package com.twitter.frigate.pushservice.util

import com.google.common.io.BaseEncoding

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.pushservice.model.PushTypes.PushCandidate

import com.twitter.frigate.pushservice.model.PushTypes.Target

import com.twitter.frigate.pushservice.params.InlineActionsEnum

import com.twitter.frigate.pushservice.params.PushParams

import com.twitter.frigate.pushservice.params.PushFeatureSwitchParams

import com.twitter.ibis2.lib.util.JsonMarshal

import com.twitter.notifications.platform.thriftscala.\_

import com.twitter.notificationservice.thriftscala.CreateGenericNotificationResponse

import com.twitter.scrooge.BinaryThriftStructSerializer

import com.twitter.util.Future

/\*\*

\* This class provides utility functions for inline action for push

\*/

object InlineActionUtil {

def scopedStats(statsReceiver: StatsReceiver): StatsReceiver =

statsReceiver.scope(getClass.getSimpleName)

/\*\*

\* Util function to build web inline actions for Ibis

\* @param actions list of inline actions to be hydrated depending on the CRT

\* @param enableForDesktopWeb if web inline actions should be shown on desktop RWeb, for experimentation purpose

\* @param enableForMobileWeb if web inline actions should be shwon on mobile RWeb, for experimentation purpose

\* @return Params for web inline actions to be consumed by `smart.inline.actions.web.mustache` in Ibis

\*/

def getGeneratedTweetInlineActionsForWeb(

actions: Seq[InlineActionsEnum.Value],

enableForDesktopWeb: Boolean,

enableForMobileWeb: Boolean

): Map[String, String] = {

if (!enableForDesktopWeb && !enableForMobileWeb) {

Map.empty

} else {

val inlineActions = buildEnrichedInlineActionsMap(actions) ++ Map(

"enable\_for\_desktop\_web" -> enableForDesktopWeb.toString,

"enable\_for\_mobile\_web" -> enableForMobileWeb.toString

)

Map(

"inline\_action\_details\_web" -> JsonMarshal.toJson(inlineActions),

)

}

}

def getGeneratedTweetInlineActionsV1(

actions: Seq[InlineActionsEnum.Value]

): Map[String, String] = {

val inlineActions = buildEnrichedInlineActionsMap(actions)

Map(

"inline\_action\_details" -> JsonMarshal.toJson(inlineActions)

)

}

private def buildEnrichedInlineActionsMap(

actions: Seq[InlineActionsEnum.Value]

): Map[String, Seq[Map[String, Any]]] = {

Map(

"actions" -> actions

.map(\_.toString.toLowerCase)

.zipWithIndex

.map {

case (a: String, i: Int) =>

Map("action" -> a) ++ Map(

s"use\_${a}\_stringcenter\_key" -> true,

"last" -> (i == (actions.length - 1))

)

}.seq

)

}

def getGeneratedTweetInlineActionsV2(

actions: Seq[InlineActionsEnum.Value]

): Map[String, String] = {

val v2CustomActions = actions

.map {

case InlineActionsEnum.Favorite =>

NotificationCustomAction(

Some("mr\_inline\_favorite\_title"),

CustomActionData.LegacyAction(LegacyAction(ActionIdentifier.Favorite))

)

case InlineActionsEnum.Follow =>

NotificationCustomAction(

Some("mr\_inline\_follow\_title"),

CustomActionData.LegacyAction(LegacyAction(ActionIdentifier.Follow)))

case InlineActionsEnum.Reply =>

NotificationCustomAction(

Some("mr\_inline\_reply\_title"),

CustomActionData.LegacyAction(LegacyAction(ActionIdentifier.Reply)))

case InlineActionsEnum.Retweet =>

NotificationCustomAction(

Some("mr\_inline\_retweet\_title"),

CustomActionData.LegacyAction(LegacyAction(ActionIdentifier.Retweet)))

case \_ =>

NotificationCustomAction(

Some("mr\_inline\_favorite\_title"),

CustomActionData.LegacyAction(LegacyAction(ActionIdentifier.Favorite))

)

}

val notifications = NotificationCustomActions(v2CustomActions)

Map("serialized\_inline\_actions\_v2" -> serializeActionsToBase64(notifications))

