package com.twitter.visibility.rules

import com.twitter.abdecider.LoggingABDecider

import com.twitter.timelines.configapi.HasParams.DependencyProvider

import com.twitter.timelines.configapi.Params

import com.twitter.visibility.configapi.params.RuleParam

import com.twitter.visibility.configapi.params.RuleParams

import com.twitter.visibility.configapi.params.RuleParams.EnableLikelyIvsUserLabelDropRule

import com.twitter.visibility.features.\_

import com.twitter.visibility.models.UserLabelValue

import com.twitter.visibility.models.UserLabelValue.LikelyIvs

import com.twitter.visibility.rules.Condition.\_

import com.twitter.visibility.rules.Reason.Unspecified

import com.twitter.visibility.rules.RuleActionSourceBuilder.UserSafetyLabelSourceBuilder

import com.twitter.visibility.rules.State.\_

import com.twitter.visibility.util.NamingUtils

trait WithGate {

def enabled: Seq[RuleParam[Boolean]] = Seq(RuleParams.True)

def isEnabled(params: Params): Boolean =

enabled.forall(enabledParam => params(enabledParam))

def holdbacks: Seq[RuleParam[Boolean]] = Seq(RuleParams.False)

final def shouldHoldback: DependencyProvider[Boolean] =

holdbacks.foldLeft(DependencyProvider.from(RuleParams.False)) { (dp, holdbackParam) =>

dp.or(DependencyProvider.from(holdbackParam))

}

protected def enableFailClosed: Seq[RuleParam[Boolean]] = Seq(RuleParams.False)

def shouldFailClosed(params: Params): Boolean =

enableFailClosed.forall(fcParam => params(fcParam))

}

abstract class ActionBuilder[T <: Action] {

def actionType: Class[\_]

val actionSeverity: Int

def build(evaluationContext: EvaluationContext, featureMap: Map[Feature[\_], \_]): RuleResult

}

object ActionBuilder {

def apply[T <: Action](action: T): ActionBuilder[T] = action match {

case \_: InterstitialLimitedEngagements => new PublicInterestActionBuilder()

case \_ => new ConstantActionBuilder(action)

}

}

class ConstantActionBuilder[T <: Action](action: T) extends ActionBuilder[T] {

private val result = RuleResult(action, Evaluated)

def actionType: Class[\_] = action.getClass

override val actionSeverity = action.severity

def build(evaluationContext: EvaluationContext, featureMap: Map[Feature[\_], \_]): RuleResult =

result

}

object ConstantActionBuilder {

def unapply[T <: Action](builder: ConstantActionBuilder[T]): Option[Action] = Some(

builder.result.action)

}

abstract class Rule(val actionBuilder: ActionBuilder[\_ <: Action], val condition: Condition)

extends WithGate {

import Rule.\_

def isExperimental: Boolean = false

def actionSourceBuilder: Option[RuleActionSourceBuilder] = None

lazy val name: String = NamingUtils.getFriendlyName(this)

val featureDependencies: Set[Feature[\_]] = condition.features

val optionalFeatureDependencies: Set[Feature[\_]] = condition.optionalFeatures

def preFilter(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], Any],

abDecider: LoggingABDecider

): PreFilterResult =

condition.preFilter(evaluationContext, featureMap)

def actWhen(evaluationContext: EvaluationContext, featureMap: Map[Feature[\_], \_]): Boolean =

condition(evaluationContext, featureMap).asBoolean

val fallbackActionBuilder: Option[ActionBuilder[\_ <: Action]] = None

final def evaluate(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): RuleResult = {

val missingFeatures = featureDependencies.filterNot(featureMap.contains)

if (missingFeatures.nonEmpty) {

fallbackActionBuilder match {

case Some(fallbackAction) =>

fallbackAction.build(evaluationContext, featureMap)

case None =>

RuleResult(NotEvaluated, MissingFeature(missingFeatures))

}

} else {

try {

val act = actWhen(evaluationContext, featureMap)

if (!act) {

EvaluatedRuleResult

} else if (shouldHoldback(evaluationContext)) {

HeldbackRuleResult

} else {

actionBuilder.build(evaluationContext, featureMap)

