package com.twitter.product\_mixer.core.service.candidate\_feature\_transformer\_executor

import com.twitter.finagle.stats.StatsReceiver

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

import com.twitter.product\_mixer.core.functional\_component.transformer.CandidateFeatureTransformer

import com.twitter.product\_mixer.core.model.common.identifier.TransformerIdentifier

import com.twitter.product\_mixer.core.service.Executor

import com.twitter.product\_mixer.core.service.Executor.\_

import com.twitter.stitch.Arrow

import javax.inject.Inject

import javax.inject.Singleton

@Singleton

class CandidateFeatureTransformerExecutor @Inject() (override val statsReceiver: StatsReceiver)

extends Executor {

def arrow[Result](

transformers: Seq[CandidateFeatureTransformer[Result]],

context: Executor.Context

): Arrow[Seq[Result], CandidateFeatureTransformerExecutorResult] = {

if (transformers.isEmpty) {

// must always return a Seq of FeatureMaps, even if there are no Transformers

Arrow.map[Seq[Result], CandidateFeatureTransformerExecutorResult] { candidates =>

CandidateFeatureTransformerExecutorResult(candidates.map(\_ => FeatureMap.empty), Seq.empty)

}

} else {

val transformerArrows: Seq[Arrow[Seq[Result], Seq[(TransformerIdentifier, FeatureMap)]]] =

transformers.map { transformer =>

val transformerContext = context.pushToComponentStack(transformer.identifier)

val liftNonValidationFailuresToFailedFeatures =

Arrow.handle[FeatureMap, FeatureMap] {

case NotAMisconfiguredFeatureMapFailure(e) =>

featureMapWithFailuresForFeatures(transformer.features, e, transformerContext)

}

val underlyingArrow = Arrow

.map(transformer.transform)

.map(validateFeatureMap(transformer.features, \_, transformerContext))

val observedArrowWithoutTracing =

wrapPerCandidateComponentWithExecutorBookkeepingWithoutTracing(

context,

transformer.identifier)(underlyingArrow)

val seqArrow =

Arrow.sequence(

observedArrowWithoutTracing

.andThen(liftNonValidationFailuresToFailedFeatures)

.map(transformer.identifier -> \_)

)

wrapComponentsWithTracingOnly(context, transformer.identifier)(seqArrow)

}

Arrow.collect(transformerArrows).map { results =>

/\*\*

\* Inner Seqs are a given Transformer applied to all the candidates

\*

\* We want to merge the FeatureMaps for each candidate

\* from all the Transformers. We do this by merging all the FeatureMaps at

\* each index `i` of each Seq in `results` by `transpose`-ing the `results`

\* so the inner Seq becomes all the FeatureMaps for Candidate

\* at index `i` in the input Seq.

\*

\* {{{

\* Seq(

\* Seq(transformer1FeatureMapCandidate1, ..., transformer1FeatureMapCandidateN),

\* ...,

\* Seq(transformerMFeatureMapCandidate1, ..., transformerMFeatureMapCandidateN)

\* ).transpose == Seq(

\* Seq(transformer1FeatureMapCandidate1, ..., transformerMFeatureMapCandidate1),

\* ...,

\* Seq(transformer1FeatureMapCandidateN, ..., transformerMFeatureMapCandidateN)

\* )

\* }}}

\*

\* we could avoid the transpose if we ran each candidate through all the transformers

\* one-after-the-other, but then we couldn't have a single tracing span for all applications

\* of a Transformer, so instead we apply each transformer to all candidates together, then

\* move onto the next transformer.

\*

\* It's worth noting that the outer Seq is bounded by the number of Transformers that are

\* applied which will typically be small.

\*/

val transposed = results.transpose

val combinedMaps = transposed.map(featureMapsForSingleCandidate =>

FeatureMap.merge(featureMapsForSingleCandidate.map { case (\_, maps) => maps }))

CandidateFeatureTransformerExecutorResult(combinedMaps, transposed.map(\_.toMap))

}

}

}

}