package com.twitter.unified\_user\_actions.adapter

import com.twitter.inject.Test

import com.twitter.socialgraph.thriftscala.Action

import com.twitter.socialgraph.thriftscala.BlockGraphEvent

import com.twitter.socialgraph.thriftscala.FollowGraphEvent

import com.twitter.socialgraph.thriftscala.FollowRequestGraphEvent

import com.twitter.socialgraph.thriftscala.FollowRetweetsGraphEvent

import com.twitter.socialgraph.thriftscala.LogEventContext

import com.twitter.socialgraph.thriftscala.MuteGraphEvent

import com.twitter.socialgraph.thriftscala.ReportAsAbuseGraphEvent

import com.twitter.socialgraph.thriftscala.ReportAsSpamGraphEvent

import com.twitter.socialgraph.thriftscala.SrcTargetRequest

import com.twitter.socialgraph.thriftscala.WriteEvent

import com.twitter.socialgraph.thriftscala.WriteRequestResult

import com.twitter.unified\_user\_actions.adapter.social\_graph\_event.SocialGraphAdapter

import com.twitter.unified\_user\_actions.thriftscala.\_

import com.twitter.util.Time

import org.scalatest.prop.TableDrivenPropertyChecks

import org.scalatest.prop.TableFor1

import org.scalatest.prop.TableFor3

class SocialGraphAdapterSpec extends Test with TableDrivenPropertyChecks {

trait Fixture {

val frozenTime: Time = Time.fromMilliseconds(1658949273000L)

val testLogEventContext: LogEventContext = LogEventContext(

timestamp = 1001L,

hostname = "",

transactionId = "",

socialGraphClientId = "",

loggedInUserId = Some(1111L),

)

val testWriteRequestResult: WriteRequestResult = WriteRequestResult(

request = SrcTargetRequest(

source = 1111L,

target = 2222L

)

)

val testWriteRequestResultWithValidationError: WriteRequestResult = WriteRequestResult(

request = SrcTargetRequest(

source = 1111L,

target = 2222L

),

validationError = Some("action unsuccessful")

)

val baseEvent: WriteEvent = WriteEvent(

context = testLogEventContext,

action = Action.AcceptFollowRequest

)

val sgFollowEvent: WriteEvent = baseEvent.copy(

action = Action.Follow,

follow = Some(List(FollowGraphEvent(testWriteRequestResult))))

val sgUnfollowEvent: WriteEvent = baseEvent.copy(

action = Action.Unfollow,

follow = Some(List(FollowGraphEvent(testWriteRequestResult))))

val sgFollowRedundantEvent: WriteEvent = baseEvent.copy(

action = Action.Follow,

follow = Some(

List(

FollowGraphEvent(

result = testWriteRequestResult,

redundantOperation = Some(true)

))))

val sgFollowRedundantIsFalseEvent: WriteEvent = baseEvent.copy(

action = Action.Follow,

follow = Some(

List(

FollowGraphEvent(

result = testWriteRequestResult,

redundantOperation = Some(false)

))))

val sgUnfollowRedundantEvent: WriteEvent = baseEvent.copy(

action = Action.Unfollow,

follow = Some(

List(

FollowGraphEvent(

result = testWriteRequestResult,

redundantOperation = Some(true)

))))

val sgUnfollowRedundantIsFalseEvent: WriteEvent = baseEvent.copy(

action = Action.Unfollow,

follow = Some(

List(

FollowGraphEvent(

result = testWriteRequestResult,

redundantOperation = Some(false)

))))

val sgUnsuccessfulFollowEvent: WriteEvent = baseEvent.copy(

action = Action.Follow,

follow = Some(List(FollowGraphEvent(testWriteRequestResultWithValidationError))))

val sgUnsuccessfulUnfollowEvent: WriteEvent = baseEvent.copy(

action = Action.Unfollow,

follow = Some(List(FollowGraphEvent(testWriteRequestResultWithValidationError))))

val sgBlockEvent: WriteEvent = baseEvent.copy(

action = Action.Block,

block = Some(List(BlockGraphEvent(testWriteRequestResult))))

val sgUnsuccessfulBlockEvent: WriteEvent = baseEvent.copy(

action = Action.Block,

block = Some(List(BlockGraphEvent(testWriteRequestResultWithValidationError))))

val sgUnblockEvent: WriteEvent = baseEvent.copy(

action = Action.Unblock,

block = Some(List(BlockGraphEvent(testWriteRequestResult))))

