package com.twitter.tweetypie

package hydrator

import com.twitter.featureswitches.v2.FeatureSwitchResults

import com.twitter.stitch.Stitch

import com.twitter.tweetypie.FieldId

import com.twitter.tweetypie.TweetId

import com.twitter.tweetypie.core.ValueState

import com.twitter.tweetypie.repository.TweetCountKey

import com.twitter.tweetypie.repository.TweetCountsRepository

import com.twitter.tweetypie.thriftscala.EditControl

import com.twitter.tweetypie.thriftscala.StatusCounts

import com.twitter.tweetypie.thriftscala.\_

/\*

\* A constructor for a ValueHydrator that hydrates `previous\_counts`

\* information. Previous counts are applied to edit tweets, they

\* are the summation of all the status\_counts in an edit chain up to

\* but not including the tweet being hydrated.

\*

\*/

object PreviousTweetCountsHydrator {

case class Ctx(

editControl: Option[EditControl],

featureSwitchResults: Option[FeatureSwitchResults],

underlyingTweetCtx: TweetCtx)

extends TweetCtx.Proxy

type Type = ValueHydrator[Option[StatusCounts], Ctx]

val hydratedField: FieldByPath = fieldByPath(Tweet.PreviousCountsField)

/\*

\* Params:

\* tweetId: The tweet being hydrated.

\* editTweetIds: The sorted list of all edits in an edit chain.

\*

\* Returns: tweetIds in an edit chain from the initial tweet up to but not including

\* the tweet being hydrated (`tweetId`)

\*/

def previousTweetIds(tweetId: TweetId, editTweetIds: Seq[TweetId]): Seq[TweetId] = {

editTweetIds.takeWhile(\_ < tweetId)

}

/\* An addition operation for Option[Long] \*/

def sumOptions(A: Option[Long], B: Option[Long]): Option[Long] =

(A, B) match {

case (None, None) => None

case (Some(a), None) => Some(a)

case (None, Some(b)) => Some(b)

case (Some(a), Some(b)) => Some(a + b)

}

/\* An addition operation for StatusCounts \*/

def sumStatusCounts(A: StatusCounts, B: StatusCounts): StatusCounts =

StatusCounts(

retweetCount = sumOptions(A.retweetCount, B.retweetCount),

replyCount = sumOptions(A.replyCount, B.replyCount),

favoriteCount = sumOptions(A.favoriteCount, B.favoriteCount),

quoteCount = sumOptions(A.quoteCount, B.quoteCount),

bookmarkCount = sumOptions(A.bookmarkCount, B.bookmarkCount)

)

def apply(repo: TweetCountsRepository.Type, shouldHydrateBookmarksCount: Gate[Long]): Type = {

/\*

\* Get a StatusCount representing the summed engagements of all previous

\* StatusCounts in an edit chain. Only `countsFields` that are specifically requested

\* are included in the aggregate StatusCount, otherwise those fields are None.

\*/

def getPreviousEngagementCounts(

tweetId: TweetId,

editTweetIds: Seq[TweetId],

countsFields: Set[FieldId]

): Stitch[ValueState[StatusCounts]] = {

val editTweetIdList = previousTweetIds(tweetId, editTweetIds)

// StatusCounts for each edit tweet revision

val statusCountsPerEditVersion: Stitch[Seq[ValueState[StatusCounts]]] =

Stitch.collect(editTweetIdList.map { tweetId =>

// Which tweet count keys to request, as indicated by the tweet options.

val keys: Seq[TweetCountKey] =

TweetCountsHydrator.toKeys(tweetId, countsFields, None)

// A separate StatusCounts for each count field, for `tweetId`

// e.g. Seq(StatusCounts(retweetCounts=5L), StatusCounts(favCounts=6L))

val statusCountsPerCountField: Stitch[Seq[ValueState[StatusCounts]]] =

Stitch.collect(keys.map(key => TweetCountsHydrator.statusCountsRepo(key, repo)))

// Reduce the per-field counts into a single StatusCounts for `tweetId`

statusCountsPerCountField.map { vs =>

// NOTE: This StatusCounts reduction uses different logic than

// `sumStatusCounts`. This reduction takes the latest value for a field.

// instead of summing the fields.

ValueState.sequence(vs).map(TweetCountsHydrator.reduceStatusCounts)

}

})

// Sum together the StatusCounts for each edit tweet revision into a single Status Count

statusCountsPerEditVersion.map { vs =>

ValueState.sequence(vs).map { statusCounts =>

// Reduce a list of StatusCounts into a single StatusCount by summing their fields.

statusCounts.reduce { (a, b) => sumStatusCounts(a, b) }

}

}

}

ValueHydrator[Option[StatusCounts], Ctx] { (inputStatusCounts, ctx) =>

val countsFields: Set[FieldId] = TweetCountsHydrator.filterRequestedCounts(

ctx.opts.forUserId.getOrElse(ctx.userId),

ctx.opts.include.countsFields,

shouldHydrateBookmarksCount,

ctx.featureSwitchResults

)

ctx.editControl match {

case Some(EditControl.Edit(edit)) =>

edit.editControlInitial match {

case Some(initial) =>

val previousStatusCounts: Stitch[ValueState[StatusCounts]] =

getPreviousEngagementCounts(ctx.tweetId, initial.editTweetIds, countsFields)

// Add the new aggregated StatusCount to the TweetData and return it

previousStatusCounts.map { valueState =>

valueState.map { statusCounts => Some(statusCounts) }

}

case None =>

// EditControlInitial is not hydrated within EditControlEdit

// This means we cannot provide aggregated previous counts, we will

// fail open and return the input data unchanged.

Stitch.value(ValueState.partial(inputStatusCounts, hydratedField))

}

case \_ =>

// If the tweet has an EditControlInitial - it's the first Tweet in the Edit Chain

// or has no EditControl - it could be an old Tweet from when no Edit Controls existed

// then the previous counts are set to be equal to None.

Stitch.value(ValueState.unit(None))

}

}.onlyIf { (\_, ctx: Ctx) =>

// only run if the CountsField was requested; note this is ran both on read and write path

TweetCountsHydrator

.filterRequestedCounts(

ctx.opts.forUserId.getOrElse(ctx.userId),

ctx.opts.include.countsFields,

shouldHydrateBookmarksCount,

ctx.featureSwitchResults

).nonEmpty

}

}

}