URI Template Standard

Version 1

October 21st, 2015

The text for this standard is available at http://tsds.org/uri_templates_v1. Discussion of the standard and links to additional resources are available at http://tsds.org/uri_templates.

1. Introduction

A URI template is a string that contains special fields. The special fields allow a time range to be associated with each URI in the collection. This specification describes the special fields needed to capture the file and directory naming conventions used for data and images in heliophysics science data archives, but could describe any set of URIs containing date elements that follow a broad range of conventions. Note that the templates themselves are not URIs, but instead they simply describe how URIs are generated or interpreted.

In order to be describable by a URI template, a data collection must have files and directories that are named such that the date elements in the names unambiguously identify the time range covered by each file. The templates allow an association to be made between an individual URI and a time range. Conceptually, this is similar to indexing data files by time range.

Consider this collection of daily files with the year, month, and month day in the file names:

```
data_2015_02_23.dat
data_2015_02_24.dat
data_2015_02_25.dat
data_2015_02_26.dat
```

The URI template would be data_$Y_$m_$d.dat.

The intended use for the templates is within applications or services that can locate files for various data sources given a time range (Figure 1).

The template specification itself does not indicate how a service goes about finding files, but it is designed to support multiple ways of associating a time range with a URI (and thus finding out which URIs are actually present in the user's time range). Therefore the specification supports two key capabilities:

1. GENERATION: Use a URI template and a time range to create a list of possible URIs for that time range.
2. PARSING: Given a list of URIs and a URI template, extract a time range from each URI, effectively identifying the URIs that fall within the given time range.

Generation and Parsing Explained

The templates are designed to provide enough information about the date-related elements in the file names that, given a template and a time range, all the possible URIs for the given time range could be

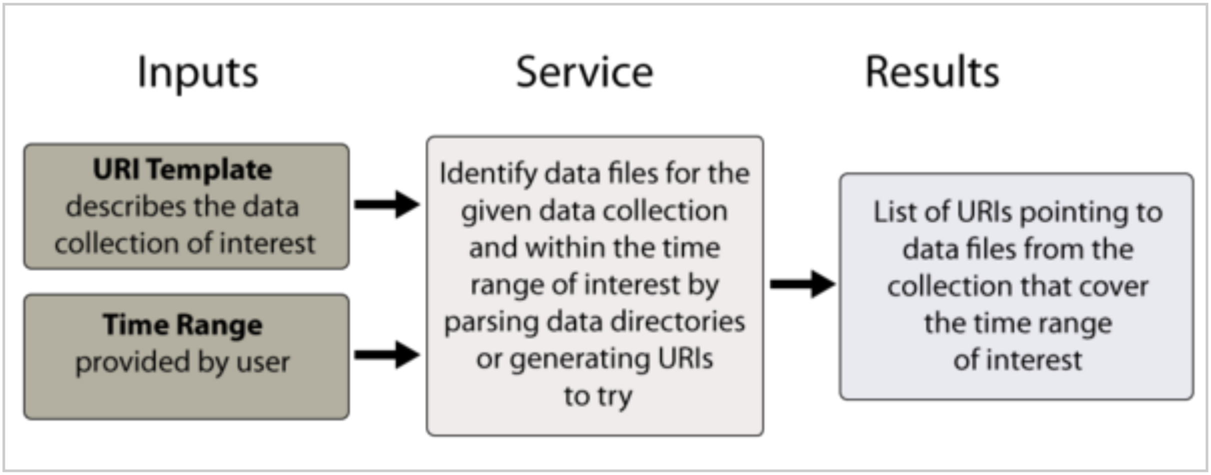
Inputs
Service
Results
URI Template describes the data collection of interest
Time Range provided by user
Identify data files for the given data collection and within the time range of interest by parsing data directories or generating URIs to try
List of URIs pointing to data files from the collection that cover the time range of interest

Figure 1: The templates are designed to describe the file names in data collections in order to support generic file discovery mechanisms.

generated. Note that this **generation** capability provides a list of *possible* URIs, and whether or not a service does a check to see if all these URIs are indeed present is up to the service. The template specification does not constrain the service behavior, but is focused on capturing enough information about the filenames to enable the generation process.

The templates are also designed to support a **parsing** capability. In this scenario, the service can use a template to extract a time range from an individual URI that the service has already obtained. Assuming that the service could obtain a bulk listing of many URIs from a data collection, this parse-to-get-time-range operation could be performed on all discovered URIs in order to determine which URIs are relevant for a particular time range. Figure 2 illustrates both the generation and parsing activities. The figure emphasizes that the focus of the parsing capability is the interpretation of a single URI. How a service uses this ability is up to the service.

