CHAPTER 3

Testing Spring Applications

Before an application is delivered to the client, it must be tested and validated by a team of professionals called *testers*. As you can imagine, testing an application after development is complete is a little too late, because perhaps specifications were not understood correctly, or were not complete. Also, the behavior of an application in an isolated development system differs considerably from the behavior in a production system. This is why there are multiple testing steps that have to be taken—some of them before development. (Before development, a project must be designed; the design can be tested as well). And there is the human factor. Since no one is perfect, mistakes are made, and testing helps find those mistakes and fix them before the application reaches the end user, thus ensuring the quality of the software. The purpose of software testing is to verify that an application satisfies the functional (application provides the expected functions) and nonfunctional (application provides the expected functions as fast as expected and does not require more memory than is available on the system) requirements and to detect errors, and all activities of planning, preparation, evaluation, and validation are part of it.

There are specific courses and certifications for testers that are designed to train them in functional and software testing processes that they can use to test an application, and the ISTQB[1] is the organization that provides the infrastructure for training and examination.

[1] International Software Testing Qualification Board: `http://www.istqb.org/`.

I. Cosmina, *Pivotal Certified Professional Core Spring 5 Developer Exam*,
https://doi.org/10.1007/978-1-4842-5136-2_3

A Few Types of Testing

There are multiple types of testing classified by the development step in which they are executed, or by their implementation, but it is not the object of this book to cover them all. Only those that imply writing actual code using testing libraries and frameworks will be covered.

Test-Driven Development

Quality starts at the beginning of a project. Requirements and specifications are decided and validated, and based on them a process called **Test-Driven Development** can be executed. This process implies creation of tests before development of code. The tests will initially fail, but will start to pass one by one as the code is developed. The tests decide how the application will behave, and thus this type of testing is called test-driven development. This type of testing ensures that the specifications were understood and implemented correctly. The tests for this process are designed by business analysts and implemented by developers. This approach puts the design under question: if tests are difficult to write, the design should be reconsidered. It is more suitable to JavaScript applications (because there is no compilation of the code), but it can be used in Java applications too when the development is done using interfaces.

In Figure 3-1, the test-driven development process is described for exactly one test case.

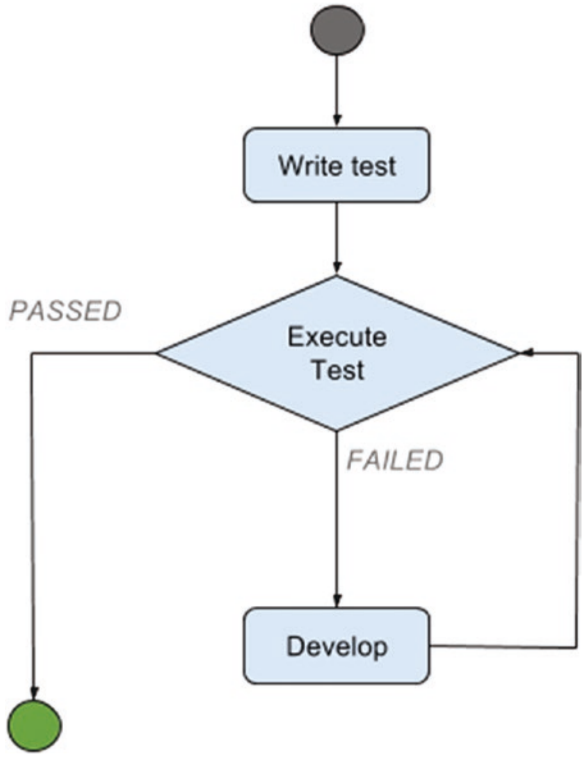


Figure 3-1. *Test-driven development logical schema*

This testing technique is good for finding problems early in the development process. And considering that the effort to solve a problem grows exponentially in proportion to the time it takes to find it, no project should ever be developed without it. Also, the tests should be run automatically using a continuous integration tool like Jenkins, Hudson, or Bamboo. Test-driven development can produce applications of high quality in less time than is possible with older methods, but it has its limitations. Sometimes tests might be incorrectly conceived or applied. This may result in units that do not perform as expected in the real world. Even if all the units work perfectly in isolation and in all anticipated scenarios, end users may encounter situations not imagined by the developers and testers. And since testing units was mentioned, this section will end here to cover the next one.

Unit and Integration Testing

Unit testing implies testing the smallest testable parts of an application individually and independently, and isolated from any other units that might affect their behavior in an unpredictable way. The dependencies are kept to a minimum, and most of them will be replaced with pseudo-units reproducing the expected behavior. This means that the unit of functionality is taken out of context. The unit tests are written by developers and are run using automated tools, although they can be run manually too in case of need.

A unit test exercises a single scenario with a provided input and compares the expected results for that input with the actual results. If they match, the test passes; if they don't, the test fails. This method of testing is fast and is often used in many projects. The tests are written by developers, and the recommended practice is to cover every method in a class with positive and negative tests. *Positive tests* are the ones that test valid inputs for the unit, while *negative tests* test invalid inputs for the unit. These are tests that cover a failure of the unit. They are considered to have failed if the unit does not actually fail when tested. The core framework helping developers to easily write and execute unit tests in Java since 2000 is `JUnit`. The current version when this chapter is being written is `4.13-beta-2`, but probably this will be the last version since `JUnit Jupiter` is already at version `5.5.0-M1`.

There are not many JUnit extensions, because there is little that this framework is missing; but there is a framework called Hamcrest that the Spring team is quite fond of, which is interesting because it provides a set of matchers that can be combined to create flexible expressions of intent. It originated as a framework based on `JUnit`, and it was used for writing tests in Java, but managed to break the language barrier, and currently it is provided for most currently used languages such as Python and Swift. Learn more about it on the official site at `http://hamcrest.org/`.

Also, a small open source library named AssertJ[2] is gaining ground because of its simplistic and intuitive syntax.

Running a suite of unit tests together in a context with all their real dependencies provided is called *integration testing*. As the name of this technique implies, the infrastructure needed to connect objects being tested is also a part of the context in which tests are executed. In Figure 3-2, is a simple diagram for comparing unit and integration testing concepts.

[2]More information is on the official site: `http://joel-costigliola.github.io/assertj/`

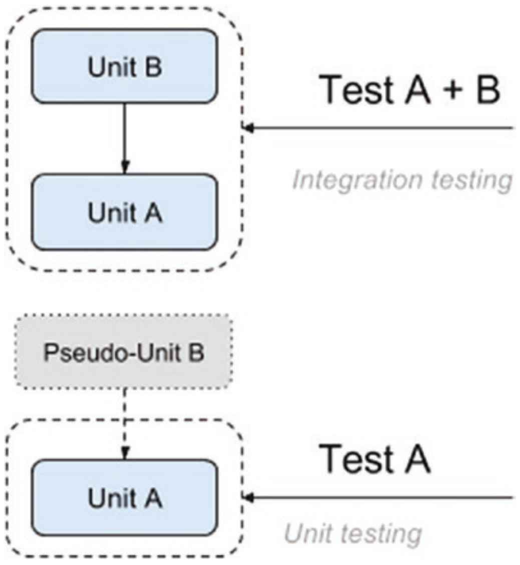


Figure 3-2. *Unit and integration testing concepts*

For functional units to be tested in isolation, dependencies must be replaced with pseudo-dependencies, fake objects with simple implementation that mimics the behavior of the real dependency as far as the dependent is concerned. The pseudo-dependencies can be *stubs* or *mocks*. Both perform the same function—to replace a real dependency, but the way that they are created is what sets them apart.

Testing with Stubs

Stubs are created by the developer; they do not require extra dependencies. A stub is a concrete class implementing the same interface as the original dependency of the unit being tested, so its methods produce dummy results of the correct types. They should be designed to exhibit a small part or the whole behavior of the actual dependency. If this approach seems familiar is because the book started with a similar example that was relevant for old style, pre-Spring Java development.

For example, let's try to test one of the service classes that were introduced in the previous chapter, the `SimpleCriminalCaseService`.

```
 package com.apress.cems.pojos.services.impl;
...

public class SimpleCriminalCaseService
     extends SimpleAbstractService<CriminalCase>
     implements CriminalCaseService {

    private CriminalCaseRepo repo;

    @Override
    public CriminalCase createCriminalCase(CaseType type,
             String shortDescription, String detailedDescription,
             CaseStatus caseStatus, String notes,
             Set<Evidence> evidenceSet, Detective leadInvestigator) {
        CriminalCase criminalCase = new CriminalCase();
        // call setters
        ...

        repo.save(criminalCase);
        return criminalCase;
    }

    @Override
    public Set<CriminalCase> findByLeadInvestigator(Detective detective) {
        return repo.findByLeadInvestigator(detective);
    }

    @Override
    public Optional<CriminalCase> findByNumber(String caseNumber) {
        return Optional.empty();
    }

    @Override
    public Set<CriminalCase> findByStatus(CaseStatus status) {
        return repo.findByStatus(status);
    }
    @Override
    public Set<CaseType> findByType(CaseType type) {
        return repo.findByType(type);
    }

    public void setRepo(CriminalCaseRepo repo) {
        this.repo = repo;
    }

    @Override
     AbstractRepo<CriminalCase> getRepo() {
        return repo;
    }
}

//abstract class containing concrete general implementations
public abstract class SimpleAbstractService<T extends AbstractEntity>
    implements AbstractService<T> {

    public void save(T entity) {
        getRepo().save(entity);
    }

    public T findById(Long entityId){
        return getRepo().findById(entityId);
    }

    @Override
    public void delete(T entity) {
        getRepo().delete(entity);
    }

    @Override
    public void deleteById(Long entityId) {
        getRepo().deleteById(entityId);
    }

    abstract AbstractRepo<T> getRepo();
}
```

```
// the interface defining the specific CriminalCase behavior
public interface CriminalCaseService extends
            AbstractService<CriminalCase> {
    CriminalCase createCriminalCase(CaseType type,
          String shortDescription, String detailedDescription,
          CaseStatus caseStatus, String notes,
          Set<Evidence> evidenceSet,
          Detective leadInvestigator);
}

// the interface defining the general service behavior
public interface AbstractService<T> {
    void save(T entity);

    T findById(Long entityId);

    void delete(T entity);

    void deleteById(Long entityId);
}
```

Although the number of tests varies depending on developer experience and what the code actually does, also related in a funny short story by Alberto Savoia, posted on the Google official blog,[3] my recommendation is to start unit testing by trying to write at least two tests for each method: one positive and one negative, for methods that can be tested in this way. There are four methods in `SimpleCriminalCaseService`, besides the getter and setter for the repository, so the test class should have approximately eight tests. When testing the service class, the concern is that the class should interact correctly with the repository class. The behavior of the repository class is assumed known, tested, and immutable. The stub class will implement the typical repository behavior but without a database connection needed, because interaction with a database introduces an undesired lag in test execution. The implementation presented here will use a `java.util.Map` to simulate a database. Like the application, there are more repository classes extending `SimpleAbstractService<T>`. The stubs will follow the same inheritance design, so the abstract class will be stubbed as well.

[3]Here it is, in case you are curious: http://googletesting.blogspot.ro/2010/07/ code-coverage-goal-80-and-no-less.html.

```
package com.apress.cems.stub.repo;
...
public abstract class StubAbstractRepo
     <T extends AbstractEntity> implements AbstractRepo<T> {

    protected Map<Long, T> records = new HashMap<>();

    @Override
    public void save(T entity) {
        if (entity.getId() == null) {
            Long id = (long) records.size() + 1;
            entity.setId(id);
        }
        records.put(entity.getId(), entity);
    }

    @Override
    public void delete(T entity) throws NotFoundException {
        records.remove(findById(entity.getId()).getId());
    }

    @Override
    public void deleteById(Long entityId) throws NotFoundException {
        records.remove(findById(entityId).getId());
    }

    @Override
    public T findById(Long entityId) {
        if(records.containsKey(entityId)) {
            return records.get(entityId);
        } else {
            throw new NotFoundException("Entity with id "
    + entityId + " could not be processed because it does not exist.");
        }
    }
}
```

The StubCriminalCaseRepo class extends the previous stub class, adding its specific behavior. And since Map contains <Detective,Set<CriminalCase>> pairs, neither of the specific CriminalCaseRepo methods can be stubbed, so a new map is needed.

```
package com.apress.cems.stub.repo;
...
public class StubCriminalCaseRepo extends
      StubAbstractRepo<CriminalCase> implements CriminalCaseRepo {

    protected Map<Detective, Set<CriminalCase>> records2 = new HashMap<>();

    public void init(){
        // create a few entries to play with
        final Detective detective = buildDetective
              ("Sherlock", "Holmes", Rank.INSPECTOR);
        this.save(buildCase(detective,
            CaseType.FELONY, CaseStatus.UNDER_INVESTIGATION));
        this.save(buildCase(detective,
            CaseType.MISDEMEANOR, CaseStatus.SUBMITTED));
    }

    @Override
    public void save(CriminalCase criminalCase) {
        super.save(criminalCase);
        addWithLeadInvestigator(criminalCase);
    }

    private void addWithLeadInvestigator(CriminalCase criminalCase){
        if (criminalCase.getLeadInvestigator()!= null) {
            Detective lead = criminalCase.getLeadInvestigator();
            if (records2.containsKey(lead)) {
                records2.get(lead).add(criminalCase);
            } else {
                Set<CriminalCase> ccSet = new HashSet<>();
                ccSet.add(criminalCase);
                records2.put(lead, ccSet);
            }
        }
    }
    @Override
    public Set<CriminalCase> findByLeadInvestigator(Detective detective) {
        return records2.get(detective);
    }

    @Override
    public Optional<CriminalCase> findByNumber(String caseNumber) {

        final CriminalCase[] result = new CriminalCase[1];

       records2.values().forEach(set -> set.stream()
                .filter(c -> c.getNumber().equalsIgnoreCase(caseNumber))

                    .findFirst().ifPresent(c -> result[0] = c)
       );
       return Optional.of(result[0]);

     }

...
}
```

Now that we have the stubs, they have to be used.

