package com.twitter.home\_mixer.functional\_component.side\_effect

import com.twitter.conversions.DurationOps.\_

import com.twitter.home\_mixer.functional\_component.decorator.HomeQueryTypePredicates

import com.twitter.home\_mixer.functional\_component.decorator.builder.HomeTweetTypePredicates

import com.twitter.home\_mixer.model.HomeFeatures.AccountAgeFeature

import com.twitter.home\_mixer.model.HomeFeatures.SuggestTypeFeature

import com.twitter.home\_mixer.model.HomeFeatures.VideoDurationMsFeature

import com.twitter.home\_mixer.model.request.FollowingProduct

import com.twitter.home\_mixer.model.request.ForYouProduct

import com.twitter.home\_mixer.model.request.ListTweetsProduct

import com.twitter.home\_mixer.model.request.SubscribedProduct

import com.twitter.product\_mixer.component\_library.side\_effect.ScribeClientEventSideEffect.ClientEvent

import com.twitter.product\_mixer.component\_library.side\_effect.ScribeClientEventSideEffect.EventNamespace

import com.twitter.product\_mixer.core.feature.featuremap.FeatureMap

import com.twitter.product\_mixer.core.model.common.presentation.CandidateWithDetails

import com.twitter.product\_mixer.core.model.common.presentation.ItemCandidateWithDetails

import com.twitter.product\_mixer.core.pipeline.PipelineQuery

import com.twitter.timelines.injection.scribe.InjectionScribeUtil

private[side\_effect] sealed trait ClientEventsBuilder {

private val FollowingSection = Some("latest")

private val ForYouSection = Some("home")

private val ListTweetsSection = Some("list")

private val SubscribedSection = Some("subscribed")

protected def section(query: PipelineQuery): Option[String] = {

query.product match {

case FollowingProduct => FollowingSection

case ForYouProduct => ForYouSection

case ListTweetsProduct => ListTweetsSection

case SubscribedProduct => SubscribedSection

case other => throw new UnsupportedOperationException(s"Unknown product: $other")

}

}

protected def count(

candidates: Seq[CandidateWithDetails],

predicate: FeatureMap => Boolean = \_ => true,

queryFeatures: FeatureMap = FeatureMap.empty

): Option[Long] = Some(candidates.view.count(item => predicate(item.features ++ queryFeatures)))

protected def sum(

candidates: Seq[CandidateWithDetails],

predicate: FeatureMap => Option[Int],

queryFeatures: FeatureMap = FeatureMap.empty

): Option[Long] =

Some(candidates.view.flatMap(item => predicate(item.features ++ queryFeatures)).sum)

}

private[side\_effect] object ServedEventsBuilder extends ClientEventsBuilder {

private val ServedTweetsAction = Some("served\_tweets")

private val ServedUsersAction = Some("served\_users")

private val InjectedComponent = Some("injected")

private val PromotedComponent = Some("promoted")

private val WhoToFollowComponent = Some("who\_to\_follow")

private val WhoToSubscribeComponent = Some("who\_to\_subscribe")

private val WithVideoDurationComponent = Some("with\_video\_duration")

private val VideoDurationSumElement = Some("video\_duration\_sum")

private val NumVideosElement = Some("num\_videos")

def build(

query: PipelineQuery,

injectedTweets: Seq[ItemCandidateWithDetails],

promotedTweets: Seq[ItemCandidateWithDetails],

whoToFollowUsers: Seq[ItemCandidateWithDetails],

whoToSubscribeUsers: Seq[ItemCandidateWithDetails]

): Seq[ClientEvent] = {

val baseEventNamespace = EventNamespace(

section = section(query),

action = ServedTweetsAction

)

val overallServedEvents = Seq(

ClientEvent(baseEventNamespace, eventValue = count(injectedTweets ++ promotedTweets)),

