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

import com.twitter.clientapp.thriftscala.EventNamespace

import com.twitter.clientapp.thriftscala.{Item => LogEventItem}

import com.twitter.clientapp.thriftscala.ItemType

import com.twitter.clientapp.thriftscala.LogEvent

import com.twitter.clientapp.thriftscala.NotificationTabDetails

import com.twitter.clientapp.thriftscala.ReportDetails

import com.twitter.clientapp.thriftscala.SearchDetails

import com.twitter.clientapp.thriftscala.SuggestionDetails

import com.twitter.inject.Test

import com.twitter.logbase.thriftscala.ClientEventReceiver

import com.twitter.reportflow.thriftscala.ReportType

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

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

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.TableFor2

import scala.language.implicitConversions

class ClientEventAdapterSpec extends Test with TableDrivenPropertyChecks {

// Tests for invalid client-events

test("should ignore events") {

new TestFixtures.ClientEventFixture {

val eventsToBeIgnored: TableFor2[String, LogEvent] = Table(

("namespace", "event"),

("ddg", ddgEvent),

("qig\_ranker", qigRankerEvent),

("timelnemixer", timelineMixerEvent),

("timelineservice", timelineServiceEvent),

("tweetconvosvc", tweetConcServiceEvent),

("item-type is non-tweet", renderNonTweetItemTypeEvent)

)

forEvery(eventsToBeIgnored) { (\_: String, event: LogEvent) =>

val actual = ClientEventAdapter.adaptEvent(event)

assert(actual.isEmpty)

}

}

}

test("Tests for ItemType filter") {

/// Tweet events

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val events = Table(

("itemType", "expectedUUA"),

(Some(ItemType.Tweet), Seq(expectedTweetRenderDefaultTweetUUA)),

(Some(ItemType.QuotedTweet), Seq(expectedTweetRenderDefaultTweetUUA)),

(Some(ItemType.Topic), Nil),

(None, Nil)

)

forEvery(events) { (itemTypeOpt: Option[ItemType], expected: Seq[UnifiedUserAction]) =>

val actual = ClientEventAdapter.adaptEvent(

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceRenderEventNamespace),

itemTypeOpt = itemTypeOpt

))

assert(expected === actual)

}

}

}

/// Topic events

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val expected: UnifiedUserAction = mkExpectedUUAForActionTowardTopicEvent(

topicId = topicId,

clientEventNamespace = Some(uuaTopicFollowClientEventNamespace1),

actionType = ActionType.ClientTopicFollow

)

val events = Table(

("itemType", "expectedUUA"),

(Some(ItemType.Tweet), Seq(expected)),

(Some(ItemType.QuotedTweet), Seq(expected)),

(Some(ItemType.Topic), Seq(expected)),

(None, Nil)

)

forEvery(events) { (itemTypeOpt: Option[ItemType], expected: Seq[UnifiedUserAction]) =>

val actual = ClientEventAdapter.adaptEvent(

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceTopicFollow1),

itemId = None,

suggestionDetails =

Some(SuggestionDetails(decodedControllerData = Some(homeTweetControllerData()))),

itemTypeOpt = itemTypeOpt

))

assert(expected === actual)

}

}

}

}

// Tests for ClientTweetRenderImpression

test("ClientTweetRenderImpression") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvents = Table(

("actionTweetType", "clientEvent", "expectedUUAEvent"),

(

"Default",

actionTowardDefaultTweetEvent(eventNamespace = Some(ceRenderEventNamespace)),

Seq(expectedTweetRenderDefaultTweetUUA)),

(

"Reply",

actionTowardReplyEvent(eventNamespace = Some(ceRenderEventNamespace)),

Seq(expectedTweetRenderReplyUUA)),

(

"Retweet",

actionTowardRetweetEvent(eventNamespace = Some(ceRenderEventNamespace)),

Seq(expectedTweetRenderRetweetUUA)),

(

"Quote",

actionTowardQuoteEvent(

eventNamespace = Some(ceRenderEventNamespace),

quotedAuthorId = Some(456L)),

Seq(expectedTweetRenderQuoteUUA1, expectedTweetRenderQuoteUUA2)),

(

"Retweet of a reply that quoted another Tweet",

actionTowardRetweetEventWithReplyAndQuote(eventNamespace =

Some(ceRenderEventNamespace)),

Seq(

expectedTweetRenderRetweetWithReplyAndQuoteUUA1,

expectedTweetRenderRetweetWithReplyAndQuoteUUA2))

)

forEvery(clientEvents) {

(\_: String, event: LogEvent, expectedUUA: Seq[UnifiedUserAction]) =>

val actual = ClientEventAdapter.adaptEvent(event)

actual should contain theSameElementsAs expectedUUA

}

}

}

}

test("ClientTweetGallery/DetailImpression") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvents = Table(

("actionTweetType", "clientEvent", "expectedUUAEvent"),

(

"DetailImpression: tweet::tweet::impression",

actionTowardDefaultTweetEvent(eventNamespace = Some(ceTweetDetailsEventNamespace1)),

expectedTweetDetailImpressionUUA1),

(

"GalleryImpression: gallery:photo:impression",

actionTowardDefaultTweetEvent(eventNamespace = Some(ceGalleryEventNamespace)),

expectedTweetGalleryImpressionUUA),

)

forEvery(clientEvents) { (\_: String, event: LogEvent, expectedUUA: UnifiedUserAction) =>

val actual = ClientEventAdapter.adaptEvent(event)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientTweetLingerImpression

test("ClientTweetLingerImpression") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvents = Table(

("actionTweetType", "clientEvent", "expectedUUAEvent"),

("Default", lingerDefaultTweetEvent, expectedTweetLingerDefaultTweetUUA),

("Reply", lingerReplyEvent, expectedTweetLingerReplyUUA),

("Retweet", lingerRetweetEvent, expectedTweetLingerRetweetUUA),

("Quote", lingerQuoteEvent, expectedTweetLingerQuoteUUA),

(

"Retweet of a reply that quoted another Tweet",

lingerRetweetWithReplyAndQuoteEvent,

expectedTweetLingerRetweetWithReplyAndQuoteUUA),

)

forEvery(clientEvents) { (\_: String, event: LogEvent, expectedUUA: UnifiedUserAction) =>

val actual = ClientEventAdapter.adaptEvent(event)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientTweetClickQuote

test(

"ClickQuote, which is the click on the quote button, results in setting retweeting, inReplyTo, quoted tweet ids") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val actual = ClientEventAdapter.adaptEvent(

// there shouldn't be any quotingTweetId in CE when it is "quote"

actionTowardRetweetEventWithReplyAndQuote(eventNamespace = Some(

EventNamespace(

action = Some("quote")

))))

assert(Seq(expectedTweetClickQuoteUUA) === actual)

}

}

}

// Tests for ClientTweetQuote

test(

"Quote, which is sending the quote, results in setting retweeting, inReplyTo, quoted tweet ids") {

new TestFixtures.ClientEventFixture {

val actions: TableFor1[String] = Table(

"action",

"send\_quote\_tweet",

"retweet\_with\_comment"

)

Time.withTimeAt(frozenTime) { \_ =>

forEvery(actions) { action =>

val actual = ClientEventAdapter.adaptEvent(

// there shouldn't be any quotingTweetId in CE when it is "quote"

actionTowardRetweetEventWithReplyAndQuote(eventNamespace = Some(

EventNamespace(

action = Some(action)

))))

assert(Seq(expectedTweetQuoteUUA(action)) === actual)

}

}

}

}

// Tests for ClientTweetFav and ClientTweetUnfav

test("ClientTweetFav and ClientTweetUnfav") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvents = Table(

