package com.twitter.timelineranker.adapter

import com.twitter.timelineranker.model.\_

import com.twitter.timelines.model.tweet.HydratedTweet

import com.twitter.timelines.model.TweetId

import com.twitter.timelineservice.model.TimelineId

import com.twitter.timelineservice.model.core

import com.twitter.timelineservice.{model => tls}

import com.twitter.timelineservice.{thriftscala => tlsthrift}

import com.twitter.timelineservice.model.core.\_

import com.twitter.util.Return

import com.twitter.util.Throw

import com.twitter.util.Try

/\*\*

\* Enables TLR model objects to be converted to/from TLS model/thrift objects.

\*/

object TimelineServiceAdapter {

def toTlrQuery(

id: Long,

tlsRange: tls.TimelineRange,

getTweetsFromArchiveIndex: Boolean = true

): ReverseChronTimelineQuery = {

val timelineId = TimelineId(id, TimelineKind.home)

val maxCount = tlsRange.maxCount

val tweetIdRange = tlsRange.cursor.map { cursor =>

TweetIdRange(

fromId = cursor.tweetIdBounds.bottom,

toId = cursor.tweetIdBounds.top

)

}

val options = ReverseChronTimelineQueryOptions(

getTweetsFromArchiveIndex = getTweetsFromArchiveIndex

)

ReverseChronTimelineQuery(timelineId, Some(maxCount), tweetIdRange, Some(options))

}

def toTlsQuery(query: ReverseChronTimelineQuery): tls.TimelineQuery = {

val tlsRange = toTlsRange(query.range, query.maxCount)

tls.TimelineQuery(

id = query.id.id,

kind = query.id.kind,

range = tlsRange

)

}

def toTlsRange(range: Option[TimelineRange], maxCount: Option[Int]): tls.TimelineRange = {

val cursor = range.map {

case tweetIdRange: TweetIdRange =>

RequestCursor(

top = tweetIdRange.toId.map(CursorState.fromTweetId),

bottom = tweetIdRange.fromId.map(core.CursorState.fromTweetId)

)

case \_ =>

throw new IllegalArgumentException(s"Only TweetIdRange is supported. Found: $range")

}

maxCount

.map { count => tls.TimelineRange(cursor, count) }

.getOrElse(tls.TimelineRange(cursor))

}

/\*\*

\* Converts TLS timeline to a Try of TLR timeline.

\*

\* TLS timeline not only contains timeline entries/attributes but also the retrieval state;

\* whereas TLR timeline only has entries/attributes. Therefore, the TLS timeline is

\* mapped to a Try[Timeline] where the Try part captures retrieval state and

\* Timeline captures entries/attributes.

\*/

def toTlrTimelineTry(tlsTimeline: tls.Timeline[tls.TimelineEntry]): Try[Timeline] = {

require(

tlsTimeline.kind == TimelineKind.home,

s"Only home timelines are supported. Found: ${tlsTimeline.kind}"

)

tlsTimeline.state match {

case Some(TimelineHit) | None =>

val tweetEnvelopes = tlsTimeline.entries.map {

case tweet: tls.Tweet =>

TimelineEntryEnvelope(Tweet(tweet.tweetId))

case entry =>

throw new Exception(s"Only tweet timelines are supported. Found: $entry")

}

Return(Timeline(TimelineId(tlsTimeline.id, tlsTimeline.kind), tweetEnvelopes))

case Some(TimelineNotFound) | Some(TimelineUnavailable) =>

Throw(new tls.core.TimelineUnavailableException(tlsTimeline.id, Some(tlsTimeline.kind)))

}

}

def toTlsTimeline(timeline: Timeline): tls.Timeline[tls.Tweet] = {

val entries = timeline.entries.map { entry =>

entry.entry match {

case tweet: Tweet => tls.Tweet(tweet.id)

case entry: HydratedTweetEntry => tls.Tweet.fromThrift(entry.tweet)

case \_ =>

throw new IllegalArgumentException(

s"Only tweet timelines are supported. Found: ${entry.entry}"

)

}

}

tls.Timeline(

id = timeline.id.id,

kind = timeline.id.kind,

entries = entries

)

}

def toTweetIds(timeline: tlsthrift.Timeline): Seq[TweetId] = {

timeline.entries.map {

case tlsthrift.TimelineEntry.Tweet(tweet) =>

tweet.statusId

case entry =>

throw new IllegalArgumentException(s"Only tweet timelines are supported. Found: ${entry}")

}

}

def toTweetIds(timeline: Timeline): Seq[TweetId] = {

timeline.entries.map { entry =>

entry.entry match {

case tweet: Tweet => tweet.id

case entry: HydratedTweetEntry => entry.tweet.id

case \_ =>

throw new IllegalArgumentException(

s"Only tweet timelines are supported. Found: ${entry.entry}"

)

}

}

}

def toHydratedTweets(timeline: Timeline): Seq[HydratedTweet] = {

timeline.entries.map { entry =>

entry.entry match {

case hydratedTweet: HydratedTweet => hydratedTweet

case \_ =>

throw new IllegalArgumentException(s"Expected hydrated tweet. Found: ${entry.entry}")

}

}

}

}