KIP-378: Enable Dependency Injection for Kafka Streams handlers

Status

Current state: *Discarded* (subsumed by KIP-832: Allow creating a producer/consumer using a producer/consumer config)

Discussion thread: here

JIRA: KAFKA-7527

Motivation

Whenever there is a need to introduce dependencies for handlers into a Kafka Streams project, additional steps used to be required to ensure this process. One possibility was to provide dependencies inside a **Map** for a **configure** method during application configuration process and another one was to create your own **StreamsConfig** class implementation with one method being overridden.

In *Kafka 2.0.0*, the latter option is marked as deprecated, which means it won't be available in upcoming versions. This implies, that projects which have been using dependency injection for Kafka Streams handlers, in the future will be forced to use a **Map** for the configure method during application configuration process instead.

Currently, the introduction of dependencies for Kafka Streams handler proceeds the following way:

1. Registration of an exception handler class in Kafka Streams configuration
2. Kafka Streams invokes a **default constructor** and creates an object out of provided class using reflection
3. Kafka Streams passes dependency configuration **Map** to the new instance's **configure** method
4. Dependencies are retrieved from the **Map** and have to be **casted** to a particular dependency type

Therefore, if your exception handler needs some other dependency, you have to construct it ahead of time and insert into the Kafka Streams config *Properti es*.

Afterwards, you need to retrieve it back in a configure method of your exception handler by extracting it from the *Map* and then *casting* it to an appropriate interface/class.

In addition, the newly introduced **TopologyTestDriver** is also affected. There is no straightforward, easy to maintain and developer-friendly possibility to benefit from dependency injection frameworks.

With respect to the mentioned above, developers experience major complication during testing and maintenance of Kafka Streams applications.

Public Interfaces

As a part of the proposed change, a deprecation annotation for three constructors in ***KafkaStreams*** class *could be removed, in particular:*

- `public KafkaStreams(final Topology topology, final StreamsConfig config)`
- `public KafkaStreams(final Topology topology, final StreamsConfig config, final KafkaClientSupplier clientSupplier)`
- `public KafkaStreams(final Topology topology, final StreamsConfig config, final Time time)`

To enable easy testing with dependency injection frameworks (e.g., Spring), three additional constructors for the **TopologyTestDriver** class could be introduced:

- `public TopologyTestDriver(final Topology topology, final StreamsConfig config)`
- `public TopologyTestDriver(final Topology topology, final StreamsConfig config, final long initialWallClockTimeMs)`
- `private TopologyTestDriver(final InternalTopologyBuilder builder, final StreamsConfig config, final long initialWallClockTimeMs)`

Proposed Changes

One possible option is to override a method inside the **StreamsConfig** class and replace a reflection-based creation of a handler class by the means of Spring dependency injection. Please consult example below:

```
public class MyStreamsConfig extends StreamsConfig {

  private final ApplicationContext applicationContext;

  SpringAwareStreamsConfig(final Properties properties, final ApplicationContext applicationContext) {
    super(properties);
    this.applicationContext = applicationContext;
  }

  @Override
  public <T> T getConfiguredInstance(final String key, final Class<T> type) {
    String[] beanNamesForType = applicationContext.getBeanNamesForType(type);
    if (beanNamesForType.length > 0) {
      return applicationContext.getBean(type);
    }
    return super.getConfiguredInstance(key, type);
  }
}
```

This offers two main advantages:

1. Spring can create dependencies for your beans. So that you don't need to construct and provide them inside a Kafka Streams configuration, as well as extract and cast it on handler's side.
2. You are obtaining an automated control over new dependencies, introduced Kafka Streams handlers

One minor advantage is that your dependencies can be set into final fields.

The second option it to create an additional interface as indicated in the example below:

Java

```
public interface ConfiguredStreamsFactory {
    <T> T getConfiguredInstance(String key, Class<T> t);
        <T> List<T> getConfiguredInstances(String key, Class<T> t);
        <T> List<T> getConfiguredInstances(String key, Class<T> t, Map<String, Object> configOverrides);
        <T> List<T> getConfiguredInstances(List<String> classNames, Class<T> t, Map<String, Object>
configOverrides);
}
```

And then provide the implementation of **ConfiguredStreamsFactory** while creating **StreamsConfig**:

Java

```
new StreamsConfig(final Properties config, final ConfiguredStreamsFactory configuredStreamsFactory);
```

A default implementation could be provided as well, based on the current implementation:

Java

```
public <T> T getConfiguredInstance(String key, Class<T> t) {
    Class<?> c = getClass(key);
    if (c == null)
        return null;
    Object o = Utils.newInstance(c);
    if (!t.isInstance(o))
        throw new KafkaException(c.getName() + " is not an instance of " + t.getName());
    if (o instanceof Configurable)
        ((Configurable) o).configure(originals());
    return t.cast(o);
}


public <T> List<T> getConfiguredInstances(String key, Class<T> t) {
    return getConfiguredInstances(key, t, Collections.emptyMap());
}


public <T> List<T> getConfiguredInstances(String key, Class<T> t, Map<String, Object> configOverrides) {
    return getConfiguredInstances(getList(key), t, configOverrides);
}


public <T> List<T> getConfiguredInstances(List<String> classNames, Class<T> t, Map<String, Object>
configOverrides) {
    List<T> objects = new ArrayList<>();
    if (classNames == null)
        return objects;
    Map<String, Object> configPairs = originals();
    configPairs.putAll(configOverrides);
    for (Object klass : classNames) {
        Object o;
        if (klass instanceof String) {
            try {
                o = Utils.newInstance((String) klass, t);
            } catch (ClassNotFoundException e) {
                throw new KafkaException(klass + " ClassNotFoundException exception occurred", e);
            }
        } else if (klass instanceof Class<?>) {
            o = Utils.newInstance((Class<?>) klass);
        } else
            throw new KafkaException("List contains element of type " + klass.getClass().getName() + ",
expected String or Class");
        if (!t.isInstance(o))
            throw new KafkaException(klass + " is not an instance of " + t.getName());
        if (o instanceof Configurable)
            ((Configurable) o).configure(configPairs);
        objects.add(t.cast(o));
    }
    return objects;
}
```

Compatibility, Deprecation, and Migration Plan

In case the first option is chosen, it is necessary to remove deprecation from three **KafkaStreams** constructors described in "Public interfaces" section.

Alternatively, the second option can be applied. Although, the second option needs to be studied in detail.

Rejected Alternatives

None at this point of time.