package com.twitter.tweetypie

package media

import com.twitter.mediainfo.server.{thriftscala => mis}

import com.twitter.mediaservices.commons.mediainformation.thriftscala.UserDefinedProductMetadata

import com.twitter.mediaservices.commons.photurkey.thriftscala.PrivacyType

import com.twitter.mediaservices.commons.servercommon.thriftscala.{ServerError => CommonServerError}

import com.twitter.mediaservices.commons.thriftscala.ProductKey

import com.twitter.mediaservices.commons.thriftscala.MediaKey

import com.twitter.servo.util.FutureArrow

import com.twitter.thumbingbird.{thriftscala => ifs}

import com.twitter.tweetypie.backends.MediaInfoService

import com.twitter.tweetypie.backends.UserImageService

import com.twitter.tweetypie.core.UpstreamFailure

import com.twitter.user\_image\_service.{thriftscala => uis}

import com.twitter.user\_image\_service.thriftscala.MediaUpdateAction

import com.twitter.user\_image\_service.thriftscala.MediaUpdateAction.Delete

import com.twitter.user\_image\_service.thriftscala.MediaUpdateAction.Undelete

import java.nio.ByteBuffer

import scala.util.control.NoStackTrace

/\*\*

\* The MediaClient trait encapsulates the various operations we make to the different media services

\* backends.

\*/

trait MediaClient {

import MediaClient.\_

/\*\*

\* On tweet creation, if the tweet contains media upload ids, we call this operation to process

\* that media and get back metadata about the media.

\*/

def processMedia: ProcessMedia

/\*\*

\* On the read path, when hydrating a MediaEntity, we call this operation to get metadata

\* about existing media.

\*/

def getMediaMetadata: GetMediaMetadata

def deleteMedia: DeleteMedia

def undeleteMedia: UndeleteMedia

}

/\*\*

\* Request type for the MediaClient.updateMedia operation.

\*/

private case class UpdateMediaRequest(

mediaKey: MediaKey,

action: MediaUpdateAction,

tweetId: TweetId)

case class DeleteMediaRequest(mediaKey: MediaKey, tweetId: TweetId) {

private[media] def toUpdateMediaRequest = UpdateMediaRequest(mediaKey, Delete, tweetId)

}

case class UndeleteMediaRequest(mediaKey: MediaKey, tweetId: TweetId) {

private[media] def toUpdateMediaRequest = UpdateMediaRequest(mediaKey, Undelete, tweetId)

}

/\*\*

\* Request type for the MediaClient.processMedia operation.

\*/

case class ProcessMediaRequest(

mediaIds: Seq[MediaId],

userId: UserId,

tweetId: TweetId,

isProtected: Boolean,

productMetadata: Option[Map[MediaId, UserDefinedProductMetadata]]) {

private[media] def toProcessTweetMediaRequest =

uis.ProcessTweetMediaRequest(mediaIds, userId, tweetId)

private[media] def toUpdateProductMetadataRequests(mediaKeys: Seq[MediaKey]) =

productMetadata match {

case None => Seq()

case Some(map) =>

mediaKeys.flatMap { mediaKey =>

map.get(mediaKey.mediaId).map { metadata =>

uis.UpdateProductMetadataRequest(ProductKey(tweetId.toString, mediaKey), metadata)

}

}

}

}

/\*\*

\* Request type for the MediaClient.getMediaMetdata operation.

\*/

case class MediaMetadataRequest(

mediaKey: MediaKey,

tweetId: TweetId,

isProtected: Boolean,

extensionsArgs: Option[ByteBuffer]) {

private[media] def privacyType = MediaClient.toPrivacyType(isProtected)

/\*\*

\* For debugging purposes, make a copy of the byte buffer at object

\* creation time, so that we can inspect the original buffer if there

\* is an error.

\*

\* Once we have found the problem, this method should be removed.

\*/

val savedExtensionArgs: Option[ByteBuffer] =

extensionsArgs.map { buf =>

val b = buf.asReadOnlyBuffer()

val ary = new Array[Byte](b.remaining)

b.get(ary)

ByteBuffer.wrap(ary)

}

private[media] def toGetTweetMediaInfoRequest =

mis.GetTweetMediaInfoRequest(

mediaKey = mediaKey,

tweetId = Some(tweetId),

privacyType = privacyType,

stratoExtensionsArgs = extensionsArgs

)

}

object MediaClient {

import MediaExceptions.\_

/\*\*

\* Operation type for processing uploaded media during tweet creation.

\*/

type ProcessMedia = FutureArrow[ProcessMediaRequest, Seq[MediaKey]]

/\*\*

\* Operation type for deleting and undeleting tweets.

\*/

private[media] type UpdateMedia = FutureArrow[UpdateMediaRequest, Unit]

type UndeleteMedia = FutureArrow[UndeleteMediaRequest, Unit]

type DeleteMedia = FutureArrow[DeleteMediaRequest, Unit]

/\*\*

\* Operation type for getting media metadata for existing media during tweet reads.

\*/

type GetMediaMetadata = FutureArrow[MediaMetadataRequest, MediaMetadata]

/\*\*

\* Builds a UpdateMedia FutureArrow using UserImageService endpoints.

