001/\*  
002 \* Licensed to the Apache Software Foundation (ASF) under one or more  
003 \* contributor license agreements. See the NOTICE file distributed with  
004 \* this work for additional information regarding copyright ownership.  
005 \* The ASF licenses this file to You under the Apache License, Version 2.0  
006 \* (the "License"); you may not use this file except in compliance with  
007 \* the License. You may obtain a copy of the License at  
008 \*  
009 \* http://www.apache.org/licenses/LICENSE-2.0  
010 \*  
011 \* Unless required by applicable law or agreed to in writing, software  
012 \* distributed under the License is distributed on an "AS IS" BASIS,  
013 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
014 \* See the License for the specific language governing permissions and  
015 \* limitations under the License.  
016 \*/  
017package org.apache.commons.beanutils.converters;  
018  
019import java.text.DateFormat;  
020import java.text.ParsePosition;  
021import java.text.SimpleDateFormat;  
022import java.util.Calendar;  
023import java.util.Date;  
024import java.util.Locale;  
025import java.util.TimeZone;  
026  
027import org.apache.commons.beanutils.ConversionException;  
028  
029/\*\*  
030 \* {@link org.apache.commons.beanutils.Converter} implementaion  
031 \* that handles conversion to and from <b>date/time</b> objects.  
032 \* <p>  
033 \* This implementation handles conversion for the following  
034 \* <i>date/time</i> types.  
035 \* <ul>  
036 \* <li><code>java.util.Date</code></li>  
037 \* <li><code>java.util.Calendar</code></li>  
038 \* <li><code>java.sql.Date</code></li>  
039 \* <li><code>java.sql.Time</code></li>  
040 \* <li><code>java.sql.Timestamp</code></li>  
041 \* </ul>  
042 \*  
043 \* <h3>String Conversions (to and from)</h3>  
044 \* This class provides a number of ways in which date/time  
045 \* conversions to/from Strings can be achieved:  
046 \* <ul>  
047 \* <li>Using the SHORT date format for the default Locale, configure using:</li>  
048 \* <ul>  
049 \* <li><code>setUseLocaleFormat(true)</code></li>  
050 \* </ul>  
051 \* <li>Using the SHORT date format for a specified Locale, configure using:</li>  
052 \* <ul>  
053 \* <li><code>setLocale(Locale)</code></li>  
054 \* </ul>  
055 \* <li>Using the specified date pattern(s) for the default Locale, configure using:</li>  
056 \* <ul>  
057 \* <li>Either <code>setPattern(String)</code> or  
058 \* <code>setPatterns(String[])</code></li>  
059 \* </ul>  
060 \* <li>Using the specified date pattern(s) for a specified Locale, configure using:</li>  
061 \* <ul>  
062 \* <li><code>setPattern(String)</code> or  
063 \* <code>setPatterns(String[]) and...</code></li>  
064 \* <li><code>setLocale(Locale)</code></li>  
065 \* </ul>  
066 \* <li>If none of the above are configured the  
067 \* <code>toDate(String)</code> method is used to convert  
068 \* from String to Date and the Dates's  
069 \* <code>toString()</code> method used to convert from  
070 \* Date to String.</li>  
071 \* </ul>  
072 \*  
073 \* <p>  
074 \* The <b>Time Zone</b> to use with the date format can be specified  
075 \* using the <code>setTimeZone()</code> method.  
076 \*  
077 \* @version $Id$  
078 \* @since 1.8.0  
079 \*/  
080public abstract class DateTimeConverter extends AbstractConverter {  
081  
082 private String[] patterns;  
083 private String displayPatterns;  
084 private Locale locale;  
085 private TimeZone timeZone;  
086 private boolean useLocaleFormat;  
087  
088  
089 // ----------------------------------------------------------- Constructors  
090  
091 /\*\*  
092 \* Construct a Date/Time <i>Converter</i> that throws a  
093 \* <code>ConversionException</code> if an error occurs.  