Unit Testing Using JUnit

To write multiple unit tests and execute them together as a suite, a dependency is needed to make the implementation run more easily. This dependency is JUnit,[4] a Java framework to write repeatable unit tests. It provides annotations to prepare and run unit test suites.

The recommended practice is to create a class named the same as the class to be tested but postfixed with `Test`, so the test class in this case is named `SimpleCriminalCaseServiceTest`. Only a few test examples will be depicted here. For more, look in the code attached to the book for this chapter, project `chapter03/junit-tests`. The full structure of the project is depicted in Figure 3-3; the location of the `SimpleCriminalCaseServiceTest` is obvious.

[4]JUnit official site: `http://junit.org/junit4/`.

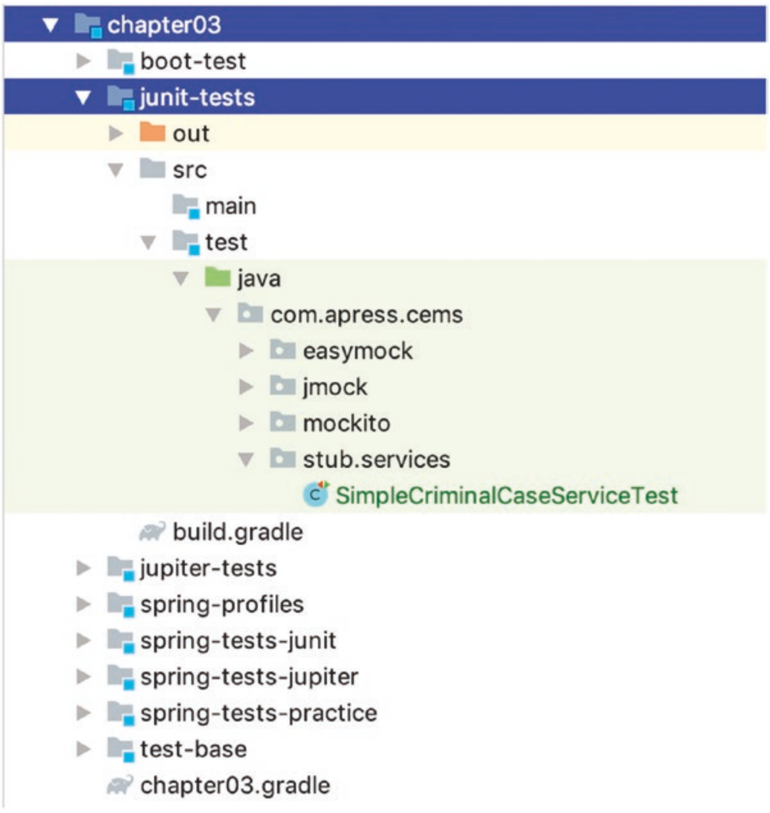


Figure 3-3. *Chapter 3 testing projects*

```
package com.apress.cems.stub.services;

import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertNull;
...

public class SimpleCriminalCaseServiceTest {

    static final Long CASE_ID = 1L;
    final Detective detective = buildDetective("Sherlock",
          "Holmes", Rank.INSPECTOR, "TS1234");
```

```
StubCriminalCaseRepo repo = new StubCriminalCaseRepo();

// object to be tested
 SimpleCriminalCaseService service = new SimpleCriminalCaseService();

@Before
public void setUp(){
    repo.init();

    //create object to be tested
    service = new SimpleCriminalCaseService();
    service.setRepo(repo);
}

  //positive test, we know that a Case with ID=1 exists
@Test
public void findByIdPositive() {
    CriminalCase criminalCase = service.findById(CASE_ID);
    assertNotNull(criminalCase);
}

//negative test, we know that a Case with ID=99 does not exist
@Test(expected = NotFoundException.class)
public void findByIdNegative() {
    CriminalCase criminalCase = service.findById(99L);
    assertNull(criminalCase);
}

 //negative test, we know that cases for this detective do not exist
@Test
 void findByLeadNegative() {
    Detective detective = buildDetective("Jake", "Peralta", Rank.
    JUNIOR, "TS1122");
    Set<CriminalCase> result =  service.findByLeadInvestigator(detective);
   assertNull(result);
}
```

```
    //positive test, we know that cases for this detective exist and how
    many
    @Test
    public void findByLeadPositive() {
        Set<CriminalCase> result =  service.findByLeadInvestigator(detective);
        assertEquals(result.size(), 2);
    }

    //positive case, deleting existing case record
    @Test
    public void deleteByIdPositive() {
        service.deleteById(CASE_ID);

        try {
            CriminalCase criminalCase = service.findById(CASE_ID);
            assertNull(criminalCase);
        } catch (NotFoundException nfe){
          assertTrue(nfe.getMessage().contains(
             "Entity with id 1 could not be processed because it does not
             exist"));
        }
    }

    //negative case, attempt to delete non-existing case
    @Test(expected = NotFoundException.class)
    public void deleteByIdNegative() {
        service.deleteById(99L);
    }

    @After
    public void tearDown(){
        repo.clear();
    }
}
```

There are special methods like the `findByIdNegative()` test method which is a negative test that expects for a type of exception to be thrown, not for a result to be returned that can be used in an assertion. Since this behavior is known, the test methods are written using the `expected` attribute of the `@Test` annotation and the type of exception is provided as argument.

In the preceding code snippet, the following JUnit components were used.

- The `@Before` annotation is used on methods that should be executed before executing each test method. These prepare the context for that test to execute in. All objects used in the test methods should be created and initialized in this method. Methods annotated with `@Before` are executed before every method annotated with `@Test`.
- The `@Test` annotation is the annotation that tells JUnit that the code in this method should be run as a test case. In JUnit 4, methods annotated with `@Test` should always be public and return void. It can also be used to treat expected exceptions.
- The `org.junit.jupiter.api.Assertions.*` static methods are defined in the `org.junit.Assert` class and can simplify the code of a test method. Without them, the user would have to write the code that specifies when the test should pass or fail.
- The `@After` annotation is used on methods that should be executed after executing each test method. These methods clean up the context after the test was executed. All objects initialized in the method `@Before` should be destroyed in the method annotated with `After`, because tests might modify them and cause unpredictable results when running the tests.

All the annotations and utilities are declared in this case under package `org.junit`.

After a quick analysis of the code, one thing should be obvious: testing code with stubs is time-consuming, and writing stubs seems to take as much time as development itself. Indeed, testing using stubs is applicable only for really simple applications and scenarios. The worst thing about testing with stubs, though, is that if the interface changes, the stub implementation must change too. So, not only do you have to adapt the tests, but the stubs too. Since we are using Java 11, there is the option of using default methods in interfaces, but this might lead to your tests not being accurate. The second-worst thing is that all methods have to be implemented when stubs are used, even those that are not used by a test scenario. For example,

```
@Override
 public Set<CriminalCase> findByType(CaseType type) {
      throw new NotImplementedException("Not needed for this stub.");
 }
```

This method was not involved in any test scenario, and to avoid providing an implementation, the decision was made to throw a `NotImplementedException`. The third-worst thing about stubs is that if we have a hierarchy of stubs, refactoring the one at the base of the hierarchy means refactoring all the stubs based on them, or else tests will fail. And this is the case in our example as well, since all stubs are based on `StubAbstractRepo`.

Unit Testing Using JUnit 5

JUnit 5 is also known as JUnit Jupiter, and compared to previous versions of JUnit, it is composed of several different modules from three different subprojects.

- **JUnit Platform**: A foundation for launching testing frameworks on the JVM.
- **JUnit Jupiter**: The new library for writing JUnit 5 tests.
- **JUnit Vintage**: A library providing engines that support execution of JUnit 3 and JUnit 4 tests.

The previous section covered how to write unit tests using stubs and JUnit 4. This section will cover how the same tests can be written using JUnit 5. Before discussing everything let's introduce the JUnit 5 version of `SimpleCriminalCaseServiceTest`.

```
package com.apress.cems.jupiter.services;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
...
```

```java
public class SimpleCriminalCaseServiceTest {
    static final Long CASE_ID = 1L;
    final Detective detective = buildDetective("Sherlock", "Holmes", Rank.
    INSPECTOR,
        "TS1234");

    StubCriminalCaseRepo repo = new StubCriminalCaseRepo();

     SimpleCriminalCaseService service = new SimpleCriminalCaseService();

    @BeforeEach
    public void setUp(){
        repo.init();

        //create object to be tested
        service = new SimpleCriminalCaseService();
        service.setRepo(repo);
    }

    //positive test, we know that a Case with ID=1 exists
    @Test
    public void findByIdPositive() {
        CriminalCase criminalCase = service.findById(CASE_ID);
        assertNotNull(criminalCase);
    }

    //negative test, we know that a Case with ID=99 does not exist
    @Test
    public void findByIdNegative() {
        assertThrows( NotFoundException.class, () ->
                service.findById(99L), "No such case exists");
    }

    ... // other test methods available on GitHub repository

    @AfterEach
    public void tearDown(){
        repo.clear();
    }
```

There are a few differences, but overall the logic is the same: there must be a method that is executed before every test method (in this case, annotated with `@BeforeEach`), and there must be a method to be executed after a test method (in this case, annotated with `@AfterEach`).

A difference that can be easily observed is the way the negative test was written using JUnit 5. The `assertThrows(..)` method allows developers to test assumptions about the exceptions being thrown when a certain method is executed. The method receives as a parameter the type of exception that is expected to be thrown. An instance of type `org.junit.jupiter.api.function.Executable`, which is an interface annotated with `@FunctionalInterface`, can be used within a lambda expression that simplifies the code to be written. There is an `assertThrows(..)` in JUnit 4 as well that was used in version 4.13; it receives a `ThrowingRunnable` instance instead of an executable. So in JUnit 4, you have more than one way of testing an assumption that involves exceptions.

All the annotations and utilities are declared in this case under package `org.junit.jupiter.api`. Actually, a comparison table would be better, right?

In Table 3-1, you can see a small set of components used for the same purpose in JUnit 4 and JUnit 5.

Table 3-1. *JUnit 4 Compare to JUnit 5 Equivalences*

JUnit 4	JUnit 5	Description
`org.junit`	`org.junit.jupiter.ap`	iBase package.
`@Test`	`@Test`	Annotation that marks test methods. Test methods must not be private or static and must not return a value. In JUnit 5, the test methods can also have *package* access.
`@Test(expected = Exception.class)`	`@Test plus assertThrows(..)`	Annotation that marks test methods containing assumptions about exceptions being thrown.
`@BeforeClass`	`@BeforeAll`	Used on static public (also *package* in JUnit 5) methods, mark them for execution before all test methods in the class, to create a mutable context for tests to be executed in.

(*continued*)

Table 3-1. (*continued*)

JUnit 4	JUnit 5	Description
`@Before`	`@BeforeEach`	Annotation that marks methods to be execute before every test method in a class. These methods must not be private or static and must not return a value. In JUnit 5, the test methods can also have *package* access.
`@AfterClass`	`@AfterAll`	Used on static public methods, mark them for execution after all test methods in the class, to destroy the mutable context the tests were executed in. In JUnit 5, the test methods can also have *package* access.
`@After`	`@AfterEach`	Annotation that marks methods to be execute after every test method in a class. These methods must not be private or static and must not return a value.
`org.junit.Assert.assertNotNull`	`org.junit.jupiter.api.Assertions.assertNotNull`	Static method used to test the assumption that the tested snippet of code does not return a `null` value.
`@Ignore`	`@Disabled`	Annotations to be used on a class or method that will cause for the test method(s) not to be executed.

JUnit 5 has more annotations than JUnit 4; the same goes for assertion/assumption methods. Some of them are used later in this book, when the context requires it. Explanations are provided on the spot, and since the focus of the book is on Spring testing, this section will end here.

! Executing JUnit 4 or JUnit 5 tests require different dependencies on the classpath of the application, and the configuration for this is all taken care of by Gradle. The sources attached to the book provide a working configuration for both. That is why, there are two projects one for each version: `chapter03/junit-tests` for tests written using JUnit 4 and `chapter03/jupiter-tests` for tests written using JUnit 5.

Conclusion: Stubs make testing seem like a tedious job, so let's see what *mocks* can do to improve the situation.

Testing with Mocks

A mock object is also a pseudo object, replacing the dependency we are not interested in testing and helping to isolate the object in which we are interested. Mock code does not have to be written, because there are a few libraries and frameworks that can generate mock objects. The mock object will implement the dependent interface on the fly. Before a mock object is generated, the developer can configure its behavior: what methods will be called and what will they return. The mock object can be used after that, and then expectations can be checked to decide the test result.

There are other libraries and frameworks for mock generation written in Java applications. The one that you use is up to you.