val sgUnsuccessfulUnblockEvent: WriteEvent = baseEvent.copy(

action = Action.Unblock,

block = Some(List(BlockGraphEvent(testWriteRequestResultWithValidationError))))

val sgMuteEvent: WriteEvent = baseEvent.copy(

action = Action.Mute,

mute = Some(List(MuteGraphEvent(testWriteRequestResult))))

val sgUnsuccessfulMuteEvent: WriteEvent = baseEvent.copy(

action = Action.Mute,

mute = Some(List(MuteGraphEvent(testWriteRequestResultWithValidationError))))

val sgUnmuteEvent: WriteEvent = baseEvent.copy(

action = Action.Unmute,

mute = Some(List(MuteGraphEvent(testWriteRequestResult))))

val sgUnsuccessfulUnmuteEvent: WriteEvent = baseEvent.copy(

action = Action.Unmute,

mute = Some(List(MuteGraphEvent(testWriteRequestResultWithValidationError))))

val sgCreateFollowRequestEvent: WriteEvent = baseEvent.copy(

action = Action.CreateFollowRequest,

followRequest = Some(List(FollowRequestGraphEvent(testWriteRequestResult)))

)

val sgCancelFollowRequestEvent: WriteEvent = baseEvent.copy(

action = Action.CancelFollowRequest,

followRequest = Some(List(FollowRequestGraphEvent(testWriteRequestResult)))

)

val sgAcceptFollowRequestEvent: WriteEvent = baseEvent.copy(

action = Action.AcceptFollowRequest,

followRequest = Some(List(FollowRequestGraphEvent(testWriteRequestResult)))

)

val sgAcceptFollowRetweetEvent: WriteEvent = baseEvent.copy(

action = Action.FollowRetweets,

followRetweets = Some(List(FollowRetweetsGraphEvent(testWriteRequestResult)))

)

val sgAcceptUnfollowRetweetEvent: WriteEvent = baseEvent.copy(

action = Action.UnfollowRetweets,

followRetweets = Some(List(FollowRetweetsGraphEvent(testWriteRequestResult)))

)

val sgReportAsSpamEvent: WriteEvent = baseEvent.copy(

action = Action.ReportAsSpam,

reportAsSpam = Some(

List(

ReportAsSpamGraphEvent(

result = testWriteRequestResult

))))

val sgReportAsAbuseEvent: WriteEvent = baseEvent.copy(

action = Action.ReportAsAbuse,

reportAsAbuse = Some(

List(

ReportAsAbuseGraphEvent(

result = testWriteRequestResult

))))

def getExpectedUUA(

userId: Long,

actionProfileId: Long,

sourceTimestampMs: Long,

actionType: ActionType,

socialGraphAction: Option[Action] = None

): UnifiedUserAction = {

val actionItem = socialGraphAction match {

case Some(sgAction) =>

Item.ProfileInfo(

ProfileInfo(

actionProfileId = actionProfileId,

profileActionInfo = Some(

ProfileActionInfo.ServerProfileReport(

ServerProfileReport(reportType = sgAction)

))

)

)

case \_ =>

Item.ProfileInfo(

ProfileInfo(

actionProfileId = actionProfileId

)

)

}

UnifiedUserAction(

userIdentifier = UserIdentifier(userId = Some(userId)),

item = actionItem,

actionType = actionType,

eventMetadata = EventMetadata(

sourceTimestampMs = sourceTimestampMs,

receivedTimestampMs = frozenTime.inMilliseconds,

sourceLineage = SourceLineage.ServerSocialGraphEvents

)

)

}

val expectedUuaFollow: UnifiedUserAction = getExpectedUUA(

userId = 1111L,

actionProfileId = 2222L,

sourceTimestampMs = 1001L,

actionType = ActionType.ServerProfileFollow

)

val expectedUuaUnfollow: UnifiedUserAction = getExpectedUUA(

userId = 1111L,

actionProfileId = 2222L,

sourceTimestampMs = 1001L,

actionType = ActionType.ServerProfileUnfollow

)

val expectedUuaMute: UnifiedUserAction = getExpectedUUA(

userId = 1111L,

actionProfileId = 2222L,

sourceTimestampMs = 1001L,

actionType = ActionType.ServerProfileMute

)

val expectedUuaUnmute: UnifiedUserAction = getExpectedUUA(

userId = 1111L,

actionProfileId = 2222L,

sourceTimestampMs = 1001L,

actionType = ActionType.ServerProfileUnmute

)

val expectedUuaBlock: UnifiedUserAction = getExpectedUUA(

userId = 1111L,

actionProfileId = 2222L,

sourceTimestampMs = 1001L,

actionType = ActionType.ServerProfileBlock

)

val expectedUuaUnblock: UnifiedUserAction = getExpectedUUA(

userId = 1111L,

actionProfileId = 2222L,

sourceTimestampMs = 1001L,

actionType = ActionType.ServerProfileUnblock

)

val expectedUuaReportAsSpam: UnifiedUserAction = getExpectedUUA(

userId = 1111L,

actionProfileId = 2222L,

sourceTimestampMs = 1001L,

actionType = ActionType.ServerProfileReport,

socialGraphAction = Some(Action.ReportAsSpam)

