component
@RequiredArgsConstructor
public class ElectionListener {
    private final ElectionsService electionService;

    @EventListener ②
    public void newVote(NewVoteEvent newVoteEvent) { ①
        electionService.addVote(newVoteEvent.getVote()); ③
    }
}
```

① method accepts `NewVoteEvent` POJO

② method annotated with `@EventListener` looking for application events

③ method invokes `addVote` of `ElectionsService` when `NewVoteEvent` occurs

At this point we are ready to look at some of the implementation details of the Elections Service.

Chapter 6. JPA Integration

In this section we will go through the steps of adding the necessary dependencies to implement a JPA repository and simulate that with an in-memory RDBMS during unit integration testing.

6.1. JPA Core Maven Dependencies

The Elections Service uses a relational database and interfaces with that using Spring Data and Java Persistence API (JPA). To do that, we need the following core dependencies defined. The starter sets up the default JDBC DataSource and JPA layer. The `postgresql` dependency provides a client for Postgres and one that takes responsibility for Postgres-formatted JDBC URLs.

RDBMS Core Dependencies

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```

There are too many (~20) dependencies to list that come in from the `spring-boot-starter-data-jpa` dependency. You can run `mvn dependency:tree` yourself to look, but basically it brings in Hibernate and connection pooling. The supporting libraries for Hibernate and JPA are quite substantial.

6.2. JPA Test Dependencies

During integration unit testing we add the H2 database dependency to provide another option.

JPA Test Dependencies

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

6.3. JPA Properties

The test properties include a direct reference to the in-memory H2 JDBC URL. I will explain the use of Flyway next, but this is considered optional for this case because Spring Data will trigger auto-schema population for in-memory databases.

JPA Test Properties

```
#rdbms
spring.datasource.url=jdbc:h2:mem:users ①
spring.jpa.show-sql=true ②

# optional: in-memory DB will automatically get schema generated
spring.flyway.enabled=true ③
```

① JDBC in-memory H2 URL

② show SQL so we can see what is occurring between service and database

③ optionally turn on Flyway migrations

6.4. Database Schema Migration

Unlike the NoSQL MongoDB, relational databases have a strict schema that defines how data is stored. That must be accounted for in all environments. However — the way we do it can vary:

- Auto-Generation - the simplest way to configure a development environment is to use JPA/Hibernate auto-generation. This will delegate the job of populating the schema to Hibernate at startup. This is perfect for dynamic development stages where schema is changing constantly. This is unacceptable for production and other environments where we cannot loose all of our data when we restart our application.
- Manual Schema Manipulation - relational database schema can get more complex than what can get auto-generated and event auto-generated schema normally passes through the review of human eyes before making it to production. Deployment can be a manually intensive and likely the choice of many production environments where database admins must review, approve, and possibly execute the changes.

Once our schema stabilizes, we can capture the changes to a versioned file and use the Flyway plugin to automate the population of schema. If we do this during integration unit testing, we get a chance to supply a more tested product for production deployment.

6.5. Flyway RDBMS Schema Migration

Flyway is a schema migration library that can do forward (free) and reverse (at a cost) RDBMS schema migrations. We include Flyway by adding the following dependency to the application.

Flyway Maven Dependency

```xml
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-core</artifactId>
    <scope>runtime</scope>
</dependency>
```

The Flyway test properties include the JDBC URL that we are using for the application and a flag to enable.

Flyway Test Properties

```
spring.datasource.url=jdbc:h2:mem:users ①
spring.flyway.enabled=true
```

① Flyway makes use of the Spring Boot database URL

6.6. Flyway RDBMS Schema Migration Files

We feed the Flyway plugin schema migrations that move the database from version N to version N+1, etc. The default directory for the migrations is in `db/migration` of the classpath. The directory is populated with files that are executed in order according to a name syntax that defaults to `V#_#_#__description` (double underscore between last digit of version and first character of description; the number of digits in the version is not mandatory)

Flyway Migration File Structure