- **EasyMock** is a framework that provides an easy way to replace collaborators of the unit under test. Learn more about it on their official site at `http://easymock.org/`. This library is very popular and was used extensively in Spring development until version 3.1 made the switch to Mockito.[5]
- **jMock** is a small library, and the team that created it brags about making it quick and easy to define mock objects. The API is very nice and is suitable for complex stateful logic. See the official site at `www.jmock.org`.

[5]More information on the Spring IO Official blog: `https://spring.io/blog/2012/11/07/spring-framework-3-2-rc1-new-testing-features`

- **Mockito** is a mocking framework that provides a really clean and simple API for writing tests. The interesting thing about it is that it provides the possibility of partial mocking; real methods are invoked but still can be verified and stubbed. More about it on their official site at `http://mockito.org`.
- **PowerMock** is a framework that extends other mock libraries such as EasyMock with more powerful capabilities. It was created to provide a way of testing code considered untestable. It uses a custom classloader and bytecode manipulation to enable mocking of static methods, constructors, final classes and methods, private methods, removal of static initializers, and more. You can read more about it at `https://github.com/jayway/powermock`.

Each of these mocking tools will be covered in this chapter. All of them also provide annotations when used in a Spring context, when doing integration testing. But until Spring testing is covered, we will stick to simple unit testing.

EasyMock

The class to test with mocks generated by `EasyMock` is `SimpleDetectiveService`. This class inherits all methods from `SimpleAbstractService<T>` and provides its own for detective-specific behavior.

```
package com.apress.cems.services.impl;
...

import java.util.Optional;
import java.util.Set;

public class SimpleDetectiveService extends
    SimpleAbstractService<Detective> implements DetectiveService {

    private DetectiveRepo repo;

    @Override
    public Detective createDetective(Person person, Rank rank) {
        Detective detective = new Detective();
        //set properties
        repo.save(detective);
        return detective;
    }

    @Override
    public Optional<Detective> findByBadgeNumber(String badgeNumber) {
        return repo.findByBadgeNumber(badgeNumber);
    }

    @Override
    public Set<Detective> findByRank(Rank rank) {
        return repo.findbyRank(rank);
    }

    public void setRepo(DetectiveRepo repo) {
        this.repo = repo;
    }

    @Override AbstractRepo<Detective> getRepo() {
        return repo;
    }
}

// interface for Detective specific behavior
public interface DetectiveService extends AbstractService<Detective> {

    Detective createDetective(Person person, Rank rank);

    Optional<Detective> findByBadgeNumber(String badgeNumber);

    Set<Detective> findByRank(Rank rank);
}
```

To perform a test using mocks generated with EasyMock, the following steps have to be completed:

1. Declare the mock.
2. Create the mock.
3. Inject the mock.
4. Record what the mock is supposed to do.

5. Tell the mock the actual testing that is being done.

6. Test.

7. Make sure that the methods were called on the mock.

8. Validate the execution.

The following test class is named `SimpleDetectiveServiceTest`. In the following test snippet, the `findById()` method is tested, and the preceding steps are underlined.

```
package com.apress.cems.easymock;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
...
import static org.easymock.EasyMock.*;
import static org.easymock.EasyMock.verify;
import static org.junit.jupiter.api.Assertions.*;

public class SimpleDetectiveServiceTest {
    static final Long DETECTIVE_ID = 1L;

    (1)private DetectiveRepo mockRepo;
    private SimpleDetectiveService service;

    @BeforeEach
    public void setUp() {
    (2)mockRepo = createMock(DetectiveRepo.class);

     //create object to be tested
     service = new SimpleDetectiveService();

     (3) service.setRepo(mockRepo);
    }

    @Test
    public void findByNamePositive() {
        //record what we want the mock to do
       Detective simpleDetective = buildDetective("Sherlock", "Holmes",
          Rank.INSPECTOR, "TS1234");
        simpleDetective.setId(DETECTIVE_ID);
```

```
         (4)expect(mockRepo.findById(DETECTIVE_ID)).
        andReturn(simpleDetective);
         (5)replay(mockRepo);

         (6)Detective detective = service.findById(DETECTIVE_ID);
         (7)verify(mockRepo);
         (8)assertAll(
                () -> assertNotNull(detective),
                () -> assertEquals(detective.getId(), simpleDetective.getId()),
                () -> assertEquals(detective.getBadgeNumber(),
                                         simpleDetective.getBadgeNumber())
         );
    }
}
```

The EasyMock framework provides static methods to process the mock. When multiple mocks are needed, the `replay` and `verify` methods are replaced with `replyAll` and `verifyAll`, and the mocks are picked up and processed without direct reference to any of them. The main advantage of using mocks is that there is no need to maintain any extra classes, because when using mocks, the behavior needed from the dependency is defined on the spot, inside the test method body, and the generating framework takes care of mimicking the behavior. Inexperienced developers might have difficulty understanding how mocking works, but if you are a mentor, just ask them to create stubs first and then switch them to mocks. They will understand more easily and will be enchanted by the possibility of not needing to write too much extra code to test an object.

EasyMock can be used with JUnit 4 and JUnit 5, but the syntax differs accordingly.

Another thing about the previous example is the `assertAll(..)` method, which was introduced in JUnit 5 and receives as argument an array of `Executable` instances. Executable is a functional interface that can be used in lambda expressions. The advantage of using `assertAll(..)` is that all assertions provided as arguments will be executed, even if one of more fails. In JUnit 4, there is no such grouping method, which means that if more than one assertion is used within the same test method, the execution stops at the first failure.

Let's test that, shall we? Try modifying the `assertAll(..)` as follows.

```
assertAll(
         () -> assertNotNull(detective),
         () -> assertEquals(detective.getId(), 2L),
         () -> assertEquals(detective.getBadgeNumber(), "SH2211")
);
```

If you are not familiar with lambda expressions, you might be asking yourself right about now: *Heeeey, where are those Executable instances that were mentioned earlier?* Well, the previous code listing depicts the collapsed version of the `assertAll(..)` call. Each line in its body represents an `Executable` instance. If we expand the code, and not use lambdas, we get the following.

```
import org.junit.jupiter.api.function.Executable;
...

        assertAll(
                new Executable() {
                    @Override
                    public void execute() throws Throwable {
                        assertNotNull(detective);
                    }
                },
                new Executable() {
                    @Override
                    public void execute() throws Throwable {
                        assertEquals(detective.getId(), 2L);
                    }
                },
                new Executable() {
                    @Override
                    public void execute() throws Throwable {
                        assertEquals(detective.getBadgeNumber(), "SH2211");
                    }
                }
        );
```

Not so simple and easy to read anymore, right?

Anyway, if the previous code snippet is run (collapsed form or not), the following is seen in the console.

```
expected: <1> but was: <2>
Comparison Failure:
Expected :1
Actual   :2
<Click to see difference>

expected: <TS1234> but was: <SH2211>
Comparison Failure:
Expected :TS1234
Actual   :aaa
<Click to see difference>

org.opentest4j.MultipleFailuresError: Multiple Failures (2 failures)
        expected: <1> but was: <2>
        expected: <TS1234> but was: <SH2211>
```

Quite practical, right?

jMock

The class to test with mocks generated by `jMock` is `SimpleEvidenceService`. This class inherits all methods from `SimpleAbstractService<T>` and provides its own for requesting specific behavior.

```
package com.apress.cems.services.impl;
 ...
public class SimpleEvidenceService extends
    SimpleAbstractService<Evidence> implements EvidenceService {
    private EvidenceRepo repo;

    public SimpleEvidenceService() {
    }
    public SimpleEvidenceService(EvidenceRepo repo) {
        this.repo = repo;
    }

    @Override
    public Evidence createEvidence(CriminalCase criminalCase,
        Storage storage, String itemName) {
        Evidence evidence = new Evidence();
        evidence.setCriminalCase(criminalCase);
        evidence.setNumber(NumberGenerator.getEvidenceNumber());
        evidence.setItemName(itemName);
        evidence.setStorage(storage);
        repo.save(evidence);
        return evidence;
    }

    @Override
    public Set<Evidence> findByCriminalCase(CriminalCase criminalCase) {
        return repo.findByCriminalCase(criminalCase);
    }

    @Override
    public Optional<Evidence> findByNumber(String evidenceNumber) {
        return repo.findByNumber(evidenceNumber);
    }

    public void setRepo(EvidenceRepo repo) {
        this.repo = repo;
    }

    @Override
    AbstractRepo<Evidence> getRepo() {
        return repo;
    }
}
```

```
// interface for Request specific behavior
public interface EvidenceService extends AbstractService<Evidence> {
    Evidence createEvidence(CriminalCase criminalCase, Storage storage,
                String itemName);

    Set<Evidence> findByCriminalCase(CriminalCase criminalCase);

    Optional<Evidence> findByNumber(String evidenceNumber);
}
```

To perform a test using mocks generated with `jMock`, the following steps have to be completed.

1. Declare the mock.
2. Declare and define the context of the object under test, an instance of the `org.jmock.Mockery` class.
3. Create the mock.
4. Inject the mock.
5. Define the expectations we have from the mock.
6. Test.
7. Check that the mock was used.
8. Validate the execution.

The test class is named `SimpleEvidenceServiceTest`, and in the following test snippet, the creation of an`Evidence` instance is tested and the preceding steps are underlined.

```
package com.ps.repo.services;

import org.jmock.Expectations;
import org.jmock.Mockery;
...

public class SimpleEvidenceServiceTest  {

     (1)private EvidenceRepo mockRepo;

     (2)private Mockery mockery = new JUnit5Mockery();
```

```
    private SimpleEvidenceService service;

    @BeforeEach
    public void setUp() {
         (3)mockRepo = mockery.mock(EvidenceRepo.class);

        service = new SimpleEvidenceService();
         (4)service.setRepo(mockRepo);
    }

    @Test
    public void testCreateEvidence() {
        Detective detective = buildDetective("Sherlock", "Holmes",
             Rank.INSPECTOR, "TS1234");
        CriminalCase criminalCase = buildCase(detective, CaseType.FELONY,
             CaseStatus.UNDER_INVESTIGATION);
        Evidence evidence = new Evidence();
        evidence.setNumber("123445464");
        evidence.setItemName("Red Bloody Knife");
        evidence.setId(EVIDENCE_ID);
        evidence.setCriminalCase(criminalCase);

         (5)mockery.checking(new Expectations() {{
            allowing(mockRepo).findById(EVIDENCE_ID);
            will(returnValue(evidence));
        }});

         (6)Evidence result = service.findById(EVIDENCE_ID);
         (7)mockery.assertIsSatisfied();
         (8)assertAll(
              () -> Assertions.assertNotNull(result),
              () -> Assertions.assertEquals(result.getId(), evidence.
                    getId()),
              () -> Assertions.assertEquals(result.getNumber(), evidence.
                    getNumber())
        );
    }
}
```

When multiple mocks are used, or multiple operations are executed by the same mock, defining the expectations can become a bit cumbersome. Still, it is easier than defining stubs.

There are two jMock libraries: jmock-junit4 for JUnit 4 and jmock-junit5 for JUnit 5. As you can see, the central class to create mocks with jMock is the Mockery class. Each version of the jMock library provides a particular implementation, JUnit5Mockery and JUnit4Mockery.

Mockito

Mockito has the advantage of mocking behavior by writing code that is readable and very intuitive. The collection of methods provided was designed so well that even somebody without extensive programming knowledge can understand what is happening in that code, if that person also understands English. The class that will be tested with Mockito is SimpleStorageService. This class inherits all methods from SimpleAbstractService<T> and provides its own for request specific behavior.

```
package com.apress.cems.services.impl;
...

public class SimpleStorageService extends
   SimpleAbstractService<Storage> implements StorageService {
    private StorageRepo repo;

    @Override
    public Storage createStorage(String name, String location) {
        Storage storage = new Storage();
        storage.setName(name);
        storage.setLocation(location);
        repo.save(storage);
        return storage;
    }

    @Override
    public Optional<Storage> findByName(String name) {
        return repo.findByName(name);
    }
    @Override
    public Optional<Storage> findByLocation(String location) {
        return repo.findByLocation(location);
    }

    public void setRepo(StorageRepo repo) {
        this.repo = repo;
    }

    @Override
    AbstractRepo<Storage> getRepo() {
        return repo;
    }
}

// interface for Service specific behavior
public interface StorageService extends AbstractService<Storage> {

    Storage createStorage(String name, String location);

    Optional<Storage> findByName(String name);

    Optional<Storage> findByLocation(String location);
}
```

To perform a test using mocks generated with Mockito, the following steps have to be completed:

1. Declare and create the mock
2. Inject the mock
3. Define the behavior of the mock
4. Test
5. Validate the execution

The test class will be named SimpleStorageServiceTest3, and in the following test snippet, the findById method is tested and the preceding steps are underlined.

```
package com.apress.cems.mockito;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;

import static org.junit.jupiter.api.Assertions.*;
import org.mockito.junit.jupiter.MockitoExtension;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.when;
...

@ExtendWith(MockitoExtension.class)
public class SimpleStorageServiceTest3 {
    public static final Long STORAGE_ID = 1L;

    @Mock //Creates mock instance of the field it annotates
    (1)private StorageRepo mockRepo;

     (2)@InjectMocks
    private SimpleStorageService storageService;

    @Test
    public void findByIdPositive() {
        Storage storage = new Storage();
        storage.setId(STORAGE_ID);
        (3) when(mockRepo.findById(any(Long.class))).thenReturn(storage);

        (4) Storage result = storageService.findById(STORAGE_ID);
        (5) assertAll(
                () -> assertNotNull(result),
                () -> assertEquals(storage.getId(), result.getId())
        );
   }
}
```

The `org.mockito.junit.jupiter.MockitoExtension` class initializes mocks and handles strict stubbings. It is used with the `@ExtendWith` JUnit 5 annotation so that Junit 5 annotations are recognized within a Mockito context.