}

def getDislikeInlineAction(

candidate: PushCandidate,

ntabResponse: CreateGenericNotificationResponse

): Option[NotificationCustomAction] = {

ntabResponse.successKey.map(successKey => {

val urlParams = Map[String, String](

"answer" -> "dislike",

"notification\_hash" -> successKey.hashKey.toString,

"upstream\_uid" -> candidate.impressionId,

"notification\_timestamp" -> successKey.timestampMillis.toString

)

val urlParamsString = urlParams.map(kvp => f"${kvp.\_1}=${kvp.\_2}").mkString("&")

val httpPostRequest = HttpRequest.PostRequest(

PostRequest(url = f"/2/notifications/feedback.json?$urlParamsString", bodyParams = None))

val httpRequestAction = HttpRequestAction(

httpRequest = httpPostRequest,

scribeAction = Option("dislike\_scribe\_action"),

isAuthorizationRequired = Option(true),

isDestructive = Option(false),

undoable = None

)

val dislikeAction = CustomActionData.HttpRequestAction(httpRequestAction)

NotificationCustomAction(title = Option("mr\_inline\_dislike\_title"), action = dislikeAction)

})

}

/\*\*

\* Given a serialized inline action v2, update the action at index to the given new action.

\* If given index is bigger than current action length, append the given inline action at the end.

\* @param serialized\_inline\_actions\_v2 the original action in serialized version

\* @param actionOption an Option of the new action to replace the old one

\* @param index the position where the old action will be replaced

\* @return a new serialized inline action v2

\*/

def patchInlineActionAtPosition(

serialized\_inline\_actions\_v2: String,

actionOption: Option[NotificationCustomAction],

index: Int

): String = {

val originalActions: Seq[NotificationCustomAction] = deserializeActionsFromString(

serialized\_inline\_actions\_v2).actions

val newActions = actionOption match {

case Some(action) if index >= originalActions.size => originalActions ++ Seq(action)

case Some(action) => originalActions.updated(index, action)

case \_ => originalActions

}

serializeActionsToBase64(NotificationCustomActions(newActions))

}

/\*\*

\* Return list of available inline actions for ibis2 model

\*/

def getGeneratedTweetInlineActions(

target: Target,

statsReceiver: StatsReceiver,

actions: Seq[InlineActionsEnum.Value],

): Map[String, String] = {

val scopedStatsReceiver = scopedStats(statsReceiver)

val useV1 = target.params(PushFeatureSwitchParams.UseInlineActionsV1)

val useV2 = target.params(PushFeatureSwitchParams.UseInlineActionsV2)

if (useV1 && useV2) {

scopedStatsReceiver.counter("use\_v1\_and\_use\_v2").incr()

getGeneratedTweetInlineActionsV1(actions) ++ getGeneratedTweetInlineActionsV2(actions)

} else if (useV1 && !useV2) {

scopedStatsReceiver.counter("only\_use\_v1").incr()

getGeneratedTweetInlineActionsV1(actions)

} else if (!useV1 && useV2) {

scopedStatsReceiver.counter("only\_use\_v2").incr()

getGeneratedTweetInlineActionsV2(actions)

} else {

scopedStatsReceiver.counter("use\_neither\_v1\_nor\_v2").incr()

Map.empty[String, String]

}

}

/\*\*

\* Return Tweet inline action ibis2 model values after applying experiment logic

\*/

def getTweetInlineActionValue(target: Target): Future[Map[String, String]] = {

if (target.isLoggedOutUser) {

Future(

Map(

"show\_inline\_action" -> "false"

)

)

} else {

val showInlineAction: Boolean = target.params(PushParams.MRAndroidInlineActionOnPushCopyParam)

Future(

Map(

"show\_inline\_action" -> s"$showInlineAction"

)

)

}

}

private val binaryThriftStructSerializer: BinaryThriftStructSerializer[

NotificationCustomActions

] = BinaryThriftStructSerializer.apply(NotificationCustomActions)

private val base64Encoding = BaseEncoding.base64()

def serializeActionsToBase64(notificationCustomActions: NotificationCustomActions): String = {

val actionsAsByteArray: Array[Byte] =

binaryThriftStructSerializer.toBytes(notificationCustomActions)

base64Encoding.encode(actionsAsByteArray)

}

def deserializeActionsFromString(serializedInlineActionV2: String): NotificationCustomActions = {

val actionAsByteArray = base64Encoding.decode(serializedInlineActionV2)

binaryThriftStructSerializer.fromBytes(actionAsByteArray)

}

}