package com.twitter.search.feature\_update\_service;

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

import javax.inject.Inject;

import javax.inject.Named;

import scala.runtime.BoxedUnit;

import com.google.common.collect.ImmutableMap;

import com.google.common.collect.Lists;

import org.apache.kafka.clients.producer.ProducerRecord;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.common.util.Clock;

import com.twitter.common\_internal.text.version.PenguinVersion;

import com.twitter.decider.Decider;

import com.twitter.finagle.mux.ClientDiscardedRequestException;

import com.twitter.finagle.thrift.ClientId;

import com.twitter.finatra.kafka.producers.BlockingFinagleKafkaProducer;

import com.twitter.inject.annotations.Flag;

import com.twitter.search.common.decider.DeciderUtil;

import com.twitter.search.common.indexing.thriftjava.ThriftVersionedEvents;

import com.twitter.search.common.metrics.SearchCounter;

import com.twitter.search.common.metrics.SearchRateCounter;

import com.twitter.search.common.schema.thriftjava.ThriftIndexingEvent;

import com.twitter.search.common.schema.thriftjava.ThriftIndexingEventType;

import com.twitter.search.feature\_update\_service.modules.EarlybirdUtilModule;

import com.twitter.search.feature\_update\_service.modules.FinagleKafkaProducerModule;

import com.twitter.search.feature\_update\_service.stats.FeatureUpdateStats;

import com.twitter.search.feature\_update\_service.thriftjava.FeatureUpdateRequest;

import com.twitter.search.feature\_update\_service.thriftjava.FeatureUpdateResponse;

import com.twitter.search.feature\_update\_service.thriftjava.FeatureUpdateResponseCode;

import com.twitter.search.feature\_update\_service.thriftjava.FeatureUpdateService;

import com.twitter.search.feature\_update\_service.util.FeatureUpdateValidator;

import com.twitter.search.ingester.model.IngesterThriftVersionedEvents;

import com.twitter.tweetypie.thriftjava.GetTweetFieldsOptions;

import com.twitter.tweetypie.thriftjava.GetTweetFieldsRequest;

import com.twitter.tweetypie.thriftjava.TweetInclude;

import com.twitter.tweetypie.thriftjava.TweetService;

import com.twitter.tweetypie.thriftjava.TweetVisibilityPolicy;

import com.twitter.util.ExecutorServiceFuturePool;

import com.twitter.util.Function;

import com.twitter.util.Future;

import com.twitter.util.Futures;

import static com.twitter.tweetypie.thriftjava.Tweet.\_Fields.CORE\_DATA;

public class FeatureUpdateController implements FeatureUpdateService.ServiceIface {

private static final Logger LOG = LoggerFactory.getLogger(FeatureUpdateController.class);

private static final Logger REQUEST\_LOG =

LoggerFactory.getLogger("feature\_update\_service\_requests");

private static final String KAFKA\_SEND\_COUNT\_FORMAT = "kafka\_%s\_partition\_%d\_send\_count";

private static final String WRITE\_TO\_KAFKA\_DECIDER\_KEY = "write\_events\_to\_kafka\_update\_events";

private static final String WRITE\_TO\_KAFKA\_DECIDER\_KEY\_REALTIME\_CG =

"write\_events\_to\_kafka\_update\_events\_realtime\_cg";

private final SearchRateCounter droppedKafkaUpdateEvents =

SearchRateCounter.export("dropped\_kafka\_update\_events");

private final SearchRateCounter droppedKafkaUpdateEventsRealtimeCg =

SearchRateCounter.export("dropped\_kafka\_update\_events\_realtime\_cg");

private final Clock clock;

private final Decider decider;

private final BlockingFinagleKafkaProducer<Long, ThriftVersionedEvents> kafkaProducer;

private final BlockingFinagleKafkaProducer<Long, ThriftVersionedEvents> kafkaProducerRealtimeCg;

private final List<PenguinVersion> penguinVersions;

private final FeatureUpdateStats stats;

private final String kafkaUpdateEventsTopicName;

private final String kafkaUpdateEventsTopicNameRealtimeCg;

private final ExecutorServiceFuturePool futurePool;

private final TweetService.ServiceIface tweetService;