Defining the behavior of the mock is so intuitive that looking at the line marked with (3), you can directly figure out how the mock works: when the `findById` method is called on it with any `Long` argument, it will return the `storage` object declared previously. When multiple mocks are used or multiple methods of the same mock are called, then more `when` statements must be written.

The `@InjectMocks` has a behavior similar to the Spring IoC, because its role is to instantiate testing object instances and to try to inject fields annotated with `@Mock` or `@Spy` into private fields of the testing object.

Also, Mockito provides matchers that can replace any variables needed for mocking environment preparation. These matchers are static methods in the `org.mockito.ArgumentMatchers` class and can be called to replace any argument with a pseudo-value of the required type. For common types, the method names are prefixed with `any`, (`anyString()`, `anyLong()`, and others), while for every other object type, `any(Class<T>)` can be used. So the line

```
Mockito.when(criminalCaseRepo.findByLeadInvestigator(detective))
     .thenReturn(sample);
```

can be written using a matcher, and no *detective* variable is needed.

```
Mockito.when(criminalCaseRepo.findByLeadInvestigator(any(Detective.class)))
    .thenReturn(sample);
```

When a mock method is being called multiple times, `Mockito` also has the possibility to check how many times the method was called with a certain argument using a combination of `verify` and `times` methods. So a check for the number of calls can be added to the assertions block.

```
when(mockRepo.findById(any(Long.class))).thenReturn(storage);

Storage result = storageService.findById(STORAGE_ID);

verify(mockRepo, times(1)).findById(any(Long.class));
```

Quite practical, right? Probably this is the reason why the Spring team switched from EasyMock to Mockito. Of course, there is more than one way to test using Mockito mocks and those are depicted in the code attached to this book in classes SimpleStorageServiceTest and SimpleStorageServiceTest2. Make sure that you do not mix and match with your classpath, because if mockito-all and mockito-junit-jupiter are both used as dependencies, you will get an ugly exception as they both contain classes with the same name in packages with the same name. Because of some confusion when configuring my projects, I initially got the exception, and you can see the error message in the following snippet.

```
java.lang.NoSuchMethodError: org.mockito.Mockito.mockitoSession()
    Lorg/mockito/session/MockitoSessionBuilder;
  at org.mockito.junit.jupiter.MockitoExtension.
  beforeEach(MockitoExtension.java:112)
  at org.junit.jupiter.engine.descriptor.TestMethodTestDescriptor.lambda
     $invokeBeforeEachCallbacks$0(TestMethodTestDescriptor.java:129)
  at org.junit.jupiter.engine.execution.ThrowableCollector.execute(
      ThrowableCollector.java:40)
  at org.junit.jupiter.engine.descriptor.TestMethodTestDescriptor
      .invokeBeforeMethodsOrCallbacksUntilExceptionOccurs(
      TestMethodTestDescriptor.java:155)
```

PowerMock

PowerMock was born because sometimes code is not testable, perhaps because of bad design or because of some necessity. The following is a list of untestable elements.

- static methods
- classes with static initializers
- final classes and final methods; sometimes there is need for an insurance that the code will not be misused or to make sure that an object is constructed correctly
- private methods and fields

`PowerMock` is not that useful in a Spring application, since you will rarely find static elements there, but there is always legacy code and third-party libraries/frameworks that might require mocking, so it is only suitable to know that it is possible to do so and the tool to use for this. If you want to know more, go to their official site at `https://github.com/jayway/powermock`.

When it comes to testing applications, the technique and tools to use are defined by the complexity of the application, the experience of the developer, and ultimately legal limitations, because there are companies that are restricted to using software under a certain license. During a development career, you will probably get to use all the techniques and libraries/frameworks mentioned in this book. Favor mocks for nontrivial dependencies and nontrivial interfaces. Favor stubs when the interfaces are simple with repetitive behavior, but also because stubs can log and record their usage.

That said, you can switch over to the next section, which shows you how to use these things to test a Spring application.

Testing with Spring

Spring provides a module called `spring-test` that contains Spring JUnit 4 and JUnit 5 test support classes that can make testing Spring applications a manageable operation. The next two sections will cover how to test a Spring application using both JUnit versions.

Testing with Spring and JUnit 4

The core class of the `spring-test` for working with JUnit 4 is `org.springframework.test.context.junit4.SpringJUnit4ClassRunner` (lately known as its alias `SpringRunner`), which caches an `ApplicationContext` across JUnit 4 test methods. All the tests are run in the same context, using the same dependencies; thus, this is integration testing.

To define a test class for running in a Spring context, the following steps have to be done.

1. Annotate the test class with `@RunWith(SpringJUnit4ClassRunner.class)` or `@RunWith(SpringRunner.class)`.

2. Annotate the class with `@ContextConfiguration` to tell the runner class where the bean definitions come from

```
// bean definitions are provided by class ReposConfig
@ContextConfiguration(classes = {ReposConfig.class})
public class RepositoryTest {...}
// bean definitions are loaded from file all-config.xml
@ContextConfiguration(locations = {"classpath:spring/all-config.xml"})
public class RepositoryTest {...}
```

CC If `@ContextConfiguration` is used without any attributes defined, the default behavior of Spring is to search for a file named `{testClassName}-context.xml` in the same location as the test class and load bean definitions from there if found. So, not only this annotation can be used with XML configuration files, but it is specific to JUnit 4, in case it was not obvious.

The easiest way to previous affirmations is by writing a very simple test. Take a look at the following test class.

```
package com.apress.cems;
...
@RunWith(SpringRunner.class)
@ContextConfiguration
public class ContextLoadingTest {

    @Autowired
    ApplicationContext ctx;

    @Test
    public void testContext() {
      assertNotNull(ctx);
    }
}
```

The `ContextLoadingTest` has a single test method declared that checks if a Spring application context is created. Since the `@ContextConfiguration` annotation does not specify where that configuration is coming from, when the method is executed, the test fails and the console output is as follows.

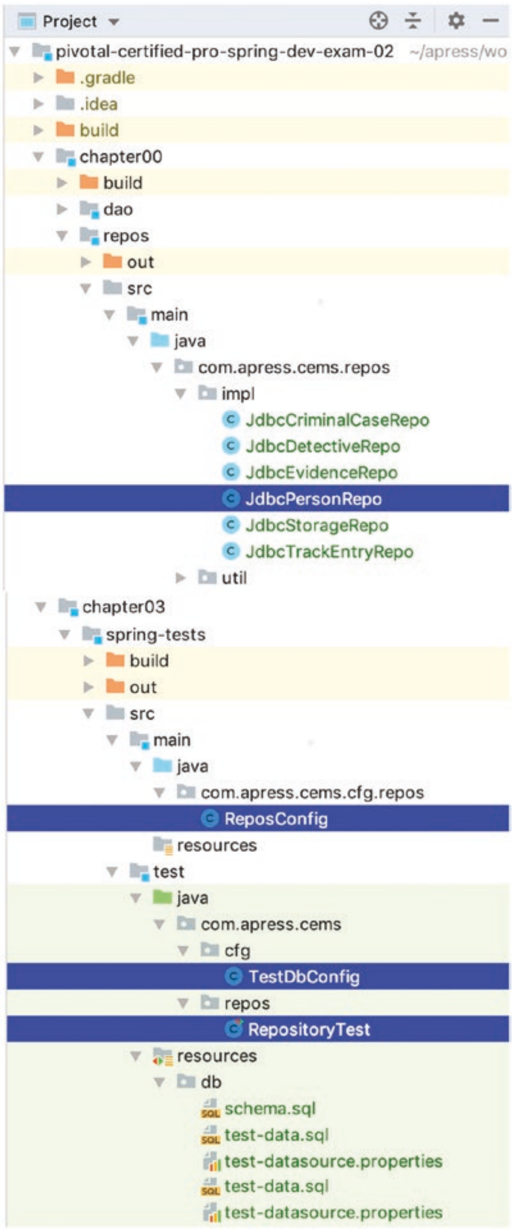
```
DEBUG o.s.t.c.j.SpringJUnit4ClassRunner -
SpringJUnit4ClassRunner constructor
        called with [class com.apress.cems.
        ContextLoadingTest]
...
DEBUG o.s.t.c.s.AbstractContextLoader - Did not detect
default resource location for
    test class [com.apress.cems.ContextLoadingTest]: class
    path resource
[com/apress/cems/ContextLoadingTest-context.xml] does not
exist
INFO  o.s.t.c.s.AbstractContextLoader - Could not detect
default resource locations
    for test class [com.apress.cems.ContextLoadingTest]:
      no resource found for suffixes {-context.xml}.
...
INFO  o.s.t.c.s.AnnotationConfigContextLoaderUtils - Could
not detect default
    configuration classes for test class [com.apress.cems.
    ContextLoadingTest]:
    ContextLoadingTest does not declare any static, non-
    private, non-final,
    nested classes annotated with @Configuration.
...
```

The log clearly shows that a Spring test context cannot be created because there is no configuration, of any kind provided for the Spring Container to do so.

---

3. Use `@Autowired` to inject beans to be tested.

This method of testing was introduced to test Java configuration-based applications in the previous chapter; but in this chapter, you will find out how you can manipulate the configuration so that tests can be run in a test context. The following code snippet tests the class `JdbcPersonRepo` in a Spring context defined by two Spring configuration classes. The test uses an H2 in-memory database; a bean of type `DataSource` is declared as a bean and injected in `JdbcPersonRepo` indirectly by first being injected in a bean of type `JdbcTemplate`. In Figure 3-4, the classes and files involved in defining the test context and running the test are depicted.



```
Project
pivotal-certified-pro-spring-dev-exam-02  ~/apress/wo
  .gradle
  .idea
  build
  chapter00
    build
    dao
    repos
      out
      src
        main
          java
            com.apress.cems.repos
              impl
                JdbcCriminalCaseRepo
                JdbcDetectiveRepo
                JdbcEvidenceRepo
                JdbcPersonRepo
                JdbcStorageRepo
                JdbcTrackEntryRepo
              util
  chapter03
    spring-tests
      build
      out
      src
        main
          java
            com.apress.cems.cfg.repos
              ReposConfig
          resources
        test
          java
            com.apress.cems
              cfg
                TestDbConfig
              repos
                RepositoryTest
          resources
            db
              schema.sql
              test-data.sql
              test-datasource.properties
            test-data.sql
            test-datasource.properties
```

Figure 3-4. Spring test context for testing JdbcPersonRepo

The ReposConfig is a configuration class for all the repository beans declared in the com.apress.cems.repos.

```
package com.apress.cems.cfg.repos;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;

import javax.sql.DataSource;

@Configuration
@ComponentScan(basePackages = {"com.apress.cems.repos"})
public class ReposConfig {

    @Autowired
    DataSource dataSource;

    @Bean
    public JdbcTemplate userJdbcTemplate() {
        return new JdbcTemplate(dataSource);
    }

}
```

The configuration decouples the database from the rest of the application, because a bean of this type is injected into this configuration using @Autowired. This means that a bean of this type is provided in a different configuration class, depending on the execution context. In this case, we are using a test context, and the dependency for executing this test is provided by the class TestDbConfig.

```
package com.apress.cems.cfg;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.PropertySource;
import org.springframework.context.support.
PropertySourcesPlaceholderConfigurer;
import org.springframework.core.env.Environment;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.jdbc.datasource.init.DatabasePopulator;
import org.springframework.jdbc.datasource.init.DatabasePopulatorUtils;
import org.springframework.jdbc.datasource.init.ResourceDatabasePopulator;

import org.springframework.core.io.Resource;
import javax.sql.DataSource;

@PropertySource("classpath:db/test-datasource.properties")
public class TestDbConfig {

    @Autowired
    Environment environment;

    @Bean
    public static PropertySourcesPlaceholderConfigurer
        propertySourcesPlaceholderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
    }

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName(environment.getProperty("db.
        driverClassName"));
        ds.setUrl(environment.getProperty("db.url"));
        ds.setUsername(environment.getProperty("db.username"));
        ds.setPassword(environment.getProperty("db.password"));
        DatabasePopulatorUtils.execute(databasePopulator(), ds);
        return ds;
    }
    @Value("classpath:db/schema.sql")
    private Resource schemaScript;

    @Value("classpath:db/test-data.sql")
    private Resource dataScript;

    private DatabasePopulator databasePopulator() {
        ResourceDatabasePopulator populator = new
        ResourceDatabasePopulator();
        populator.addScript(schemaScript);
        populator.addScript(dataScript);
        return populator;
    }

}
```

The `schema.sql` contains the SQL DDL script to create the `PERSON` table. The `test-data.sql` contains a DML script to insert two records in the `Person` table. A bean of type `PersonRepo` can manipulate entries in this table.