```
dockercompose-votes-svc/src/main/resources/
`-- db
    `-- migration
        |-- V1.0.0__initial_schema.sql
        `-- V1.0.1__expanding_choice_column.sql
```

The following is an example of a starting schema (V1_0_0).

Create Table/Index Example Migration #1

```
create table vote (
id varchar(50) not null,
choice varchar(40),
date timestamp,
source varchar(40),
constraint vote_pkey primary key(id)
);

comment on table vote is 'countable votes for election';
```

The following is an example of a follow-on migration after it was determined that the original `choice` column size was too small.

Expand Table Column Size Example Migration #2

```
alter table vote alter column choice type varchar(60);
```

6.7. Flyway RDBMS Schema Migration Output

The following is an example Flyway migration occurring during startup.

Example Flyway Schema Migration Output

```
Database: jdbc:h2:mem:users (H2 1.4)
Successfully validated 2 migrations (execution time 00:00.022s)
Creating Schema History table "PUBLIC"."flyway_schema_history" ...
Current version of schema "PUBLIC": << Empty Schema >>
Migrating schema "PUBLIC" to version 1.0.0 - initial schema
Migrating schema "PUBLIC" to version 1.0.1 - expanding choice column
Successfully applied 2 migrations to schema "PUBLIC" (execution time 00:00.069s)
```

For our integration unit test — we end up at the same place as auto-generation, except we are taking the opportunity to dry-run and regression test the schema migrations prior to them reaching production.

6.8. JPA Repository

The following shows an example of our JPA/ElectionRepository. Similar to the MongoDB repository — this extension will provide us with many core CRUD and query methods. However, the one aggregate query targeted for this database cannot be automatically supplied without some help. We must provide the JPA Query that translates into SQL query to return the choice, vote count, and latest vote data for that choice.

JPA/ElectionRepository

```
...
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

public interface ElectionRepository extends JpaRepository<VoteBO, String> {
    @Query("select choice, count(id), max(date) from VoteBO group by choice order by
count(id) DESC") ①
    public List<Object[]> countVotes(); ②
}
```

① JPA query language to return choices aggregated with vote count and latest vote for each choice

② a list of arrays — one per result row — with raw DB types is returned to caller

6.9. Example VoteBO Entity Class

The following shows the example JPA Entity class used by the repository and service. This is a standard JPA definition that defines a table override, primary key, and mapping aspects for each property in the class.

Example VoteBO Entity Class

```
...
import javax.persistence.*;

@Entity ①
@Table(name="VOTE") ②
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class VoteBO {
    @Id ③
    @Column(length = 50) ④
    private String id;
    @Temporal(TemporalType.TIMESTAMP)
    private Date date;
    @Column(length = 40)
    private String source;
    @Column(length = 40)
    private String choice;
}
```

① `@Entity` annotation required by JPA

② overriding default table name (`VOTEBO`)

③ JPA requires valid Entity classes to have primary key marked by `@Id`

④ column size specifications only used when generating schema — otherwise depends on migration to match

6.10. Sample JPA/ElectionRepository Calls

The following is an example service class that is injected with the `ElectionRepository` and is able to make a few sample calls. `save()` is pretty straight forward but notice that `countVotes()` requires some extra processing. The repository method returns a list of Object[] values populated with raw values from the database — representing choice, voteCount, and lastDate. The newest lastDate is used as the date of the election results. The other two values are stored within a `VoteCountDTO` object within `ElectionResultsDTO`.

Elections Service Class

```
@Service
@RequiredArgsConstructor
public class ElectionsServiceImpl implements ElectionsService {
    private final ElectionRepository votesRepository;

    @Override
    @Transactional(value = Transactional.TxType.REQUIRED)
    public void addVote(VoteDTO voteDTO) {
        VoteBO vote = map(voteDTO);
        votesRepository.save(vote); ①
    }

    @Override
    public ElectionResultsDTO getVoteCounts() {
        List<Object[]> counts = votesRepository.countVotes(); ②

