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

import com.twitter.clientapp.thriftscala.SuggestionDetails

import com.twitter.clientapp.thriftscala.\_

import com.twitter.search.common.constants.thriftscala.ThriftQuerySource

import com.twitter.search.common.constants.thriftscala.TweetResultSource

import com.twitter.search.common.constants.thriftscala.UserResultSource

import com.twitter.suggests.controller\_data.search\_response.item\_types.thriftscala.ItemTypesControllerData

import com.twitter.suggests.controller\_data.search\_response.request.thriftscala.RequestControllerData

import com.twitter.suggests.controller\_data.search\_response.thriftscala.SearchResponseControllerData

import com.twitter.suggests.controller\_data.search\_response.tweet\_types.thriftscala.TweetTypesControllerData

import com.twitter.suggests.controller\_data.search\_response.user\_types.thriftscala.UserTypesControllerData

import com.twitter.suggests.controller\_data.search\_response.v1.thriftscala.{

SearchResponseControllerData => SearchResponseControllerDataV1

}

import com.twitter.suggests.controller\_data.thriftscala.ControllerData

import com.twitter.suggests.controller\_data.v2.thriftscala.{ControllerData => ControllerDataV2}

import com.twitter.util.mock.Mockito

import org.junit.runner.RunWith

import org.scalatest.funsuite.AnyFunSuite

import org.scalatest.matchers.should.Matchers

import org.scalatest.prop.TableDrivenPropertyChecks

import org.scalatestplus.junit.JUnitRunner

import com.twitter.unified\_user\_actions.adapter.client\_event.SearchInfoUtils

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

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

import org.scalatest.prop.TableFor2

@RunWith(classOf[JUnitRunner])

class SearchInfoUtilsSpec

extends AnyFunSuite

with Matchers

with Mockito

with TableDrivenPropertyChecks {

trait Fixture {

def mkControllerData(

queryOpt: Option[String],

querySourceOpt: Option[Int] = None,

traceId: Option[Long] = None,

requestJoinId: Option[Long] = None

): ControllerData = {

ControllerData.V2(

ControllerDataV2.SearchResponse(

SearchResponseControllerData.V1(

SearchResponseControllerDataV1(requestControllerData = Some(

RequestControllerData(

rawQuery = queryOpt,

querySource = querySourceOpt,

traceId = traceId,

requestJoinId = requestJoinId

)))

)))

}

def mkTweetTypeControllerData(bitmap: Long, topicId: Option[Long] = None): ControllerData.V2 = {

ControllerData.V2(

ControllerDataV2.SearchResponse(

SearchResponseControllerData.V1(

SearchResponseControllerDataV1(itemTypesControllerData = Some(

ItemTypesControllerData.TweetTypesControllerData(

TweetTypesControllerData(

tweetTypesBitmap = Some(bitmap),

topicId = topicId

))

))

)))

}

def mkUserTypeControllerData(bitmap: Long): ControllerData.V2 = {

ControllerData.V2(

ControllerDataV2.SearchResponse(

SearchResponseControllerData.V1(

SearchResponseControllerDataV1(itemTypesControllerData = Some(

ItemTypesControllerData.UserTypesControllerData(UserTypesControllerData(

userTypesBitmap = Some(bitmap)

))

))

)))

}

}

test("getQueryOptFromControllerDataFromItem should return query if present in controller data") {

new Fixture {

val controllerData: ControllerData = mkControllerData(Some("twitter"))

val suggestionDetails: SuggestionDetails =

SuggestionDetails(decodedControllerData = Some(controllerData))

val item: Item = Item(suggestionDetails = Some(suggestionDetails))

val result: Option[String] = new SearchInfoUtils(item).getQueryOptFromControllerDataFromItem

result shouldEqual Option("twitter")

}

}

test("getRequestJoinId should return requestJoinId if present in controller data") {

new Fixture {

val controllerData: ControllerData = mkControllerData(

Some("twitter"),

traceId = Some(11L),

requestJoinId = Some(12L)

