package com.twitter.tweetypie.federated.columns

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.stitch.MapGroup

import com.twitter.stitch.Stitch

import com.twitter.strato.catalog.Fetch

import com.twitter.strato.catalog.OpMetadata

import com.twitter.strato.config.AllowAll

import com.twitter.strato.config.ContactInfo

import com.twitter.strato.config.Policy

import com.twitter.strato.data.Conv

import com.twitter.strato.data.Description.PlainText

import com.twitter.strato.data.Lifecycle.Production

import com.twitter.strato.fed.StratoFed

import com.twitter.strato.opcontext.OpContext

import com.twitter.strato.response.Err

import com.twitter.strato.thrift.ScroogeConv

import com.twitter.tweetypie.TweetId

import com.twitter.tweetypie.client\_id.PreferForwardedServiceIdentifierForStrato

import com.twitter.tweetypie.thriftscala.GetTweetFieldsOptions

import com.twitter.tweetypie.thriftscala.GetTweetFieldsRequest

import com.twitter.tweetypie.thriftscala.GetTweetFieldsResult

import com.twitter.tweetypie.thriftscala.TweetVisibilityPolicy

import com.twitter.util.Future

import com.twitter.util.Try

/\*\*

\* Strato federated column implementing GetTweetFields as a Fetch.

\*/

class GetTweetFieldsColumn(

handler: GetTweetFieldsRequest => Future[Seq[GetTweetFieldsResult]],

stats: StatsReceiver)

extends StratoFed.Column(GetTweetFieldsColumn.Path)

with StratoFed.Fetch.StitchWithContext {

/\*\*

\* At this point, this fetch op will reject any requests that specify

\* visibilityPolicy other than USER\_VISIBLE, so no access control is needed.

\*/

override val policy: Policy = AllowAll

override type Key = TweetId

override type View = GetTweetFieldsOptions

override type Value = GetTweetFieldsResult

override val keyConv: Conv[Key] = Conv.ofType

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

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

override val contactInfo: ContactInfo = TweetypieContactInfo

override val metadata: OpMetadata = OpMetadata(

lifecycle = Some(Production),

description =

Some(PlainText("Get of tweets that allows fetching only specific subsets of the data.")),

)

val safetyOpContextOnlyCounter = stats.counter("safety\_op\_context\_only")

val safetyOpContextOnlyValueScope = stats.scope("safety\_op\_context\_only\_value")

val safetyOpContextOnlyCallerScope = stats.scope("safety\_op\_context\_only\_caller")

val safetyViewOnlyCounter = stats.counter("safety\_view\_only")

val safetyViewOnlyValueScope = stats.scope("safety\_view\_only\_value")

val safetyViewOnlyCallerScope = stats.scope("safety\_view\_only\_caller")

val safetyLevelInconsistencyCounter = stats.counter("safety\_level\_inconsistency")

val safetyLevelInconsistencyValueScope = stats.scope("safety\_level\_inconsistency\_value")

val safetyLevelInconsistencyCallerScope = stats.scope("safety\_level\_inconsistency\_caller")

override def fetch(key: Key, view: View, ctx: OpContext): Stitch[Result[Value]] = {

compareSafetyLevel(view, ctx)

checkVisibilityPolicyUserVisible(view).flatMap { \_ =>

Stitch.call(key, Group(view))

}

}

/\*\*

\* Only allow [[TweetVisibilityPolicy.UserVisible]] visibilityPolicy.

\*

\* This column requires access policy in order to serve requests with visibilityPolicy

\* other than [[TweetVisibilityPolicy.UserVisible]]. Before we support access control,

\* reject all requests that are not safe.

\*/

private def checkVisibilityPolicyUserVisible(view: View): Stitch[Unit] =

view.visibilityPolicy match {

case TweetVisibilityPolicy.UserVisible => Stitch.value(Unit)

case otherValue =>

Stitch.exception(

Err(

Err.BadRequest,

"GetTweetFields does not support access control on Strato yet. "

+ s"Hence visibilityPolicy can only take the default ${TweetVisibilityPolicy.UserVisible} value, "

+ s"got: ${otherValue}."

))

}

/\*\* Compare the SafetyLevels in the View and OpContext \*/

private def compareSafetyLevel(view: View, ctx: OpContext): Unit =

(view.safetyLevel, ctx.safetyLevel) match {

case (None, None) =>

case (Some(viewSafety), None) => {

safetyViewOnlyCounter.incr()

safetyViewOnlyValueScope.counter(viewSafety.name).incr()

PreferForwardedServiceIdentifierForStrato.serviceIdentifier

.foreach(serviceId => safetyViewOnlyCallerScope.counter(serviceId.toString).incr())

}

case (None, Some(ctxSafety)) => {

safetyOpContextOnlyCounter.incr()

safetyOpContextOnlyValueScope.counter(ctxSafety.name).incr()

PreferForwardedServiceIdentifierForStrato.serviceIdentifier

.foreach(serviceId => safetyOpContextOnlyCallerScope.counter(serviceId.toString).incr())

}

case (Some(viewSafety), Some(ctxSafety)) =>

def safeStringEquals(a: String, b: String) =

a.toLowerCase().trim().equals(b.toLowerCase().trim())

if (!safeStringEquals(viewSafety.name, ctxSafety.name)) {

safetyLevelInconsistencyCounter.incr()

safetyLevelInconsistencyValueScope.counter(viewSafety.name + '-' + ctxSafety.name).incr()

PreferForwardedServiceIdentifierForStrato.serviceIdentifier

.foreach(serviceId =>

safetyLevelInconsistencyCallerScope.counter(serviceId.toString).incr())

}

}

/\*\*

\* Means of batching of [[GetTweetFieldsColumn]] calls.

\*

\* Only calls issued against the same instance of [[GetTweetFieldsColumn]]

\* are batched as Stitch clusters group objects based on equality,

\* and nested case class implicitly captures [[GetTweetFieldsColumn]] reference.

\*/

private case class Group(view: GetTweetFieldsOptions)

extends MapGroup[TweetId, Fetch.Result[GetTweetFieldsResult]] {

/\*\*

\* Batches given [[TweetId]] lookups in a single [[GetTweetFieldsRequest]]

\* and returns a result mapped by [[TweetId]].

\*/

override protected def run(

keys: Seq[TweetId]

): Future[TweetId => Try[Fetch.Result[GetTweetFieldsResult]]] =

handler(

GetTweetFieldsRequest(

// Sorting the keys makes for simpler matchers in the tests

// as matching on a Seq needs to be in order.

tweetIds = keys.sorted,

options = view,

)).map(groupByTweetId)

/\*\*

\* Groups given [[GetTweetFieldsResult]] objects by [[TweetId]] and returns the mapping.

\*/

private def groupByTweetId(

allResults: Seq[GetTweetFieldsResult]

): TweetId => Try[Fetch.Result[GetTweetFieldsResult]] = {

allResults

.groupBy(\_.tweetId)

.mapValues {

case Seq(result) => Try(Fetch.Result.found(result))

case manyResults =>

Try {

throw Err(

Err.Dependency,

s"Expected one result per tweeet ID, got ${manyResults.length}")

}

}

}

}

}

object GetTweetFieldsColumn {

val Path = "tweetypie/getTweetFields.Tweet"

}