Integration Tests

MidPoint as a large number of integration tests. Even though it is not entirely typical, midPoint integration tests are implemented in unit test framework TestNG. The integration tests are executed in almost the same way as unit tests are executed. The most important difference is that the integration test instantiates almost all midpoint components. It in fact creates embedded midPoint instance inside the test.

There are several "families" of the integration tests:

	Source path	Purpose
Model Integration Tests	model /model-intest	Test the system by using IDM Model Interface and similar interfaces on that level (namely ModelInteractionService). All model integration tests share parts of a common set-up. This is good for efficient writing of a new tests and also for maintenance. However writing a new tests requires familiarity with the common configuration, which is currently quite rich. Also it is quite difficult to test exotic configurations or configurations that are not entirely compatible with the common configuration. Almost all o the tests are using Dummy Resource. See Model Integration Tests page for more details.
Story Tests	testing /story	Each story test is focused on one particular functionality aspect or on a specific configuration. Some of the tests are inspired by existing midPoint deployments, some are purely fictional. But each "story" has mostly separate configuration. The tests usually use combination of Dummy Resource and embedded LDAP resources. Some of the tests are also documented (see Story Tests) Note: The story tests are not executed during normal build. There is a special `exttest` maven profile to execute them.
Longtests	testing /long	Integration tests that are taking too long to execute to run during normal build. Currently this is mostly an LDAP tests with many entries. Note: The longtests are not executed during normal build. There is a special `exttest` maven profile to execute them.
Sanity tests	testing /sanity	Relatively short and compact sanity tests that should tell whether the system is roughly OK or whether some basic functionality is broken. The run of sanity tests is much shorter than that of model-intest. But they are no that comprehensive. The sanity tests were a great help in early midPoint development. However as midPoint is currently quite comprehensive their value is quite limited. But they are still good tests and therefore they are still executed and maintained.
Webservice tests	testing /wstest	Very special tests for end-to-end testing of the SOAP-based web service. There are not really stand-alone tests. They require a separate running and somehow pre-configured midPoint instance. There is a script in our bamboo to set up an environment before these are executed.
REST tests	testing /rest	Tests for the REST service. These tests are running embedded midPoint instance similarly to other integration tests.
GUI integration tests	gui/admin-gui	Integration tests for the GUI testing "from the inside". These tests are **not** meant to test what GUI really displays. These tests are testing internal GUI APIs. E.g. they are testing object wrappers. However they construct and initialize full midPoint instance which makes them a real integration tests.
Conntest	testing /contest	Full end-to-end tests focused at testing integration of midPoint and resources, including test of the real connectors with the real resources. E.g. there are tests that do the same operations on several LDAP servers, tests with real Active Directory instance and so on. Obviously these require test environment and that environment is not accessible outside Evolveum network. Therefore the purpose of these tests are mostly just for internal midPoint testing by Evolveum.
Selenide tests	testing /seleniditest	Tests that are checking what GUI really displays on the browser level. These are based on the Selenide framework. **WORK IN PROGRESS**

There are also integration or semi-integration tests in other components. But these are the most prominent.

Almost all of the integration tests are based on the same principles that are described in this page. Please look at the pages for the individual test families (if there are any) for more details.

Embedded midPoint Instance

Almost all integration tests are creating and initializing embedded midPoint instance. That means that the tests are creating instances of all the midPoint components that they can. In fact this initialization is mostly done by the spring context. You may notice spring configuration files in the src/test/resources directories. These are used to create the embedded midPoint instance.

The embedded instance starts with an empty H2 database. The tests must populate this database with a configuration. This creates the test configuration and environment. Many test families already have superclasses that create the part of the configuration that is common for that particular family.

Test Superclasses

There are several test superclasses that contains the code that initializes the embedded midPoint. The supeclasses also contain a lot of utility methods to manipulate repository objects, lot of assertion methods and so on. The superclasses are:

Class Name	Component	Description
AbstractIntegrationTest	repo-test-util	The most basic integration test. Contains instance of repository service and task manager.
AbstractModelIntegrationTest	model-test	Almost complete integration test. Contains instance of model service and all the lower-level components (repo, task, provisioning, security, ...)
AbstractGuiIntegrationTest	admin-gui	Test superclass for GUI integration tests

Your test code can directly subclass any of these abstract superclasses. The tests come with empty repository, no resources and just the basic connectors.