094 \*/  
095 public DateTimeConverter() {  
096 super();  
097 }  
098  
099 /\*\*  
100 \* Construct a Date/Time <i>Converter</i> that returns a default  
101 \* value if an error occurs.  
102 \*  
103 \* @param defaultValue The default value to be returned  
104 \* if the value to be converted is missing or an error  
105 \* occurs converting the value.  
106 \*/  
107 public DateTimeConverter(final Object defaultValue) {  
108 super(defaultValue);  
109 }  
110  
111  
112 // --------------------------------------------------------- Public Methods  
113  
114 /\*\*  
115 \* Indicate whether conversion should use a format/pattern or not.  
116 \*  
117 \* @param useLocaleFormat <code>true</code> if the format  
118 \* for the locale should be used, otherwise <code>false</code>  
119 \*/  
120 public void setUseLocaleFormat(final boolean useLocaleFormat) {  
121 this.useLocaleFormat = useLocaleFormat;  
122 }  
123  
124 /\*\*  
125 \* Return the Time Zone to use when converting dates  
126 \* (or <code>null</code> if none specified.  
127 \*  
128 \* @return The Time Zone.  
129 \*/  
130 public TimeZone getTimeZone() {  
131 return timeZone;  
132 }  
133  
134 /\*\*  
135 \* Set the Time Zone to use when converting dates.  
136 \*  
137 \* @param timeZone The Time Zone.  
138 \*/  
139 public void setTimeZone(final TimeZone timeZone) {  
140 this.timeZone = timeZone;  
141 }  
142  
143 /\*\*  
144 \* Return the Locale for the <i>Converter</i>  
145 \* (or <code>null</code> if none specified).  
146 \*  
147 \* @return The locale to use for conversion  
148 \*/  
149 public Locale getLocale() {  
150 return locale;  
151 }  
152  
153 /\*\*  
154 \* Set the Locale for the <i>Converter</i>.  
155 \*  
156 \* @param locale The Locale.  
157 \*/  
158 public void setLocale(final Locale locale) {  
159 this.locale = locale;  
160 setUseLocaleFormat(true);  
161 }  
162  
163 /\*\*  
164 \* Set a date format pattern to use to convert  
165 \* dates to/from a <code>java.lang.String</code>.  
166 \*  
167 \* @see SimpleDateFormat  
168 \* @param pattern The format pattern.  
169 \*/  
170 public void setPattern(final String pattern) {  
171 setPatterns(new String[] {pattern});  
172 }  
173  
174 /\*\*  
175 \* Return the date format patterns used to convert  
176 \* dates to/from a <code>java.lang.String</code>  
177 \* (or <code>null</code> if none specified).  
178 \*  
179 \* @see SimpleDateFormat  
180 \* @return Array of format patterns.  
181 \*/  
182 public String[] getPatterns() {  
183 return patterns;  
184 }  
185  
186 /\*\*  
187 \* Set the date format patterns to use to convert  
188 \* dates to/from a <code>java.lang.String</code>.  
189 \*  
190 \* @see SimpleDateFormat  
191 \* @param patterns Array of format patterns.  
192 \*/  
193 public void setPatterns(final String[] patterns) {  
194 this.patterns = patterns;  
195 if (patterns != null && patterns.length > 1) {  
196 final StringBuilder buffer = new StringBuilder();  
197 for (int i = 0; i < patterns.length; i++) {  
198 if (i > 0) {  
199 buffer.append(", ");  
200 }  
201 buffer.append(patterns[i]);  
202 }  
203 displayPatterns = buffer.toString();  
204 }  
205 setUseLocaleFormat(true);  
206 }  
207  
208 // ------------------------------------------------------ Protected Methods  
209  
210 /\*\*  
211 \* Convert an input Date/Calendar object into a String.  
212 \* <p>  
213 \* <b>N.B.</b>If the converter has been configured to with  
214 \* one or more patterns (using <code>setPatterns()</code>), then  
215 \* the first pattern will be used to format the date into a String.  