        ElectionResultsDTO electionResults = new ElectionResultsDTO();
        //...
        return electionResults;
    }
```

① `save()` inserts a new row into the database

② `countVotes()` returns a list of Object[] with raw values from the DB

Chapter 7. Unit Integration Test

Stepping outside of the application and looking at the actual unit integration test — we see the majority of the magical meat in the first several lines.

- `@SpringBootTest` is used to define an application context that includes our complete application plus a test configuration that is used to inject necessary test objects that could be configured differently for certain types of tests (e.g., security filter)
- The port number is randomly generated and injected into the constructor to form baseUrls. We will look at a different technique in the Testcontainers lecture that allows for more first-class support for late-binding properties.

Example Integration Unit Test

```
@SpringBootTest( classes = {ClientTestConfiguration.class, VotesExampleApp.class},
        webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT, ①
        properties = "test=true") ②
@ActiveProfiles("test") ③
@DisplayName("votes integration unit test")
public class VotesTemplateNTest {
    @Autowired ④
    private RestTemplate restTemplate;
    private final URI baseVotesUrl;
    private final URI baseElectionsUrl;

    public VotesTemplateNTest(@LocalServerPort int port) ①
        throws URISyntaxException {
        baseVotesUrl = new URI( ⑤
            String.format("http://localhost:%d/api/votes", port));
        baseElectionsUrl = new URI(
            String.format("http://localhost:%d/api/elections", port));
    }
...
```

① configuring a local web environment with the random port# injected into constructor

② adding a `test=true` property that can be used to turn off conditional logic during tests

③ activating the `test` profile and its associated `application-test.properties`

④ `restTemplate` injected for cases where we may need authentication or other filters added

⑤ constructor forming reusable baseUrls with supplied random port value

7.1. ClientTestConfiguration

The following shows how the `restTemplate` was formed. In this case — it is extremely simple. However, as you have seen in other cases, we could have required some authentication and logging filters to the instance and this is the best place to do that when required.

ClientTestConfiguration

```
@SpringBootConfiguration()
@EnableAutoConfiguration       //needed to setup logging
public class ClientTestConfiguration {
    @Bean
    public RestTemplate anonymousUser(RestTemplateBuilder builder) {
        RestTemplate restTemplate = builder.build();
        return restTemplate;
    }
}
```

7.2. Example Test

The following shows a very basic example of an end-to-end test of the Votes Service. We use the baseUrl to cast a vote and then verify that is was accurately recorded.

Example test

```
@Test
public void cast_vote() {
    //given - a vote to cast
    Instant before = Instant.now();
    URI url = baseVotesUrl;
    VoteDTO voteCast = create_vote("voter1","quisp");
    RequestEntity<VoteDTO> request = RequestEntity.post(url).body(voteCast);

    //when - vote is casted
    ResponseEntity<VoteDTO> response = restTemplate.exchange(request, VoteDTO.class);

    //then - vote is created
    then(response.getStatusCode()).isEqualTo(HttpStatus.CREATED);
    VoteDTO recordedVote = response.getBody();
    then(recordedVote.getId()).isNotEmpty();
    then(recordedVote.getDate()).isAfterOrEqualTo(before);
    then(recordedVote.getSource()).isEqualTo(voteCast.getSource());
    then(recordedVote.getChoice()).isEqualTo(voteCast.getChoice());
}
```

At this point in the lecture we have completed covering the important aspects of forming an integration unit test with embedded resources in order to implement end-to-end testing on a small scale.

Chapter 8. Summary

At this point we should have a good handle on how to add external resources (e.g., MongoDB, Postgres, ActiveMQ) to our application and configure our integration unit tests to operate end-to-end using either simulated or in-memory options for the real resource. This gives us the ability to identify more issues early before we go into more manually intensive integration or production. In this following lectures — I will be expanding on this topic to take on several Docker-based approaches to integration testing.

In this module we learned:

- how to integrate MongoDB into a Spring Boot application
    - and how to integration unit test MongoDB code using Flapdoodle
- how to integrate a ActiveMQ server into a Spring Boot application
    - and how to integration unit test JMS code using an embedded ActiveMQ server
- how to integrate a Postgres into a Spring Boot application
    - and how to integration unit test relational code using an in-memory H2 database
- how to implement an integration unit test using embedded resources