("actionTweetType", "clientEvent", "expectedUUAEvent"),

(

"Default Tweet favorite",

actionTowardDefaultTweetEvent(eventNamespace = Some(ceFavoriteEventNamespace)),

expectedTweetFavoriteDefaultTweetUUA),

(

"Reply Tweet favorite",

actionTowardReplyEvent(eventNamespace = Some(ceFavoriteEventNamespace)),

expectedTweetFavoriteReplyUUA),

(

"Retweet Tweet favorite",

actionTowardRetweetEvent(eventNamespace = Some(ceFavoriteEventNamespace)),

expectedTweetFavoriteRetweetUUA),

(

"Quote Tweet favorite",

actionTowardQuoteEvent(eventNamespace = Some(ceFavoriteEventNamespace)),

expectedTweetFavoriteQuoteUUA),

(

"Retweet of a reply that quoted another Tweet favorite",

actionTowardRetweetEventWithReplyAndQuote(eventNamespace =

Some(ceFavoriteEventNamespace)),

expectedTweetFavoriteRetweetWithReplyAndQuoteUUA),

(

"Default Tweet unfavorite",

actionTowardDefaultTweetEvent(

eventNamespace = Some(EventNamespace(action = Some("unfavorite"))),

),

mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(ClientEventNamespace(action = Some("unfavorite"))),

actionType = ActionType.ClientTweetUnfav

))

)

forEvery(clientEvents) { (\_: String, event: LogEvent, expectedUUA: UnifiedUserAction) =>

val actual = ClientEventAdapter.adaptEvent(event)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientTweetClickReply

test("ClientTweetClickReply") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvents = Table(

("actionTweetType", "clientEvent", "expectedUUAEvent"),

(

"Default",

actionTowardDefaultTweetEvent(eventNamespace = Some(ceClickReplyEventNamespace)),

expectedTweetClickReplyDefaultTweetUUA),

(

"Reply",

actionTowardReplyEvent(eventNamespace = Some(ceClickReplyEventNamespace)),

expectedTweetClickReplyReplyUUA),

(

"Retweet",

actionTowardRetweetEvent(eventNamespace = Some(ceClickReplyEventNamespace)),

expectedTweetClickReplyRetweetUUA),

(

"Quote",

actionTowardQuoteEvent(eventNamespace = Some(ceClickReplyEventNamespace)),

expectedTweetClickReplyQuoteUUA),

(

"Retweet of a reply that quoted another Tweet",

actionTowardRetweetEventWithReplyAndQuote(eventNamespace =

Some(ceClickReplyEventNamespace)),

expectedTweetClickReplyRetweetWithReplyAndQuoteUUA)

)

forEvery(clientEvents) { (\_: String, event: LogEvent, expectedUUA: UnifiedUserAction) =>

val actual = ClientEventAdapter.adaptEvent(event)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientTweetReply

test("ClientTweetReply") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvents = Table(

("actionTweetType", "clientEvent", "expectedUUAEvent"),

("DefaultOrReply", replyToDefaultTweetOrReplyEvent, expectedTweetReplyDefaultTweetUUA),

("Retweet", replyToRetweetEvent, expectedTweetReplyRetweetUUA),

("Quote", replyToQuoteEvent, expectedTweetReplyQuoteUUA),

(

"Retweet of a reply that quoted another Tweet",

replyToRetweetWithReplyAndQuoteEvent,

expectedTweetReplyRetweetWithReplyAndQuoteUUA)

)

forEvery(clientEvents) { (\_: String, event: LogEvent, expectedUUA: UnifiedUserAction) =>

val actual = ClientEventAdapter.adaptEvent(event)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientTweetRetweet and ClientTweetUnretweet

test("ClientTweetRetweet and ClientTweetUnretweet") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvents = Table(

("actionTweetType", "clientEvent", "expectedUUAEvent"),

(

"Default Tweet retweet",

actionTowardDefaultTweetEvent(eventNamespace = Some(ceRetweetEventNamespace)),

expectedTweetRetweetDefaultTweetUUA),

(

"Reply Tweet retweet",

actionTowardReplyEvent(eventNamespace = Some(ceRetweetEventNamespace)),

expectedTweetRetweetReplyUUA),

(

"Retweet Tweet retweet",

actionTowardRetweetEvent(eventNamespace = Some(ceRetweetEventNamespace)),

expectedTweetRetweetRetweetUUA),

(

"Quote Tweet retweet",

actionTowardQuoteEvent(eventNamespace = Some(ceRetweetEventNamespace)),

expectedTweetRetweetQuoteUUA),

(

"Retweet of a reply that quoted another Tweet retweet",

actionTowardRetweetEventWithReplyAndQuote(eventNamespace =

Some(ceRetweetEventNamespace)),

expectedTweetRetweetRetweetWithReplyAndQuoteUUA),

(

"Default Tweet unretweet",

actionTowardDefaultTweetEvent(

eventNamespace = Some(EventNamespace(action = Some("unretweet"))),

),

mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(ClientEventNamespace(action = Some("unretweet"))),

actionType = ActionType.ClientTweetUnretweet

))

)

forEvery(clientEvents) { (\_: String, event: LogEvent, expectedUUA: UnifiedUserAction) =>

val actual = ClientEventAdapter.adaptEvent(event)

assert(Seq(expectedUUA) === actual)

}

}

}

}

test("include Topic Id") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val actual = ClientEventAdapter.adaptEvent(renderDefaultTweetWithTopicIdEvent)

assert(Seq(expectedTweetRenderDefaultTweetWithTopicIdUUA) === actual)

}

}

}

// Tests for ClientTweetVideoPlayback0, 25, 50, 75, 95, 100 PlayFromTap, QualityView,

// VideoView, MrcView, ViewThreshold

test("ClientTweetVideoPlayback\*") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvents = Table(

("ceNamespace", "uuaNamespace", "uuaActionType"),

(

ceVideoPlayback25,

uuaVideoPlayback25ClientEventNamespace,

ActionType.ClientTweetVideoPlayback25),

(

ceVideoPlayback50,

uuaVideoPlayback50ClientEventNamespace,

ActionType.ClientTweetVideoPlayback50),

(

ceVideoPlayback75,

uuaVideoPlayback75ClientEventNamespace,

ActionType.ClientTweetVideoPlayback75),

(

ceVideoPlayback95,

uuaVideoPlayback95ClientEventNamespace,

ActionType.ClientTweetVideoPlayback95),

(

ceVideoPlayFromTap,

uuaVideoPlayFromTapClientEventNamespace,

ActionType.ClientTweetVideoPlayFromTap),

(

ceVideoQualityView,

uuaVideoQualityViewClientEventNamespace,

ActionType.ClientTweetVideoQualityView),

(ceVideoView, uuaVideoViewClientEventNamespace, ActionType.ClientTweetVideoView),

(ceVideoMrcView, uuaVideoMrcViewClientEventNamespace, ActionType.ClientTweetVideoMrcView),

(

ceVideoViewThreshold,

uuaVideoViewThresholdClientEventNamespace,

ActionType.ClientTweetVideoViewThreshold),

(

ceVideoCtaUrlClick,

uuaVideoCtaUrlClickClientEventNamespace,

ActionType.ClientTweetVideoCtaUrlClick),

(

ceVideoCtaWatchClick,

uuaVideoCtaWatchClickClientEventNamespace,

ActionType.ClientTweetVideoCtaWatchClick),

)

for (element <- videoEventElementValues) {

forEvery(clientEvents) {

(

ceNamespace: EventNamespace,

uuaNamespace: ClientEventNamespace,

uuaActionType: ActionType

) =>

val event = actionTowardDefaultTweetEvent(

eventNamespace = Some(ceNamespace.copy(element = Some(element))),

mediaDetailsV2 = Some(mediaDetailsV2),

clientMediaEvent = Some(clientMediaEvent),

cardDetails = Some(cardDetails)

)

val expectedUUA = mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(uuaNamespace.copy(element = Some(element))),

actionType = uuaActionType,

tweetActionInfo = Some(videoMetadata)

)

val actual = ClientEventAdapter.adaptEvent(event)

assert(Seq(expectedUUA) === actual)

}

}

}

}

}

// Tests for ClientTweetPhotoExpand

test("Client Tweet Photo Expand") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvent = actionTowardDefaultTweetEvent(eventNamespace = Some(cePhotoExpand))

val expectedUUA = mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(uuaPhotoExpandClientEventNamespace),

actionType = ActionType.ClientTweetPhotoExpand

)

assert(Seq(expectedUUA) === ClientEventAdapter.adaptEvent(clientEvent))

}

}

}

// Tests for ClientCardClick

test("Client Card Related") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvents = Table(