216 \* Otherwise the default <code>DateFormat</code> for the default locale  
217 \* (and <i>style</i> if configured) will be used.  
218 \*  
219 \* @param value The input value to be converted  
220 \* @return the converted String value.  
221 \* @throws Throwable if an error occurs converting to a String  
222 \*/  
223 @Override  
224 protected String convertToString(final Object value) throws Throwable {  
225  
226 Date date = null;  
227 if (value instanceof Date) {  
228 date = (Date)value;  
229 } else if (value instanceof Calendar) {  
230 date = ((Calendar)value).getTime();  
231 } else if (value instanceof Long) {  
232 date = new Date(((Long)value).longValue());  
233 }  
234  
235 String result = null;  
236 if (useLocaleFormat && date != null) {  
237 DateFormat format = null;  
238 if (patterns != null && patterns.length > 0) {  
239 format = getFormat(patterns[0]);  
240 } else {  
241 format = getFormat(locale, timeZone);  
242 }  
243 logFormat("Formatting", format);  
244 result = format.format(date);  
245 if (log().isDebugEnabled()) {  
246 log().debug(" Converted to String using format '" + result + "'");  
247 }  
248 } else {  
249 result = value.toString();  
250 if (log().isDebugEnabled()) {  
251 log().debug(" Converted to String using toString() '" + result + "'");  
252 }  
253 }  
254 return result;  
255 }  
256  
257 /\*\*  
258 \* Convert the input object into a Date object of the  
259 \* specified type.  
260 \* <p>  
261 \* This method handles conversions between the following  
262 \* types:  
263 \* <ul>  
264 \* <li><code>java.util.Date</code></li>  
265 \* <li><code>java.util.Calendar</code></li>  
266 \* <li><code>java.sql.Date</code></li>  
267 \* <li><code>java.sql.Time</code></li>  
268 \* <li><code>java.sql.Timestamp</code></li>  
269 \* </ul>  
270 \*  
271 \* It also handles conversion from a <code>String</code> to  
272 \* any of the above types.  
273 \* <p>  
274 \*  
275 \* For <code>String</code> conversion, if the converter has been configured  
276 \* with one or more patterns (using <code>setPatterns()</code>), then  
277 \* the conversion is attempted with each of the specified patterns.  
278 \* Otherwise the default <code>DateFormat</code> for the default locale  
279 \* (and <i>style</i> if configured) will be used.  
280 \*  
281 \* @param <T> The desired target type of the conversion.  
282 \* @param targetType Data type to which this value should be converted.  
283 \* @param value The input value to be converted.  
284 \* @return The converted value.  
285 \* @throws Exception if conversion cannot be performed successfully  
286 \*/  
287 @Override  
288 protected <T> T convertToType(final Class<T> targetType, final Object value) throws Exception {  
289  
290 final Class<?> sourceType = value.getClass();  
291  
292 // Handle java.sql.Timestamp  
293 if (value instanceof java.sql.Timestamp) {  
294  
295 // ---------------------- JDK 1.3 Fix ----------------------  
296 // N.B. Prior to JDK 1.4 the Timestamp's getTime() method  
297 // didn't include the milliseconds. The following code  
298 // ensures it works consistently accross JDK versions  
299 final java.sql.Timestamp timestamp = (java.sql.Timestamp)value;  
300 long timeInMillis = ((timestamp.getTime() / 1000) \* 1000);  
301 timeInMillis += timestamp.getNanos() / 1000000;  
302 // ---------------------- JDK 1.3 Fix ----------------------  
303 return toDate(targetType, timeInMillis);  
304 }  
305  
306 // Handle Date (includes java.sql.Date & java.sql.Time)  
307 if (value instanceof Date) {  
308 final Date date = (Date)value;  
309 return toDate(targetType, date.getTime());  
310 }  
311  
312 // Handle Calendar  
