package com.twitter.tweetypie

package repository

import com.twitter.gizmoduck.thriftscala.UserResponseState

import com.twitter.spam.rtf.thriftscala.{SafetyLevel => ThriftSafetyLevel}

import com.twitter.stitch.NotFound

import com.twitter.stitch.Stitch

import com.twitter.tweetypie.core.\_

import com.twitter.tweetypie.thriftscala.UserIdentity

import com.twitter.visibility.interfaces.tweets.UserUnavailableStateVisibilityLibrary

import com.twitter.visibility.interfaces.tweets.UserUnavailableStateVisibilityRequest

import com.twitter.visibility.models.SafetyLevel

import com.twitter.visibility.models.UserUnavailableStateEnum

import com.twitter.visibility.models.ViewerContext

import com.twitter.visibility.thriftscala.UserVisibilityResult

/\*\*

\* Some types of user (e.g. frictionless users) may not

\* have profiles, so a missing UserIdentity may mean that the user

\* does not exist, or that the user does not have a profile.

\*/

object UserIdentityRepository {

type Type = UserKey => Stitch[UserIdentity]

def apply(repo: UserRepository.Type): Type = { key =>

val opts = UserQueryOptions(Set(UserField.Profile), UserVisibility.Mentionable)

repo(key, opts)

.map { user =>

user.profile.map { profile =>

UserIdentity(

id = user.id,

screenName = profile.screenName,

realName = profile.name

)

}

}

.lowerFromOption()

}

}

object UserProtectionRepository {

type Type = UserKey => Stitch[Boolean]

def apply(repo: UserRepository.Type): Type = {

val opts = UserQueryOptions(Set(UserField.Safety), UserVisibility.All)

userKey =>

repo(userKey, opts)

.map(user => user.safety.map(\_.isProtected))

.lowerFromOption()

}

}

/\*\*

\* Query Gizmoduck to check if a user `forUserId` can see user `userKey`.

\* If forUserId is Some(), this will also check protected relationship,

\* if it's None, it will check others as per UserVisibility.Visible policy in

\* UserRepository.scala. If forUserId is None, this doesn't verify any

\* relationships, visibility is determined based solely on user's

\* properties (eg. deactivated, suspended, etc)

\*/

object UserVisibilityRepository {

type Type = Query => Stitch[Option[FilteredState.Unavailable]]

case class Query(

userKey: UserKey,

forUserId: Option[UserId],

tweetId: TweetId,

isRetweet: Boolean,

isInnerQuotedTweet: Boolean,

safetyLevel: Option[ThriftSafetyLevel])

def apply(

repo: UserRepository.Type,

userUnavailableAuthorStateVisibilityLibrary: UserUnavailableStateVisibilityLibrary.Type

): Type =

query => {

repo(

query.userKey,

UserQueryOptions(

Set(),

UserVisibility.Visible,

forUserId = query.forUserId,

filteredAsFailure = true,

safetyLevel = query.safetyLevel

)

)

// We don't actually care about the response here (User's data), only whether

// it was filtered or not

.map { case \_ => None }

.rescue {

case fs: FilteredState.Unavailable => Stitch.value(Some(fs))

case UserFilteredFailure(state, reason) =>

userUnavailableAuthorStateVisibilityLibrary

.apply(

UserUnavailableStateVisibilityRequest(

query.safetyLevel

.map(SafetyLevel.fromThrift).getOrElse(SafetyLevel.FilterDefault),

query.tweetId,

ViewerContext.fromContextWithViewerIdFallback(query.forUserId),

toUserUnavailableState(state, reason),

query.isRetweet,

query.isInnerQuotedTweet

)

).map(VisibilityResultToFilteredState.toFilteredStateUnavailable)

case NotFound => Stitch.value(Some(FilteredState.Unavailable.Author.NotFound))

}

}

def toUserUnavailableState(

userResponseState: UserResponseState,

userVisibilityResult: Option[UserVisibilityResult]

): UserUnavailableStateEnum = {

(userResponseState, userVisibilityResult) match {

case (UserResponseState.DeactivatedUser, \_) => UserUnavailableStateEnum.Deactivated

case (UserResponseState.OffboardedUser, \_) => UserUnavailableStateEnum.Offboarded

case (UserResponseState.ErasedUser, \_) => UserUnavailableStateEnum.Erased

case (UserResponseState.SuspendedUser, \_) => UserUnavailableStateEnum.Suspended

case (UserResponseState.ProtectedUser, \_) => UserUnavailableStateEnum.Protected

case (\_, Some(result)) => UserUnavailableStateEnum.Filtered(result)

case \_ => UserUnavailableStateEnum.Unavailable

}

}

}

object UserViewRepository {

type Type = Query => Stitch[User]

case class Query(

userKey: UserKey,

forUserId: Option[UserId],

visibility: UserVisibility,

queryFields: Set[UserField] = Set(UserField.View))

def apply(repo: UserRepository.Type): UserViewRepository.Type =

query =>

repo(query.userKey, UserQueryOptions(query.queryFields, query.visibility, query.forUserId))

}