KIP-448: Add State Stores Unit Test Support to Kafka Streams Test Utils

Status

Current state: *"Under Discussion"*

Discussion thread: *here*

JIRA: *KAFKA-6460*

Motivation

This is for adding mock testing support for StateStore, StoreBuilder, StoreSupplier and other store related components which are used in Streams unit testing.

We'd like to use mocks for different types of state stores: KV, window, session - that can be used to record the number of expected put / get calls used in the DSL operator unit testing. These will provide convenience for developers when they are writing unit test for *Kafka stream* and other related modules.

For example, in the current *streams/TopologyTest.java:*

```
public class TopologyTest {

    private final StoreBuilder storeBuilder = EasyMock.createNiceMock(StoreBuilder.class);
    private final KeyValueStoreBuilder globalStoreBuilder = EasyMock.createNiceMock(KeyValueStoreBuilder.class);
    private final Topology topology = new Topology();
...

        @Test
    public void shouldFailIfSinkIsParent() {
        topology.addSource("source", "topic-1");
        topology.addSink("sink-1", "topic-2", "source");
        try {
            topology.addSink("sink-2", "topic-3", "sink-1");
            fail("Should throw TopologyException for using sink as parent");
        } catch (final TopologyException expected) { }
    }

    @Test(expected = TopologyException.class)
    public void shouldNotAllowToAddStateStoreToNonExistingProcessor() {
        mockStoreBuilder();
        EasyMock.replay(storeBuilder);
        topology.addStateStore(storeBuilder, "no-such-processor");
    }

    @Test
    public void shouldNotAllowToAddStateStoreToSource() {
        mockStoreBuilder();
        EasyMock.replay(storeBuilder);
        topology.addSource("source-1", "topic-1");
        try {
            topology.addStateStore(storeBuilder, "source-1");
            fail("Should have thrown TopologyException for adding store to source node");
        } catch (final TopologyException expected) { }
    }

        private void mockStoreBuilder() {
        EasyMock.expect(storeBuilder.name()).andReturn("store").anyTimes();
        EasyMock.expect(storeBuilder.logConfig()).andReturn(Collections.emptyMap());
        EasyMock.expect(storeBuilder.loggingEnabled()).andReturn(false);
    }
}
```

One of the goal is to replace the in-test vanilla mockStoreBuilder with a more general purpose mockStoreBuilder class, which simplify writing unit test and can be reuse later.

After the improvements, we will replace the easyMock StoreBuilder with our mockStoreBuilder.

```
public class TopologyTest {

    private final StoreBuilder storeBuilder = EasyMock.createNiceMock(StoreBuilder.class);
    private final Topology topology = new Topology();
    private final MockStoreFactory mockStoreFactory = new MockStoreFactory<>();
    private final KeyValueStoreBuilder keyValueStoreBuilder = mockStoreFactory.createKeyValueStoreBuilder(
            Stores.inMemoryKeyValueStore("store"),
            Serdes.Bytes(),
            Serdes.Bytes(),
            false,
                        Time.System);

...
    @Test(expected = TopologyException.class)
    public void shouldNotAllowToAddStateStoreToNonExistingProcessor() {
        topology.addStateStore(keyValueStoreBuilder, "no-such-processor");
    }
}
```

Public Interfaces

We add some new classes to a *state* package (*org.apache.kafka.streams.state*) under *streams/test-utils.*

We will provide a MockStoreFactory to generate mock store builders, I will use the KeyValueStoreBuilder as an example. Window and Session will have a similar structure.