There are also abstract superclasses that are pre-configured for a specific setup, e.g. many tests in model-intest component use AbstractInitializedModelIntegrationTest that contains a code that pre-populate the repository with a common set of objects.

initSystem method

The integration tests are initialized using the `initSystem()` method. This method can be used to insert initial objects into repository, initialized the resources, and so on.

Test method execution

Tests methods are executed sequentially in an alphabetical order. It is a convention to start the name of a test method with `testXXX` where `XXX` is a decimal number. The test execution is ordered by number. The number should be followed with a short name that describes what the test is trying to do. E. g.:

- `test100CreateUserAccount`
- `test125AssignRolePirateToJack`

The methods in a test class should form a story. Contrary to the frequently recommended practice the tests methods **may** depend on one another. E.g. one method may assign a role to a user which creates an account, following method may try to retrieve the account, next method may try to modify the account and the last method may try to unassign the role which deletes the account. This makes the test code easy to write and reasonably easy to maintain. The drawback is that if the first tests fails other tests will also fail. But we can live with that.

There is a convention to use test method `test000Sanity` to check that the test environment was initialized properly. This tests is like a pre-condition for the entire test class.

Description of the Test

Every test is testing some kind of scenario or story. If the story is not obvious from the test code at the first sight then please describe the scenario in one or two sentences. Like this:

```
/**
 * Make a native modification to an account and read it again. Make sure that
 * fresh data are returned - even though caching may be in effect.
 * MID-3481
 */
@Test
public void test106GetModifiedAccount() throws Exception {
            ...
    }
```

Please also specify **JIRA issue identifier** if it is applicable. This may be JIRA issue that refers to the bug that you are trying to reproduce in the test. Or it may be ID of a feature that you are trying to implement.

GIVEN, WHEN and THEN

Good practice is to divide each test method to three parts:

- **GIVEN** part: set up the environment, do a preparation (e.g. create Task and OperationResult, create query, etc.)
- **WHEN** part: do the thing that you want to test. Usually a single command.
- **THEN** part: Check that the test command in the WHEN part went well. Check that the result is not an error. Check that the object was really modified. Check that the accounts were created. Etc.

It is recommended to visibly mark the tests parts with comments `// GIVEN`, `// WHEN` and `// THEN`. Especially the WHEN mark is important. Code is the best documentation. Marking the core part of the test allows the developer to figure out what the test does at a first sight. Test maintenance is a major task. Therefore please save the time of your colleagues (and yourself). There are also appropriate methods to dump label for a started test and when/then section. These methods dump markers to logfiles and test output. Therefore it is easier to locate corresponding parts of test output after it is executed.

The test code should look like this:

```
    @Test
    public void test106GetModifiedAccount() throws Exception {
        final String TEST_NAME = "test106GetModifiedAccount";
        TestUtil.displayTestTile(TEST_NAME);

        // GIVEN

                Prepare test environment here

        // WHEN
                TestUtil.displayWhen(TEST_NAME);

        Do the thing that the test does. Ideally on a single line.

        // THEN
                TestUtil.displayThen(TEST_NAME);

        Check test results, make assertions, check environment
    }
```

Display methods

The test should be completely self-sufficient when it comes to checking the tests results. E.g. use assert methods to check that the test did what it should do. But, the tests need to be maintained and it is often good to display entire objects to help diagnose the test problems. There is a variety of `display`() methods especially for this purpose. All the display methods write the data both to test log and to standard output.

method	class	
display()	com.evolveum.midpoint.test.IntegrationTestTools	Display almost any value in a human-readable form
displayTestTitle()	com.evolveum.midpoint.test.util.TestUtil	Display a visual mark that divides individual tests in logfiles
displayWhen(), displayThen()	com.evolveum.midpoint.test.util.TestUtil	Display visual mart that divides test parts in logs

The display() method can be used to display almost any value that is used in midPoint. The method will make sure that the value is displayed in human-readable form. Also a title can be specified. E.g.:

```
display("Repository shadow", shadow)
```

will display the full dump of the PrismObject<ShadowType> object that is in the shadow variable. This a good way how to display diagnostic information about the test progress.