All the test methods will run in the same Spring test context. The object to be tested and its dependencies are created only once, when the application context is created, and they are used by all methods. The object to be tested is a bean, and it is injected in the class testing it using `@Autowired`. The following test methods test integration between the declared datasource and a bean of type `JdbcPersonRepo`.

```
package com.apress.cems.repos;

import com.apress.cems.cfg.TestDbConfig;
import com.apress.cems.cfg.repos.ReposConfig;
import com.apress.cems.dao.Person;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringRunner;
import java.util.Set;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {TestDbConfig.class, ReposConfig.class})
public class RepositoryTest{

     public static final Long PERSON_ID = 1L;;

    @Autowired
    PersonRepo personRepo;

    //positive test, we know that a Person with ID=1 exists
    @Test
    public void findByIdPositive() {
         Person person = personRepo.findById(PERSON_ID);
        assertNotNull(person);
        assertEquals("Sherlock", person.getFirstName());
    }

    //positive test, we know that a person matching the criteria exists
     @Test
    public void testFindByComplete(){
        Set<Person> personSet = personRepo.findByCompleteName("Sherlock",
        "Holmes");
        assertNotNull(personSet);
        assertEquals(1, personSet.size());
    }
}
```

The Spring test context is created using the two configuration classes, which will make sure the test `DataSource` dependency is the only one available to be injected into the `jdbcPersonRepo` bean. As you can see, a method annotated with `@Before` is not really necessary, because there is nothing else needed to have a context for the tests to execute in. Also, there is no longer any need to manipulate the `jdbcPersonRepo` object, since it is created, initialized, and injected by Spring. So all that is left for the developer to do is to inject the bean being tested and jump right to writing tests.

In Spring 3.1, the `org.springframework.test.context.support.AnnotationConfigContextLoader` class was introduced. This class loads bean definitions from static nested annotated classes. So a configuration class specific to a test scenario can be created in the body test class to create the beans needed to complete the test context. The class must be internal to the test class and static, and the `AnnotationConfigContextLoader` class must be used as a value for the `loader` attribute of the `@ContextConfiguration` annotation.

Please look at the following code snippet.

```
package com.ps.integration;
import
  org.springframework.test.context.support.AnnotationConfigContextLoader;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabase;
import org.springframework.jdbc.datasource.embedded.
EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
 ...
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(loader=AnnotationConfigContextLoader.class)
public class RepositoryTest2 {

    public static final Long PET_ID = 1L;

    @Configuration
    static class TestCtxConfig {

      @Bean
        PersonRepo jdbcPersonRepo() {
            return new JdbcPersonRepo(jdbcTemplate());
        }

        @Bean
        JdbcTemplate jdbcTemplate() {
            return new JdbcTemplate(dataSource());
        }
        @Bean
        public DataSource dataSource() {
            EmbeddedDatabaseBuilder builder = new
            EmbeddedDatabaseBuilder();
            EmbeddedDatabase db = builder
                    .setType(EmbeddedDatabaseType.H2)
                    .addScript("db/schema.sql")
                    .addScript("db/test-data.sql")
                    .build();
            return db;
        }
    }

     @Autowired
    PersonRepo personRepo;

    @Before
    public void setUp() {
        assertNotNull(personRepo);
    }

    @Test
    public void testFindByIdPositive() {
        Person person = personRepo.findById(PERSON_ID);
        assertNotNull(person);
        assertEquals("Sherlock", person.getFirstName());
    }

    @Test
    public void testFindByComplete() {
        Set<Person> personSet = personRepo.findByCompleteName("Sherlock",
        "Holmes");
        assertNotNull(personSet);
        assertEquals(1, personSet.size());
    }
}
```

As you can see, the dataSource bean is an embedded database that is created under the hood by Spring. This bean can be easily replaced by a stub or a mock if necessary.

To cover this specific scenario, concrete implementations for the repository classes must be used. Since their implementation is irrelevant to this chapter and is covered in Chapter 5, the full code will not be presented here. To manage Person objects, a repository class is needed. The most basic way to provide access to a database in Spring applications is to use a bean of type org.springframework.jdbc.core.JdbcTemplate, so repository classes will be created with a dependency of that type. The methods querying the database from this bean require an object that can map database records to entities, which is why each repository class should define an internal class (or external) implementing the Spring-specific mapping interface org.springframework.jdbc.core.RowMapper<T>.

```
//abstract repository base class
 package com.ps.repos.impl;
 import org.springframework.jdbc.core.JdbcTemplate;
 ...
public class JdbcAbstractRepo<T extends AbstractEntity>
     implements AbstractRepo<T> {

    protected JdbcTemplate jdbcTemplate;

    public JdbcAbstractRepo(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
 ...
 }

@Repository
public class JdbcPersonRepo extends
    JdbcAbstractRepo<Person> implements PersonRepo {
    private RowMapper<Person> rowMapper
         = new PersonRowMapper();

    @Autowired
    public JdbcPersonRepo(JdbcTemplate jdbcTemplate) {
        super(jdbcTemplate);
    }
...
```

```
    @Override
    public Person findById(Long id) {
        String sql = "select id, username, firstname, lastname,
             password, hiringdate from person where id= ?";
        return jdbcTemplate.queryForObject(sql, rowMapper, id);
    }
...
    // the DB record to entity mapper class
   class PersonRowMapper implements RowMapper<Person> {
    @Override
    public Person mapRow(ResultSet rs, int rowNum) throws SQLException {
        Long id = rs.getLong("ID");
        String username = rs.getString("USERNAME");
        String firstname = rs.getString("FIRSTNAME");
        String lastname = rs.getString("LASTNAME");
        String password = rs.getString("PASSWORD");
        String hiringDate = rs.getString("HIRINGDATE");

        Person person = new Person();
        person.setId(id);
        person.setUsername(username);
        person.setFirstName(firstname);
        person.setLastName(lastname);
        person.setPassword(password);
        person.setHiringDate(toDate(hiringDate));
        return person;
    }
  }
}
```

A configuration class that provides a mock that replaces the `jdbcTemplate` could be used, thus this will allow unit testing within a Spring context. The mock is created by a static method in the `org.mockito.Mockito` class called `mock(Class<T>)`. In the following example, the `jdbcPersonRepo` bean is tested in isolation within a Spring Context created using the `MockCtxConfig` nested class.

```
package com.apress.cems.mock;

import com.apress.cems.dao.Person;
import com.apress.cems.repos.PersonRepo;
import com.apress.cems.repos.impl.JdbcPersonRepo;
import com.apress.cems.repos.util.PersonRowMapper;
import org.junit.ClassRule;
import org.junit.Rule;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mockito;
import org.mockito.runners.MockitoJUnitRunner;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.rules.SpringClassRule;
import org.springframework.test.context.junit4.rules.SpringMethodRule;
import org.springframework.test.context.support.
AnnotationConfigContextLoader;

import static org.junit.Assert.assertNotNull;
import static org.mockito.Matchers.*;
import static org.mockito.Mockito.mock;
...
@RunWith(MockitoJUnitRunner.class)
@ContextConfiguration(loader = AnnotationConfigContextLoader.class)
public class SpringUnitTest {
    public static final Long PERSON_ID = 1L;

    @ClassRule
    public static final SpringClassRule SPRING_CLASS_RULE = new
    SpringClassRule();

    @Rule
    public final SpringMethodRule springMethodRule = new
    SpringMethodRule();
    @Autowired
    PersonRepo personRepo;

    // mocking the database
    @Autowired
    JdbcTemplate jdbcTemplate;

    @Test
    public void testFindByIdPositive() {
        Mockito.when(jdbcTemplate.queryForObject(was
            any(PersonRowMapper.class), anyLong())).thenReturn(new Person());

        Person person = personRepo.findById(PERSON_ID);
        assertNotNull(person);
    }

    @Configuration
    static class MockCtxConfig {
        @Bean
        JdbcTemplate jdbcTemplate() {
            return mock(JdbcTemplate.class);
        }

        @Bean
        PersonRepo jdbcPersonRepo() {
            return new JdbcPersonRepo(jdbcTemplate());
        }

    }
}
```

To be executed correctly, the test class must respect the following two rules.

- It must be executed with `MockitoJUnitRunner` runner class.
- Two Spring-specific components must be added to the test class so the context can be loaded.

```
@ClassRule
public static final SpringClassRule SPRING_CLASS_RULE = new
SpringClassRule();
```

```
@Rule
public final SpringMethodRule springMethodRule = new
SpringMethodRule();
```

`org.springframework.test.context.junit4.rules.SpringClassRule` is an implementation of JUnit `org.junit.rules.TestRule` that supports class-level features of the Spring TestContext Framework. The `@ClassRule` annotation is used on fields that reference rules or methods that return them. Fields must be public, static and a subtype of `org.junit.rules.TestRule`. Methods must be public, static and return an implementation of `TestRule`.

The `org.springframework.test.context.junit4.rules.SpringMethodRule` is an implementation of `JUnit org.junit.rules`. MethodRule that supports instance-level and method-level features of the Spring TestContext Framework. The `@Rule` annotation is used on fields that reference rules or methods that return them. Fields must be public and not static and an implementation of `TestRule` or `MethodRule`; methods must be public and not static and return an implementation of `TestRule` or `MethodRule`.

If you haven't figured out yet what happens in the previous example, I'll explain. A test context was created containing all the beans in the `MockCtxConfig` configuration class. One of the beans is the `jdbcTemplate`. It is a mock defined as a bean and is injected automatically by Spring where needed (in the repository bean).

Testing with Spring and JUnit 5

After the launch of JUnit 5, a new package was added to the `spring-test` library: `org.springframework.test.context.junit.jupiter`. This package contains a set of special classes and annotation that write JUnit 5 tests that run in a Spring context.

`@RunWith(SpringRunner.class)` must be dropped since the annotation and the runner class are specific to JUnit 4. So to migrate our tests to JUnit 5, they have to be replaced with their JUnit 5 equivalents which are: `@ExtendWith(SpringExtension.class)`. The `SpringExtension` class integrates the Spring test context into the Junit 5 test context.

@ContextConfiguration is Spring-test specific, so it will work with JUnit 5 as well. All that is left is to replace the annotations inside the class. So the RepositoryTest for execution with JUnit 5 will look like the following.

```
package com.apress.cems.repos;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
...

@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = {TestDbConfig.class, ReposConfig.class})
public class RepositoryTest {

    public static final Long PERSON_ID = 1L;


    @Autowired
    PersonRepo personRepo;

    @BeforeEach
    public void setUp(){
        assertNotNull(personRepo);
    }

    @Test
    public void testFindByIdPositive(){
        Person person = personRepo.findById(PERSON_ID);
        assertNotNull(person);
        assertEquals("Sherlock", person.getFirstName());
    }
}
```

If you want something simpler, you can use the @SpringJUnitConfig[6] annotation, which is a composed annotation that combines @ExtendWith(SpringExtension.class) and @ContextConfiguration.

```
package com.apress.cems.repos;

import org.springframework.test.context.junit.jupiter.SpringJUnitConfig;
...

@SpringJUnitConfig(classes = {TestDbConfig.class, ReposConfig.class})
public class RepositoryTest {

    public static final Long PERSON_ID = 1L;

    @Autowired
    PersonRepo personRepo;

    @BeforeEach
    public void setUp(){
        assertNotNull(personRepo);
    }

    @Test
    public void testFindByIdPositive(){
        Person person = personRepo.findById(PERSON_ID);
        assertNotNull(person);
        assertEquals("Sherlock", person.getFirstName());
    }
}
```

The same goes for Spring test classes where mocks are used, for Mockito there is an extension that can be used. If @RunWith(MockitoJUnitRunner.class) is replaced with @ExtendWith(MockitoExtension.class), the specific test annotations are migrated to JUnit 5. If the proper JUnit 5 dependencies are added in the classpath, then the SpringUnitTest class is executed as intended, but on a Jupiter test engine. (Test samples are provided for you in the GitHub repository for this book.)

[6]The code of SpringJUnitConfig annotation is available at https://github.com/spring-projects/spring-framework/blob/master/spring-test/src/main/java/org/springframework/test/context/junit/jupiter/SpringJUnitConfig.java

This is all that there is. The most important thing that there is to know here is how to create a Spring shared context for your tests to run in. How complex the tests are and how they are executed really depends on your knowledge and understanding of the additional testing libraries introduced so far. But that is not a topic for this book, so let's continue with Spring-related testing details, shall we?

A Few Other Useful Spring Test Annotations

So far in our tests, all the test data was declared in a single file named `test-data.sql`. But what if we need to execute tests on different data sets, and we do not want to lose time initializing the database all at once before executing the tests? Well, there are a couple of Spring useful annotations that can come in handy: the `@Sql` family: `@Sql`, `@SqlConfig`, and `@SqlGroup`. Let's see how they can be used.

```
package com.apress.cems.testrepos;

 import org.springframework.test.context.jdbc.Sql;
 import org.springframework.test.context.jdbc.SqlConfig;
 ...

@SpringJUnitConfig(classes = RepositoryTest3.TestCtxConfig.class)
public class RepositoryTest3 {
 public static final Long PERSON_ID = 1L;

    @Autowired
    PersonRepo personRepo;

    @BeforeEach
    public void setUp() {
        assertNotNull(personRepo);
    }

    @Test
    @Sql(
            scripts = "classpath:db/test-data-one.sql",
            config = @SqlConfig(commentPrefix = "`", separator = "@@")
     )
```

```
public void testFindByIdPositive() {
    Person person = personRepo.findById(PERSON_ID);
    assertNotNull(person);
    assertEquals("Sherlock", person.getFirstName());
}

@Test
@Sql({"classpath:db/test-data-two.sql"})
public void testFindByComplete() {
    Set<Person> personSet = personRepo.findByCompleteName("Irene",
    "Adler");
    assertNotNull(personSet);
    assertEquals(1, personSet.size());
}

@Configuration
static class TestCtxConfig {
    @Bean
    PersonRepo jdbcPersonRepo() {
        return new JdbcPersonRepo(jdbcTemplate());
    }

    @Bean
    JdbcTemplate jdbcTemplate() {
        return new JdbcTemplate(dataSource());
    }

    @Bean
    public DataSource dataSource() {
        EmbeddedDatabaseBuilder builder = new
        EmbeddedDatabaseBuilder();
        EmbeddedDatabase db = builder
                .setType(EmbeddedDatabaseType.H2)
                .generateUniqueName(true)
                .addScript("db/schema.sql")
                .build();
```

```
            return db;
        }
    }

}
```

The `@Sql` annotation is used on methods annotated with `@Test`, which will cause the script or statement referred by it to be executed in the test database before the test is executed because it is the default setting.

