package com.twitter.home\_mixer.federated

import com.twitter.gizmoduck.{thriftscala => gd}

import com.twitter.home\_mixer.marshaller.request.HomeMixerRequestUnmarshaller

import com.twitter.home\_mixer.model.request.HomeMixerRequest

import com.twitter.home\_mixer.{thriftscala => hm}

import com.twitter.product\_mixer.core.functional\_component.configapi.ParamsBuilder

import com.twitter.product\_mixer.core.pipeline.product.ProductPipelineRequest

import com.twitter.product\_mixer.core.pipeline.product.ProductPipelineResult

import com.twitter.product\_mixer.core.product.registry.ProductPipelineRegistry

import com.twitter.product\_mixer.core.{thriftscala => pm}

import com.twitter.stitch.Arrow

import com.twitter.stitch.Stitch

import com.twitter.strato.callcontext.CallContext

import com.twitter.strato.catalog.OpMetadata

import com.twitter.strato.config.\_

import com.twitter.strato.data.\_

import com.twitter.strato.fed.StratoFed

import com.twitter.strato.generated.client.auth\_context.AuditIpClientColumn

import com.twitter.strato.generated.client.gizmoduck.CompositeOnUserClientColumn

import com.twitter.strato.graphql.timelines.{thriftscala => gql}

import com.twitter.strato.thrift.ScroogeConv

import com.twitter.timelines.render.{thriftscala => tr}

import com.twitter.util.Try

import javax.inject.Inject

import javax.inject.Singleton

@Singleton

class HomeMixerColumn @Inject() (

homeMixerRequestUnmarshaller: HomeMixerRequestUnmarshaller,

compositeOnUserClientColumn: CompositeOnUserClientColumn,

auditIpClientColumn: AuditIpClientColumn,

paramsBuilder: ParamsBuilder,

productPipelineRegistry: ProductPipelineRegistry)

extends StratoFed.Column(HomeMixerColumn.Path)

with StratoFed.Fetch.Arrow {

override val contactInfo: ContactInfo = ContactInfo(

contactEmail = "",

ldapGroup = "",

slackRoomId = ""

)

override val metadata: OpMetadata =

OpMetadata(

lifecycle = Some(Lifecycle.Production),

description =

Some(Description.PlainText("Federated Strato column for Timelines served via Home Mixer"))

)

private val bouncerAccess: Seq[Policy] = Seq(BouncerAccess())

private val finatraTestServiceIdentifiers: Seq[Policy] = Seq(

ServiceIdentifierPattern(

role = "",

service = "",

env = "",

zone = Seq(""))

)

override val policy: Policy = AnyOf(bouncerAccess ++ finatraTestServiceIdentifiers)

override type Key = gql.TimelineKey

override type View = gql.HomeTimelineView

override type Value = tr.Timeline

override val keyConv: Conv[Key] = ScroogeConv.fromStruct[gql.TimelineKey]

override val viewConv: Conv[View] = ScroogeConv.fromStruct[gql.HomeTimelineView]

override val valueConv: Conv[Value] = ScroogeConv.fromStruct[tr.Timeline]

private def createHomeMixerRequestArrow(

compositeOnUserClientColumn: CompositeOnUserClientColumn,

auditIpClientColumn: AuditIpClientColumn

): Arrow[(Key, View), hm.HomeMixerRequest] = {

val populateUserRolesAndIp: Arrow[(Key, View), (Option[Set[String]], Option[String])] = {

val gizmoduckView: (gd.LookupContext, Set[gd.QueryFields]) =

(gd.LookupContext(), Set(gd.QueryFields.Roles))

val populateUserRoles = Arrow

.flatMap[(Key, View), Option[Set[String]]] { \_ =>

Stitch.collect {

CallContext.twitterUserId.map { userId =>

compositeOnUserClientColumn.fetcher

.callStack(HomeMixerColumn.FetchCallstack)

.fetch(userId, gizmoduckView).map(\_.v)

.map {

\_.flatMap(\_.roles.map(\_.roles.toSet)).getOrElse(Set.empty)

}

}

}

}

val populateIpAddress = Arrow

.flatMap[(Key, View), Option[String]](\_ =>

auditIpClientColumn.fetcher

.callStack(HomeMixerColumn.FetchCallstack)