ClientEvent(

baseEventNamespace.copy(component = InjectedComponent),

eventValue = count(injectedTweets)),

ClientEvent(

baseEventNamespace.copy(component = PromotedComponent),

eventValue = count(promotedTweets)),

ClientEvent(

baseEventNamespace.copy(component = WhoToFollowComponent, action = ServedUsersAction),

eventValue = count(whoToFollowUsers)),

ClientEvent(

baseEventNamespace.copy(component = WhoToSubscribeComponent, action = ServedUsersAction),

eventValue = count(whoToSubscribeUsers)),

)

val tweetTypeServedEvents = HomeTweetTypePredicates.PredicateMap.map {

case (tweetType, predicate) =>

ClientEvent(

baseEventNamespace.copy(component = InjectedComponent, element = Some(tweetType)),

eventValue = count(injectedTweets, predicate, query.features.getOrElse(FeatureMap.empty))

)

}.toSeq

val suggestTypeServedEvents = injectedTweets

.flatMap(\_.features.getOrElse(SuggestTypeFeature, None))

.map {

InjectionScribeUtil.scribeComponent

}

.groupBy(identity).map {

case (suggestType, group) =>

ClientEvent(

baseEventNamespace.copy(component = suggestType),

eventValue = Some(group.size.toLong))

}.toSeq

// Video duration events

val numVideosEvent = ClientEvent(

baseEventNamespace.copy(component = WithVideoDurationComponent, element = NumVideosElement),

eventValue = count(injectedTweets, \_.getOrElse(VideoDurationMsFeature, None).nonEmpty)

)

val videoDurationSumEvent = ClientEvent(

baseEventNamespace

.copy(component = WithVideoDurationComponent, element = VideoDurationSumElement),

eventValue = sum(injectedTweets, \_.getOrElse(VideoDurationMsFeature, None))

)

val videoEvents = Seq(numVideosEvent, videoDurationSumEvent)

overallServedEvents ++ tweetTypeServedEvents ++ suggestTypeServedEvents ++ videoEvents

}

}

private[side\_effect] object EmptyTimelineEventsBuilder extends ClientEventsBuilder {

private val EmptyAction = Some("empty")

private val AccountAgeLessThan30MinutesComponent = Some("account\_age\_less\_than\_30\_minutes")

private val ServedNonPromotedTweetElement = Some("served\_non\_promoted\_tweet")

def build(

query: PipelineQuery,

injectedTweets: Seq[ItemCandidateWithDetails]

): Seq[ClientEvent] = {

val baseEventNamespace = EventNamespace(

section = section(query),

action = EmptyAction

)

// Empty timeline events

val accountAgeLessThan30Minutes = query.features

.flatMap(\_.getOrElse(AccountAgeFeature, None))

.exists(\_.untilNow < 30.minutes)

val isEmptyTimeline = count(injectedTweets).contains(0L)

val predicates = Seq(

None -> isEmptyTimeline,

AccountAgeLessThan30MinutesComponent -> (isEmptyTimeline && accountAgeLessThan30Minutes)

)

for {

(component, predicate) <- predicates

if predicate

} yield ClientEvent(

baseEventNamespace.copy(component = component, element = ServedNonPromotedTweetElement))

}

}

private[side\_effect] object QueryEventsBuilder extends ClientEventsBuilder {

private val ServedSizePredicateMap: Map[String, Int => Boolean] = Map(

("size\_is\_empty", \_ <= 0),

("size\_at\_most\_5", \_ <= 5),

("size\_at\_most\_10", \_ <= 10),

("size\_at\_most\_35", \_ <= 35)

)

def build(

query: PipelineQuery,

injectedTweets: Seq[ItemCandidateWithDetails]

): Seq[ClientEvent] = {

val baseEventNamespace = EventNamespace(

section = section(query)

)

val queryFeatureMap = query.features.getOrElse(FeatureMap.empty)

val servedSizeQueryEvents =

for {

(queryPredicateName, queryPredicate) <- HomeQueryTypePredicates.PredicateMap

if queryPredicate(queryFeatureMap)

(servedSizePredicateName, servedSizePredicate) <- ServedSizePredicateMap

if servedSizePredicate(injectedTweets.size)

} yield ClientEvent(

baseEventNamespace

.copy(component = Some(servedSizePredicateName), action = Some(queryPredicateName)))

servedSizeQueryEvents.toSeq

}

}