package com.twitter.tweetypie

package repository

import com.twitter.gizmoduck.thriftscala.LookupContext

import com.twitter.gizmoduck.thriftscala.UserResponseState

import com.twitter.gizmoduck.thriftscala.UserResult

import com.twitter.servo.cache.ScopedCacheKey

import com.twitter.servo.json.syntax.\_

import com.twitter.spam.rtf.thriftscala.SafetyLevel

import com.twitter.stitch.NotFound

import com.twitter.stitch.SeqGroup

import com.twitter.stitch.Stitch

import com.twitter.stitch.compat.LegacySeqGroup

import com.twitter.tweetypie.backends.Gizmoduck

import com.twitter.tweetypie.core.\_

import com.twitter.util.Base64Long.toBase64

import com.twitter.util.logging.Logger

import com.twitter.visibility.thriftscala.UserVisibilityResult

import scala.util.control.NoStackTrace

sealed trait UserKey

object UserKey {

def byId(userId: UserId): UserKey = UserIdKey(userId)

def byScreenName(screenName: String): UserKey = ScreenNameKey.toLowerCase(screenName)

def apply(userId: UserId): UserKey = UserIdKey(userId)

def apply(screenName: String): UserKey = ScreenNameKey.toLowerCase(screenName)

}

case class UserIdKey(userId: UserId)

extends ScopedCacheKey("t", "usr", 1, "id", toBase64(userId))

with UserKey

object ScreenNameKey {

def toLowerCase(screenName: String): ScreenNameKey = ScreenNameKey(screenName.toLowerCase)

}

/\*\*

\* Use UserKey.apply(String) instead of ScreenNameKey(String) to construct a key,

\* as it will down-case the screen-name to better utilize the user cache.

\*/

case class ScreenNameKey private (screenName: String)

extends ScopedCacheKey("t", "usr", 1, "sn", screenName)

with UserKey

/\*\*

\* A set of flags, used in UserQuery, which control whether to include or filter out

\* users in various non-standard states.

\*/

case class UserVisibility(

filterProtected: Boolean,

filterSuspended: Boolean,

filterDeactivated: Boolean,

filterOffboardedAndErased: Boolean,

filterNoScreenName: Boolean,

filterPeriscope: Boolean,

filterSoft: Boolean)

object UserVisibility {

/\*\*

\* No filtering, can see every user that gizmoduck can return.

\*/

val All: UserVisibility = UserVisibility(

filterProtected = false,

filterSuspended = false,

filterDeactivated = false,

filterOffboardedAndErased = false,

filterNoScreenName = false,

filterPeriscope = false,

filterSoft = false

)

/\*\*

\* Only includes users that would be visible to a non-logged in user,

\* or a logged in user where the following graph is checked for

\* protected users.

\*

\* no-screen-name, soft, and periscope users are visible, but not

\* mentionable.

\*/

val Visible: UserVisibility = UserVisibility(

filterProtected = true,

filterSuspended = true,

filterDeactivated = true,

filterOffboardedAndErased = true,

filterNoScreenName = false,

filterPeriscope = false,

filterSoft = false

)

val MediaTaggable: UserVisibility = UserVisibility(

filterProtected = false,

filterSuspended = true,

filterDeactivated = true,

filterOffboardedAndErased = true,

filterNoScreenName = true,

filterPeriscope = true,

filterSoft = true

)

/\*\*

\* Includes all mentionable users (filter deactivated/offboarded/erased/no-screen-name users)

\*/

val Mentionable: UserVisibility = UserVisibility(

filterProtected = false,

filterSuspended = false,

filterDeactivated = false,

filterOffboardedAndErased = true,

filterNoScreenName = true,

filterPeriscope = true,

filterSoft = true

)

}

/\*\*

\* The `visibility` field includes a set of flags that indicate whether users in

\* various non-standard states should be included in the `found` results, or filtered

\* out. By default, "filtered out" means to treat them as `notFound`, but if `filteredAsFailure`

\* is true, then the filtered users will be indicated in a [[UserFilteredFailure]] result.

\*/

case class UserQueryOptions(

queryFields: Set[UserField] = Set.empty,

visibility: UserVisibility,

forUserId: Option[UserId] = None,

filteredAsFailure: Boolean = false,

safetyLevel: Option[SafetyLevel] = None) {

def toLookupContext: LookupContext =

LookupContext(

includeFailed = true,

forUserId = forUserId,

includeProtected = !visibility.filterProtected,

includeSuspended = !visibility.filterSuspended,

includeDeactivated = !visibility.filterDeactivated,

includeErased = !visibility.filterOffboardedAndErased,

includeNoScreenNameUsers = !visibility.filterNoScreenName,

includePeriscopeUsers = !visibility.filterPeriscope,

includeSoftUsers = !visibility.filterSoft,

includeOffboarded = !visibility.filterOffboardedAndErased,

safetyLevel = safetyLevel

)

}

case class UserLookupFailure(message: String, state: UserResponseState) extends RuntimeException {

override def getMessage(): String =

s"$message: responseState = $state"

}

/\*\*

\* Indicates a failure due to the user being filtered.