("ceNamespace", "ceItemType", "uuaNamespace", "uuaActionType"),

(

ceCardClick,

ItemType.Tweet,

uuaCardClickClientEventNamespace,

ActionType.ClientCardClick),

(

ceCardClick,

ItemType.User,

uuaCardClickClientEventNamespace,

ActionType.ClientCardClick),

(

ceCardOpenApp,

ItemType.Tweet,

uuaCardOpenAppClientEventNamespace,

ActionType.ClientCardOpenApp),

(

ceCardAppInstallAttempt,

ItemType.Tweet,

uuaCardAppInstallAttemptClientEventNamespace,

ActionType.ClientCardAppInstallAttempt),

(

cePollCardVote1,

ItemType.Tweet,

uuaPollCardVote1ClientEventNamespace,

ActionType.ClientPollCardVote),

(

cePollCardVote2,

ItemType.Tweet,

uuaPollCardVote2ClientEventNamespace,

ActionType.ClientPollCardVote),

)

forEvery(clientEvents) {

(

ceNamespace: EventNamespace,

ceItemType: ItemType,

uuaNamespace: ClientEventNamespace,

uuaActionType: ActionType

) =>

val event = actionTowardDefaultTweetEvent(

eventNamespace = Some(ceNamespace),

itemTypeOpt = Some(ceItemType),

authorId = Some(authorId)

)

val expectedUUA = mkExpectedUUAForCardEvent(

id = Some(itemTweetId),

clientEventNamespace = Some(uuaNamespace),

actionType = uuaActionType,

itemType = Some(ceItemType),

authorId = Some(authorId)

)

val actual = ClientEventAdapter.adaptEvent(event)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientTweetClickMentionScreenName

test("ClientTweetClickMentionScreenName") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val userHandle = "someHandle"

val clientEvent = actionTowardDefaultTweetEvent(

eventNamespace = Some(ceMentionClick),

targets = Some(

Seq(

LogEventItem(

itemType = Some(ItemType.User),

id = Some(userId),

name = Some(userHandle)))))

val expectedUUA = mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(uuaMentionClickClientEventNamespace),

actionType = ActionType.ClientTweetClickMentionScreenName,

tweetActionInfo = Some(

TweetActionInfo.ClientTweetClickMentionScreenName(

ClientTweetClickMentionScreenName(actionProfileId = userId, handle = userHandle)))

)

assert(Seq(expectedUUA) === ClientEventAdapter.adaptEvent(clientEvent))

}

}

}

// Tests for Topic Follow/Unfollow actions

test("Topic Follow/Unfollow Actions") {

// The Topic Id is mostly from TimelineTopic controller data or HomeTweets controller data!

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvents = Table(

("clientEventNamesapce", "expectedUUANamespace", "controllerData", "actionType"),

(

ceTopicFollow1,

uuaTopicFollowClientEventNamespace1,

timelineTopicControllerData(),

ActionType.ClientTopicFollow

),

(

ceTopicFollow1,

uuaTopicFollowClientEventNamespace1,

homeTweetControllerData(),

ActionType.ClientTopicFollow),

(

ceTopicFollow2,

uuaTopicFollowClientEventNamespace2,

timelineTopicControllerData(),

ActionType.ClientTopicFollow

),

(

ceTopicFollow2,

uuaTopicFollowClientEventNamespace2,

homeTweetControllerData(),

ActionType.ClientTopicFollow),

(

ceTopicFollow3,

uuaTopicFollowClientEventNamespace3,

timelineTopicControllerData(),

ActionType.ClientTopicFollow

),

(

ceTopicFollow3,

uuaTopicFollowClientEventNamespace3,

homeTweetControllerData(),

ActionType.ClientTopicFollow),

(

ceTopicUnfollow1,

uuaTopicUnfollowClientEventNamespace1,

timelineTopicControllerData(),

ActionType.ClientTopicUnfollow

),

(

ceTopicUnfollow1,

uuaTopicUnfollowClientEventNamespace1,

homeTweetControllerData(),

ActionType.ClientTopicUnfollow),

(

ceTopicUnfollow2,

uuaTopicUnfollowClientEventNamespace2,

timelineTopicControllerData(),

ActionType.ClientTopicUnfollow

),

(

ceTopicFollow2,

uuaTopicFollowClientEventNamespace2,

homeTweetControllerData(),

ActionType.ClientTopicFollow),

(

ceTopicUnfollow3,

uuaTopicUnfollowClientEventNamespace3,

timelineTopicControllerData(),

ActionType.ClientTopicUnfollow

),

(

ceTopicUnfollow3,

uuaTopicUnfollowClientEventNamespace3,

homeTweetControllerData(),

ActionType.ClientTopicUnfollow),

)

forEvery(clientEvents) {

(

eventNamespace: EventNamespace,

uuaNs: ClientEventNamespace,

controllerData: ControllerData,

actionType: ActionType

) =>

val event = actionTowardDefaultTweetEvent(

eventNamespace = Some(eventNamespace),

itemId = None,

suggestionDetails =

Some(SuggestionDetails(decodedControllerData = Some(controllerData)))

)

val expectedUUA = mkExpectedUUAForActionTowardTopicEvent(

topicId = topicId,

traceId = None,

clientEventNamespace = Some(uuaNs),

actionType = actionType

)

val actual = ClientEventAdapter.adaptEvent(event)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for Topic NotInterestedIn & its Undo actions

test("Topic NotInterestedIn & its Undo actions") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvents = Table(

("clientEventNamesapce", "expectedUUANamespace", "controllerData", "actionType"),

(

ceTopicNotInterestedIn1,

uuaTopicNotInterestedInClientEventNamespace1,

timelineTopicControllerData(),

ActionType.ClientTopicNotInterestedIn

),

(

ceTopicNotInterestedIn1,

uuaTopicNotInterestedInClientEventNamespace1,

homeTweetControllerData(),

ActionType.ClientTopicNotInterestedIn),

(

ceTopicNotInterestedIn2,

uuaTopicNotInterestedInClientEventNamespace2,

timelineTopicControllerData(),

ActionType.ClientTopicNotInterestedIn

),

(

ceTopicNotInterestedIn2,

uuaTopicNotInterestedInClientEventNamespace2,

homeTweetControllerData(),

ActionType.ClientTopicNotInterestedIn),

(

ceTopicUndoNotInterestedIn1,

uuaTopicUndoNotInterestedInClientEventNamespace1,

timelineTopicControllerData(),

ActionType.ClientTopicUndoNotInterestedIn

),

(

ceTopicUndoNotInterestedIn1,

uuaTopicUndoNotInterestedInClientEventNamespace1,

homeTweetControllerData(),

ActionType.ClientTopicUndoNotInterestedIn),

(

ceTopicUndoNotInterestedIn2,

uuaTopicUndoNotInterestedInClientEventNamespace2,

timelineTopicControllerData(),

ActionType.ClientTopicUndoNotInterestedIn

),

(

ceTopicUndoNotInterestedIn2,

uuaTopicUndoNotInterestedInClientEventNamespace2,

homeTweetControllerData(),

ActionType.ClientTopicUndoNotInterestedIn),

)

forEvery(clientEvents) {

(

eventNamespace: EventNamespace,

uuaNs: ClientEventNamespace,

controllerData: ControllerData,

actionType: ActionType

) =>

val event = actionTowardDefaultTweetEvent(

eventNamespace = Some(eventNamespace),

itemId = None,

suggestionDetails =

Some(SuggestionDetails(decodedControllerData = Some(controllerData)))

)

val expectedUUA = mkExpectedUUAForActionTowardTopicEvent(

topicId = topicId,

traceId = None,

clientEventNamespace = Some(uuaNs),

actionType = actionType

)

val actual = ClientEventAdapter.adaptEvent(event)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for authorInfo

test("authorInfo") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvents = Table(

("authorIdOpt", "isFollowedByActingUser", "isFollowingActingUser"),

(Some(authorId), true, false),

(Some(authorId), true, true),

(Some(authorId), false, true),

(Some(authorId), false, false),

(None, true, true),

)

forEvery(clientEvents) {

(

authorIdOpt: Option[Long],

isFollowedByActingUser: Boolean,

isFollowingActingUser: Boolean

) =>

val actual = ClientEventAdapter.adaptEvent(

renderDefaultTweetUserFollowStatusEvent(

authorId = authorIdOpt,

isFollowedByActingUser = isFollowedByActingUser,

isFollowingActingUser = isFollowingActingUser

))

val expected =

expectedTweetRenderDefaultTweetWithAuthorInfoUUA(authorInfo = authorIdOpt.map { id =>