The displayTestTitle(), displayWhen() and displayThen() will mark tests and test parts. It is very good when examining test logfile, because simple test search can be used to skip to the start of the test or start of test part.

Test Directories

Tests often use files to initialize the repository, store queries, deltas and so on. The files should be placed in src/test/resources subdirectories. However placing all the files in one directory will create a chaos. Therefore we use subdirectories:

- The directory `src/test/resources/common` contains files that are shared by several tests. E.g. a resource definition, common roles, tasks, etc.
- Each test or a group of related tests has its own subdirectory. E.g. `src/test/resources/security`, `src/test/resources/entitlements`

Repository

Each test class should initialize a new instance of repository. The repository instance is slightly different that the repository used in standalone midPoint. But it is still embedded H2 database with full SQL repository component implementation. The repository starts as completely empty. There is even no system configuration or an administrator user. These should be explicitly added in initSystem() method if they are needed.

Make sure there is `@DirtiesContext(classMode = ClassMode.AFTER_CLASS)` annotation at the class level for each integration test. This makes sure that spring context is re-initialized after the tests and that the test will not leave dirty repository for the following tests.

Provisioning, Connectors and Dummy Resource

Model integration tests contain an initialized instance of provisioning components. This includes the usual set of connectors (LDAP, DB, CSV). There is also a special connector and an associated resource especially designed for integration tests: Dummy Resource.

The Dummy Resource simulates a real resource by using in-memory maps and lists. Therefore is is very lightweight. It supports accounts, groups and privilege object classes. It has extensible schema. It can behave in very tolerant and also in a very strict modes (e.g. being case in-sensitive, tolerate duplicates or not tolerate anything at all). This can all be configured by using connector configuration properties. It can also simulate network (and other errors). Therefore it is ideal for the tests. Have a look at `TestDummy` (provisioning-impl) or any test in `model-intest` component to see how to the dummy connector and resource are used.

There is also option to use embedded LDAP server (OpenDJ) and database (Derby) in tests. Have a look at TestOpenDJ and TestDBTable in provisioning-impl.

Logging and Output

All the tests log to the same log file:

`target/test.log`

The logging levels are usually controlled by the file:

`src/test/resources/logback-test.xml`

This file controls logging in majority of integration tests. However, there are few exception that need to initialize the system in slightly a different way. In that case the logging configuration from the SystemConfiguration object in the repository is used.

Test output (stdout and stderr) are stored in files:

`target/surefire-reports/*-output.txt`

The test output usually contains just the things that are printed by display() methods. Therefore the test output is usually much faster way to diagnose test problems as the objects are seen almost at the first sight. However the test output does not contain the details. Therefore the usual procedure is to look at the test output to get overall idea what is going on and then have a look at the log files. If the display() methods are used properly that the log files also contains all the test output, test markers, test part markers and so on. So simply searching for a test name in the log file will get you at the right place in the logfile.

Disabling Tests

Though shalt not disable the tests! Do not disable the tests unless there is very very very good reason for it. Perhaps the only good reason for disabling a test is this: you have created a new test and that test have found a nasty bug that you cannot fix right now. The bug is not critical and you do not want the whole build to fail. Then do this:

1. Create JIRA issu for the bug
2. Disable the test
3. Put the ID of the JIRA issue in the comment right after the disable statement. Like this:

```
@Test(enabled=false) // MID-3483
public void test031ModifyUserOnExistingAccountTest() throws Exception {
            ...
    }
```

Never ever commit a disable test without the reference to the JIRA issue.

See Also

- Model Integration Tests
- Story Tests
- Dummy Resource