The `@Sql` annotation provides an attribute named `executionPhase` that specify when a script or statement should be executed. Its default value is set to `Sql.ExecutionPhase.BEFORE_TEST_METHOD` (a member of the `ExecutionPhase` declared in the body of the `@Sql` annotation), so the script or statement is executed before the test method is executed, but it can be changed to be executed after it by setting its value to `Sql.ExecutionPhase.AFTER_TEST_METHOD`.

In the previous example, the `@Sql` was used to refer to files on the classpath containing SQL statements, but statements can be specified as arguments for the annotation directly. For example, `@Sql({"classpath:db/test-data-two.sql"})` is equivalent to

```
@Sql(statements = {"INSERT INTO PERSON(ID, USERNAME, FIRSTNAME,
      LASTNAME, PASSWORD, HIRINGDATE, VERSION, CREATED_AT,
      MODIFIED_AT)
    VALUES (2, 'irene.adler', 'Irene', 'Adler', 'id123ds', '1990-08-18', 1,
           '1990-07-18', '1998-01-18');"})
```

The `@Sql` annotation can also be used at the class level, and in this case, the script is executed before each test method (actually, before the method annotated with `@BeforeEach` if such method exists), so you might need another script to clean up the database after each test method. This can be done easily with `@Sql`, because executions of SQL scripts can be linked to test execution phases using the `executionPhase` attribute. Take a look at the following example.

```
package com.apress.cems.testrepos;
...
import org.springframework.test.context.jdbc.Sql;
import org.springframework.test.context.jdbc.SqlConfig;
...
```

```
@Sql(
        scripts = "classpath:db/test-data.sql",
        config = @SqlConfig(commentPrefix = "`", separator = "@@")
)
@Sql(statements = "DELETE FROM PERSON",
       executionPhase = Sql.ExecutionPhase.AFTER_TEST_METHOD)
@SpringJUnitConfig(classes = RepositoryTest5.TestCtxConfig.class)
public class RepositoryTest5 {
    static final Long PERSON_ID = 1L;

    @Autowired
    PersonRepo personRepo;

    @BeforeEach
    void setUp() {
        assertNotNull(personRepo);
    }

    @Test
    void testFindByIdPositive() {
        Person person = personRepo.findById(PERSON_ID);
        assertNotNull(person);
        assertEquals("Sherlock", person.getFirstName());
    }

    @Test
    void testFindAll(){
        Set<Person> all = personRepo.findAll();
        assertAll(
                () -> assertNotNull(all),
                () -> assertEquals(2, all.size())
        );
    }

    @Configuration
    static class TestCtxConfig {
    ... //same as the previous code listing
    }

}
```

When the previous code is executed, both tests pass because both are executed on a database that is initialized with the content declared in the `test-data.sql` file and cleaned right after the test execution method ends. And, we can do something even smarter: we can use the class level `Sql` annotation to create the database structure, or only the table that is required in the test context.

In the next code sample, we annotate the `RepositoryTest5` class with an `Sql` annotation that is declared to be executed before each test method.

```
package com.apress.cems.testrepos;
...
import org.springframework.test.context.jdbc.Sql;
import org.springframework.test.context.jdbc.SqlConfig;
...
@Sql(
        scripts = {"classpath:db/person-schema.sql",
            "classpath:db/test-data.sql"},
        executionPhase = Sql.ExecutionPhase.BEFORE_TEST_METHOD

)
@SpringJUnitConfig(classes = RepositoryTest5.TestCtxConfig.class)
public class RepositoryTest5 {
    ... // same code as in previous code listing
}
```

If we run this class, the test methods will pass as before, because the `person-schema.sql` script starts with a declaration to drop the PERSON table if it exists. So each test method is executed on a new version of the table.

The `@SqlConfig` annotation provides extra information about the syntax used in the SQL script file provided as argument to an `@Sql` annotation.

The `SqlGroup` groups multiple `Sql` annotations, which use different scripts/statements. You can use it on test classes or methods. The syntax should be similar to the one in the next code snippet.

```
@SqlGroup({
    @Sql(scripts = "classpath:db/test-data-one.sql",
             config = @SqlConfig(commentPrefix = "`")),
    @Sql({"classpath:db/test-data-two.sql"})
})
```

When the application is more complex and transactions are in place, integration testing might require test methods to be executed within a transactional context. This means test methods will be annotated with `@Transactional` (from package `org.springframework.transaction.annotation`). And because controlling the context matters in a test environment, Spring Test provides annotations to use to mark methods for executing before (`@BeforeTransaction`) and after a transaction is closed (`@AfterTransaction`).

There is also an annotation used to require a rollback (`@Rollback`) of the last transaction. After a test has been executed, it is quite useful to preserve the state of the test database so that other tests are executed on the same set of data. This is the reason why rolling back a transaction after the execution of a test method is the default behavior. But in some integration tests, you might need a "dirty" database, and in those cases, you can annotate your test methods with `@Rollback("false")`. There is also the case when you might need a transactional test method to be committed after the test method has completed; the `@Commit` annotation is suitable in that case.

All of these annotations are used in Chapter 5**,** when integration tests using persistence context is created.

Using Profiles

In the previous chapter, out of necessity, using profiles for customizing an application behavior was introduced. In Spring version 3.1, the `@Profile` annotation became available. With this annotation, classes become eligible for registration when one or more profiles are activated at runtime. Spring profiles have the same purpose as Maven profiles, but they are much easier to configure and are not linked to an application's builder tool. Different environments require different configurations, and much care should be used during development so that components are decoupled enough, and they can be swapped depending on the context in which processes are executed.

Spring profiles help considerably in this case. For example, during development, tests are run on development machines, and a database is not really needed; or if one is needed, an in-memory simple and fast implementation should be used. This can be set up by creating a test datasource configuration file that is used only when the development profile is active. The datasource classes for production and test environments are depicted in the following code snippet.

```
//production dataSource
 package com.ps.config;
 import org.springframework.context.annotation.Profile;
 ...
@Configuration
@PropertySource("classpath:db/datasource.properties")
@Profile("prod")
public class ProdDbConfig {

    @Bean("connectionProperties")
    Properties connectionProperties(){
        try {
            return PropertiesLoaderUtils.loadProperties(
                    new ClassPathResource("db/prod-datasource.
                    properties"));
        } catch (IOException e) {
   throw new ConfigurationException("Could not retrieve connection
   properties!", e);
        }
    }

    @Bean
    public DataSource dataSource() {
        try {
            OracleDataSource ds = new OracleDataSource();
            ds.setConnectionProperties(connectionProperties());
            return ds;
        } catch (SQLException e) {
    throw new ConfigurationException("Could not configure Oracle
    database!", e);
        }
    }
}

// development dataSource
@Configuration
@Profile("dev")
```

```
public class TestDbConfig {

    @Bean
    public DataSource dataSource() {
        EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
        EmbeddedDatabase db = builder
                .setType(EmbeddedDatabaseType.H2)
                .generateUniqueName(true)
                .addScript("db/schema.sql")
                .addScript("db/test-data.sql")
                .build();
        return db;
    }
}
```

In the preceding sample, we have two configuration classes, each declaring a bean named `dataSource`, each bean specific to a different environment. The profiles are named simply *prod*, for the production environment, and *dev*, for the development environment. In the test class, we can activate the development profile by annotating the test class with `@ActiveProfiles` annotation and giving the profile name as argument. Thus, in the following test context, only the beans defined in classes annotated with `@Profile("dev")` will be created and injected.

The test class is depicted in the following code snippet.

```
import org.springframework.test.context.ActiveProfiles;
 ...
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = {TestDbConfig.class,
    ProdDbConfig, ReposConfig.class})
@ActiveProfiles("dev")
public class RepositoryTest {

    public static final Long PERSON_ID = 1L;

    @Autowired
    PersonRepo personRepo;

    //positive test, we know that a Person with ID=1 exists
 @Test
    public void testFindByIdPositive(){
        Person person = personRepo.findById(PERSON_ID);
        assertNotNull(person);
        assertEquals("Sherlock", person.getFirstName());
    }
}
```

The `ProdDbConfig` looks out of place in the previous example, but it is there to prove that only beans specific to the `dev` profile are created.

The advantage of using profiles become obvious when the beans within the application context must be replaced in the test context, but the configuration does not allow that because it is not decoupled enough. In the previous code sample, the datasource is configured in its own file. But what if the datasource is declared in a configuration class named `AllConfig` that declares all other application beans as well, either by using `@Bean` annotation or by using component scanning? If the configuration is not decoupled, how do we override the bean declarations we are interested in within a test context? Well, easy really, the `@Profile` annotation can be used directly on bean declarations as well. We have to make sure that the `dataSource` bean is declared as specific to the `prod` profile by annotating it with `@Profile("prod")`. Yes, the `@Profile` annotation can be used both at the class level and at the method level.

```
package com.apress.cems.cfg;
 import org.springframework.context.annotation.Profile
 ....

@Configuration
@ComponentScan(basePackages = {"com.apress.cems.repos"})
public class AllConfig {

    @Bean
    public JdbcTemplate userJdbcTemplate() {
        return new JdbcTemplate(dataSource());
    }
    @Bean("connectionProperties")
    Properties connectionProperties(){
        try {
            return PropertiesLoaderUtils.loadProperties(
                    new ClassPathResource("db/prod-datasource.
                    properties"));
        } catch (IOException e) {
            throw new ConfigurationException(
               "Could not retrieve connection properties!", e);
        }
    }

    @Profile("prod")
    @Bean
    public DataSource dataSource() {
        try {
            OracleDataSource ds = new OracleDataSource();
            ds.setConnectionProperties(connectionProperties());
            return ds;
        } catch (SQLException e) {
            throw new ConfigurationException("Could not configure Oracle
            database!", e);
        }
    }
}
```

After this is done, we can create a test context using this class and the configuration class containing the dataSource bean specific to the dev environment, the TestDbConfig class and activate the dev profile.

```
@ContextConfiguration(classes = {TestDbConfig.class, AllConfig.class})
@ActiveProfiles("dev")
public class RepositoryTest {
...
}
```

By annotating the test class `@ActiveProfiles("dev")`, the development profile is activated, so when the context is created, bean definitions are picked up from the configuration class (or files) specified by the `@ContextConfiguration` annotation and all the configuration classes annotated with `@Profile("dev")`. Thus, when running the test, the in-memory database will be used, making the execution fast and practical.

Any bean that is not specific to a profile is created and added to the application context, regardless if there is an active profile at runtime or not. There is a test class named `SampleProfileConfigTest` in the `chapter03/spring-profiles` project. It contains an internal Spring configuration class, which declares a few beans, some of them specific to profiles. You can play with it; activate and deactivate profiles to check the beans created in each scenario.

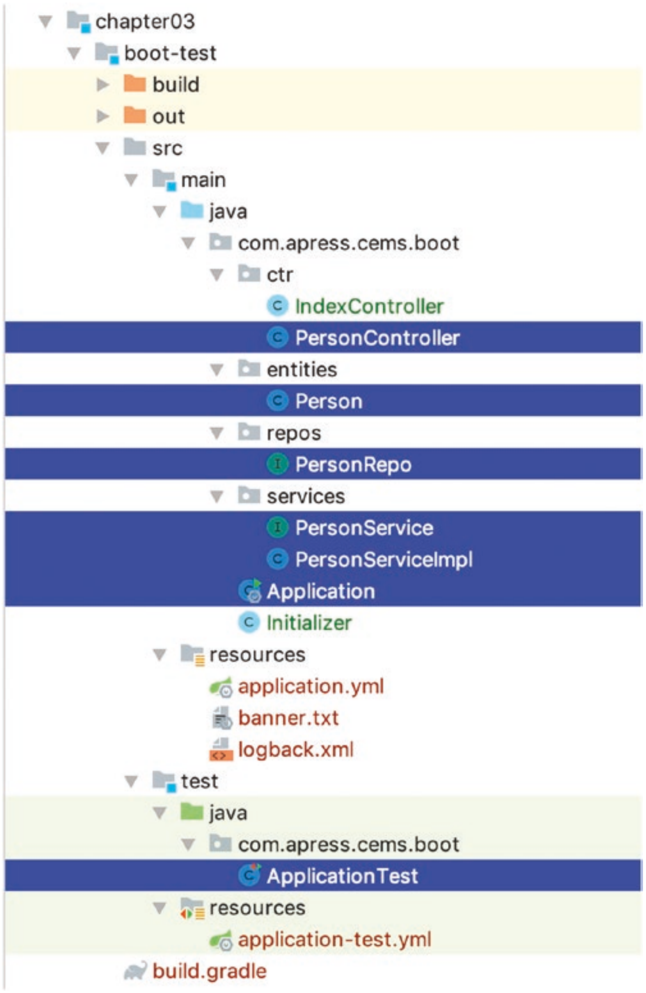
In conclusion, using the Spring-provided test classes to test a Spring application is definitely easier than not doing it, since no external container is needed to define the context in which the tests run. If the configuration is decoupled enough, pieces of it can be shared between the test and production environment. For basic unit testing, Spring is not needed, but to implement proper integration testing, the ability to set up a test context in record time is surely useful.

