package com.twitter.product\_mixer.core.feature.featuremap

import com.fasterxml.jackson.databind.annotation.JsonSerialize

import com.twitter.product\_mixer.core.feature.Feature

import com.twitter.product\_mixer.core.feature.FeatureWithDefaultOnFailure

import com.twitter.product\_mixer.core.feature.featurestorev1.featurevalue.FeatureStoreV1Response

import com.twitter.product\_mixer.core.feature.featurestorev1.featurevalue.{

FeatureStoreV1ResponseFeature => FSv1Feature

}

import com.twitter.util.Return

import com.twitter.util.Throw

import com.twitter.util.Try

/\*\*

\* A set of features and their values. Associated with a specific instance of an Entity, though

\* that association is maintained by the framework.

\*

\* [[FeatureMapBuilder]] is used to build new FeatureMap instances

\*/

@JsonSerialize(using = classOf[FeatureMapSerializer])

case class FeatureMap private[feature] (

private[core] val underlyingMap: Map[Feature[\_, \_], Try[\_]]) {

/\*\*

\* Returns the [[Value]] associated with the Feature

\*

\* If the Feature is missing from the feature map, it throws a [[MissingFeatureException]].

\* If the Feature failed and isn't a [[FeatureWithDefaultOnFailure]] this will throw the underlying exception

\* that the feature failed with during hydration.

\*/

def get[Value](feature: Feature[\_, Value]): Value =

getOrElse(feature, throw MissingFeatureException(feature), None)

/\*\*

\* Returns the [[Value]] associated with the Feature with the same semantics as

\* [[FeatureMap.get()]], but the underlying [[Try]] is returned to allow for checking the success

\* or error state of a feature hydration. This is helpful for implementing fall-back behavior in

\* case the feature is missing or hydration failed without a [[FeatureWithDefaultOnFailure]] set.

\*

\* @note The [[FeatureMap.getOrElse()]] logic is duplicated here to avoid unpacking and repacking

\* the [[Try]] that is already available in the [[underlyingMap]]

\*/

def getTry[Value](feature: Feature[\_, Value]): Try[Value] =

underlyingMap.get(feature) match {

case None => Throw(MissingFeatureException(feature))

case Some(value @ Return(\_)) => value.asInstanceOf[Return[Value]]

case Some(value @ Throw(\_)) =>

feature match {

case f: FeatureWithDefaultOnFailure[\_, Value] @unchecked => Return(f.defaultValue)

case \_ => value.asInstanceOf[Throw[Value]]

}

}

/\*\*

\* Returns the [[Value]] associated with the feature or a default if the key is not contained in the map

\* `default` can also be used to throw an exception.

\*

\* e.g. `.getOrElse(feature, throw new MyCustomException())`

\*

\* @note for [[FeatureWithDefaultOnFailure]], the [[FeatureWithDefaultOnFailure.defaultValue]]

\* will be returned if the [[Feature]] failed, but if it is missing/never hydrated,

\* then the `default` provided here will be used.

\*/

def getOrElse[Value](feature: Feature[\_, Value], default: => Value): Value = {

getOrElse(feature, default, Some(default))

}

/\*\*

\* Private helper for getting features from the feature map, allowing us to define a default

\* when the feature is missing from the feature map, vs when its in the feature map but failed.

\* In the case of failed features, if the feature is a [FeatureWithDefaultOnFailure], it will

\* prioritize that default.

\* @param feature The feature to retrieve

\* @param missingDefault The default value to use when the feature is missing from the map.

\* @param failureDefault The default value to use when the feature is present but failed.

\* @tparam Value The value type of the feature.

\* @return The value stored in the map.