@Inject

public FeatureUpdateController(

Clock clock,

Decider decider,

@Named("KafkaProducer")

BlockingFinagleKafkaProducer<Long, ThriftVersionedEvents> kafkaProducer,

@Named("KafkaProducerRealtimeCg")

BlockingFinagleKafkaProducer<Long, ThriftVersionedEvents> kafkaProducerRealtimeCg,

@Flag(EarlybirdUtilModule.PENGUIN\_VERSIONS\_FLAG) String penguinVersions,

FeatureUpdateStats stats,

@Flag(FinagleKafkaProducerModule.KAFKA\_TOPIC\_NAME\_UPDATE\_EVENTS\_FLAG)

String kafkaUpdateEventsTopicName,

@Flag(FinagleKafkaProducerModule.KAFKA\_TOPIC\_NAME\_UPDATE\_EVENTS\_FLAG\_REALTIME\_CG)

String kafkaUpdateEventsTopicNameRealtimeCg,

ExecutorServiceFuturePool futurePool,

TweetService.ServiceIface tweetService

) {

this.clock = clock;

this.decider = decider;

this.kafkaProducer = kafkaProducer;

this.kafkaProducerRealtimeCg = kafkaProducerRealtimeCg;

this.penguinVersions = getPenguinVersions(penguinVersions);

this.stats = stats;

this.kafkaUpdateEventsTopicName = kafkaUpdateEventsTopicName;

this.kafkaUpdateEventsTopicNameRealtimeCg = kafkaUpdateEventsTopicNameRealtimeCg;

this.futurePool = futurePool;

this.tweetService = tweetService;

}

@Override

public Future<FeatureUpdateResponse> process(FeatureUpdateRequest featureUpdate) {

long requestStartTimeMillis = clock.nowMillis();

// Export overall and per-client request rate stats

final String requestClientId;

if (featureUpdate.getRequestClientId() != null

&& !featureUpdate.getRequestClientId().isEmpty()) {

requestClientId = featureUpdate.getRequestClientId();

} else if (ClientId.current().nonEmpty()) {

requestClientId = ClientId.current().get().name();

} else {

requestClientId = "unknown";

}

stats.clientRequest(requestClientId);

REQUEST\_LOG.info("{} {}", requestClientId, featureUpdate);

FeatureUpdateResponse errorResponse = FeatureUpdateValidator.validate(featureUpdate);

if (errorResponse != null) {

stats.clientResponse(requestClientId, errorResponse.getResponseCode());

LOG.warn("client error: clientID {} - reason: {}",

requestClientId, errorResponse.getDetailMessage());

return Future.value(errorResponse);

}

ThriftIndexingEvent event = featureUpdate.getEvent();

return writeToKafka(event, requestStartTimeMillis)

.map(responsesList -> {

stats.clientResponse(requestClientId, FeatureUpdateResponseCode.SUCCESS);

// only when both Realtime & RealtimeCG succeed, then it will return a success flag

return new FeatureUpdateResponse(FeatureUpdateResponseCode.SUCCESS);

})

.handle(Function.func(throwable -> {

FeatureUpdateResponseCode responseCode;

// if either Realtime or RealtimeCG throws an exception, it will return a failure

if (throwable instanceof ClientDiscardedRequestException) {

responseCode = FeatureUpdateResponseCode.CLIENT\_CANCEL\_ERROR;

LOG.info("ClientDiscardedRequestException received from client: " + requestClientId,

throwable);

} else {

responseCode = FeatureUpdateResponseCode.TRANSIENT\_ERROR;

LOG.error("Error occurred while writing to output stream: "

+ kafkaUpdateEventsTopicName + ", "

+ kafkaUpdateEventsTopicNameRealtimeCg, throwable);

}

stats.clientResponse(requestClientId, responseCode);

return new FeatureUpdateResponse(responseCode)

.setDetailMessage(throwable.getMessage());

}));

}

/\*\*

\* In writeToKafka(), we use Futures.collect() to aggregate results for two RPC calls

\* Futures.collect() means that if either one of the Future fails then it will return an Exception

\* only when both Realtime & RealtimeCG succeed, then it will return a success flag