.fetch((), ()).map(\_.v))

Arrow.join(

populateUserRoles,

populateIpAddress

)

}

Arrow.zipWithArg(populateUserRolesAndIp).map {

case ((key, view), (roles, ipAddress)) =>

val deviceContextOpt = Some(

hm.DeviceContext(

isPolling = CallContext.isPolling,

requestContext = view.requestContext,

latestControlAvailable = view.latestControlAvailable,

autoplayEnabled = view.autoplayEnabled

))

val seenTweetIds = view.seenTweetIds.filter(\_.nonEmpty)

val (product, productContext) = key match {

case gql.TimelineKey.HomeTimeline(\_) | gql.TimelineKey.HomeTimelineV2(\_) =>

(

hm.Product.ForYou,

hm.ProductContext.ForYou(

hm.ForYou(

deviceContextOpt,

seenTweetIds,

view.dspClientContext,

view.pushToHomeTweetId

)

))

case gql.TimelineKey.HomeLatestTimeline(\_) | gql.TimelineKey.HomeLatestTimelineV2(\_) =>

(

hm.Product.Following,

hm.ProductContext.Following(

hm.Following(deviceContextOpt, seenTweetIds, view.dspClientContext)))

case gql.TimelineKey.CreatorSubscriptionsTimeline(\_) =>

(

hm.Product.Subscribed,

hm.ProductContext.Subscribed(hm.Subscribed(deviceContextOpt, seenTweetIds)))

case \_ => throw new UnsupportedOperationException(s"Unknown product: $key")

}

val clientContext = pm.ClientContext(

userId = CallContext.twitterUserId,

guestId = CallContext.guestId,

guestIdAds = CallContext.guestIdAds,

guestIdMarketing = CallContext.guestIdMarketing,

appId = CallContext.clientApplicationId,

ipAddress = ipAddress,

userAgent = CallContext.userAgent,

countryCode = CallContext.requestCountryCode,

languageCode = CallContext.requestLanguageCode,

isTwoffice = CallContext.isInternalOrTwoffice,

userRoles = roles,

deviceId = CallContext.deviceId,

mobileDeviceId = CallContext.mobileDeviceId,

mobileDeviceAdId = CallContext.adId,

limitAdTracking = CallContext.limitAdTracking

)

hm.HomeMixerRequest(

clientContext = clientContext,

product = product,

productContext = Some(productContext),

maxResults = Try(view.count.get.toInt).toOption.orElse(HomeMixerColumn.MaxCount),

cursor = view.cursor.filter(\_.nonEmpty)

)

}

}

override val fetch: Arrow[(Key, View), Result[Value]] = {

val transformThriftIntoPipelineRequest: Arrow[

(Key, View),

ProductPipelineRequest[HomeMixerRequest]

] = {

Arrow

.identity[(Key, View)]

.andThen {

createHomeMixerRequestArrow(compositeOnUserClientColumn, auditIpClientColumn)

}

.map {

case thriftRequest =>

val request = homeMixerRequestUnmarshaller(thriftRequest)

val params = paramsBuilder.build(

clientContext = request.clientContext,

product = request.product,

featureOverrides =

request.debugParams.flatMap(\_.featureOverrides).getOrElse(Map.empty),

)

ProductPipelineRequest(request, params)

}

}

val underlyingProduct: Arrow[

ProductPipelineRequest[HomeMixerRequest],

ProductPipelineResult[tr.TimelineResponse]

] = Arrow

.identity[ProductPipelineRequest[HomeMixerRequest]]

.map { pipelineRequest =>

val pipelineArrow = productPipelineRegistry

.getProductPipeline[HomeMixerRequest, tr.TimelineResponse](

pipelineRequest.request.product)

.arrow

(pipelineArrow, pipelineRequest)

}.applyArrow

transformThriftIntoPipelineRequest.andThen(underlyingProduct).map {

\_.result match {

case Some(result) => found(result.timeline)

case \_ => missing

}

}

}

}

object HomeMixerColumn {

val Path = "home-mixer/homeMixer.Timeline"

private val FetchCallstack = s"$Path:fetch"

private val MaxCount: Option[Int] = Some(100)

}