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

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

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

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

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.Strong

/\*

\* RichTextMarkupUtil facilitates building a Product Mixer URT RichText object out of

\* a string with inline XML markup.

\*

\* This allows us to use a string like "Our system <a href="#promix">Product Mixer</a> is the <b>best</b>". Using

\* inline markup like this is advantageous since the string can go through translation/localization and the

\* translators will move the tags around in each language as appropriate.

\*

\* This class is derived from the OCF (onboarding/serve)'s RichTextUtil, but they diverge because:

\* - We generate ProMix URT structures, not OCF URT structures

\* - The OCF supports some internal OCF tags, like <data>

\* - The OCF has additional legacy support and processing that we don't need

\*/

object RichTextMarkupUtil {

// Matches a anchor element, extracting the 'a' tag and the display text.

// First group is the tag

// Second group is the display text

// Allows any character in the display text, but matches reluctantly

private val LinkAnchorRegex = """(?i)(?s)<a\s+href\s\*=\s\*"#([\w-]\*)">(.\*?)</a>""".r

// Matches a <b>bold text section</b>

private val BoldRegex = """(?i)(?s)<b>(.\*?)</b>""".r

def richTextFromMarkup(

text: String,

linkMap: Map[String, String],

rtl: Option[Boolean] = None,

alignment: Option[RichTextAlignment] = None,

linkTypeMap: Map[String, UrlType] = Map.empty

): RichText = {

// Mutable!

var currentText = text

val entities = scala.collection.mutable.ArrayBuffer.empty[RichTextEntity]

// Using a while loop since we want to execute the regex after each iteration, so our indexes remain consistent

// Handle Links

var matchOpt = LinkAnchorRegex.findFirstMatchIn(currentText)

while (matchOpt.isDefined) {

matchOpt.foreach { linkMatch =>

val tag = linkMatch.group(1)

val displayText = linkMatch.group(2)

currentText = currentText.substring(0, linkMatch.start) + displayText + currentText

.substring(linkMatch.end)

adjustEntities(

entities,

linkMatch.start,

linkMatch.end - (linkMatch.start + displayText.length))

entities.append(

RichTextEntity(

fromIndex = linkMatch.start,

toIndex = linkMatch.start + displayText.length,

ref = linkMap.get(tag).map { url =>

Url(

urlType = linkTypeMap.getOrElse(tag, ExternalUrl),

url = url

)

},

format = None

)

)

}

matchOpt = LinkAnchorRegex.findFirstMatchIn(currentText)

}

// Handle Bold

matchOpt = BoldRegex.findFirstMatchIn(currentText)

while (matchOpt.isDefined) {

matchOpt.foreach { boldMatch =>

val text = boldMatch.group(1)

currentText =

currentText.substring(0, boldMatch.start) + text + currentText.substring(boldMatch.end)

adjustEntities(entities, boldMatch.start, boldMatch.end - (boldMatch.start + text.length))

entities.append(

RichTextEntity(

fromIndex = boldMatch.start,

toIndex = boldMatch.start + text.length,

ref = None,

format = Some(Strong),

)

)

}

matchOpt = BoldRegex.findFirstMatchIn(currentText)

}

RichText(

currentText,

entities.sortBy(\_.fromIndex).toList, // always return immutable copies!

rtl,

alignment

)

}

/\* When we create a new entity, we need to adjust

\* any already existing entities that have been moved.

\* Entities cannot overlap, so we can just compare start positions.

\*/

private def adjustEntities(

entities: scala.collection.mutable.ArrayBuffer[RichTextEntity],

start: Int,

length: Int

): Unit = {

for (i <- entities.indices) {

if (entities(i).fromIndex > start) {

val old = entities(i)

entities.update(

i,

entities(i).copy(

fromIndex = old.fromIndex - length,

toIndex = old.toIndex - length

))

}

}

}

}