**Implementing Tests**

This page summarizes points to consider when adding or revisiting tests for some feature or functionality. Tests should be omitted only if there are certain reasons for not implementing tests of some kind.

1. **Single threaded full API tests**
   a. Single threaded API calls with proper assertions
   b. Single threaded async API calls with proper assertions
   c. Input parameters validation
   d. Configuration parameters validation
   e. Single node topology
   f. Multi nodes topology
   g. Operations from server nodes
   h. Operations from client nodes
   i. Cache modes
      i. partitioned, replicated, local
      ii. atomic, transactional
      iii. tx concurrency and isolation modes
   j. Cache memory modes
      i. onheap
      ii. offheap
   k. Cache backups count - 0, 1, 2, ..., N
   l. Cache with store
   m. Cache with swap
   n. Cache with eviction policies
   o. Cache with expiry policies
   p. Marshellers - JDK, Binary, Optimized
   q. Peer class loading - On, Off
   r. Different key and value types - primitives, arrays, collections, POJOs, Serializables, Externalizables, Binarylizables.

2. **Failover tests**
   a. Multithreaded various API calls involving random branch choices with proper assertions
   b. Should be performed on changing topology
   c. May involve following techniques
      i. Certain messages delaying
      ii. Communication problems simulation
      iii. Exception throwing from user implemented logic (e.g. SPIs, listeners, filters, interceptors etc).
   d. For combinations to cover see pt.1
   e. Tests should also be targeted to discover memory usage problems and possible memory leaks (in case of batching, deferred buffers, etc)

3. **Benchmarks and performance tests**
   a. Yardstick benchmark should be added

4. **Long running performance tests**
   a. May involve fault-tolerance test scenarios
   b. Should also address possible memory usage problems
   c. Should check that system parameters (throughput, latency, CPU, memory, IO, etc) are stable.

### Configuration variations test framework

The framework provides an ability to write a test-case once and run it in multiple Ignite and Cache configurations.

**How to use the framework:**

- **Test class has to extend** `IgniteConfigVariationsAbstractTest` or `IgniteCacheConfigVariationsAbstractTest`.
- **TestSuite has to be built using** `ConfigVariationsTestSuiteBuilder`.

As an example see `IgniteCacheBasicConfigVariationsFullApiTestSuite`.

**Examples:**

1. The following code will test `SomeFunctionalityTest` with basic set of IgniteConfiguration variations (BinaryMarshaller / Optimized marshaller and enabled / disabled per class loading), will be used Ignite cluster with 3 nodes, the second started node is client (client with idx =1), each test-method will be run with `testedGrid=0` and `testedGrid=1` (see `testedGrid()` method).

```java
TestSuite suite = new ConfigVariationsTestSuiteBuilder(
    "Basic Ignite Configuration Variations Some Functionality Test Suite",
    SomeFunctionalityTest.class
).gridsCount(3)
  .testedNodesCount(2).withClients()
  .build();
```
2. The following code will test `SomeCacheFunctionalityTest` with basic set of `IgniteConfiguration` variations (see above) and basic set of `CacheConfiguration` variations (different cache modes, atomicity modes and memory modes). Ignite cluster with 5 nodes will be used, the second started node is client (with nodeIdx=1) and the third started node is client with near-only cache (with nodeIdx=2), each test-method will be run with `testedGrid=0`, `testedGrid=1` and `testedGrid=2` (see `testedGrid()` and `jcache()` methods).

```java
TestSuite suite = new ConfigVariationsTestSuiteBuilder("Basic Cache Configuration Variations Some Cache Functionality Test Suite", SomeCacheFunctionalityTest.class)
  .withBasicCacheParams()
  .gridsCount(5).backups(1)
  .testedNodesCount(3).withClients()
  .build();
```

Run in all data modes

`IgniteConfigVariationsAbstractTest` has methods `key(int i)` and `value(int i)`. To run a test scenario in all supported by the framework data modes (Serializable, `Externalizable`, plane, mixed and etc. objects), it's enough to write test scenario using these methods to generate different keys and values and to place test scenario inside `runInAllDataModes(TestRunnable runnable)` method.

For example, the following code test cache's put-get scenario in all supported by the framework data modes:

```java
public void testPutGet() throws Exception {
  runInAllDataModes(new TestRunnable() {
    @Override public void run() throws Exception {
      IgniteCache cache = jcache();
      Object key = key(1);
      Object val = value(1);
      cache.put(key, val);
      assertEquals(val, cache.get(key));
    }
  });
}
```

Implementation information

The main idea of Configuration Variations framework is a using of a matrix of possible variants of configuration properties.

Let's imagine there's a need to look over all possible variations of `IgniteConfiguration` where marshaller property can be `BinaryMarshaller` / `OptimizedMarshaller` and `peerClassLoadingEnabled` can be `true` / `false`. So, there's the following matrix:

<table>
<thead>
<tr>
<th><code>IgniteConfiguration Property</code></th>
<th>Variant 1</th>
<th>Variant 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>marshaller</td>
<td>BinaryMarshaller</td>
<td>OptimizedMarshaller</td>
</tr>
<tr>
<td>peerClassLoadingEnabled</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

The framework has `VariationsIterator` which will produce the following 4 variation vectors for the matrix above:

- `[0, 0]` - that means the framework will set marshaller property to `BinaryMarshaller` and `peerClassLoadingEnabled` flag to `true`.
- `[1, 0]` - that means the framework will set marshaller property to `OptimizedMarshaller` and `peerClassLoadingEnabled` flag to `true`.
- `[0, 1]` - that means the framework will set marshaller property to `BinaryMarshaller` and `peerClassLoadingEnabled` flag to `false`.
- `[1, 1]` - that means the framework will set marshaller property to `OptimizedMarshaller` and `peerClassLoadingEnabled` flag to `false`.

The following methods should be used to provide custom matrices:

- `ConfigVariationsTestSuiteBuilder.igniteParams(ConfigParameter<IgniteConfiguration>[]... igniteParams)`
- `ConfigVariationsTestSuiteBuilder.cacheParams(ConfigParameter<CacheConfiguration>[]... cacheParams)`

`ConfigVariations` - contains ready to use configuration parameters matrixes.

`Parameters` - contains util methods to build parameters matrixes.

Filtration of configurations

`ConfigVariationsTestSuiteBuilder` also provides possibility to filter `Ignite` and `Cache` configurations. See `withIgniteConfigFilters` (`IgnitePredicate<IgniteConfiguration>... filters`) and `withCacheConfigFilters` (`IgnitePredicate<CacheConfiguration>... filters`) methods.
Full API coverage enhancements

There is a plan to have Full API test coverage of Ignite functionality. See:

IGNITE-2521 - Need to increase test coverage (support all configurations combinations) OPEN

The purpose of this ticket is to make sure that all ignite operations should work for any configuration properties combination.