Spring Boot Testing

Spring Boot was introduced in the previous chapter. Just in case you skipped that one, let me remind you that Spring Boot is a preconfigured set of frameworks/technologies designed to reduce boilerplate configuration (infrastructure) and provide a quick way to have a Spring web application up and running. As expected, Spring Boot includes a library that makes testing of Spring Boot Application super easy.

Adding the library `spring-boot-starter-test` as a dependency adds multiple testing libraries to the classpath: JUnit 4 and 5, Hamcrest, Mockito, AssertJ. In this chapter, a small but complete Spring Boot application is created to manage `Person` instances. We have an Oracle database, a special JPA management interface named `PersonRepo` to manipulate `Person` instances, and a class named `PersonServiceImpl` (that implements interface `PersonService`). A bean of type PersonService uses a bean of type `PersonRepo` to persist changes to the database. There is also a bean of type `PersonController` that receives web requests and forwards specific actions to a bean of type PersonService. The connection between these beans makes for a perfect application to write unit and integration tests for.

Not all classes are depicted in Figure 3-5, because some of them contain implementations specific to other chapters. Figure 3-5 depicts all the classes of interest for the application.



```
▼ chapter03
  ▼ boot-test
    ▶ build
    ▶ out
    ▼ src
      ▼ main
        ▼ java
          ▼ com.apress.cems.boot
            ▼ ctr
                IndexController
                PersonController
            ▼ entities
                Person
            ▼ repos
                PersonRepo
            ▼ services
                PersonService
                PersonServiceImpl
              Application
              Initializer
        ▼ resources
            application.yml
            banner.txt
            logback.xml
      ▼ test
        ▼ java
          ▼ com.apress.cems.boot
              ApplicationTest
        ▼ resources
            application-test.yml
    build.gradle
```

Figure 3-5. *Test-driven development logical schema*

The Spring Boot application class annotated with `@SpringBootApplication` also enables component scanning on package `com.apress.cems.boot` and its subpackages, so all the bean declarations will be discovered and all beans will be created.

```
package com.apress.cems.boot;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication
public class Application {

    private static Logger logger = LoggerFactory.getLogger(Application.class);

    public static void main(String... args) {
        ConfigurableApplicationContext ctx =
            SpringApplication.run(Application.class, args);
        ctx.registerShutdownHook();
        logger.info("Application Started ...");
    }
}
```

When executing this class, (if you have Oracle Database installed locally, or in a Docker container[7] and a database user as specified in the properties file) if no exception appears in the console, it means that the context was initialized correctly. You can confirm that the application is up and running by accessing `http://localhost:8080/` in your browser. You should see the text *This works!*.[8] The details of how everything works under the hood will be clear to you by the end of the book. For now, let's focus on testing this application.

To test this application, we need to create a Spring Boot context based on the `Application` class. By default, any test class annotated with `@SpringBootTest` in the package `com.apress.cems.boot` and subpackages looks for an application class annotated with `@SpringBootApplication` to use for bootstrapping a test context. The test context can be customized either by specific annotations configurations of the

[7]Instructions for installing Docker and setting up a Docker container with an Oracle database are provided together with the sources associated to the chapter.

[8]This text is returned by a method of a web specialized bean of type *IndexController*.

boot test class, or by properties declared in specific test properties files (like `src/test/resources/application.yml` or `src/test/resources/application.properties`).

The awesome part is that Spring Boot has embraced JUnit 5, because if we were to look into the `@SpringBootTest` code, we see that this annotation is meta-annotated with `@ExtendWith(SpringExtension.class)`. This means that tests annotated with `@SpringBootTest` are automatically picked up and executed by the JUnit 5 Jupiter engine. Let's quickly write a simple test to check that the `PersonService` class is implemented properly.

```
package com.apress.cems.boot;

import com.apress.cems.boot.entities.Person;
import com.apress.cems.boot.services.PersonService;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(webEnvironment =
    SpringBootTest.WebEnvironment.NONE)
public class ApplicationTest {

    @Autowired
    private PersonService personService;

    @Test
    public void testSavePerson(){
        Person person = new Person();
        person.setFirstName("Irene");
        person.setLastName("Adler");
        personService.save(person);

        Assertions.assertEquals(personService.count(), 2);
    }
}
```

The previous class tests the ability of the `personService` bean to create an instance of type `Person`. The reason that we expect two entries is because one of them was created by a class named `Initializer`, which is declared within the application.

The personService bean, calls methods of the personRepo bean which makes use of an H2 data base. The first to beans are production specific beans, the database is declared in the test configuration file. This is an integration test that covers the application's service and DAO layers. As you can see, we do not need a full web environment booted up to run our test, which is why the SpringBootTest.WebEnvironment.NONE argument is used.

Earlier, the @Transactional and @Rollback annotations were mentioned. Since we now have a transactional service, we can actually make use of them. Let's modify the previous test class to roll back all changes done by the testSavePerson() and add another test that counts the entries in the test database to make sure their number is unchanged. Sure, the @Rollback annotation is redundant since the default test behavior is to rollback anyway, but for teaching purposes, it will be used anyway. Examples that are more relevant are in Chapter 4.

```
package com.apress.cems.boot;

import com.apress.cems.boot.entities.Person;
import com.apress.cems.boot.services.PersonService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import import org.springframework.transaction.annotation.Transactional;
import org.springframework.test.annotation.Rollback;

import static org.junit.jupiter.api.Assertions.assertEquals;

@SpringBootTest(webEnvironment =
           SpringBootTest.WebEnvironment.NONE)
public class ApplicationTest {

    @Autowired
    private PersonService personService;

    @Rollback   @Transactional
    @Test
    public void testSavePerson(){
        Person person = new Person();
        person.setFirstName("Irene");
        person.setLastName("Adler");
        personService.save(person);

       assertEquals(2, personService.count());
    }

    @Test
    void testCount(){
        //get all persons, list should only have one
        assertEquals( 1, personService.count());
    }
}
```

So, the `@SpringBootTest` is basically a `@ContextConfiguration` on steroids. Under the hood, when no loader is specified (like in the `@ContextConfiguration(loader = AnnotationConfigContextLoader.class)` examples from previous sections, a `SpringBootContextLoader` loads a Spring Boot configuration from a class annotated with `@SpringBootConfiguration` or any specialization of it, like `@SpringBootApplication`. It looks for properties on the test classpath to be injected in the `Environment` and registers a `TestRestTemplate` and `WebTestClient` beans, can be used to test web applications. You will become familiar with this in Chapter 6.

Tests that cover integration with the web layer can be written as well. And there are specific classes for simulating a web request and matcher to test the contents of the response that do not requiring starting up the web server. The `MockMvcBuilders` class is part of the spring-test module and provides a series of builder instances that simulate a call to web specialized beans called *controllers*. The `standaloneSetup(..)` method returns a builder of type `StandaloneMockMvcBuilder` to register one or more controller beans and configure the Spring MVC infrastructure programmatically. Other relevant examples are covered in Chapter 6.

```
package com.apress.cems.boot;

import com.apress.cems.boot.ctr.PersonController;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import static org.hamcrest.Matchers.*;
import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.*;

@SpringBootTest(webEnvironment =
     SpringBootTest.WebEnvironment.NONE)
public class ApplicationWebTest {

    @Autowired
    PersonController personController;

    private MockMvc mockMvc;

    @BeforeEach
    public void setup() {
         this.mockMvc = MockMvcBuilders.
             standaloneSetup(this.personController).build();
    }

    @Test
    public void testfindAll() throws Exception {

        mockMvc.perform(get(String.format("/person/all")))
                .andExpect(status().isOk())
                .andExpect(content().contentTypeCompatibleWith
                    (MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$", hasSize(1)))
                .andExpect(jsonPath("$..firstName",
                hasItem(is("Sherlock"))));

    }
}
```

In the previous test, we use an instance of type `MockMvc`, created by the `StandaloneMockMvcBuilder` instance to simulate a GET request to `http://localhost/person/all`. There is no need for a port in the test context because the `mockMvc` bean is automatically configured by Spring. Pretty neat, right?

`MockMvc` is designed very well, and the method making the request can be chained with the methods testing the contents of the response. In the previous test a bunch of matcher methods form the Hamcrest library are used too, for testing assumptions made regarding the number of items in the response and the value of the properties.

The `jsonPath(..)` method is a static method from the `MockMvcResultMatchers` class that can navigate the structure of a JSON text to extract values to test assumptions on. The two dots in the previous example are called *deep scan operators*[9] and are used to access the first JSON element in a JSON array.

The value returned by `personController` is `[{"firstName":"Sherlock","lastName":"Holmes","id":1}]` and since the test passes, the `jsonPath(..)` call works as expected.

Once a test class is annotated with `@SpringBootTest` a test context is set up and available, so the tests that can be written can make use of any library at your disposal. You will probably hear that if it is difficult to test an application, the test environment setup is too complex. This is a clear sign that the application's design is bad. The fact that testing Spring Boot applications is so easy is clearly a sign that Spring Boot promotes good design.

Summary

After reading this chapter you should possess enough knowledge to test a Spring application using unit and integration testing and you should be able to answer to the following questions.

- What is unit testing, and which frameworks are useful for writing unit tests in Java?
- What is a stub?
- What is a mock?

[9]If you are interested in more information about *jsonPath(..)*, check out the official repository: `https://github.com/json-path/JsonPath`

- What is integration testing?
- How does one set up a Spring Test Context?
- What is the purpose of the `@SpringBootTest` annotation?

Quick Quiz

Question 1. Given the following test class declaration,

```
@RunWith(SpringRunner.class)
@ContextConfiguration
public class SimpleTest {
 //test methods here
}
```

Which of the following affirmations is true? (Choose one.)

A. The tests will be executed on a JUnit 4 engine.

B. The tests will be executed on a JUnit 5 engine.

C. The tests will be executed on a JUnit engine found in the class path.

Question 2. Given the following unit test, what is missing from the class definition that prevents the test from being executed correctly? (Choose all that apply.)

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
...

public class SimplePersonServiceTest {

  @InjectMocks
  SimplePersonService personService;

  @Mock
  PersonRepo personRepo;
```

```
    @Test
    public void findById() {
        Mockito.when(petRepo.findById(1L)).thenReturn(new Pet());
        Pet pet = personService.findById(1L);
        assertNotNull(pet);
    }
}
```

A. The `@ExtendWith(MockitoExtension.class)` annotation.

B. Nothing. The test will be executed correctly, and it will pass.

C. A `setUp` method is missing with the following content.

