package com.twitter.product\_mixer.component\_library.decorator.urt.builder.richtext.twitter\_text

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

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

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

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

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

import scala.annotation.tailrec

import scala.collection.mutable

object TwitterTextRenderer {

/\*\*

\* Creates a new [[TwitterTextRenderer]] instance.

\* @param text The initial text representation

\* @param rtl Defines whether this text is in an RTL language

\* @param alignment Assigns the [[RichTextAlignment]] of the given text for display

\* @return A new [[TwitterTextRenderer]] instance

\*/

def apply(

text: String,

rtl: Option[Boolean] = None,

alignment: Option[RichTextAlignment] = None

): TwitterTextRenderer = {

TwitterTextRenderer(rtl, alignment).append(text)

}

/\*\*

\* Creates a new [[TwitterTextRenderer]] instance from a product-mixer [[RichText]] object.

\* Converts Unicode entity indexes into JVM String indexes.

\* @param richText The product-mixer [[RichText]] representation

\* @return A new [[TwitterTextRenderer]] instance

\*/

def fromRichText(richText: RichText): TwitterTextRenderer = {

val builder = TwitterTextRenderer(richText.text, richText.rtl, richText.alignment)

richText.entities.foreach { e =>

val startIndex = richText.text.offsetByCodePoints(0, e.fromIndex)

val endIndex = richText.text.offsetByCodePoints(0, e.toIndex)

e.format.foreach { f =>

builder.setFormat(startIndex, endIndex, f)

}

e.ref.foreach { r =>

builder.setRefObject(startIndex, endIndex, r)

}

}

builder

}

private def buildRichTextEntity(

text: String,

entity: TwitterTextRendererEntity[\_]

): RichTextEntity = {

val fromIndex = text.codePointCount(0, entity.startIndex)

val toIndex = text.codePointCount(0, entity.endIndex)

entity.value match {

case format: RichTextFormat =>

RichTextEntity(fromIndex, toIndex, ref = None, format = Some(format))

case ref: ReferenceObject =>

RichTextEntity(fromIndex, toIndex, ref = Some(ref), format = None)

}

}

}

case class TwitterTextRenderer(

rtl: Option[Boolean],

alignment: Option[RichTextAlignment],

) {

private[this] val textBuilder = new mutable.StringBuilder()

private[richtext] val formatBuffer =

mutable.ArrayBuffer[TwitterTextRendererEntity[RichTextFormat]]()

private[richtext] val refObjectBuffer =

mutable.ArrayBuffer[TwitterTextRendererEntity[ReferenceObject]]()

/\*\*

\* Appends a string with attached [[RichTextFormat]] and [[ReferenceObject]] information.

\* @param string The text to append to the end of the existing text

\* @param format The [[RichTextFormat]] assigned to the new text

\* @param refObject The [[ReferenceObject]] assigned to the new text

\* @return this

\*/

def append(

string: String,

format: Option[RichTextFormat] = None,

refObject: Option[ReferenceObject] = None

): TwitterTextRenderer = {

if (string.nonEmpty) {

val start = textBuilder.length

val end = start + string.length

format.foreach { f =>

formatBuffer.append(TwitterTextRendererEntity(start, end, f))

}

refObject.foreach { r =>

refObjectBuffer.append(TwitterTextRendererEntity(start, end, r))

}

textBuilder.append(string)

}

this

}

/\*\*

\* Builds a new [[RichText]] thrift instance with Unicode entity ranges.

\*/

def build: RichText = {

val richTextString = this.text

val richTextEntities = this.entities

.map { e =>

TwitterTextRenderer.buildRichTextEntity(richTextString, e)

}

RichText(

text = richTextString,

rtl = rtl,

alignment = alignment,

entities = richTextEntities.toList

)

}

/\*\*

\* Modifies the TwitterTextRenderer with the provided [[TwitterTextRendererProcessor]]

\*/

def transform(twitterTextProcessor: TwitterTextRendererProcessor): TwitterTextRenderer = {

twitterTextProcessor.process(this)

}

/\*\*

\* Builds and returns a sorted list of [[TwitterTextRendererEntity]] with JVM String index entity ranges.

\*/

def entities: Seq[TwitterTextRendererEntity[\_]] = {

buildEntities(formatBuffer.toList, refObjectBuffer.toList)

}

/\*\*

\* Assigns a [[RichTextFormat]] to the given range while keeping existing formatting information.

\* New formatting will only be assigned to unformatted text ranges.