313 if (value instanceof Calendar) {  
314 final Calendar calendar = (Calendar)value;  
315 return toDate(targetType, calendar.getTime().getTime());  
316 }  
317  
318 // Handle Long  
319 if (value instanceof Long) {  
320 final Long longObj = (Long)value;  
321 return toDate(targetType, longObj.longValue());  
322 }  
323  
324 // Convert all other types to String & handle  
325 final String stringValue = value.toString().trim();  
326 if (stringValue.length() == 0) {  
327 return handleMissing(targetType);  
328 }  
329  
330 // Parse the Date/Time  
331 if (useLocaleFormat) {  
332 Calendar calendar = null;  
333 if (patterns != null && patterns.length > 0) {  
334 calendar = parse(sourceType, targetType, stringValue);  
335 } else {  
336 final DateFormat format = getFormat(locale, timeZone);  
337 calendar = parse(sourceType, targetType, stringValue, format);  
338 }  
339 if (Calendar.class.isAssignableFrom(targetType)) {  
340 return targetType.cast(calendar);  
341 } else {  
342 return toDate(targetType, calendar.getTime().getTime());  
343 }  
344 }  
345  
346 // Default String conversion  
347 return toDate(targetType, stringValue);  
348  
349 }  
350  
351 /\*\*  
352 \* Convert a long value to the specified Date type for this  
353 \* <i>Converter</i>.  
354 \* <p>  
355 \*  
356 \* This method handles conversion to the following types:  
357 \* <ul>  
358 \* <li><code>java.util.Date</code></li>  
359 \* <li><code>java.util.Calendar</code></li>  
360 \* <li><code>java.sql.Date</code></li>  
361 \* <li><code>java.sql.Time</code></li>  
362 \* <li><code>java.sql.Timestamp</code></li>  
363 \* </ul>  
364 \*  
365 \* @param <T> The target type  
366 \* @param type The Date type to convert to  
367 \* @param value The long value to convert.  
368 \* @return The converted date value.  
369 \*/  
370 private <T> T toDate(final Class<T> type, final long value) {  
371  
372 // java.util.Date  
373 if (type.equals(Date.class)) {  
374 return type.cast(new Date(value));  
375 }  
376  
377 // java.sql.Date  
378 if (type.equals(java.sql.Date.class)) {  
379 return type.cast(new java.sql.Date(value));  
380 }  
381  
382 // java.sql.Time  
383 if (type.equals(java.sql.Time.class)) {  
384 return type.cast(new java.sql.Time(value));  
385 }  
386  
387 // java.sql.Timestamp  
388 if (type.equals(java.sql.Timestamp.class)) {  
389 return type.cast(new java.sql.Timestamp(value));  
390 }  
391  
392 // java.util.Calendar  
393 if (type.equals(Calendar.class)) {  
394 Calendar calendar = null;  
395 if (locale == null && timeZone == null) {  
396 calendar = Calendar.getInstance();  
397 } else if (locale == null) {  
398 calendar = Calendar.getInstance(timeZone);  
399 } else if (timeZone == null) {  
400 calendar = Calendar.getInstance(locale);  
401 } else {  
402 calendar = Calendar.getInstance(timeZone, locale);  
403 }  
404 calendar.setTime(new Date(value));  
405 calendar.setLenient(false);  
406 return type.cast(calendar);  
407 }  
408  
409 final String msg = toString(getClass()) + " cannot handle conversion to '"  
410 + toString(type) + "'";  
411 if (log().isWarnEnabled()) {  
412 log().warn(" " + msg);  
413 }  
414 throw new ConversionException(msg);  
415 }  
416  
417 /\*\*  
418 \* Default String to Date conversion.  
419 \* <p>  
420 \* This method handles conversion from a String to the following types:  
421 \* <ul>  
422 \* <li><code>java.sql.Date</code></li>  
423 \* <li><code>java.sql.Time</code></li>  
424 \* <li><code>java.sql.Timestamp</code></li>  
425 \* </ul>  
426 \* <p>  
427 \* <strong>N.B.</strong> No default String conversion  
428 \* mechanism is provided for <code>java.util.Date</code>  
429 \* and <code>java.util.Calendar</code> type.  