)

val expectedUuaReportAsAbuse: UnifiedUserAction = getExpectedUUA(

userId = 1111L,

actionProfileId = 2222L,

sourceTimestampMs = 1001L,

actionType = ActionType.ServerProfileReport,

socialGraphAction = Some(Action.ReportAsAbuse)

)

}

test("SocialGraphAdapter ignore events not in the list") {

new Fixture {

Time.withTimeAt(frozenTime) { \_ =>

val ignoredSocialGraphEvents: TableFor1[WriteEvent] = Table(

"ignoredSocialGraphEvents",

sgAcceptUnfollowRetweetEvent,

sgAcceptFollowRequestEvent,

sgAcceptFollowRetweetEvent,

sgCreateFollowRequestEvent,

sgCancelFollowRequestEvent,

)

forEvery(ignoredSocialGraphEvents) { writeEvent: WriteEvent =>

val actual = SocialGraphAdapter.adaptEvent(writeEvent)

assert(actual.isEmpty)

}

}

}

}

test("Test SocialGraphAdapter consuming Write events") {

new Fixture {

Time.withTimeAt(frozenTime) { \_ =>

val socialProfileActions: TableFor3[String, WriteEvent, UnifiedUserAction] = Table(

("actionType", "event", "expectedUnifiedUserAction"),

("ProfileFollow", sgFollowEvent, expectedUuaFollow),

("ProfileUnfollow", sgUnfollowEvent, expectedUuaUnfollow),

("ProfileBlock", sgBlockEvent, expectedUuaBlock),

("ProfileUnBlock", sgUnblockEvent, expectedUuaUnblock),

("ProfileMute", sgMuteEvent, expectedUuaMute),

("ProfileUnmute", sgUnmuteEvent, expectedUuaUnmute),

("ProfileReportAsSpam", sgReportAsSpamEvent, expectedUuaReportAsSpam),

("ProfileReportAsAbuse", sgReportAsAbuseEvent, expectedUuaReportAsAbuse),

)

forEvery(socialProfileActions) {

(\_: String, event: WriteEvent, expected: UnifiedUserAction) =>

val actual = SocialGraphAdapter.adaptEvent(event)

assert(Seq(expected) === actual)

}

}

}

}

test("SocialGraphAdapter ignore redundant follow/unfollow events") {

new Fixture {

Time.withTimeAt(frozenTime) { \_ =>

val socialGraphActions: TableFor3[String, WriteEvent, Seq[UnifiedUserAction]] = Table(

("actionType", "ignoredRedundantFollowUnfollowEvents", "expectedUnifiedUserAction"),

("ProfileFollow", sgFollowRedundantEvent, Nil),

("ProfileFollow", sgFollowRedundantIsFalseEvent, Seq(expectedUuaFollow)),

("ProfileUnfollow", sgUnfollowRedundantEvent, Nil),

("ProfileUnfollow", sgUnfollowRedundantIsFalseEvent, Seq(expectedUuaUnfollow))

)

forEvery(socialGraphActions) {

(\_: String, event: WriteEvent, expected: Seq[UnifiedUserAction]) =>

val actual = SocialGraphAdapter.adaptEvent(event)

assert(expected === actual)

}

}

}

}

test("SocialGraphAdapter ignore Unsuccessful SocialGraph events") {

new Fixture {

Time.withTimeAt(frozenTime) { \_ =>

val unsuccessfulSocialGraphEvents: TableFor1[WriteEvent] = Table(

"ignoredSocialGraphEvents",

sgUnsuccessfulFollowEvent,

sgUnsuccessfulUnfollowEvent,

sgUnsuccessfulBlockEvent,

sgUnsuccessfulUnblockEvent,

sgUnsuccessfulMuteEvent,

sgUnsuccessfulUnmuteEvent

)

forEvery(unsuccessfulSocialGraphEvents) { writeEvent: WriteEvent =>

val actual = SocialGraphAdapter.adaptEvent(writeEvent)

assert(actual.isEmpty)

}

}

}

}

}