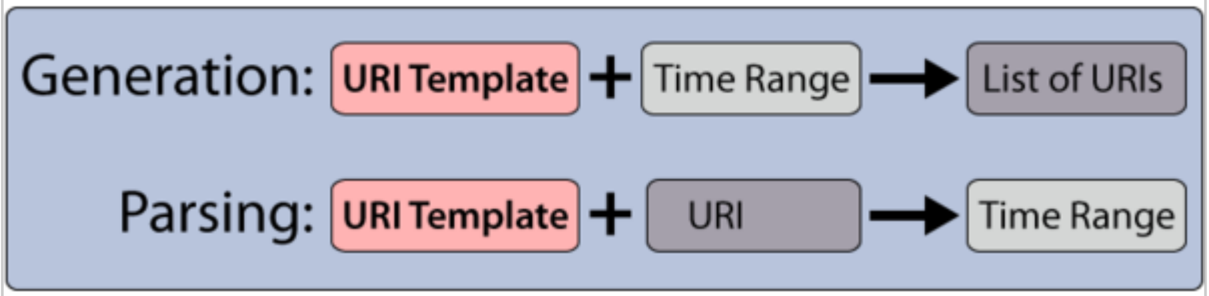
Generation: URI Template + Time Range → List of URIs
Parsing: URI Template + URI → Time Range

Figure 2: The URI Template Specification captures enough information about file names to support the both the generation of URIs given a time range and the parsing of URIs to obtain a time range for each one.

Both parsing and generation capabilities are important, because each capability alone is insufficient for some cases. For example, if a listing of available files cannot be obtained (perhaps the web server does not allow directory listings), then the generation capability is the only way to get started -- the service must first generate a list of potential filenames, and then possibly do an existence test done for each generated URI. But sometimes the generation of URIs is not possible because the file names contain characters that cannot be described or known apriori. Unpredictable file name components include elements like mission elapsed time integers, unusually formatted version numbers, or even strange processing ids that end up looking like random numbers. In these cases, the URIs must be obtained (from some type of listing mechanism available for that URI), and then the template can be used to parse each URI to obtain a time range. The random characters are skipped over in the parsing process since they do not affect the time. Note that if file names cannot be generated AND a listing is unavailable, then of course no access is possible.

A service might decide to use only the parsing capability. In this case, it would only work with data collections for which a listing is available, but such a service could still fully support the URI template specification. If a service only used the generation capability, there are a few template features that such a service could not support. The template codes that preclude URI generation are indicated in the discussion further below.

The following examples illustrate the generation and parsing capabilities.

1.1. Generation Example

Suppose there is a data collection where each file contains one year of information and consists of the following URIs:

```
http://example.com/data_1998.dat
http://example.com/data_1999.dat
http://example.com/data_2000.dat
http://example.com/data_2001.dat
http://example.com/data_2002.dat
http://example.com/data_2003.dat
http://example.com/data_2004.dat
http://example.com/data_2005.dat
```

Given a URI template of http://example.com/data_$Y.dat and a date range of 2001-03-22 to 2004-08-18, it would be possible to generate this list of URIs:

```
http://example.com/data_2001.dat
http://example.com/data_2002.dat
http://example.com/data_2003.dat
http://example.com/data_2004.dat
```

There is enough information in this example template for a service to generate just the URIs that fall in the time range of interest. And because each URI covers a year of data, there is little need to check if the generated URIs exist, since a year-long data gap seems unlikely. But for files of shorter duration (e.g., daily files), gaps are more probable, and so the list of generated URIs will likely contain more URIs than actually exist. The template specification does not indicate how a service would handle this situation. The URI generation aspect of the URI template specification requires that all possible URIs be generated, but it does not require services to check that they exist. This is outside the scope of the URI template specification, and is left as a (potentially important!) service implementation detail.

1.2. Parsing Example

If a filename contains unpredictable characters, the URI template may specify this using a wildcard character, and then it is impossible to generate URIs for this collection of files. But the filenames can still be parsed in order to determine the time range covered by the file. Consider these files:

```
http://example.com/2015/data_2015_361_id79242.cdf
http://example.com/2015/data_2015_364_id49304.cdf
http://example.com/2015/data_2015_365_id93039.cdf
http://example.com/2016/data_2016_001_id54040.cdf
http://example.com/2016/data_2016_002_id13487.cdf
```

The template http://example.com/$Y/data_$Y_j_idx.cdf would be used to describe these files, where the $x code indicates the wildcard element in the filename. Given this template, a service would be expected to obtain a listing of available URIs, and parse them with knowledge from the template to see which URIs fall inside a user-specified time range. For the time range 2015-362T00:00:00 to 2016-001T00:00:00, the following files would be included:

```
http://example.com/2015/data_2015_364_id49304.cdf
http://example.com/2015/data_2015_365_id93039.cdf
```

1.3. Time Range Boundary Issues

In the Parsing Example above, note that no files from 2016 are included in the results list. This is because the time range provided has an overlap of zero seconds with data in 2016, but this type of boundary condition may vary from service to service, since the URI template spec does not specify how services treat user input values and boundary values. It is suggested that services treat the time boundaries of the data files as slightly fuzzy, so it might make sense to add some safety padding to a user's request, so that slightly more files than requested are returned. But again, this is a service specific detail. The URI template spec only indicates what the time range is for each URI in the dataset.