\* @param start Start index to apply formatting (inclusive)

\* @param end End index to apply formatting (exclusive)

\* @param format The format to assign

\* @return this

\*/

def mergeFormat(start: Int, end: Int, format: RichTextFormat): TwitterTextRenderer = {

validateRange(start, end)

var injectionIndex: Option[Int] = None

var entity = TwitterTextRendererEntity(start, end, format)

val buffer = mutable.ArrayBuffer[TwitterTextRendererEntity[RichTextFormat]]()

val iterator = formatBuffer.zipWithIndex.reverseIterator

while (iterator.hasNext && injectionIndex.isEmpty) {

iterator.next match {

case (e, i) if e.startIndex >= end =>

buffer.append(e)

case (e, i) if e.enclosedIn(entity.startIndex, entity.endIndex) =>

val endEntity = entity.copy(startIndex = e.endIndex)

if (endEntity.nonEmpty) { buffer.append(endEntity) }

buffer.append(e)

entity = entity.copy(endIndex = e.startIndex)

case (e, i) if e.encloses(entity.startIndex, entity.endIndex) =>

buffer.append(e.copy(startIndex = entity.endIndex))

buffer.append(e.copy(endIndex = entity.startIndex))

injectionIndex = Some(i + 1)

case (e, i) if e.startsBetween(entity.startIndex, entity.endIndex) =>

buffer.append(e)

entity = entity.copy(endIndex = e.startIndex)

case (e, i) if e.endsBetween(entity.startIndex, entity.endIndex) =>

buffer.append(e)

entity = entity.copy(startIndex = e.endIndex)

injectionIndex = Some(i + 1)

case (e, i) if e.endIndex <= entity.startIndex =>

buffer.append(e)

injectionIndex = Some(i + 1)

case \_ => // do nothing

}

}

val index = injectionIndex.map(\_ - 1).getOrElse(0)

formatBuffer.remove(index, formatBuffer.length - index)

formatBuffer.appendAll(buffer.reverse)

if (entity.nonEmpty) {

formatBuffer.insert(injectionIndex.getOrElse(0), entity)

}

this

}

/\*\*

\* Removes text, formatting, and refObject information from the given range.

\* @param start Start index to apply formatting (inclusive)

\* @param end End index to apply formatting (exclusive)

\* @return this

\*/

def remove(start: Int, end: Int): TwitterTextRenderer = replace(start, end, "")

/\*\*

\* Replaces text, formatting, and refObject information in the given range.

\* @param start Start index to apply formatting (inclusive)

\* @param end End index to apply formatting (exclusive)

\* @param string The new text to insert

\* @param format The [[RichTextFormat]] assigned to the new text

\* @param refObject The [[ReferenceObject]] assigned to the new text

\* @return this

\*/

def replace(

start: Int,

end: Int,

string: String,

format: Option[RichTextFormat] = None,

refObject: Option[ReferenceObject] = None

): TwitterTextRenderer = {

validateRange(start, end)

val newEnd = start + string.length

val formatInjectIndex = removeAndOffsetFormats(start, end, string.length)

val refObjectInjectIndex = removeAndOffsetRefObjects(start, end, string.length)

format.foreach { f =>

formatBuffer.insert(formatInjectIndex, TwitterTextRendererEntity(start, newEnd, f))

}

refObject.foreach { r =>

refObjectBuffer.insert(refObjectInjectIndex, TwitterTextRendererEntity(start, newEnd, r))

}

textBuilder.replace(start, end, string)

this

}

/\*\*

\* Assigns a [[RichTextFormat]] to the given range. Trims existing format ranges that overlap the

\* new format range. Removes format ranges that fall within the new range.

\* @param start Start index to apply formatting (inclusive)

\* @param end End index to apply formatting (exclusive)

\* @param format The format to assign

\* @return this

\*/

def setFormat(start: Int, end: Int, format: RichTextFormat): TwitterTextRenderer = {

validateRange(start, end)

val bufferIndex = removeAndOffsetFormats(start, end, end - start)

formatBuffer.insert(bufferIndex, TwitterTextRendererEntity(start, end, format))

this

}

private[this] def removeAndOffsetFormats(start: Int, end: Int, newSize: Int): Int = {

val newEnd = start + newSize

val offset = newEnd - end

var injectionIndex: Option[Int] = None

val buffer = mutable.ArrayBuffer[TwitterTextRendererEntity[RichTextFormat]]()

val iterator = formatBuffer.zipWithIndex.reverseIterator

while (iterator.hasNext && injectionIndex.isEmpty) {

iterator.next match {

case (e, i) if e.startIndex >= end =>

buffer.append(e.offset(offset))

case (e, i) if e.encloses(start, end) =>

buffer.append(e.copy(startIndex = newEnd, endIndex = e.endIndex + offset))

buffer.append(e.copy(endIndex = e.endIndex + offset))

injectionIndex = Some(i + 1)

case (e, i) if e.endsBetween(start, end) =>

buffer.append(e.copy(endIndex = start))

injectionIndex = Some(i + 1)

case (e, i) if e.startsBetween(start, end) =>

buffer.append(e.copy(startIndex = newEnd, endIndex = e.endIndex + offset))

case (e, i) if e.endIndex <= start =>

buffer.append(e)

injectionIndex = Some(i + 1)

case \_ => // do nothing

}

}

val index = injectionIndex.map(\_ - 1).getOrElse(0)

formatBuffer.remove(index, formatBuffer.length - index)

formatBuffer.appendAll(buffer.reverse)

injectionIndex.getOrElse(0)

