<http://www.infoq.com/articles/testing-in-spring>

There are several other types of testing in addition to unit and integration tests. Following table lists different testing strategies and their objectives.

**Table 1. Java EE Testing Strategies**

|  |  |
| --- | --- |
| Test Strategy | Objective |
| Unit Testing | Test the application at class level to test all methods in each class. |
| Mock Objects | Mock Objects are used in Client and Service layers of the application to test the class methods without having to really connect to back-end database or other network resources. |
| ORM Testing | Verify the integrity of database table mappings defined in ORM layer. These test classes use database metadata information to check ORM mappings. |
| DB Testing | Test data access classes (DAOs) in isolation. These tests put the database tables into a known state for each test run. |
| XML Testing | Test XML documents, their validity, and compare two different documents to assert if they are same or not. |
| Integration Testing | Test website navigation, web flow, and state (session) management and transaction management. |
| Regression Testing | Test the application functionality as an end-user would use the application when it's deployed in production environment. These tests are usually run by a dedicated QA team using automated testing tools such as Mercury QuickTest Professional ([QTP](http://www.mercury.com/us/products/quality-center/functional-testing/quicktest-professional/)). |
| Load Testing | Test the scalability of the application. These performance tests are usually run by a dedicated load testing team using tools such as Mercury [LoadRunner](http://www.mercury.com/us/products/performance-center/loadrunner/), [WAPT](http://www.loadtestingtool.com/), and [JMeter](http://jakarta.apache.org/jmeter/index.html). |
| Profiling | Test for any memory leaks, memory usage, garbage collection etc at application run-time. Developers run the application through a java profiler such as [JProfiler](http://www.ej-technologies.com/products/jprofiler/overview.html), Quest [JProbe](http://www.quest.com/jprobe/), Eclipse Test and Performance Tools Platform ([TPTP](http://www.eclipse.org/tptp/)). |

There are a variety of open source testing frameworks to perform the various test strategies listed above. Following is a list of some of these frameworks.

* JUnit
* JMock
* ORMUnit
* DBUnit
* XMLUnit
* JUnitEE
* MockEJB
* Cactus
* TestNG

## Integration Testing using Spring

Spring configuration, dependency injection (DI), data access (CRUD), and transaction management are some of the concerns that we can test outside the server environment using Spring Testing framework. The data access tests are executed against a real database so no need to use any mock objects in these tests.

Spring context load time may not be an issue in small to medium size web applications. But for a large size enterprise application, it could take significant time to instantiate the classes in the application. Also, the overhead of running every single test case in every single test fixture leads to slower overall test runs and adversely affects developer productivity. Keeping these concerns in mind, Spring development team wrote some test classes that can be used as integration tests running outside the container. Since these test classes are extensions of **JUnit** API, we get all the benefits of **JUnit** right out of the box when we use the Spring test classes. These test classes setup a transaction for each test method and automatically cleanup (rolling back the transaction at end of each method) eliminating the necessity for any database setup and teardown tasks.

Following is a list of items we can verify when we run integration tests in a Spring application:

* Loading of Spring contexts and management of the contexts by caching loaded contexts between execution of each test case. Also, verify correct wiring of application contexts by Spring IoC container.
* Dependency Injection of test fixtures and Spring configuration details (To verify if a specific data access (Repository) class configuration is loaded correctly).
* Convenience variables for data access and CRUD operations (The logic of data access class to test database select and updates).
* Transaction management.
* ORM mapping file configuration (Verify if everything related to a persistence object is mapped correctly and correct lazy-loading semantics are in place).

We can run integration tests just like JUnit tests. They are slower to run compared to unit tests because we are testing the code at integration level rather than just at a class level. But these integration tests are lot faster to execute than the tests created using in-container testing frameworks such as [JUnitEE](http://www.junitee.org/) or [Cactus](http://jakarta.apache.org/cactus/) which rely on deployment of the application to the container before the tests can be executed.

Spring integration test classes were designed to address various testing concerns so there are different test classes in **org.springframework.test** package. Following table shows some of the test classes provided in Spring framework for integration testing and in what scenarios they can be used.

*Table 2. Spring Test Classes*

|  |  |
| --- | --- |
| Test Class Name | Description |
| AbstractDependencyInjection SpringContextTests | This test class injects test dependencies so we don't need to specifically perform the Spring application context lookups. It also automatically locates the corresponding object in the set of configuration files specified in the**getConfigLocations()** method. |
| AbstractTransactionalDataSource SpringContextTests | This test class is used to test the code that runs inside a transaction. It creates and rolls back a transaction for each test case. We write code with the assumption that there is a transaction present. It provides the fields such as a **JdbcTemplate** that can be used to verify database state after test operations, or verify the results of queries performed by application code. An**ApplicationContext** is also inherited, and can be used for explicit lookup if necessary. |
| AbstractJpaTests | This test class is used for testing JPA functionality. It provides an**EntityManager** instance that we can use to call JPA methods. |
| AbstractAspectjJpaTests | This class extends from **AbstractJpaTests** and it is used for load-time weaving (LTW) purposes using **AspectJ**. We override the method**getActualAopXmlLocation()** to specify the location of AspectJ's configuration xml file. |
| AbstractModelAndViewTests | This is a convenient base class for testing the presentation and controller layers (using Spring MVC) in the application. |