package com.twitter.product\_mixer.component\_library.premarshaller.urt.builder

import com.twitter.product\_mixer.core.model.marshalling.response.urt.operation.CursorOperation

import com.twitter.product\_mixer.core.model.marshalling.response.urt.Timeline

import com.twitter.product\_mixer.core.model.marshalling.response.urt.TimelineEntry

import com.twitter.product\_mixer.core.model.marshalling.response.urt.TimelineInstruction

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

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

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

import com.twitter.product\_mixer.core.util.SortIndexBuilder

trait UrtBuilder[-Query <: PipelineQuery, +Instruction <: TimelineInstruction] {

private val TimelineIdSuffix = "-Timeline"

def instructionBuilders: Seq[UrtInstructionBuilder[Query, Instruction]]

def cursorBuilders: Seq[UrtCursorBuilder[Query]]

def cursorUpdaters: Seq[UrtCursorUpdater[Query]]

def metadataBuilder: Option[BaseUrtMetadataBuilder[Query]]

// Timeline entry sort indexes will count down by this value. Values higher than 1 are useful to

// leave room in the sequence for dynamically injecting content in between existing entries.

def sortIndexStep: Int = 1

final def buildTimeline(

query: Query,

entries: Seq[TimelineEntry]

): Timeline = {

val initialSortIndex = getInitialSortIndex(query)

// Set the sort indexes of the entries before we pass them to the cursor builders, since many

// cursor implementations use the sort index of the first/last entry as part of the cursor value

val sortIndexedEntries = updateSortIndexes(initialSortIndex, entries)

// Iterate over the cursorUpdaters in the order they were defined. Note that each updater will

// be passed the timelineEntries updated by the previous cursorUpdater.

val updatedCursorEntries: Seq[TimelineEntry] =

cursorUpdaters.foldLeft(sortIndexedEntries) { (timelineEntries, cursorUpdater) =>

cursorUpdater.update(query, timelineEntries)

}

val allCursoredEntries =

updatedCursorEntries ++ cursorBuilders.flatMap(\_.build(query, updatedCursorEntries))

val instructions: Seq[Instruction] =

instructionBuilders.flatMap(\_.build(query, allCursoredEntries))

val metadata = metadataBuilder.map(\_.build(query, allCursoredEntries))

Timeline(

id = query.product.identifier.toString + TimelineIdSuffix,

instructions = instructions,

metadata = metadata

)

}

final def getInitialSortIndex(query: Query): Long =

query match {

case cursorQuery: HasPipelineCursor[\_] =>

UrtPipelineCursor

.getCursorInitialSortIndex(cursorQuery)

.getOrElse(SortIndexBuilder.timeToId(query.queryTime))

case \_ => SortIndexBuilder.timeToId(query.queryTime)

}

/\*\*

\* Updates the sort indexes in the timeline entries starting from the given initial sort index

\* value and decreasing by the value defined in the sort index step field

\*

\* @param initialSortIndex The initial value of the sort index

\* @param timelineEntries Timeline entries to update

\*/

final def updateSortIndexes(

initialSortIndex: Long,

timelineEntries: Seq[TimelineEntry]

): Seq[TimelineEntry] = {

val indexRange =

initialSortIndex to (initialSortIndex - (timelineEntries.size \* sortIndexStep)) by -sortIndexStep

// Skip any existing cursors because their sort indexes will be managed by their cursor updater.

// If the cursors are not removed first, then the remaining entries would have a gap everywhere

// an existing cursor was present.

val (cursorEntries, nonCursorEntries) = timelineEntries.partition {

case \_: CursorOperation => true

case \_ => false

}

nonCursorEntries.zip(indexRange).map {

case (entry, index) =>

entry.withSortIndex(index)

} ++ cursorEntries

}

}