AuthorInfo(

authorId = Some(id),

isFollowedByActingUser = Some(isFollowedByActingUser),

isFollowingActingUser = Some(isFollowingActingUser)

)

})

assert(Seq(expected) === actual)

}

}

}

}

// Tests for ClientTweetReport

test("ClientTweetReport") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val ceNTabTweetReport: EventNamespace =

ceTweetReport.copy(page = Some("ntab"), section = Some("all"), component = Some("urt"))

val uuaNTabTweetReport: ClientEventNamespace =

uuaTweetReport.copy(page = Some("ntab"), section = Some("all"), component = Some("urt"))

val params = Table(

(

"eventType",

"ceNamespace",

"ceNotificationTabDetails",

"ceReportDetails",

"uuaNamespace",

"uuaTweetActionInfo",

"uuaProductSurface",

"uuaProductSurfaceInfo"),

(

"ntabReportTweetClick",

ceNTabTweetReport.copy(action = Some("click")),

Some(notificationTabTweetEventDetails),

None,

uuaNTabTweetReport.copy(action = Some("click")),

reportTweetClick,

Some(ProductSurface.NotificationTab),

Some(notificationTabProductSurfaceInfo)

),

(

"ntabReportTweetDone",

ceNTabTweetReport.copy(action = Some("done")),

Some(notificationTabTweetEventDetails),

None,

uuaNTabTweetReport.copy(action = Some("done")),

reportTweetDone,

Some(ProductSurface.NotificationTab),

Some(notificationTabProductSurfaceInfo)

),

(

"defaultReportTweetDone",

ceTweetReport.copy(page = Some("tweet"), action = Some("done")),

None,

None,

uuaTweetReport.copy(page = Some("tweet"), action = Some("done")),

reportTweetDone,

None,

None

),

(

"defaultReportTweetWithReportFlowId",

ceTweetReport.copy(page = Some("tweet"), action = Some("done")),

None,

Some(ReportDetails(reportFlowId = Some(reportFlowId))),

uuaTweetReport.copy(page = Some("tweet"), action = Some("done")),

reportTweetWithReportFlowId,

None,

None

),

(

"defaultReportTweetWithoutReportFlowId",

ceTweetReport.copy(page = Some("tweet"), action = Some("done")),

None,

None,

uuaTweetReport.copy(page = Some("tweet"), action = Some("done")),

reportTweetWithoutReportFlowId,

None,

None

),

)

forEvery(params) {

(

\_: String,

ceNamespace: EventNamespace,

ceNotificationTabDetails: Option[NotificationTabDetails],

ceReportDetails: Option[ReportDetails],

uuaNamespace: ClientEventNamespace,

uuaTweetActionInfo: TweetActionInfo,

productSurface: Option[ProductSurface],

productSurfaceInfo: Option[ProductSurfaceInfo]

) =>

val actual = ClientEventAdapter.adaptEvent(

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceNamespace),

notificationTabDetails = ceNotificationTabDetails,

reportDetails = ceReportDetails))

val expectedUUA = mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(uuaNamespace),

actionType = ActionType.ClientTweetReport,

tweetActionInfo = Some(uuaTweetActionInfo),

productSurface = productSurface,

productSurfaceInfo = productSurfaceInfo

)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientTweetNotHelpful and ClientTweetUndoNotHelpful

test("ClientTweetNotHelpful & UndoNotHelpful") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val actions = Table(("action"), "click", "undo")

val element = "feedback\_givefeedback"

forEvery(actions) { action =>

val clientEvent =

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceEventNamespace(element, action)),

)

val expectedUUA = mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(uuaClientEventNamespace(element, action)),

actionType = action match {

case "click" => ActionType.ClientTweetNotHelpful

case "undo" => ActionType.ClientTweetUndoNotHelpful

}

)

val actual = ClientEventAdapter.adaptEvent(clientEvent)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientTweetNotInterestedIn and ClientTweetUndoNotInterestedIn

test("ClientTweetNotInterestedIn & UndoNotInterestedIn") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val actions = Table(("action"), "click", "undo")

val element = "feedback\_dontlike"

forEvery(actions) { action =>

val clientEvent =

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceEventNamespace(element, action)),

)

val expectedUUA = mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(uuaClientEventNamespace(element, action)),

actionType = action match {

case "click" => ActionType.ClientTweetNotInterestedIn

case "undo" => ActionType.ClientTweetUndoNotInterestedIn

}

)

val actual = ClientEventAdapter.adaptEvent(clientEvent)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientTweetNotAboutTopic & ClientTweetUndoNotAboutTopic

test("ClientTweetNotAboutTopic & ClientTweetUndoNotAboutTopic") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val actions = Table(("action"), "click", "undo")

val element = "feedback\_notabouttopic"

forEvery(actions) { action =>

val clientEvent =

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceEventNamespace(element, action)),

)

val expectedUUA = mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(uuaClientEventNamespace(element, action)),

actionType = action match {

case "click" => ActionType.ClientTweetNotAboutTopic

case "undo" => ActionType.ClientTweetUndoNotAboutTopic

}

)

val actual = ClientEventAdapter.adaptEvent(clientEvent)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientTweetNotRecent and ClientTweetUndoNotRecent

test("ClientTweetNotRecent & UndoNotRecent") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val actions = Table(("action"), "click", "undo")

val element = "feedback\_notrecent"

forEvery(actions) { action =>

val clientEvent =

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceEventNamespace(element, action)),

)

val expectedUUA = mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(uuaClientEventNamespace(element, action)),

actionType = action match {

case "click" => ActionType.ClientTweetNotRecent

case "undo" => ActionType.ClientTweetUndoNotRecent

}

)

val actual = ClientEventAdapter.adaptEvent(clientEvent)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientTweetSeeFewer and ClientTweetUndoSeeFewer

test("ClientTweetSeeFewer & ClientTweetUndoSeeFewer") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val actions = Table(("action"), "click", "undo")

val element = "feedback\_seefewer"

forEvery(actions) { action =>

val clientEvent =

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceEventNamespace(element, action)),

)

val expectedUUA = mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(uuaClientEventNamespace(element, action)),

actionType = action match {

case "click" => ActionType.ClientTweetSeeFewer

case "undo" => ActionType.ClientTweetUndoSeeFewer

}

)

val actual = ClientEventAdapter.adaptEvent(clientEvent)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for getEventMetadata

test("getEventMetadata") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvents = Table(

("clientEventNamesapce", "expectedUUANamespace", "controllerData"),

(

ceRenderEventNamespace,

uuaRenderClientEventNamespace,

homeTweetControllerData()

),

)

forEvery(clientEvents) {

(

eventNamespace: EventNamespace,

uuaNs: ClientEventNamespace,

controllerData: ControllerData

) =>

val event = actionTowardDefaultTweetEvent(

eventNamespace = Some(eventNamespace),

suggestionDetails =

Some(SuggestionDetails(decodedControllerData = Some(controllerData)))

)

val expectedEventMetaData = mkUUAEventMetadata(

clientEventNamespace = Some(uuaNs)

)

val actual = ClientEventAdapter.adaptEvent(event).head.eventMetadata

assert(expectedEventMetaData === actual)

}

}

}

}

// Tests for getSourceTimestamp

test("getSourceTimestamp") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val params = Table(

("testCase", "clientEvent", "expectedUUAEventTimestamp"),

(

"CES event with DriftAdjustedEventCreatedAtMs",

actionTowardDefaultTweetEvent(eventNamespace = Some(ceRenderEventNamespace)),

logBase.driftAdjustedEventCreatedAtMs),

(

"CES event without DriftAdjustedEventCreatedAtMs: ignore",

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceRenderEventNamespace),

logBase = logBase.unsetDriftAdjustedEventCreatedAtMs),

None),

(

"Non-CES event without DriftAdjustedEventCreatedAtMs: use logBase.timestamp",

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceRenderEventNamespace),

logBase = logBase

.copy(

clientEventReceiver =

Some(ClientEventReceiver.Unknown)).unsetDriftAdjustedEventCreatedAtMs

),

Some(logBase.timestamp))