}

} catch {

case t: Throwable =>

RuleResult(NotEvaluated, RuleFailed(t))

}

}

}

}

trait ExperimentalRule extends Rule {

override def isExperimental: Boolean = true

}

object Rule {

val HeldbackRuleResult: RuleResult = RuleResult(Allow, Heldback)

val EvaluatedRuleResult: RuleResult = RuleResult(Allow, Evaluated)

val DisabledRuleResult: RuleResult = RuleResult(NotEvaluated, Disabled)

def unapply(rule: Rule): Option[(ActionBuilder[\_ <: Action], Condition)] =

Some((rule.actionBuilder, rule.condition))

}

abstract class RuleWithConstantAction(val action: Action, override val condition: Condition)

extends Rule(ActionBuilder(action), condition)

abstract class UserHasLabelRule(action: Action, userLabelValue: UserLabelValue)

extends RuleWithConstantAction(action, AuthorHasLabel(userLabelValue)) {

override def actionSourceBuilder: Option[RuleActionSourceBuilder] = Some(

UserSafetyLabelSourceBuilder(userLabelValue))

}

abstract class ConditionWithUserLabelRule(

action: Action,

condition: Condition,

userLabelValue: UserLabelValue)

extends Rule(

ActionBuilder(action),

And(NonAuthorViewer, AuthorHasLabel(userLabelValue), condition)) {

override def actionSourceBuilder: Option[RuleActionSourceBuilder] = Some(

UserSafetyLabelSourceBuilder(userLabelValue))

}

abstract class WhenAuthorUserLabelPresentRule(action: Action, userLabelValue: UserLabelValue)

extends ConditionWithUserLabelRule(action, Condition.True, userLabelValue)

abstract class ConditionWithNotInnerCircleOfFriendsRule(

action: Action,

condition: Condition)

extends RuleWithConstantAction(

action,

And(Not(DoesHaveInnerCircleOfFriendsRelationship), condition))

abstract class AuthorLabelWithNotInnerCircleOfFriendsRule(

action: Action,

userLabelValue: UserLabelValue)

extends ConditionWithNotInnerCircleOfFriendsRule(

action,

AuthorHasLabel(userLabelValue)

) {

override def actionSourceBuilder: Option[RuleActionSourceBuilder] = Some(

UserSafetyLabelSourceBuilder(userLabelValue))

}

abstract class OnlyWhenNotAuthorViewerRule(action: Action, condition: Condition)

extends RuleWithConstantAction(action, And(NonAuthorViewer, condition))

abstract class AuthorLabelAndNonFollowerViewerRule(action: Action, userLabelValue: UserLabelValue)

extends ConditionWithUserLabelRule(action, LoggedOutOrViewerNotFollowingAuthor, userLabelValue)

abstract class AlwaysActRule(action: Action) extends Rule(ActionBuilder(action), Condition.True)

abstract class ViewerOptInBlockingOnSearchRule(action: Action, condition: Condition)

extends OnlyWhenNotAuthorViewerRule(

action,

And(condition, ViewerOptInBlockingOnSearch)

)

abstract class ViewerOptInFilteringOnSearchRule(action: Action, condition: Condition)

extends OnlyWhenNotAuthorViewerRule(

action,

And(condition, ViewerOptInFilteringOnSearch)

)

abstract class ViewerOptInFilteringOnSearchUserLabelRule(

action: Action,

userLabelValue: UserLabelValue,

prerequisiteCondition: Condition = True)

extends ConditionWithUserLabelRule(

action,

And(prerequisiteCondition, LoggedOutOrViewerOptInFiltering),

userLabelValue

)

abstract class LikelyIvsLabelNonFollowerDropRule

extends AuthorLabelAndNonFollowerViewerRule(

Drop(Unspecified),

LikelyIvs

) {

override def enabled: Seq[RuleParam[Boolean]] =

Seq(EnableLikelyIvsUserLabelDropRule)

}