430 \*  
431 \* @param <T> The target type  
432 \* @param type The date type to convert to  
433 \* @param value The String value to convert.  
434 \* @return The converted Number value.  
435 \*/  
436 private <T> T toDate(final Class<T> type, final String value) {  
437 // java.sql.Date  
438 if (type.equals(java.sql.Date.class)) {  
439 try {  
440 return type.cast(java.sql.Date.valueOf(value));  
441 } catch (final IllegalArgumentException e) {  
442 throw new ConversionException(  
443 "String must be in JDBC format [yyyy-MM-dd] to create a java.sql.Date");  
444 }  
445 }  
446  
447 // java.sql.Time  
448 if (type.equals(java.sql.Time.class)) {  
449 try {  
450 return type.cast(java.sql.Time.valueOf(value));  
451 } catch (final IllegalArgumentException e) {  
452 throw new ConversionException(  
453 "String must be in JDBC format [HH:mm:ss] to create a java.sql.Time");  
454 }  
455 }  
456  
457 // java.sql.Timestamp  
458 if (type.equals(java.sql.Timestamp.class)) {  
459 try {  
460 return type.cast(java.sql.Timestamp.valueOf(value));  
461 } catch (final IllegalArgumentException e) {  
462 throw new ConversionException(  
463 "String must be in JDBC format [yyyy-MM-dd HH:mm:ss.fffffffff] " +  
464 "to create a java.sql.Timestamp");  
465 }  
466 }  
467  
468 final String msg = toString(getClass()) + " does not support default String to '"  
469 + toString(type) + "' conversion.";  
470 if (log().isWarnEnabled()) {  
471 log().warn(" " + msg);  
472 log().warn(" (N.B. Re-configure Converter or use alternative implementation)");  
473 }  
474 throw new ConversionException(msg);  
475 }  
476  
477 /\*\*  
478 \* Return a <code>DateFormat<code> for the Locale.  
479 \* @param locale The Locale to create the Format with (may be null)  
480 \* @param timeZone The Time Zone create the Format with (may be null)  
481 \*  
482 \* @return A Date Format.  
483 \*/  
484 protected DateFormat getFormat(final Locale locale, final TimeZone timeZone) {  
485 DateFormat format = null;  
486 if (locale == null) {  
487 format = DateFormat.getDateInstance(DateFormat.SHORT);  
488 } else {  
489 format = DateFormat.getDateInstance(DateFormat.SHORT, locale);  
490 }  
491 if (timeZone != null) {  
492 format.setTimeZone(timeZone);  
493 }  
494 return format;  
495 }  
496  
497 /\*\*  
498 \* Create a date format for the specified pattern.  
499 \*  
500 \* @param pattern The date pattern  
501 \* @return The DateFormat  
502 \*/  
503 private DateFormat getFormat(final String pattern) {  
504 final DateFormat format = new SimpleDateFormat(pattern);  
505 if (timeZone != null) {  
506 format.setTimeZone(timeZone);  
507 }  
508 return format;  
509 }  
510  
511 /\*\*  
512 \* Parse a String date value using the set of patterns.  
513 \*  
514 \* @param sourceType The type of the value being converted  
515 \* @param targetType The type to convert the value to.  
516 \* @param value The String date value.  
517 \*  
518 \* @return The converted Date object.  
519 \* @throws Exception if an error occurs parsing the date.  