)

forEvery(params) { (\_: String, event: LogEvent, expectedUUAEventTimestamp: Option[Long]) =>

val actual =

ClientEventAdapter.adaptEvent(event).map(\_.eventMetadata.sourceTimestampMs).headOption

assert(expectedUUAEventTimestamp === actual)

}

}

}

}

// Tests for ServerTweetReport

test("ServerTweetReport") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val params = Table(

("eventType", "ceNamespace", "ceReportDetails", "uuaNamespace", "uuaTweetActionInfo"),

(

"ReportImpressionIsNotAdapted",

ceTweetReportFlow(page = "report\_abuse", action = "impression"),

Some(ReportDetails(reportFlowId = Some(reportFlowId))),

None,

None

),

(

"ReportSubmitIsAdapted",

ceTweetReportFlow(page = "report\_abuse", action = "submit"),

Some(

ReportDetails(

reportFlowId = Some(reportFlowId),

reportType = Some(ReportType.Abuse))),

Some(uuaTweetReportFlow(page = "report\_abuse", action = "submit")),

Some(reportTweetSubmit)

),

)

forEvery(params) {

(

\_: String,

ceNamespace: EventNamespace,

ceReportDetails: Option[ReportDetails],

uuaNamespace: Option[ClientEventNamespace],

uuaTweetActionInfo: Option[TweetActionInfo]

) =>

val actual = ClientEventAdapter.adaptEvent(

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceNamespace),

reportDetails = ceReportDetails))

val expectedUUA =

if (ceNamespace.action.contains("submit"))

Seq(

mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = uuaNamespace,

actionType = ActionType.ServerTweetReport,

tweetActionInfo = uuaTweetActionInfo

))

else Nil

assert(expectedUUA === actual)

}

}

}

}

// Tests for ClientNotificationOpen

test("ClientNotificationOpen") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvent =

pushNotificationEvent(

eventNamespace = Some(ceNotificationOpen),

notificationDetails = Some(notificationDetails))

val expectedUUA = mkExpectedUUAForNotificationEvent(

clientEventNamespace = Some(uuaNotificationOpen),

actionType = ActionType.ClientNotificationOpen,

notificationContent = tweetNotificationContent,

productSurface = Some(ProductSurface.PushNotification),

productSurfaceInfo = Some(

ProductSurfaceInfo.PushNotificationInfo(

PushNotificationInfo(notificationId = notificationId)))

)

val actual = ClientEventAdapter.adaptEvent(clientEvent)

assert(Seq(expectedUUA) === actual)

}

}

}

// Tests for ClientNotificationClick

test("ClientNotificationClick") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val params = Table(

("notificationType", "ceNotificationTabDetails", "uuaNotificationContent"),

("tweetNotification", notificationTabTweetEventDetails, tweetNotificationContent),

(

"multiTweetNotification",

notificationTabMultiTweetEventDetails,

multiTweetNotificationContent),

(

"unknownNotification",

notificationTabUnknownEventDetails,

unknownNotificationContent

),

)

forEvery(params) {

(

\_: String,

ceNotificationTabDetails: NotificationTabDetails,

uuaNotificationContent: NotificationContent

) =>

val actual = ClientEventAdapter.adaptEvent(

actionTowardNotificationEvent(

eventNamespace = Some(ceNotificationClick),

notificationTabDetails = Some(ceNotificationTabDetails)))

val expectedUUA = mkExpectedUUAForNotificationEvent(

clientEventNamespace = Some(uuaNotificationClick),

actionType = ActionType.ClientNotificationClick,

notificationContent = uuaNotificationContent,

productSurface = Some(ProductSurface.NotificationTab),

productSurfaceInfo = Some(notificationTabProductSurfaceInfo)

)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientNotificationSeeLessOften

test("ClientNotificationSeeLessOften") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val params = Table(

("notificationType", "ceNotificationTabDetails", "uuaNotificationContent"),

("tweetNotification", notificationTabTweetEventDetails, tweetNotificationContent),

(

"multiTweetNotification",

notificationTabMultiTweetEventDetails,

multiTweetNotificationContent),

("unknownNotification", notificationTabUnknownEventDetails, unknownNotificationContent),

)

forEvery(params) {

(

\_: String,

ceNotificationTabDetails: NotificationTabDetails,

uuaNotificationContent: NotificationContent

) =>

val actual = ClientEventAdapter.adaptEvent(

actionTowardNotificationEvent(

eventNamespace = Some(ceNotificationSeeLessOften),

notificationTabDetails = Some(ceNotificationTabDetails)))

val expectedUUA = mkExpectedUUAForNotificationEvent(

clientEventNamespace = Some(uuaNotificationSeeLessOften),

actionType = ActionType.ClientNotificationSeeLessOften,

notificationContent = uuaNotificationContent,

productSurface = Some(ProductSurface.NotificationTab),

productSurfaceInfo = Some(notificationTabProductSurfaceInfo)

)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientTweetClick

test("ClientTweetClick") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val params = Table(

("eventName", "page", "nTabDetails", "uuaProductSurface", "uuaProductSurfaceInfo"),

("tweetClick", "messages", None, None, None),

(

"tweetClickInNTab",

"ntab",

Some(notificationTabTweetEventDetails),

Some(ProductSurface.NotificationTab),

Some(notificationTabProductSurfaceInfo))

)

forEvery(params) {

(

\_: String,

page: String,

notificationTabDetails: Option[NotificationTabDetails],

uuaProductSurface: Option[ProductSurface],

uuaProductSurfaceInfo: Option[ProductSurfaceInfo]

) =>

val actual = ClientEventAdapter.adaptEvent(

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceTweetClick.copy(page = Some(page))),

notificationTabDetails = notificationTabDetails))

val expectedUUA = mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(uuaTweetClick.copy(page = Some(page))),

actionType = ActionType.ClientTweetClick,

productSurface = uuaProductSurface,

productSurfaceInfo = uuaProductSurfaceInfo

)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientTweetClickProfile

test("ClientTweetClickProfile") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val actual =

ClientEventAdapter.adaptEvent(

profileClickEvent(eventNamespace = Some(ceTweetClickProfile)))

val expectedUUA = mkExpectedUUAForProfileClick(

clientEventNamespace = Some(uuaTweetClickProfile),

actionType = ActionType.ClientTweetClickProfile,

authorInfo = Some(

AuthorInfo(

authorId = Some(authorId)

)))

assert(Seq(expectedUUA) === actual)

}

}

}

// Tests for ClientTweetClickShare

test("ClientTweetClickShare") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val actual =

ClientEventAdapter.adaptEvent(

actionTowardDefaultTweetEvent(

eventNamespace = Some(EventNamespace(action = Some("share\_menu\_click"))),

authorId = Some(authorId),

tweetPosition = Some(1),

promotedId = Some("promted\_123")

))

val expectedUUA = mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(ClientEventNamespace(action = Some("share\_menu\_click"))),

actionType = ActionType.ClientTweetClickShare,

authorInfo = Some(

AuthorInfo(

authorId = Some(authorId)

)),

tweetPosition = Some(1),

promotedId = Some("promted\_123")

)

assert(Seq(expectedUUA) === actual)

}

}

}

// Tests for ClientTweetShareVia\* and ClientTweetUnbookmark

test("ClientTweetShareVia and Unbookmark") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val input = Table(

("eventNamespaceAction", "uuaActionTypes"),

("bookmark", Seq(ActionType.ClientTweetShareViaBookmark, ActionType.ClientTweetBookmark)),

("copy\_link", Seq(ActionType.ClientTweetShareViaCopyLink)),

("share\_via\_dm", Seq(ActionType.ClientTweetClickSendViaDirectMessage)),

("unbookmark", Seq(ActionType.ClientTweetUnbookmark))

)

forEvery(input) { (eventNamespaceAction: String, uuaActionTypes: Seq[ActionType]) =>

val actual: Seq[UnifiedUserAction] =

ClientEventAdapter.adaptEvent(

actionTowardDefaultTweetEvent(

eventNamespace = Some(EventNamespace(action = Some(eventNamespaceAction))),

authorId = Some(authorId)))

implicit def any2iterable[A](a: A): Iterable[A] = Some(a)

val expectedUUA: Seq[UnifiedUserAction] = uuaActionTypes.flatMap { uuaActionType =>

mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace =

Some(ClientEventNamespace(action = Some(eventNamespaceAction))),

actionType = uuaActionType,

authorInfo = Some(

AuthorInfo(

authorId = Some(authorId)