\*

\* @see [[GizmoduckUserRepository.FilteredStates]]

\*/

case class UserFilteredFailure(state: UserResponseState, reason: Option[UserVisibilityResult])

extends Exception

with NoStackTrace

object UserRepository {

type Type = (UserKey, UserQueryOptions) => Stitch[User]

type Optional = (UserKey, UserQueryOptions) => Stitch[Option[User]]

def optional(repo: Type): Optional =

(userKey, queryOptions) => repo(userKey, queryOptions).liftNotFoundToOption

def userGetter(

userRepo: UserRepository.Optional,

opts: UserQueryOptions

): UserKey => Future[Option[User]] =

userKey => Stitch.run(userRepo(userKey, opts))

}

object GizmoduckUserRepository {

private[this] val log = Logger(getClass)

def apply(

getById: Gizmoduck.GetById,

getByScreenName: Gizmoduck.GetByScreenName,

maxRequestSize: Int = Int.MaxValue

): UserRepository.Type = {

case class GetBy[K](

opts: UserQueryOptions,

get: ((LookupContext, Seq[K], Set[UserField])) => Future[Seq[UserResult]])

extends SeqGroup[K, UserResult] {

override def run(keys: Seq[K]): Future[Seq[Try[UserResult]]] =

LegacySeqGroup.liftToSeqTry(get((opts.toLookupContext, keys, opts.queryFields)))

override def maxSize: Int = maxRequestSize

}

(key, opts) => {

val result =

key match {

case UserIdKey(id) => Stitch.call(id, GetBy(opts, getById))

case ScreenNameKey(sn) => Stitch.call(sn, GetBy(opts, getByScreenName))

}

result.flatMap(r => Stitch.const(toTryUser(r, opts.filteredAsFailure)))

}

}

private def toTryUser(

userResult: UserResult,

filteredAsFailure: Boolean

): Try[User] =

userResult.responseState match {

case s if s.forall(SuccessStates.contains(\_)) =>

userResult.user match {

case Some(u) =>

Return(u)

case None =>

log.warn(

s"User expected to be present, but not found in:\n${userResult.prettyPrint}"

)

// This should never happen, but if it does, treat it as the

// user being returned as NotFound.

Throw(NotFound)

}

case Some(s) if NotFoundStates.contains(s) =>

Throw(NotFound)

case Some(s) if FilteredStates.contains(s) =>

Throw(if (filteredAsFailure) UserFilteredFailure(s, userResult.unsafeReason) else NotFound)

case Some(UserResponseState.Failed) =>

def lookupFailure(msg: String) =

UserLookupFailure(msg, UserResponseState.Failed)

Throw {

userResult.failureReason

.map { reason =>

reason.internalServerError

.orElse {

reason.overCapacity.map { e =>

// Convert Gizmoduck OverCapacity to Tweetypie

// OverCapacity exception, explaining that it was

// propagated from Gizmoduck.

OverCapacity(s"gizmoduck over capacity: ${e.message}")

}

}

.orElse(reason.unexpectedException.map(lookupFailure))

.getOrElse(lookupFailure("failureReason empty"))

}

.getOrElse(lookupFailure("failureReason missing"))

}

case Some(unexpected) =>

Throw(UserLookupFailure("Unexpected response state", unexpected))

}

/\*\*

\* States that we expect to correspond to a user being returned.

\*/

val SuccessStates: Set[UserResponseState] =

Set[UserResponseState](

UserResponseState.Found,

UserResponseState.Partial

)

/\*\*

\* States that always correspond to a NotFound response.

\*/

val NotFoundStates: Set[UserResponseState] =

Set[UserResponseState](

UserResponseState.NotFound,

// These are really filtered out, but we treat them as not found

// since we don't have analogous filtering states for tweets.

UserResponseState.PeriscopeUser,

UserResponseState.SoftUser,

UserResponseState.NoScreenNameUser

)

/\*\*

\* Response states that correspond to a FilteredState

\*/

val FilteredStates: Set[UserResponseState] =

Set(

UserResponseState.DeactivatedUser,

UserResponseState.OffboardedUser,

UserResponseState.ErasedUser,

UserResponseState.SuspendedUser,

UserResponseState.ProtectedUser,

UserResponseState.UnsafeUser

)

}