)

val suggestionDetails: SuggestionDetails =

SuggestionDetails(decodedControllerData = Some(controllerData))

val item: Item = Item(suggestionDetails = Some(suggestionDetails))

val infoUtils = new SearchInfoUtils(item)

infoUtils.getTraceId shouldEqual Some(11L)

infoUtils.getRequestJoinId shouldEqual Some(12L)

}

}

test("getQueryOptFromControllerDataFromItem should return None if no suggestion details") {

new Fixture {

val suggestionDetails: SuggestionDetails = SuggestionDetails()

val item: Item = Item(suggestionDetails = Some(suggestionDetails))

val result: Option[String] = new SearchInfoUtils(item).getQueryOptFromControllerDataFromItem

result shouldEqual None

}

}

test("getQueryOptFromSearchDetails should return query if present") {

new Fixture {

val searchDetails: SearchDetails = SearchDetails(query = Some("twitter"))

val result: Option[String] = new SearchInfoUtils(Item()).getQueryOptFromSearchDetails(

LogEvent(eventName = "", searchDetails = Some(searchDetails))

)

result shouldEqual Option("twitter")

}

}

test("getQueryOptFromSearchDetails should return None if not present") {

new Fixture {

val searchDetails: SearchDetails = SearchDetails()

val result: Option[String] = new SearchInfoUtils(Item()).getQueryOptFromSearchDetails(

LogEvent(eventName = "", searchDetails = Some(searchDetails))

)

result shouldEqual None

}

}

test("getQuerySourceOptFromControllerDataFromItem should return QuerySource if present") {

new Fixture {

// 1 is Typed Query

val controllerData: ControllerData = mkControllerData(Some("twitter"), Some(1))

val item: Item = Item(

suggestionDetails = Some(

SuggestionDetails(

decodedControllerData = Some(controllerData)

))

)

new SearchInfoUtils(item).getQuerySourceOptFromControllerDataFromItem shouldEqual Some(

ThriftQuerySource.TypedQuery)

}

}

test("getQuerySourceOptFromControllerDataFromItem should return None if not present") {

new Fixture {

val controllerData: ControllerData = mkControllerData(Some("twitter"), None)

val item: Item = Item(

suggestionDetails = Some(

SuggestionDetails(

decodedControllerData = Some(controllerData)

))

)

new SearchInfoUtils(item).getQuerySourceOptFromControllerDataFromItem shouldEqual None

}

}

test("Decoding Tweet Result Sources bitmap") {

new Fixture {

TweetResultSource.list

.foreach { tweetResultSource =>

val bitmap = (1 << tweetResultSource.getValue()).toLong

val controllerData = mkTweetTypeControllerData(bitmap)

val item = Item(

suggestionDetails = Some(

SuggestionDetails(

decodedControllerData = Some(controllerData)

))

)

val result = new SearchInfoUtils(item).getTweetResultSources

result shouldEqual Some(Set(tweetResultSource))

}

}

}

test("Decoding multiple Tweet Result Sources") {

new Fixture {

val tweetResultSources: Set[TweetResultSource] =

Set(TweetResultSource.QueryInteractionGraph, TweetResultSource.QueryExpansion)

val bitmap: Long = tweetResultSources.foldLeft(0L) {

case (acc, source) => acc + (1 << source.getValue())

}

val controllerData: ControllerData.V2 = mkTweetTypeControllerData(bitmap)

val item: Item = Item(

suggestionDetails = Some(

SuggestionDetails(

decodedControllerData = Some(controllerData)

))

)

val result: Option[Set[TweetResultSource]] = new SearchInfoUtils(item).getTweetResultSources

result shouldEqual Some(tweetResultSources)

}

}

test("Decoding User Result Sources bitmap") {

new Fixture {

UserResultSource.list

.foreach { userResultSource =>

val bitmap = (1 << userResultSource.getValue()).toLong

val controllerData = mkUserTypeControllerData(bitmap)

val item = Item(

suggestionDetails = Some(

SuggestionDetails(

decodedControllerData = Some(controllerData)

))

)

val result = new SearchInfoUtils(item).getUserResultSources

result shouldEqual Some(Set(userResultSource))