))

)

}

assert(expectedUUA === actual)

}

}

}

}

// Test for ClientTweetClickHashtag

test("ClientTweetClickHashtag") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val events = Table(

("targets", "tweetActionInfo"),

(

Some(Seq(LogEventItem(name = Some("test\_hashtag")))),

Some(

TweetActionInfo.ClientTweetClickHashtag(

ClientTweetClickHashtag(hashtag = Some("test\_hashtag"))))),

(

Some(Seq.empty[LogEventItem]),

Some(TweetActionInfo.ClientTweetClickHashtag(ClientTweetClickHashtag(hashtag = None)))),

(

Some(Nil),

Some(TweetActionInfo.ClientTweetClickHashtag(ClientTweetClickHashtag(hashtag = None)))),

(

None,

Some(TweetActionInfo.ClientTweetClickHashtag(ClientTweetClickHashtag(hashtag = None))))

)

forEvery(events) {

(targets: Option[Seq[LogEventItem]], tweetActionInfo: Option[TweetActionInfo]) =>

val clientEvent =

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceClickHashtag),

targets = targets)

val expectedUUA = mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(uuaClickHashtagClientEventNamespace),

actionType = ActionType.ClientTweetClickHashtag,

tweetActionInfo = tweetActionInfo

)

assert(Seq(expectedUUA) === ClientEventAdapter.adaptEvent(clientEvent))

}

}

}

}

// Tests for ClientTweetVideoPlaybackStart and ClientTweetVideoPlaybackComplete

test("Client Tweet Video Playback Start and Complete") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val input = Table(

("ceNamespace", "uuaNamespace", "uuaActionType"),

(

ceVideoPlaybackStart,

uuaVideoPlaybackStartClientEventNamespace,

ActionType.ClientTweetVideoPlaybackStart),

(

ceVideoPlaybackComplete,

uuaVideoPlaybackCompleteClientEventNamespace,

ActionType.ClientTweetVideoPlaybackComplete),

)

for (element <- videoEventElementValues) {

forEvery(input) {

(

ceNamespace: EventNamespace,

uuaNamespace: ClientEventNamespace,

uuaActionType: ActionType

) =>

val clientEvent = actionTowardDefaultTweetEvent(

eventNamespace = Some(ceNamespace.copy(element = Some(element))),

mediaDetailsV2 = Some(mediaDetailsV2),

clientMediaEvent = Some(clientMediaEvent),

cardDetails = Some(cardDetails)

)

val expectedUUA = mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(uuaNamespace.copy(element = Some(element))),

actionType = uuaActionType,

tweetActionInfo = Some(videoMetadata)

)

assert(ClientEventAdapter.adaptEvent(clientEvent).contains(expectedUUA))

}

}

for (element <- invalidVideoEventElementValues) {

forEvery(input) {

(

ceNamespace: EventNamespace,

uuaNamespace: ClientEventNamespace,

uuaActionType: ActionType

) =>

val clientEvent = actionTowardDefaultTweetEvent(

eventNamespace = Some(ceNamespace.copy(element = Some(element))),

mediaDetailsV2 = Some(mediaDetailsV2),

clientMediaEvent = Some(clientMediaEvent)

)

val unexpectedUUA = mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(uuaNamespace.copy(element = Some(element))),

actionType = uuaActionType,

tweetActionInfo = Some(videoMetadata)

)

assert(!ClientEventAdapter.adaptEvent(clientEvent).contains(unexpectedUUA))

}

}

}

}

}

// Tests for ClientTweetNotRelevant and ClientTweetUndoNotRelevant

test("ClientTweetNotRelevant & UndoNotRelevant") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val actions = Table(("action"), "click", "undo")

val element = "feedback\_notrelevant"

forEvery(actions) { action =>

val clientEvent =

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceEventNamespace(element, action)),

)

val expectedUUA = mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(uuaClientEventNamespace(element, action)),

actionType = action match {

case "click" => ActionType.ClientTweetNotRelevant

case "undo" => ActionType.ClientTweetUndoNotRelevant

}

)

val actual = ClientEventAdapter.adaptEvent(clientEvent)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientNotificationDismiss

test("ClientNotificationDismiss") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvent =

pushNotificationEvent(

eventNamespace = Some(ceNotificationDismiss),

notificationDetails = Some(notificationDetails))

val expectedUUA = mkExpectedUUAForNotificationEvent(

clientEventNamespace = Some(uuaNotificationDismiss),

actionType = ActionType.ClientNotificationDismiss,

notificationContent = tweetNotificationContent,

productSurface = Some(ProductSurface.PushNotification),

productSurfaceInfo = Some(

ProductSurfaceInfo.PushNotificationInfo(

PushNotificationInfo(notificationId = notificationId)))

)

val actual = ClientEventAdapter.adaptEvent(clientEvent)

assert(Seq(expectedUUA) === actual)

}

}

}

// Tests for ClientTypeaheadClick

test("ClientTypeaheadClick") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val searchQuery = "searchQuery"

val input = Table(

("clientEventTargets", "typeaheadActionInfo"),

(

Some(Seq(LogEventItem(id = Some(userId), itemType = Some(ItemType.User)))),

TypeaheadActionInfo.UserResult(UserResult(profileId = userId))),

(

Some(Seq(LogEventItem(name = Some(s"$searchQuery"), itemType = Some(ItemType.Search)))),

TypeaheadActionInfo.TopicQueryResult(

TopicQueryResult(suggestedTopicQuery = s"$searchQuery")))

)

forEvery(input) {

(

clientEventTargets: Option[Seq[LogEventItem]],

typeaheadActionInfo: TypeaheadActionInfo,

) =>

val clientEvent =

actionTowardsTypeaheadEvent(

eventNamespace = Some(ceTypeaheadClick),

targets = clientEventTargets,

searchQuery = searchQuery)

val expectedUUA = mkExpectedUUAForTypeaheadAction(

clientEventNamespace = Some(uuaTypeaheadClick),

actionType = ActionType.ClientTypeaheadClick,

typeaheadActionInfo = typeaheadActionInfo,

searchQuery = searchQuery

)

val actual = ClientEventAdapter.adaptEvent(clientEvent)

assert(Seq(expectedUUA) === actual)

}

// Testing invalid target item type case

assert(

Seq() === ClientEventAdapter.adaptEvent(

actionTowardsTypeaheadEvent(

eventNamespace = Some(ceTypeaheadClick),

targets =

Some(Seq(LogEventItem(id = Some(itemTweetId), itemType = Some(ItemType.Tweet)))),

searchQuery = searchQuery)))

}

}

}

// Tests for ClientFeedbackPromptSubmit

test("ClientFeedbackPromptSubmit") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val searchQuery: String = "searchQuery"

val searchDetails = Some(SearchDetails(query = Some(searchQuery)))

val input = Table(

("logEvent", "uuaNamespace", "uuaActionType", "FeedbackPromptInfo"),

(

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceTweetRelevantToSearch),

searchDetails = searchDetails

),

uuaTweetRelevantToSearch,

ActionType.ClientFeedbackPromptSubmit,

FeedbackPromptInfo(feedbackPromptActionInfo =

FeedbackPromptActionInfo.TweetRelevantToSearch(

TweetRelevantToSearch(

searchQuery = searchQuery,

tweetId = itemTweetId,

isRelevant = Some(true))))),

(

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceTweetNotRelevantToSearch),

searchDetails = searchDetails

),

uuaTweetNotRelevantToSearch,

ActionType.ClientFeedbackPromptSubmit,

FeedbackPromptInfo(feedbackPromptActionInfo =

FeedbackPromptActionInfo.TweetRelevantToSearch(

TweetRelevantToSearch(

searchQuery = searchQuery,

tweetId = itemTweetId,

isRelevant = Some(false))))),

(

actionTowardSearchResultPageEvent(

eventNamespace = Some(ceSearchResultsRelevant),

searchDetails = searchDetails,

items = Some(Seq(LogEventItem(itemType = Some(ItemType.RelevancePrompt))))