The developers/users can provide their own store as the backend storage, and their own Serde of choice. For example, for simple testing, they can just use an InMemoryKeyValueStore.

```
package org.apache.kafka.streams.internals;

public class MockStoreFactory<K, V> {

    public final Map<String, StoreBuilder> stateStores = new LinkedHashMap<>();

    public MockStoreFactory () {
    }

    public KeyValueStoreBuilder createKeyValueStoreBuilder(KeyValueBytesStoreSupplier
keyValueBytesStoreSupplier,
                                                           final Serde<K> keySerde,
                                                           final Serde<V> valueSerde,
                                                           boolean persistent){
                String storeName = keyValueBytesStoreSupplier.name();
        stateStores.put(storeName, new MockKeyValueStoreBuilder<>(keyValueBytesStoreSupplier, keySerde,
valueSerde, persistent));
        return (KeyValueStoreBuilder)stateStores.get(storeName);
    }

        public WindowStoreBuilder createWindowStoreBuilder(KeyValueBytesStoreSupplier
keyValueBytesStoreSupplier,
                                                           final Serde<K> keySerde,
                                                           final Serde<V> valueSerde,
                                                           final Time time){
        ...
        }

        public SessionStoreBuilder createSessionStoreBuilder(KeyValueBytesStoreSupplier
keyValueBytesStoreSupplier,
                                                           final Serde<K> keySerde,
                                                           final Serde<V> valueSerde,
                                                           final Time time){
        ...
        }

    public StoreBuilder getStore(String storeName) {
        return stateStores.get(storeName);
    }
}
```

Each Store builder will have a build method:

```
package org.apache.kafka.streams.state;

import org.apache.kafka.common.serialization.Serde;

import org.apache.kafka.common.utils.Time;
import org.apache.kafka.streams.state.internals.KeyValueStoreBuilder;


public class MockKeyValueStoreBuilder<K, V> extends KeyValueStoreBuilder<K, V> {

    private final boolean persistent;
    private final KeyValueBytesStoreSupplier storeSupplier;
    final Serde<K> keySerde;
    final Serde<V> valueSerde;
    final Time time;

    public MockKeyValueStoreBuilder(final KeyValueBytesStoreSupplier storeSupplier,
                                    final Serde<K> keySerde,
                                    final Serde<V> valueSerde,
                                    final boolean persistent,
                                    final Time time) {
        super(storeSupplier, keySerde, valueSerde, time);
        this.persistent = persistent;
        this.storeSupplier = storeSupplier;
        this.keySerde = keySerde;
        this.valueSerde = valueSerde;
        this.time = time;
    }

    @Override
    public KeyValueStore<K, V> build() {
        return new MockKeyValueStore<>(storeSupplier, keySerde, valueSerde, persistent, time);
    }
}
```