```
MockitoAnnotations.initMocks(this);
```

Question 3. The `SpringJUnit4ClassRunner` class enhances a JUnit 4 test class with Spring context.

A. true

B. false

Question 4. What is the `@ContextConfiguration` used for? (Choose one.)

A. to load and configure a TestApplicationContext instance

B. to load and configure an ApplicationContext for integration testing

C. to inject beans used in unit testing

Question 5. What library is mandatory for writing unit tests for a Spring application? (Choose one.)

A. JUnit

B. spring-test

C. any mock generating library such as jMock, Mockito, or EasyMock

Question 6. Which of the affirmations describes unit testing best? (Choose one.)

A. Unit testing is a software practice of replacing dependencies with small units of mock code during application testing.

B. Unit testing is a software practice of testing how small individual units of code work together in the same context.

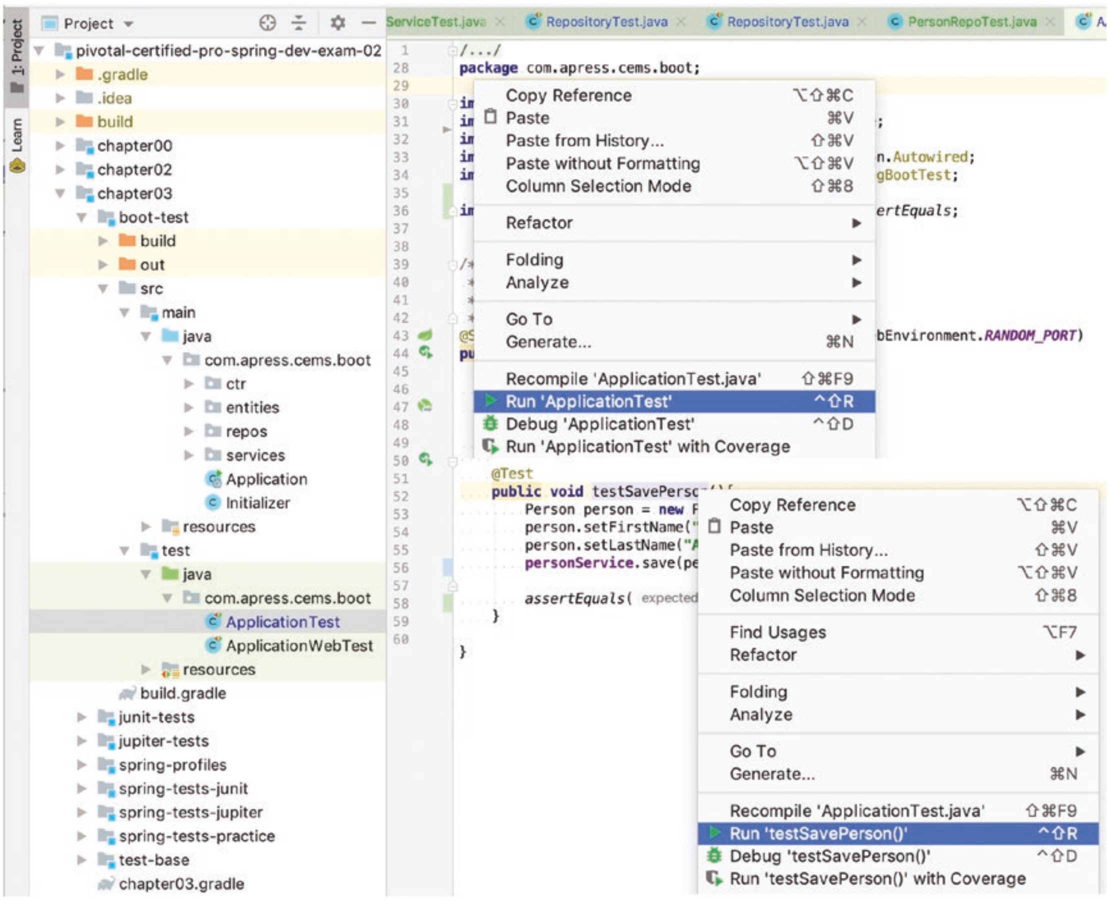
C. Unit testing is a software practice of testing small individual units of code in isolation.

Practical Exercise

The project to use to test your understanding of testing is `chapter03/spring-tests-practice`. This project contains part of the implementation depicted in the code snippets. The missing parts are marked with a TODO task and are visible in IntelliJ IDEA in the TODO view. There are six tasks for you to solve to test your acquired knowledge of testing, and they are focused on Mockito usage and Spring testing.

Tasks TODO 15 and 16 require you to complete two unit tests that test a `com.apress.cems.stub.SimpleCriminalCaseServiceTest` object using a stub. Task TODO 17 requires you to place the missing annotations in the `MockPersonServiceTest`.

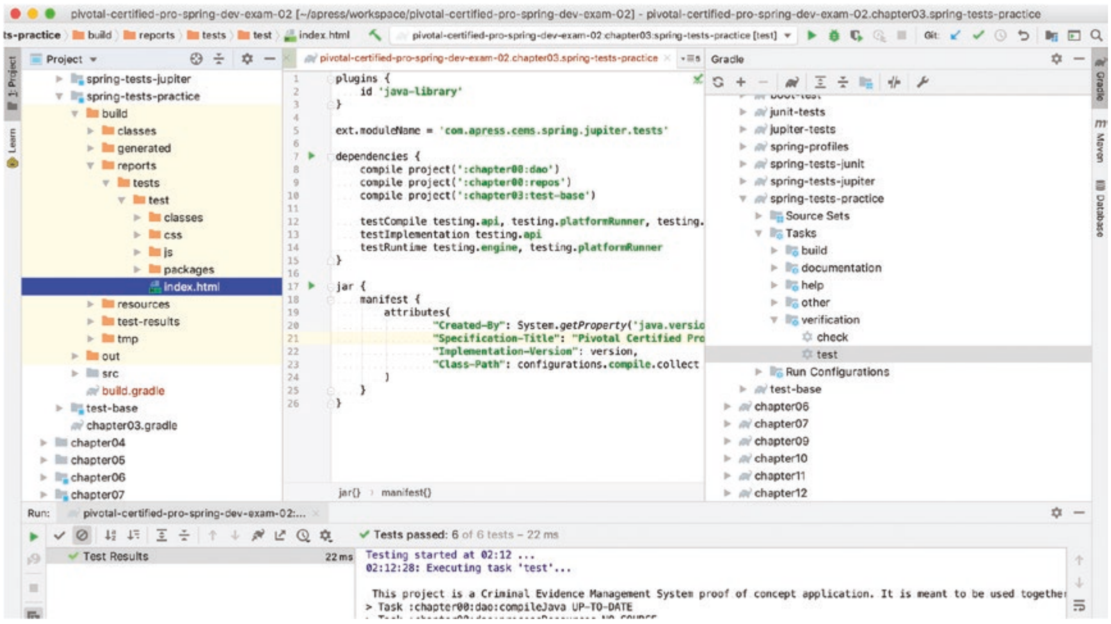
Tasks TODO 18 require you to complete the test class definitions so that the test cases can execute correctly. To run a test case, just click anywhere on the class content or on the class name in the project view and select the `Run '{TestClassName'}` option. If you want to run a single test, right-click, and from the menu select `Run '{TestMethodName}'`. The options are depicted in Figure 3-6 for a test class and a test method in the project. You can even run the tests using debug in case, you are interested in stopping the execution at specific times.



```
/.../
package com.apress.cems.boot;

...n.Autowired;
...gBootTest;
...ertEquals;

...bEnvironment.RANDOM_PORT)

    @Test
    public void testSavePerso...
        Person person = new P...
        person.setFirstName("...
        person.setLastName("A...
        personService.save(pe...

        assertEquals( expected...
    }
}
```

Figure 3-6. *How to run tests in IntelliJ IDEA*

After implementing all the solutions, you can run the Gradle `test` task to run all your tests. You should see a green check next to the test results, similar to what is depicted in Figure 3-7.



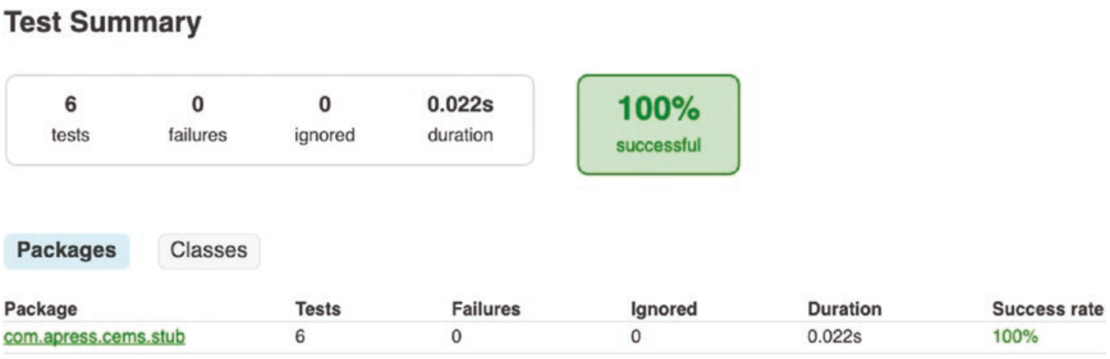
```
plugins {
    id 'java-library'
}

ext.moduleName = 'com.apress.cems.spring.jupiter.tests'

dependencies {
    compile project(':chapter00:dao')
    compile project(':chapter00:repos')
    compile project(':chapter03:test-base')

    testCompile testing.api, testing.platformRunner, testing.
    testImplementation testing.api
    testRuntime testing.engine, testing.platformRunner
}

jar {
    manifest {
        attributes(
                "Created-By": System.getProperty('java.versio
                "Specification-Title": "Pivotal Certified Pro
                "Implementation-Version": version,
                "Class-Path": configurations.compile.collect
        )
    }
}
```

```
Tests passed: 6 of 6 tests – 22 ms

Testing started at 02:12 ...
02:12:28: Executing task 'test'...

 This project is a Criminal Evidence Management System proof of concept application. It is meant to be used together
> Task :chapter00:dao:compileJava UP-TO-DATE
```

Figure 3-7. *Running all tests for a module in IntelliJ IDEA, using the Gradle test task*

In Figure 3-7, the Gradle task appears on the right in the Gradle view selected in gray. For any module it can be found under `Sub-ProjectName/Tasks/verification`.

On the left, the `index.html` file generated when Gradle tests are run is selected. You can open that file in the browser when writing the test to check percentages of the tests passing and detailed logs telling you why tests fail. If all is well at the end, you should see a page looking like the one depicted in Figure 3-8.



Test Summary

6	0	0	0.022s	100%
tests	failures	ignored	duration	successful

Packages Classes

Package	Tests	Failures	Ignored	Duration	Success rate
com.apress.cems.stub	6	0	0	0.022s	100%

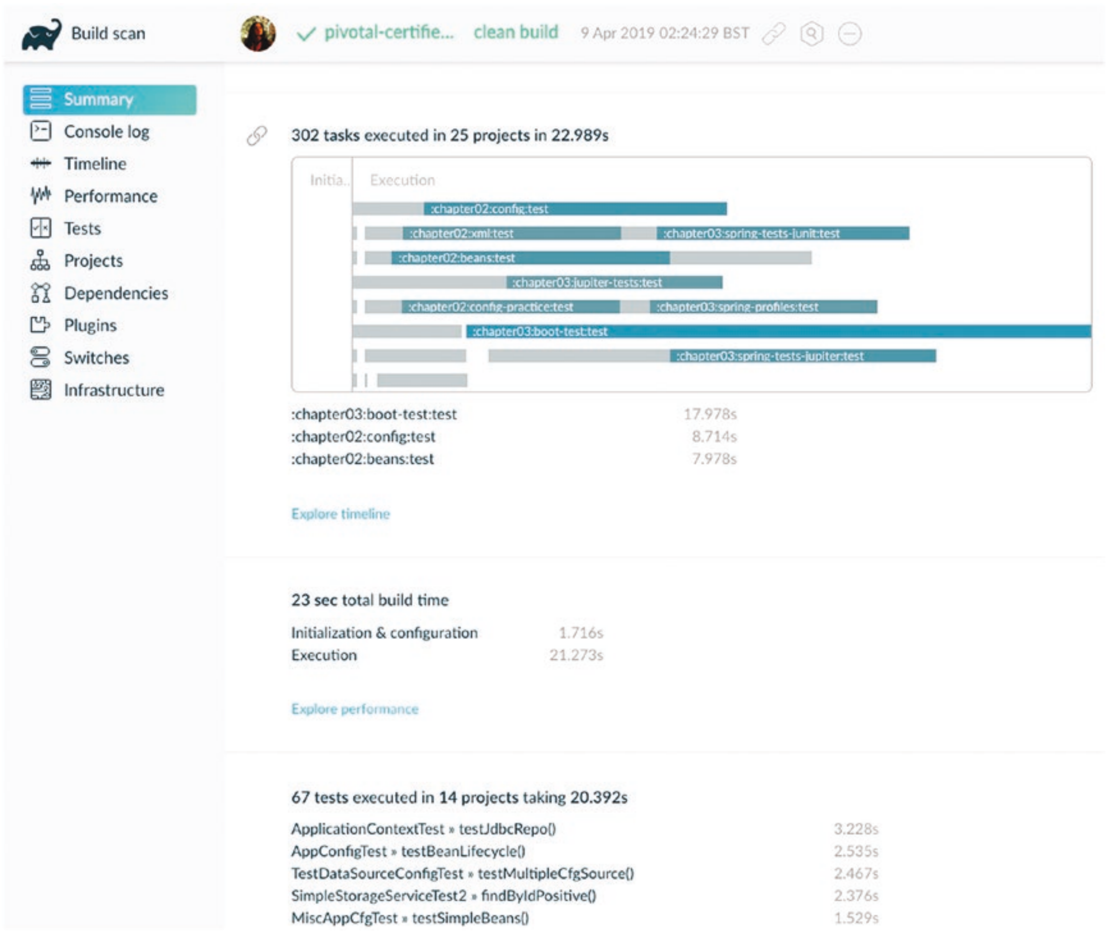
Figure 3-8. *Gradle web report after all tests have passed*

All other projects under `chapter03` contain code samples presented in this chapter including proposed solutions for the TODOS. Please compare yours with the provided ones to make sure all was implemented according to the specifications.

Also, because the project has grown, you can try doing a Gradle build scan using the `gradle clean build --scan` command and analyze its stability and execution time. Also, to build modules in parallel, you can add the `-Dorg.gradle.parallel=true` option.

```
gradle clean build --scan -Dorg.gradle.parallel=true
```

Tests in `*-practice` projects are either annotated with `@Disabled` or are missing annotations that would allow them being picked up from execution until you provide a solution and enable for them to be picked up. After agreeing with the Gradle terms, the build will end with a link that opens in your browser to a web page with the result analysis of the build. In Figure 3-9, you can see the result of my current build scan. If there are build failures, they will appear in the build scan. Feel free to click around to inspect the details of the build.



302 tasks executed in 25 projects in 22.989s

```
:chapter03:boot-test:test        17.978s
:chapter02:config:test            8.714s
:chapter02:beans:test             7.978s
```

23 sec total build time

```
Initialization & configuration    1.716s
Execution                        21.273s
```

67 tests executed in 14 projects taking 20.392s

```
ApplicationContextTest » testJdbcRepo()                3.228s
AppConfigTest » testBeanLifecycle()                    2.535s
TestDataSourceConfigTest » testMultipleCfgSource()     2.467s
SimpleStorageServiceTest2 » findByIdPositive()         2.376s
MiscAppCfgTest » testSimpleBeans()                     1.529s
```

Figure 3-9. *Official Gradle report about the CEMS project build*