\*/

private def getOrElse[Value](

feature: Feature[\_, Value],

missingDefault: => Value,

failureDefault: => Option[Value]

): Value =

underlyingMap.get(feature) match {

case None => missingDefault

case Some(Return(value)) => value.asInstanceOf[Value]

case Some(Throw(err)) =>

feature match {

case f: FeatureWithDefaultOnFailure[\_, Value] @unchecked => f.defaultValue

case \_ => failureDefault.getOrElse(throw err)

}

}

/\*\*

\* returns a new FeatureMap with

\* - the new Feature and Value pair if the Feature was not present

\* - overriding the previous Value if that Feature was previously present

\*/

def +[V](key: Feature[\_, V], value: V): FeatureMap =

new FeatureMap(underlyingMap + (key -> Return(value)))

/\*\*

\* returns a new FeatureMap with all the elements of current FeatureMap and `other`.

\*

\* @note if a [[Feature]] exists in both maps, the Value from `other` takes precedence

\*/

def ++(other: FeatureMap): FeatureMap = {

if (other.isEmpty) {

this

} else if (isEmpty) {

other

} else if (this.getFeatures.contains(FSv1Feature) && other.getFeatures.contains(FSv1Feature)) {

val mergedResponse =

FeatureStoreV1Response.merge(this.get(FSv1Feature), other.get(FSv1Feature))

val mergedResponseFeatureMap = FeatureMapBuilder().add(FSv1Feature, mergedResponse).build()

new FeatureMap(underlyingMap ++ other.underlyingMap ++ mergedResponseFeatureMap.underlyingMap)

} else {

new FeatureMap(underlyingMap ++ other.underlyingMap)

}

}

/\*\* returns the keySet of Features in the map \*/

def getFeatures: Set[Feature[\_, \_]] = underlyingMap.keySet

/\*\*

\* The Set of Features in the FeatureMap that have a successfully returned value. Failed features

\* will obviously not be here.

\*/

def getSuccessfulFeatures: Set[Feature[\_, \_]] = underlyingMap.collect {

case (feature, Return(\_)) => feature

}.toSet

def isEmpty: Boolean = underlyingMap.isEmpty

override def toString: String = s"FeatureMap(${underlyingMap.toString})"

}

object FeatureMap {

// Restrict access to the apply method.

// This shouldn't be required after scala 2.13.2 (https://github.com/scala/scala/pull/7702)

private[feature] def apply(underlyingMap: Map[Feature[\_, \_], Try[\_]]): FeatureMap =

FeatureMap(underlyingMap)

/\*\* Merges an arbitrary number of [[FeatureMap]]s from left-to-right \*/

def merge(featureMaps: TraversableOnce[FeatureMap]): FeatureMap = {

val builder = FeatureMapBuilder()

/\*\*

\* merge the current [[FSv1Feature]] with the existing accumulated one

\* and add the rest of the [[FeatureMap]]'s [[Feature]]s to the `builder`

\*/

def mergeInternal(

featureMap: FeatureMap,

accumulatedFsV1Response: Option[FeatureStoreV1Response]

): Option[FeatureStoreV1Response] = {

if (featureMap.isEmpty) {

accumulatedFsV1Response

} else {

val currentFsV1Response =

if (featureMap.getFeatures.contains(FSv1Feature))

Some(featureMap.get(FSv1Feature))

else

None

val mergedFsV1Response = (accumulatedFsV1Response, currentFsV1Response) match {

case (Some(merged), Some(current)) =>

// both present so merge them

Some(FeatureStoreV1Response.merge(merged, current))

case (merged, current) =>

// one or both are missing so use whichever is available

merged.orElse(current)

}

featureMap.underlyingMap.foreach {

case (FSv1Feature, \_) => // FSv1Feature is only added once at the very end

case (feature, value) => builder.addWithoutValidation(feature, value)

}

mergedFsV1Response

}

}

featureMaps

.foldLeft[Option[FeatureStoreV1Response]](None) {

case (fsV1Response, featureMap) => mergeInternal(featureMap, fsV1Response)

}.foreach(

// add merged `FSv1Feature` to the `builder` at the end

builder.add(FSv1Feature, \_)

)

builder.build()

}

val empty = new FeatureMap(Map.empty)

}