\* The FeatureUpdateResponse is more like an ACK message, and the upstream (feature update ingester)

\* will not be affected much even if it failed (as long as the kafka message is written)

\*/

private Future<List<BoxedUnit>> writeToKafka(ThriftIndexingEvent event,

long requestStartTimeMillis) {

return Futures.collect(Lists.newArrayList(

writeToKafkaInternal(event, WRITE\_TO\_KAFKA\_DECIDER\_KEY, droppedKafkaUpdateEvents,

kafkaUpdateEventsTopicName, -1, kafkaProducer),

Futures.flatten(getUserId(event.getUid()).map(

userId -> writeToKafkaInternal(event, WRITE\_TO\_KAFKA\_DECIDER\_KEY\_REALTIME\_CG,

droppedKafkaUpdateEventsRealtimeCg,

kafkaUpdateEventsTopicNameRealtimeCg, userId, kafkaProducerRealtimeCg)))));

}

private Future<BoxedUnit> writeToKafkaInternal(ThriftIndexingEvent event, String deciderKey,

SearchRateCounter droppedStats, String topicName, long userId,

BlockingFinagleKafkaProducer<Long, ThriftVersionedEvents> producer) {

if (!DeciderUtil.isAvailableForRandomRecipient(decider, deciderKey)) {

droppedStats.increment();

return Future.Unit();

}

ProducerRecord<Long, ThriftVersionedEvents> producerRecord = new ProducerRecord<>(

topicName,

convertToThriftVersionedEvents(userId, event));

try {

return Futures.flatten(futurePool.apply(() ->

producer.send(producerRecord)

.map(record -> {

SearchCounter.export(String.format(

KAFKA\_SEND\_COUNT\_FORMAT, record.topic(), record.partition())).increment();

return BoxedUnit.UNIT;

})));

} catch (Exception e) {

return Future.exception(e);

}

}

private List<PenguinVersion> getPenguinVersions(String penguinVersionsStr) {

String[] tokens = penguinVersionsStr.split("\\s\*,\\s\*");

List<PenguinVersion> listOfPenguinVersions = Lists.newArrayListWithCapacity(tokens.length);

for (String token : tokens) {

listOfPenguinVersions.add(PenguinVersion.valueOf(token.toUpperCase()));

}

LOG.info(String.format("Using Penguin Versions: %s", listOfPenguinVersions));

return listOfPenguinVersions;

}

private Future<Long> getUserId(long tweetId) {

TweetInclude tweetInclude = new TweetInclude();

tweetInclude.setTweetFieldId(CORE\_DATA.getThriftFieldId());

GetTweetFieldsOptions getTweetFieldsOptions = new GetTweetFieldsOptions().setTweet\_includes(

Collections.singleton(tweetInclude)).setVisibilityPolicy(

TweetVisibilityPolicy.NO\_FILTERING);

GetTweetFieldsRequest getTweetFieldsRequest = new GetTweetFieldsRequest().setTweetIds(

Arrays.asList(tweetId)).setOptions(getTweetFieldsOptions);

try {

return tweetService.get\_tweet\_fields(getTweetFieldsRequest).map(

tweetFieldsResults -> tweetFieldsResults.get(

0).tweetResult.getFound().tweet.core\_data.user\_id);

} catch (Exception e) {

return Future.exception(e);

}

}

private ThriftVersionedEvents convertToThriftVersionedEvents(

long userId, ThriftIndexingEvent event) {

ThriftIndexingEvent thriftIndexingEvent = event.deepCopy()

.setEventType(ThriftIndexingEventType.PARTIAL\_UPDATE);

ImmutableMap.Builder<Byte, ThriftIndexingEvent> versionedEventsBuilder =

new ImmutableMap.Builder<>();

for (PenguinVersion penguinVersion : penguinVersions) {

versionedEventsBuilder.put(penguinVersion.getByteValue(), thriftIndexingEvent);

}

IngesterThriftVersionedEvents thriftVersionedEvents =

new IngesterThriftVersionedEvents(userId, versionedEventsBuilder.build());

thriftVersionedEvents.setId(thriftIndexingEvent.getUid());

return thriftVersionedEvents;

}

}