),

uuaSearchResultsRelevant,

ActionType.ClientFeedbackPromptSubmit,

FeedbackPromptInfo(feedbackPromptActionInfo =

FeedbackPromptActionInfo.DidYouFindItSearch(

DidYouFindItSearch(searchQuery = searchQuery, isRelevant = Some(true))))),

(

actionTowardSearchResultPageEvent(

eventNamespace = Some(ceSearchResultsNotRelevant),

searchDetails = searchDetails,

items = Some(Seq(LogEventItem(itemType = Some(ItemType.RelevancePrompt))))

),

uuaSearchResultsNotRelevant,

ActionType.ClientFeedbackPromptSubmit,

FeedbackPromptInfo(feedbackPromptActionInfo =

FeedbackPromptActionInfo.DidYouFindItSearch(

DidYouFindItSearch(searchQuery = searchQuery, isRelevant = Some(false)))))

)

forEvery(input) {

(

logEvent: LogEvent,

uuaNamespace: ClientEventNamespace,

uuaActionType: ActionType,

feedbackPromptInfo: FeedbackPromptInfo

) =>

val actual =

ClientEventAdapter.adaptEvent(logEvent)

val expectedUUA = mkExpectedUUAForFeedbackSubmitAction(

clientEventNamespace = Some(uuaNamespace),

actionType = uuaActionType,

feedbackPromptInfo = feedbackPromptInfo,

searchQuery = searchQuery)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientProfile\*

test("ClientProfile\*") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val input = Table(

("eventName", "ceNamespace", "uuaNamespace", "uuaActionType"),

("profile\_block", ceProfileBlock, uuaProfileBlock, ActionType.ClientProfileBlock),

("profile\_unblock", ceProfileUnblock, uuaProfileUnblock, ActionType.ClientProfileUnblock),

("profile\_mute", ceProfileMute, uuaProfileMute, ActionType.ClientProfileMute),

("profile\_report", ceProfileReport, uuaProfileReport, ActionType.ClientProfileReport),

("profile\_follow", ceProfileFollow, uuaProfileFollow, ActionType.ClientProfileFollow),

("profile\_click", ceProfileClick, uuaProfileClick, ActionType.ClientProfileClick),

(

"profile\_follow\_attempt",

ceProfileFollowAttempt,

uuaProfileFollowAttempt,

ActionType.ClientProfileFollowAttempt),

("profile\_show", ceProfileShow, uuaProfileShow, ActionType.ClientProfileShow),

)

forEvery(input) {

(

eventName: String,

ceNamespace: EventNamespace,

uuaNamespace: ClientEventNamespace,

uuaActionType: ActionType

) =>

val actual =

ClientEventAdapter.adaptEvent(

actionTowardProfileEvent(

eventName = eventName,

eventNamespace = Some(ceNamespace)

))

val expectedUUA = mkExpectedUUAForProfileAction(

clientEventNamespace = Some(uuaNamespace),

actionType = uuaActionType,

actionProfileId = itemProfileId)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientTweetEngagementAttempt

test("ClientTweetEngagementAttempt") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvents = Table(

("eventName", "ceNamespace", "uuaNamespace", "uuaActionType"),

(

"tweet\_favourite\_attempt",

ceTweetFavoriteAttempt,

uuaTweetFavoriteAttempt,

ActionType.ClientTweetFavoriteAttempt),

(

"tweet\_retweet\_attempt",

ceTweetRetweetAttempt,

uuaTweetRetweetAttempt,

ActionType.ClientTweetRetweetAttempt),

(

"tweet\_reply\_attempt",

ceTweetReplyAttempt,

uuaTweetReplyAttempt,

ActionType.ClientTweetReplyAttempt),

)

forEvery(clientEvents) {

(

eventName: String,

ceNamespace: EventNamespace,

uuaNamespace: ClientEventNamespace,

uuaActionType: ActionType

) =>

val actual =

ClientEventAdapter.adaptEvent(actionTowardDefaultTweetEvent(Some(ceNamespace)))

val expectedUUA = mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(uuaNamespace),

actionType = uuaActionType)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for LoggedOut for ClientLogin\*

test("ClientLogin\*") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvents = Table(

("eventName", "ceNamespace", "uuaNamespace", "uuaActionType"),

(

"client\_click\_login",

ceClientCTALoginClick,

uuaClientCTALoginClick,

ActionType.ClientCTALoginClick),

(

"client\_click\_show",

ceClientCTALoginStart,

uuaClientCTALoginStart,

ActionType.ClientCTALoginStart),

(

"client\_login\_success",

ceClientCTALoginSuccess,

uuaClientCTALoginSuccess,

ActionType.ClientCTALoginSuccess),

)

forEvery(clientEvents) {

(

eventName: String,

ceNamespace: EventNamespace,

uuaNamespace: ClientEventNamespace,

uuaActionType: ActionType

) =>

val actual =

ClientEventAdapter.adaptEvent(

mkLogEvent(

eventName,

Some(ceNamespace),

logBase = Some(logBase1),

eventDetails = None,

pushNotificationDetails = None,

reportDetails = None,

searchDetails = None))

val expectedUUA = mkExpectedUUAForActionTowardCTAEvent(

clientEventNamespace = Some(uuaNamespace),

actionType = uuaActionType,

guestIdMarketingOpt = logBase1.guestIdMarketing

)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for LoggedOut for ClientSignup\*

test("ClientSignup\*") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvents = Table(

("eventName", "ceNamespace", "uuaNamespace", "uuaActionType"),

(

"client\_click\_signup",

ceClientCTASignupClick,

uuaClientCTASignupClick,

ActionType.ClientCTASignupClick),

(

"client\_signup\_success",

ceClientCTASignupSuccess,

uuaClientCTASignupSuccess,

ActionType.ClientCTASignupSuccess),

)

forEvery(clientEvents) {

(

eventName: String,

ceNamespace: EventNamespace,

uuaNamespace: ClientEventNamespace,

uuaActionType: ActionType

) =>

val actual =

ClientEventAdapter.adaptEvent(

mkLogEvent(

eventName,

Some(ceNamespace),

logBase = Some(logBase1),

eventDetails = None,

pushNotificationDetails = None,

reportDetails = None,

searchDetails = None))

val expectedUUA = mkExpectedUUAForActionTowardCTAEvent(

clientEventNamespace = Some(uuaNamespace),

actionType = uuaActionType,

guestIdMarketingOpt = logBase1.guestIdMarketing

)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientTweetFollowAuthor

test("ClientTweetFollowAuthor") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val testEventsList = Seq(

(ceTweetFollowAuthor1, uuaTweetFollowAuthor1, TweetAuthorFollowClickSource.CaretMenu),

(ceTweetFollowAuthor2, uuaTweetFollowAuthor2, TweetAuthorFollowClickSource.ProfileImage)

)

testEventsList.foreach {

case (eventNamespace, clientEventNamespace, followClickSource) =>

val actual =

ClientEventAdapter.adaptEvent(

tweetActionTowardAuthorEvent(

eventName = "tweet\_follow\_author",

eventNamespace = Some(eventNamespace)

))

val expectedUUA = mkExpectedUUAForTweetActionTowardAuthor(

clientEventNamespace = Some(clientEventNamespace),

actionType = ActionType.ClientTweetFollowAuthor,

authorInfo = Some(

AuthorInfo(

authorId = Some(authorId)

)),

tweetActionInfo = Some(

TweetActionInfo.ClientTweetFollowAuthor(

ClientTweetFollowAuthor(followClickSource)

))

)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientTweetUnfollowAuthor

test("ClientTweetUnfollowAuthor") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val testEventsList = Seq(

(

ceTweetUnfollowAuthor1,

uuaTweetUnfollowAuthor1,

TweetAuthorUnfollowClickSource.CaretMenu),

(

ceTweetUnfollowAuthor2,

uuaTweetUnfollowAuthor2,

TweetAuthorUnfollowClickSource.ProfileImage)

)

testEventsList.foreach {

case (eventNamespace, clientEventNamespace, unfollowClickSource) =>

val actual =

ClientEventAdapter.adaptEvent(

tweetActionTowardAuthorEvent(

eventName = "tweet\_unfollow\_author",

eventNamespace = Some(eventNamespace)