520 \*/  
521 private Calendar parse(final Class<?> sourceType, final Class<?> targetType, final String value) throws Exception {  
522 Exception firstEx = null;  
523 for (String pattern : patterns) {  
524 try {  
525 final DateFormat format = getFormat(pattern);  
526 final Calendar calendar = parse(sourceType, targetType, value, format);  
527 return calendar;  
528 } catch (final Exception ex) {  
529 if (firstEx == null) {  
530 firstEx = ex;  
531 }  
532 }  
533 }  
534 if (patterns.length > 1) {  
535 throw new ConversionException("Error converting '" + toString(sourceType) + "' to '" + toString(targetType)  
536 + "' using patterns '" + displayPatterns + "'");  
537 } else {  
538 throw firstEx;  
539 }  
540 }  
541  
542 /\*\*  
543 \* Parse a String into a <code>Calendar</code> object  
544 \* using the specified <code>DateFormat</code>.  
545 \*  
546 \* @param sourceType The type of the value being converted  
547 \* @param targetType The type to convert the value to  
548 \* @param value The String date value.  
549 \* @param format The DateFormat to parse the String value.  
550 \*  
551 \* @return The converted Calendar object.  
552 \* @throws ConversionException if the String cannot be converted.  
553 \*/  
554 private Calendar parse(final Class<?> sourceType, final Class<?> targetType, final String value, final DateFormat format) {  
555 logFormat("Parsing", format);  
556 format.setLenient(false);  
557 final ParsePosition pos = new ParsePosition(0);  
558 final Date parsedDate = format.parse(value, pos); // ignore the result (use the Calendar)  
559 if (pos.getErrorIndex() >= 0 || pos.getIndex() != value.length() || parsedDate == null) {  
560 String msg = "Error converting '" + toString(sourceType) + "' to '" + toString(targetType) + "'";  
561 if (format instanceof SimpleDateFormat) {  
562 msg += " using pattern '" + ((SimpleDateFormat)format).toPattern() + "'";  
563 }  
564 if (log().isDebugEnabled()) {  
565 log().debug(" " + msg);  
566 }  
567 throw new ConversionException(msg);  
568 }  
569 final Calendar calendar = format.getCalendar();  
570 return calendar;  
571 }  
572  
573 /\*\*  
574 \* Provide a String representation of this date/time converter.  
575 \*  
576 \* @return A String representation of this date/time converter  
577 \*/  
578 @Override  
579 public String toString() {  
580 final StringBuilder buffer = new StringBuilder();  
581 buffer.append(toString(getClass()));  
582 buffer.append("[UseDefault=");  
583 buffer.append(isUseDefault());  
584 buffer.append(", UseLocaleFormat=");  
585 buffer.append(useLocaleFormat);  
586 if (displayPatterns != null) {  
587 buffer.append(", Patterns={");  
588 buffer.append(displayPatterns);  
589 buffer.append('}');  
590 }  
591 if (locale != null) {  
592 buffer.append(", Locale=");  
593 buffer.append(locale);  
594 }  
595 if (timeZone != null) {  
596 buffer.append(", TimeZone=");  
597 buffer.append(timeZone);  
598 }  
599 buffer.append(']');  
600 return buffer.toString();  
601 }  
602  
603 /\*\*  
604 \* Log the <code>DateFormat<code> creation.  
605 \* @param action The action the format is being used for  
606 \* @param format The Date format  
607 \*/  
608 private void logFormat(final String action, final DateFormat format) {  
609 if (log().isDebugEnabled()) {  
610 final StringBuilder buffer = new StringBuilder(45);  
611 buffer.append(" ");  
612 buffer.append(action);  
613 buffer.append(" with Format");  
614 if (format instanceof SimpleDateFormat) {  
615 buffer.append("[");  
616 buffer.append(((SimpleDateFormat)format).toPattern());  
617 buffer.append("]");  
618 }  
619 buffer.append(" for ");  
620 if (locale == null) {  
621 buffer.append("default locale");  
622 } else {  
623 buffer.append("locale[");  
624 buffer.append(locale);  
625 buffer.append("]");  
626 }  
627 if (timeZone != null) {  
628 buffer.append(", TimeZone[");  
629 buffer.append(timeZone);  
630 buffer.append("]");  
631 }  
632 log().debug(buffer.toString());  
633 }  
634 }  
635}