Then in the store, we will build a wrapper around the provided backend store. We will capture each get/put/delete call, the user can write tests accordingly. We will also track if the store has been flushed or closed.

```
package org.apache.kafka.streams.state;

public class MockKeyValueStore<K, V>
        extends WrappedStateStore<KeyValueStore<Bytes, byte[]>, K, V>
        implements KeyValueStore<K, V>  {
    // keep a global counter of flushes and a local reference to which store had which
    // flush, so we can reason about the order in which stores get flushed.
    private static final AtomicInteger GLOBAL_FLUSH_COUNTER = new AtomicInteger(0);
    private final AtomicInteger instanceLastFlushCount = new AtomicInteger(-1);


    public boolean initialized = false;
    public boolean flushed = false;
    public boolean closed = true;

    public String name;
    public boolean persistent;

    protected final Time time;


    final Serde<K> keySerde;
    final Serde<V> valueSerde;
    StateSerdes<K, V> serdes;

    public final List<KeyValue<K, V>> capturedPutCalls = new LinkedList<>();
    public final List<KeyValue<K, V>> capturedGetCalls = new LinkedList<>();
    public final List<KeyValue<K, V>> capturedDeleteCalls = new LinkedList<>();
```

```
public MockKeyValueStore(final KeyValueBytesStoreSupplier keyValueBytesStoreSupplier,
                         final Serde<K> keySerde,
                         final Serde<V> valueSerde,
                         final boolean persistent,
                         final Time time) {
    super(keyValueBytesStoreSupplier.get());
    this.name = keyValueBytesStoreSupplier.name();
    this.time = time != null ? time : Time.SYSTEM;
    this.persistent = persistent;
    this.keySerde = keySerde;
    this.valueSerde = valueSerde;
}

@SuppressWarnings("unchecked")
void initStoreSerde(final ProcessorContext context) {
    serdes = new StateSerdes<>(
            ProcessorStateManager.storeChangelogTopic(context.applicationId(), name()),
            keySerde == null ? (Serde<K>) context.keySerde() : keySerde,
            valueSerde == null ? (Serde<V>) context.valueSerde() : valueSerde);
}

@Override
public String name() {
    return name;
}

@Override
public void init(final ProcessorContext context,
                 final StateStore root) {
    context.register(root, stateRestoreCallback);
    initialized = true;
    closed = false;
}

@Override
public void flush() {
    instanceLastFlushCount.set(GLOBAL_FLUSH_COUNTER.getAndIncrement());
    wrapped().flush();
    flushed = true;
}

public int getLastFlushCount() {
    return instanceLastFlushCount.get();
}

@Override
public void close() {
    wrapped().close();
    closed = true;
}

@Override
public boolean persistent() {
    return persistent;
}

@Override
public boolean isOpen() {
    return !closed;
}

public final StateRestoreCallback stateRestoreCallback = new StateRestoreCallback() {
    @Override
    public void restore(final byte[] key,
                        final byte[] value) {
    }
};

@Override
public void put(final K key, final V value) {
```

```
        capturedPutCalls.add(new KeyValue<>(key, value));
        wrapped().put(keyBytes(key), serdes.rawValue(value));
    }

    @Override
    public V putIfAbsent(final K key, final V value) {
        final V originalValue = get(key);
        if (originalValue == null) {
            put(key, value);
            capturedPutCalls.add(new KeyValue<>(key, value));
        }
        return originalValue;
    }

    @Override
    public V delete(final K key) {
        V value = outerValue(wrapped().delete(keyBytes(key)));
        capturedDeleteCalls.add(new KeyValue<>(key, value));
        return value;
    }

    @Override
    public void putAll(final List<KeyValue<K, V>> entries) {
        for (final KeyValue<K, V> entry : entries) {
            put(entry.key, entry.value);
            capturedPutCalls.add(entry);
        }
    }

    @Override
    public V get(final K key) {
        V value = outerValue(wrapped().get(keyBytes(key)));
        capturedGetCalls.add(new KeyValue<>(key, value));
        return value;
    }

    @SuppressWarnings("unchecked")
    @Override
    public KeyValueIterator<K,V> range(final K from, final K to) {
        return new MockKeyValueStore.MockKeyValueIterator(
                wrapped().range(Bytes.wrap(serdes.rawKey(from)), Bytes.wrap(serdes.rawKey(to))));
    }

    @SuppressWarnings("unchecked")
    @Override
    public KeyValueIterator<K,V>  all() {
        return new MockKeyValueStore.MockKeyValueIterator(wrapped().all());
    }

    @Override
    public long approximateNumEntries() {
        return wrapped().approximateNumEntries();
    }

    private V outerValue(final byte[] value) {
        return value != null ? serdes.valueFrom(value) : null;
    }

    private Bytes keyBytes(final K key) {
        return Bytes.wrap(serdes.rawKey(key));
    }

    private class MockKeyValueIterator implements KeyValueIterator<K, V> {

        private final KeyValueIterator<Bytes, byte[]> iter;
                ....
    }
}
```

Proposed Changes

I proposed to add:

1. A MockStoreFactory class to produce mock state store builders.
2. A mock StateStoreBuilder class for KV, Session and Window.
3. A mock StateStore class for KV, Session and Window with tracking.

Compatibility, Deprecation, and Migration Plan

Until we are refactoring with these new MockStateStores, there shouldn't be any compatibility issues. But the next phase should be refactoring, including:

1) Remove and refactor redundant MockStores (i.e. *org.apache.kafka.test.MockKeyValueStore*)

2) Examine the current tests (i.e. org.apache.kafka.streams.TopologyTest and refactor the testing code logics with the new MockStateStores.

A discussion that has been brought up is 

. In-memory Window/Session store doesn't work well in tests because in the init() method, it has internally cast ProcessorContext into InternalProcessorContext, that way the tester couldn't use MockProcessorContext. Either we twist the init() method in the mock store to accommodate or we can somehow use InternalMockProcessorContext instead?

Rejected Alternatives

1. Using the current EasyMock implementation. There is no strong argument against the current EasyMock implementation, it is easy to use and lightweight. The main argument for this KIP is to write better tests with an in-house mock state store support.