))

val expectedUUA = mkExpectedUUAForTweetActionTowardAuthor(

clientEventNamespace = Some(clientEventNamespace),

actionType = ActionType.ClientTweetUnfollowAuthor,

authorInfo = Some(

AuthorInfo(

authorId = Some(authorId)

)),

tweetActionInfo = Some(

TweetActionInfo.ClientTweetUnfollowAuthor(

ClientTweetUnfollowAuthor(unfollowClickSource)

))

)

assert(Seq(expectedUUA) === actual)

}

}

}

}

// Tests for ClientTweetMuteAuthor

test("ClientTweetMuteAuthor") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val actual =

ClientEventAdapter.adaptEvent(

tweetActionTowardAuthorEvent(

eventName = "tweet\_mute\_author",

eventNamespace = Some(ceTweetMuteAuthor)

))

val expectedUUA = mkExpectedUUAForTweetActionTowardAuthor(

clientEventNamespace = Some(uuaTweetMuteAuthor),

actionType = ActionType.ClientTweetMuteAuthor,

authorInfo = Some(

AuthorInfo(

authorId = Some(authorId)

)))

assert(Seq(expectedUUA) === actual)

}

}

}

// Tests for ClientTweetBlockAuthor

test("ClientTweetBlockAuthor") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val actual =

ClientEventAdapter.adaptEvent(

tweetActionTowardAuthorEvent(

eventName = "tweet\_block\_author",

eventNamespace = Some(ceTweetBlockAuthor)

))

val expectedUUA = mkExpectedUUAForTweetActionTowardAuthor(

clientEventNamespace = Some(uuaTweetBlockAuthor),

actionType = ActionType.ClientTweetBlockAuthor,

authorInfo = Some(

AuthorInfo(

authorId = Some(authorId)

)))

assert(Seq(expectedUUA) === actual)

}

}

}

// Tests for ClientTweetUnblockAuthor

test("ClientTweetUnblockAuthor") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val actual =

ClientEventAdapter.adaptEvent(

tweetActionTowardAuthorEvent(

eventName = "tweet\_unblock\_author",

eventNamespace = Some(ceTweetUnblockAuthor)

))

val expectedUUA = mkExpectedUUAForTweetActionTowardAuthor(

clientEventNamespace = Some(uuaTweetUnblockAuthor),

actionType = ActionType.ClientTweetUnblockAuthor,

authorInfo = Some(

AuthorInfo(

authorId = Some(authorId)

)))

assert(Seq(expectedUUA) === actual)

}

}

}

// Test for ClientTweetOpenLink

test("ClientTweetOpenLink") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val input = Table(

("url", "tweetActionInfo"),

(Some("go/url"), clientOpenLinkWithUrl),

(None, clientOpenLinkWithoutUrl)

)

forEvery(input) { (url: Option[String], tweetActionInfo: TweetActionInfo) =>

val clientEvent =

actionTowardDefaultTweetEvent(eventNamespace = Some(ceOpenLink), url = url)

val expectedUUA = mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(uuaOpenLinkClientEventNamespace),

actionType = ActionType.ClientTweetOpenLink,

tweetActionInfo = Some(tweetActionInfo)

)

assert(Seq(expectedUUA) === ClientEventAdapter.adaptEvent(clientEvent))

}

}

}

}

// Test for ClientTweetTakeScreenshot

test("Client take screenshot") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvent =

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceTakeScreenshot),

percentVisibleHeight100k = Some(100))

val expectedUUA = mkExpectedUUAForActionTowardDefaultTweetEvent(

clientEventNamespace = Some(uuaTakeScreenshotClientEventNamespace),

actionType = ActionType.ClientTweetTakeScreenshot,

tweetActionInfo = Some(clientTakeScreenshot)

)

assert(Seq(expectedUUA) === ClientEventAdapter.adaptEvent(clientEvent))

}

}

}

test("Home / Search product surface meta data") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val clientEvents = Table(

("actionTweetType", "clientEvent", "expectedUUAEvent"),

(

"homeTweetEventWithControllerData",

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceHomeFavoriteEventNamespace),

suggestionDetails = Some(

SuggestionDetails(decodedControllerData = Some(

homeTweetControllerDataV2(

injectedPosition = Some(1),

traceId = Some(traceId),

requestJoinId = Some(requestJoinId)

))))

),

expectedHomeTweetEventWithControllerData),

(

"homeTweetEventWithSuggestionType",

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceHomeFavoriteEventNamespace),

suggestionDetails = Some(

SuggestionDetails(

suggestionType = Some("Test\_type")

))),

expectedHomeTweetEventWithSuggestType),

(

"homeTweetEventWithControllerDataSuggestionType",

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceHomeFavoriteEventNamespace),

suggestionDetails = Some(

SuggestionDetails(

suggestionType = Some("Test\_type"),

decodedControllerData = Some(

homeTweetControllerDataV2(

injectedPosition = Some(1),

traceId = Some(traceId),

requestJoinId = Some(requestJoinId)))

))

),

expectedHomeTweetEventWithControllerDataSuggestType),

(

"homeLatestTweetEventWithControllerData",

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceHomeLatestFavoriteEventNamespace),

suggestionDetails = Some(

SuggestionDetails(decodedControllerData = Some(

homeTweetControllerDataV2(

injectedPosition = Some(1),

traceId = Some(traceId),

requestJoinId = Some(requestJoinId)

))))

),

expectedHomeLatestTweetEventWithControllerData),

(

"homeLatestTweetEventWithSuggestionType",

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceHomeLatestFavoriteEventNamespace),

suggestionDetails = Some(

SuggestionDetails(

suggestionType = Some("Test\_type")

))),

expectedHomeLatestTweetEventWithSuggestType),

(

"homeLatestTweetEventWithControllerDataSuggestionType",

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceHomeLatestFavoriteEventNamespace),

suggestionDetails = Some(

SuggestionDetails(

suggestionType = Some("Test\_type"),

decodedControllerData = Some(

homeTweetControllerDataV2(

injectedPosition = Some(1),

traceId = Some(traceId),

requestJoinId = Some(requestJoinId)))

))

),

expectedHomeLatestTweetEventWithControllerDataSuggestType),

(

"searchTweetEventWithControllerData",

actionTowardDefaultTweetEvent(

eventNamespace = Some(ceSearchFavoriteEventNamespace),

suggestionDetails = Some(

SuggestionDetails(decodedControllerData = Some(

mkSearchResultControllerData(

queryOpt = Some("twitter"),

traceId = Some(traceId),

requestJoinId = Some(requestJoinId)

))))

),

expectedSearchTweetEventWithControllerData),

)

forEvery(clientEvents) { (\_: String, event: LogEvent, expectedUUA: UnifiedUserAction) =>

val actual = ClientEventAdapter.adaptEvent(event)

assert(Seq(expectedUUA) === actual)

}

}

}

}

test("ClientAppExit") {

new TestFixtures.ClientEventFixture {

Time.withTimeAt(frozenTime) { \_ =>

val duration: Option[Long] = Some(10000L)

val inputTable = Table(

("eventType", "clientAppId", "section", "duration", "isValidEvent"),

("uas-iPhone", Some(129032L), Some("enter\_background"), duration, true),

("uas-iPad", Some(191841L), Some("enter\_background"), duration, true),

("uas-android", Some(258901L), None, duration, true),

("none-clientId", None, None, duration, false),

("invalid-clientId", Some(1L), None, duration, false),

("none-duration", Some(258901L), None, None, false),

("non-uas-iPhone", Some(129032L), None, duration, false)

)

forEvery(inputTable) {

(

\_: String,

clientAppId: Option[Long],

section: Option[String],

duration: Option[Long],

isValidEvent: Boolean

) =>

val actual = ClientEventAdapter.adaptEvent(

actionTowardsUasEvent(

eventNamespace = Some(ceAppExit.copy(section = section)),

clientAppId = clientAppId,

duration = duration

))

if (isValidEvent) {

// create UUA UAS event

val expectedUUA = mkExpectedUUAForUasEvent(

clientEventNamespace = Some(uuaAppExit.copy(section = section)),

actionType = ActionType.ClientAppExit,

clientAppId = clientAppId,

duration = duration

)

assert(Seq(expectedUUA) === actual)

} else {

// ignore the event and do not create UUA UAS event

assert(actual.isEmpty)

}

}

}

}

}

}