}

private[this] def validateRange(start: Int, end: Int): Unit = {

require(

start >= 0 && start < textBuilder.length && end > start && end <= textBuilder.length,

s"The start ($start) and end ($end) indexes must be within the text range (0..${textBuilder.length})"

)

}

/\*\*

\* Assigns a [[ReferenceObject]] to the given range. Since it makes little sense to trim object

\* ranges, existing intersecting or overlapping ranges are removed entirely.

\* @param start Start index to apply formatting (inclusive)

\* @param end End index to apply formatting (exclusive)

\* @param refObject The [[ReferenceObject]] to assign

\* @return this

\*/

def setRefObject(start: Int, end: Int, refObject: ReferenceObject): TwitterTextRenderer = {

validateRange(start, end)

val bufferIndex = removeAndOffsetRefObjects(start, end, end - start)

refObjectBuffer.insert(bufferIndex, TwitterTextRendererEntity(start, end, refObject))

this

}

private[this] def removeAndOffsetRefObjects(start: Int, end: Int, newSize: Int): Int = {

val newEnd = start + newSize

val offset = newEnd - end

var injectionIndex: Option[Int] = None

val buffer = mutable.ArrayBuffer[TwitterTextRendererEntity[ReferenceObject]]()

val iterator = refObjectBuffer.zipWithIndex.reverseIterator

while (iterator.hasNext && injectionIndex.isEmpty) {

iterator.next match {

case (e, i) if e.startIndex >= end => buffer.append(e.offset(offset))

case (e, i) if e.endIndex <= start => injectionIndex = Some(i + 1)

case \_ => // do nothing

}

}

val index = injectionIndex.getOrElse(0)

refObjectBuffer.remove(index, refObjectBuffer.length - index)

refObjectBuffer.appendAll(buffer.reverse)

index

}

/\*\*

\* Builds and returns the full TwitterTextRenderer text with any changes applied to the builder instance.

\*/

def text: String = {

textBuilder.mkString

}

@tailrec

private def buildEntities(

formats: List[TwitterTextRendererEntity[RichTextFormat]],

refs: List[TwitterTextRendererEntity[ReferenceObject]],

acc: List[TwitterTextRendererEntity[\_]] = List()

): Seq[TwitterTextRendererEntity[\_]] = {

(formats, refs) match {

case (Nil, Nil) => acc

case (remainingFormats, Nil) => acc ++ remainingFormats

case (Nil, remainingRefs) => acc ++ remainingRefs

case (format +: remainingFormats, ref +: remainingRefs)

if format.startIndex < ref.startIndex || (format.startIndex == ref.startIndex && format.endIndex < ref.endIndex) =>

buildEntities(remainingFormats, refs, acc :+ format)

case (format +: remainingFormats, ref +: remainingRefs)

if format.startIndex == ref.startIndex && format.endIndex == ref.endIndex =>

buildEntities(remainingFormats, remainingRefs, acc :+ format :+ ref)

case (\_, ref +: remainingRefs) =>

buildEntities(formats, remainingRefs, acc :+ ref)

}

}

}

case class TwitterTextRendererEntity[+T] private[richtext] (

startIndex: Int,

endIndex: Int,

value: T) {

require(startIndex <= endIndex, "startIndex must be <= than endIndex")

def nonEmpty: Boolean = !isEmpty

def isEmpty: Boolean = startIndex == endIndex

private[richtext] def enclosedIn(start: Int, end: Int): Boolean = {

start <= startIndex && endIndex <= end

}

private[richtext] def encloses(start: Int, end: Int): Boolean = {

startIndex < start && end < endIndex

}

private[richtext] def endsBetween(start: Int, end: Int): Boolean = {

start < endIndex && endIndex <= end && startIndex < start

}

private[richtext] def offset(num: Int): TwitterTextRendererEntity[T] = {

copy(startIndex = startIndex + num, endIndex = endIndex + num)

}

private[richtext] def startsBetween(start: Int, end: Int): Boolean = {

startIndex >= start && startIndex < end && endIndex > end

}

}