package com.twitter.tweetypie

package backends

import com.twitter.finagle.stats.NullStatsReceiver

import com.twitter.limiter.thriftscala.FeatureRequest

import com.twitter.tweetypie.backends.LimiterBackend.GetFeatureUsage

import com.twitter.tweetypie.backends.LimiterBackend.IncrementFeature

import com.twitter.tweetypie.backends.LimiterService.Feature

/\*\*

\* Why does LimiterService exist?

\*

\* The underlying Limiter thrift service doesn't support batching. This trait and implementation

\* basically exist to allow a batch like interface to the Limiter. This keeps us from having to

\* spread batching throughout our code base.

\*

\* Why is LimiterService in the backends package?

\*

\* In some ways it is like a backend if the backend supports batching. There is a modest amount of

\* business logic LimiterService, but that logic exists here to allow easier consumption throughout

\* the tweetypie code base. We did look at moving LimiterService to another package, but all likely

\* candidates (service, serverutil) caused circular dependencies.

\*

\* When I need to add functionality, should I add it to LimiterBackend or LimiterService?

\*

\* LimiterBackend is used as a simple wrapper around the Limiter thrift client. The LimiterBackend

\* should be kept as dumb as possible. You will most likely want to add the functionality in

\* LimiterService.

\*/

object LimiterService {

type MinRemaining = (UserId, Option[UserId]) => Future[Int]

type HasRemaining = (UserId, Option[UserId]) => Future[Boolean]

type Increment = (UserId, Option[UserId], Int) => Future[Unit]

type IncrementByOne = (UserId, Option[UserId]) => Future[Unit]

sealed abstract class Feature(val name: String, val hasPerApp: Boolean = false) {

def forUser(userId: UserId): FeatureRequest = FeatureRequest(name, userId = Some(userId))

def forApp(appId: AppId): Option[FeatureRequest] =

if (hasPerApp) {

Some(

FeatureRequest(

s"${name}\_per\_app",

applicationId = Some(appId),

identifier = Some(appId.toString)

)

)

} else {

None

}

}

object Feature {

case object Updates extends Feature("updates", hasPerApp = true)

case object MediaTagCreate extends Feature("media\_tag\_create")

case object TweetCreateFailure extends Feature("tweet\_creation\_failure")

}

def fromBackend(

incrementFeature: IncrementFeature,

getFeatureUsage: GetFeatureUsage,

getAppId: => Option[

AppId

], // the call-by-name here to invoke per request to get the current request's app id

stats: StatsReceiver = NullStatsReceiver

): LimiterService =

new LimiterService {

def increment(

feature: Feature

)(

userId: UserId,

contributorUserId: Option[UserId],

amount: Int

): Future[Unit] = {

Future.when(amount > 0) {

def increment(req: FeatureRequest): Future[Unit] = incrementFeature((req, amount))

val incrementUser: Option[Future[Unit]] =

Some(increment(feature.forUser(userId)))

val incrementContributor: Option[Future[Unit]] =

for {

id <- contributorUserId

if id != userId

} yield increment(feature.forUser(id))

val incrementPerApp: Option[Future[Unit]] =

for {

appId <- getAppId

req <- feature.forApp(appId)

} yield increment(req)

Future.collect(Seq(incrementUser, incrementContributor, incrementPerApp).flatten)

}

}

def minRemaining(

feature: Feature

)(

userId: UserId,

contributorUserId: Option[UserId]

): Future[Int] = {

def getRemaining(req: FeatureRequest): Future[Int] = getFeatureUsage(req).map(\_.remaining)

val getUserRemaining: Option[Future[Int]] =

Some(getRemaining(feature.forUser(userId)))

val getContributorRemaining: Option[Future[Int]] =

contributorUserId.map(id => getRemaining(feature.forUser(id)))

val getPerAppRemaining: Option[Future[Int]] =

for {

appId <- getAppId

req <- feature.forApp(appId)

} yield getRemaining(req)

Future

.collect(Seq(getUserRemaining, getContributorRemaining, getPerAppRemaining).flatten)

.map(\_.min)

}

}

}

trait LimiterService {

/\*\*

\* Increment the feature count for both the user and the contributor. If either increment fails,

\* the resulting future will be the first exception encountered.

\*

\* @param feature The feature that is incremented

\* @param userId The current user tied to the current request

\* @param contributorUserId The contributor, if one exists, tied to the current request

\* @param amount The amount that each feature should be incremented.

\*/

def increment(

feature: Feature

)(

userId: UserId,

contributorUserId: Option[UserId],

amount: Int

): Future[Unit]

/\*\*

\* Increment the feature count, by one, for both the user and the contributor. If either

\* increment fails, the resulting future will be the first exception encountered.

\*

\* @param feature The feature that is incremented

\* @param userId The current user tied to the current request

\* @param contributorUserId The contributor, if one exists, tied to the current request

\*

\* @see [[increment]] if you want to increment a feature by a specified amount

\*/

def incrementByOne(

feature: Feature

)(

userId: UserId,

contributorUserId: Option[UserId]

): Future[Unit] =

increment(feature)(userId, contributorUserId, 1)

/\*\*

\* The minimum remaining limit between the user and contributor. If an exception occurs, then the

\* resulting Future will be the first exception encountered.

\*

\* @param feature The feature that is queried

\* @param userId The current user tied to the current request

\* @param contributorUserId The contributor, if one exists, tied to the current request

\*

\* @return a `Future[Int]` with the minimum limit left between the user and contributor

\*/

def minRemaining(feature: Feature)(userId: UserId, contributorUserId: Option[UserId]): Future[Int]

/\*\*

\* Can the user and contributor increment the given feature. If the result cannot be determined

\* because of an exception, then we assume they can increment. This will allow us to continue

\* servicing requests even if the limiter service isn't responding.

\*

\* @param feature The feature that is queried

\* @param userId The current user tied to the current request

\* @param contributorUserId The contributor, if one exists, tied to the current request

\* @return a `Future[Boolean]` with true if both the user and contributor have remaining limit

\* cap.

\*

\* @see [[minRemaining]] if you would like to handle any exceptions that occur on your own

\*/

def hasRemaining(

feature: Feature

)(

userId: UserId,

contributorUserId: Option[UserId]

): Future[Boolean] =

minRemaining(feature)(userId, contributorUserId)

.map(\_ > 0)

.handle { case \_ => true }

}