\*/

private[media] object UpdateMedia {

def apply(updateTweetMedia: UserImageService.UpdateTweetMedia): UpdateMedia =

FutureArrow[UpdateMediaRequest, Unit] { r =>

updateTweetMedia(uis.UpdateTweetMediaRequest(r.mediaKey, r.action, Some(r.tweetId))).unit

}.translateExceptions(handleMediaExceptions)

}

/\*\*

\* Builds a ProcessMedia FutureArrow using UserImageService endpoints.

\*/

object ProcessMedia {

def apply(

updateProductMetadata: UserImageService.UpdateProductMetadata,

processTweetMedia: UserImageService.ProcessTweetMedia

): ProcessMedia = {

val updateProductMetadataSeq = updateProductMetadata.liftSeq

FutureArrow[ProcessMediaRequest, Seq[MediaKey]] { req =>

for {

mediaKeys <- processTweetMedia(req.toProcessTweetMediaRequest).map(\_.mediaKeys)

\_ <- updateProductMetadataSeq(req.toUpdateProductMetadataRequests(mediaKeys))

} yield {

sortKeysByIds(req.mediaIds, mediaKeys)

}

}.translateExceptions(handleMediaExceptions)

}

/\*\*

\* Sort the mediaKeys Seq based on the media id ordering specified by the

\* caller's request mediaIds Seq.

\*/

private def sortKeysByIds(mediaIds: Seq[MediaId], mediaKeys: Seq[MediaKey]): Seq[MediaKey] = {

val idToKeyMap = mediaKeys.map(key => (key.mediaId, key)).toMap

mediaIds.flatMap(idToKeyMap.get)

}

}

/\*\*

\* Builds a GetMediaMetadata FutureArrow using MediaInfoService endpoints.

\*/

object GetMediaMetadata {

private[this] val log = Logger(getClass)

def apply(getTweetMediaInfo: MediaInfoService.GetTweetMediaInfo): GetMediaMetadata =

FutureArrow[MediaMetadataRequest, MediaMetadata] { req =>

getTweetMediaInfo(req.toGetTweetMediaInfoRequest).map { res =>

MediaMetadata(

res.mediaKey,

res.assetUrlHttps,

res.sizes.toSet,

res.mediaInfo,

res.additionalMetadata.flatMap(\_.productMetadata),

res.stratoExtensionsReply,

res.additionalMetadata

)

}

}.translateExceptions(handleMediaExceptions)

}

private[media] def toPrivacyType(isProtected: Boolean): PrivacyType =

if (isProtected) PrivacyType.Protected else PrivacyType.Public

/\*\*

\* Constructs an implementation of the MediaClient interface using backend instances.

\*/

def fromBackends(

userImageService: UserImageService,

mediaInfoService: MediaInfoService

): MediaClient =

new MediaClient {

val getMediaMetadata =

GetMediaMetadata(

getTweetMediaInfo = mediaInfoService.getTweetMediaInfo

)

val processMedia =

ProcessMedia(

userImageService.updateProductMetadata,

userImageService.processTweetMedia

)

private val updateMedia =

UpdateMedia(

userImageService.updateTweetMedia

)

val deleteMedia: FutureArrow[DeleteMediaRequest, Unit] =

FutureArrow[DeleteMediaRequest, Unit](r => updateMedia(r.toUpdateMediaRequest))

val undeleteMedia: FutureArrow[UndeleteMediaRequest, Unit] =

FutureArrow[UndeleteMediaRequest, Unit](r => updateMedia(r.toUpdateMediaRequest))

}

}

/\*\*

\* Exceptions from the various media services backends that indicate bad requests (validation

\* failures) are converted to a MediaClientException. Exceptions that indicate a server

\* error are converted to a UpstreamFailure.MediaServiceServerError.

\*

\* MediaNotFound: Given media id does not exist. It could have been expired

\* BadMedia: Given media is corrupted and can not be processed.

\* InvalidMedia: Given media has failed to pass one or more validations (size, dimensions, type etc.)

\* BadRequest Request is bad, but reason not available

\*/

object MediaExceptions {

import UpstreamFailure.MediaServiceServerError

// Extends NoStackTrace because the circumstances in which the

// exceptions are generated don't yield useful stack traces

// (e.g. you can't tell from the stack trace anything about what

// backend call was being made.)

abstract class MediaClientException(message: String) extends Exception(message) with NoStackTrace

class MediaNotFound(message: String) extends MediaClientException(message)

class BadMedia(message: String) extends MediaClientException(message)

class InvalidMedia(message: String) extends MediaClientException(message)

class BadRequest(message: String) extends MediaClientException(message)

// translations from various media service errors into MediaExceptions

val handleMediaExceptions: PartialFunction[Any, Exception] = {

case uis.BadRequest(msg, reason) =>

reason match {

case Some(uis.BadRequestReason.MediaNotFound) => new MediaNotFound(msg)

case Some(uis.BadRequestReason.BadMedia) => new BadMedia(msg)

case Some(uis.BadRequestReason.InvalidMedia) => new InvalidMedia(msg)

case \_ => new BadRequest(msg)

}

case ifs.BadRequest(msg, reason) =>

reason match {

case Some(ifs.BadRequestReason.NotFound) => new MediaNotFound(msg)

case \_ => new BadRequest(msg)

}

case mis.BadRequest(msg, reason) =>

reason match {

case Some(mis.BadRequestReason.MediaNotFound) => new MediaNotFound(msg)

case \_ => new BadRequest(msg)

}

case ex: CommonServerError => MediaServiceServerError(ex)

}

}