}

}

}

test("Decoding multiple User Result Sources") {

new Fixture {

val userResultSources: Set[UserResultSource] =

Set(UserResultSource.QueryInteractionGraph, UserResultSource.ExpertSearch)

val bitmap: Long = userResultSources.foldLeft(0L) {

case (acc, source) => acc + (1 << source.getValue())

}

val controllerData: ControllerData.V2 = mkUserTypeControllerData(bitmap)

val item: Item = Item(

suggestionDetails = Some(

SuggestionDetails(

decodedControllerData = Some(controllerData)

))

)

val result: Option[Set[UserResultSource]] = new SearchInfoUtils(item).getUserResultSources

result shouldEqual Some(userResultSources)

}

}

test("getQueryFilterTabType should return correct query filter type") {

new Fixture {

val infoUtils = new SearchInfoUtils(Item())

val eventsToBeChecked: TableFor2[Option[EventNamespace], Option[SearchQueryFilterType]] =

Table(

("eventNamespace", "queryFilterType"),

(

Some(EventNamespace(client = Some("m5"), element = Some("search\_filter\_top"))),

Some(Top)),

(

Some(EventNamespace(client = Some("m5"), element = Some("search\_filter\_live"))),

Some(Latest)),

(

Some(EventNamespace(client = Some("m5"), element = Some("search\_filter\_user"))),

Some(People)),

(

Some(EventNamespace(client = Some("m5"), element = Some("search\_filter\_image"))),

Some(Photos)),

(

Some(EventNamespace(client = Some("m5"), element = Some("search\_filter\_video"))),

Some(Videos)),

(

Some(EventNamespace(client = Some("m5"), section = Some("search\_filter\_top"))),

None

), // if client is web, element determines the query filter hence None if element is None

(

Some(EventNamespace(client = Some("android"), element = Some("search\_filter\_top"))),

Some(Top)),

(

Some(EventNamespace(client = Some("android"), element = Some("search\_filter\_tweets"))),

Some(Latest)),

(

Some(EventNamespace(client = Some("android"), element = Some("search\_filter\_user"))),

Some(People)),

(

Some(EventNamespace(client = Some("android"), element = Some("search\_filter\_image"))),

Some(Photos)),

(

Some(EventNamespace(client = Some("android"), element = Some("search\_filter\_video"))),

Some(Videos)),

(

Some(EventNamespace(client = Some("m5"), section = Some("search\_filter\_top"))),

None

), // if client is android, element determines the query filter hence None if element is None

(

Some(EventNamespace(client = Some("iphone"), section = Some("search\_filter\_top"))),

Some(Top)),

(

Some(EventNamespace(client = Some("iphone"), section = Some("search\_filter\_live"))),

Some(Latest)),

(

Some(EventNamespace(client = Some("iphone"), section = Some("search\_filter\_user"))),

Some(People)),

(

Some(EventNamespace(client = Some("iphone"), section = Some("search\_filter\_image"))),

Some(Photos)),

(

Some(EventNamespace(client = Some("iphone"), section = Some("search\_filter\_video"))),

Some(Videos)),

(

Some(EventNamespace(client = Some("iphone"), element = Some("search\_filter\_top"))),

None

), // if client is iphone, section determines the query filter hence None if section is None

(

Some(EventNamespace(client = None, section = Some("search\_filter\_top"))),

Some(Top)

), // if client is missing, use section by default

(

Some(EventNamespace(client = None, element = Some("search\_filter\_top"))),

None

), // if client is missing, section is used by default hence None since section is missing

(

Some(EventNamespace(client = Some("iphone"))),

None

), // if both element and section missing, expect None

(None, None), // if namespace is missing from LogEvent, expect None

)

forEvery(eventsToBeChecked) {

(

eventNamespace: Option[EventNamespace],

searchQueryFilterType: Option[SearchQueryFilterType]

) =>

infoUtils.getQueryFilterType(

LogEvent(

eventName = "srp\_event",

eventNamespace = eventNamespace)) shouldEqual searchQueryFilterType

}

}

}

}