2. Basic Syntax

URI templates are text containing embedded date fields. Each date field is indicated using a leading marker character, namely a dollar sign $. If a URI contains a literal $ (as would happen if the $ character is present in the actual filename), this can be captured in the template string by inserting $$.

The use of the $ marker character was motivated by a desire to distinguish this standard from other similar date specifications, many of which use % as a marker character. (Consider for example, the ptime function common in many C libraries.) By using a $, the URI templates will look different, and this hopefully serves as a reminder to users that the meanings of date fields in this standard are different than those in other standards. Also, the % character in URIs indicates the start of an escape sequence (a space is escaped as %20, for example). Therefore the use of $ in template strings prevents further visual conflict with the escape sequences.

Following the marker character is a date field, which consists of a code with optional modifiers, and the modifiers may have values. A code will always start with a single letter (upper or lower case) followed by zero or more letters, digits or underscores. Modifier names follow the same convention. Most codes are single characters, and most modifiers are short words.

Some codes allow or require modifiers that change the characteristics or behavior of the code. Modifiers are separated from the field code by a semicolon, e.g., $(Y;begin) and some codes allow more than one modifier, with each modifier separated from the next using a semicolon. A modifier may have an associated value following an equals sign.

The modifier values are restricted to be one of:

1. a signed integer
2. an integer followed by one of these single letter date codes: Y, m, d, H, M, S
3. any alpha numeric string, i.e., one or more numbers, letters or underscores and also : and . and - (ISO8601 strings meet this criteria)
4. a single quoted string

If the field code is longer than one character, or if modifiers are present, the entire field must be surrounded with parentheses. Parentheses may be used for single-character codes, but this is not required.

This chart illustrates the basic syntax for date fields of varying complexity:

Full Date Field:	$(code;modifier=value; mod2=val2)	multi-character code with two modifiers, each with a different value
Simpler Possibilities:	$(code;modifier=value)	just one modifier with a value
	$(code;modifier)	one modifier with no value
	$(code)	multi-character code with no modifier - parentheses are required
	$Y or $(Y)	parentheses are optional for a single-character code

The simplest field is just a marker character followed by a single-character code, as in $Y, where the uppercase 'Y' is an actual code that denotes a field to be interpreted as a 4-digit year. For any single-character code, parentheses are not required, but can be used, as in $(Y).

The most commonly used codes are (see the next section for complete list):

- $Y zero-padded four digit year
- $j zero-padded three digit day of year
- $m zero-padded two digit month
- $d zero-padded two digit day
- $H zero-padded two digit hour
- $M zero-padded two digit minute
- $S zero-padded two digit second

Consider this example template: data_$Y-$(m;pad=none)-$d.dat. The file names will have year $Y, month number $m, and day-of-month $d elements. The modifier on the month field indicates that the month numbers are not zero-padded. The day-of-month numbers are zero-padded, because this is the default behavior of the $d code. File names generated from this template would look like this:

```
data_2015-9-28.txt  (notice that the month 9 value is not zero-padded)
data_2015-9-29.txt
data_2015-9-30.txt
data_2015-10-01.txt  (notice that the day number is zero-padded)
data_2015-10-02.txt
```

Here are a few more examples of possible templates:

data_$Y_$j.txt	four digit year and zero-padded three-digit day of year
data_$y_$(j;pad=none).txt	two-digit year and un-padded day of year
example.com/$Y /data_$Y_$m_$d.txt	data in yearly directories with year, month, and month-day in the filename

3. Full List of Codes

As mentioned in the introduction, a few of the codes will preclude a template from being used to generate URIs, namely x and v. This restriction is also highlighted at the beginning of the description section for these codes.

We take an aside here to point out that the full set of date codes in this spec allows file names that are potentially very strange or obscure. This specification is *not* meant to define a convention for naming files in future data collections. The spec was designed to capture as many of file naming schemes that are found "in the wild," and many of these represent poor choices that resulted in directory or file names that are overly compact, confusing, or difficult to process in an automated way. The basic principle is to keep filenames intuitive and uniform, but this spec is not the place to look for recommended naming conventions. A complete set of recommended naming conventions will be described in a separate document. **need reference here**

3.1. Y, y, m, b, d, j, H, M, and S

The full description for each of these codes is included below, but they have many common modifiers that are described in this section.

Common modifiers

- begin - (optional) indicates that this field belongs to the begin time (use if, for example, a filename has begin and end times in it). It is assumed that times refer to the beginning of the interval, and this should be used when end time fields precede the begin field.
- end - (optional) indicates that this field belongs to the end time (if a filename has begin and end times in it). All fields after this are considered to be part of the end time.
- delta=<integer with optional unit> - (optional) specifies the duration or span covered by this file or the increment of the code to get the next filename.
- sparse - (optional) indicates that many of the generated URIs may not actually exist; this is likely to be needed for the smaller time values, such as seconds $subsec, where it is unlikely that there is a new data file for every millisecond
- phasestart=<ISO8601 date> - (optional except when delta in year or day field) when to start the delta. The default is the smallest value that the field may take in which delta is used (e.g., 0 for H, 1 for j).
- shift=<integer with optional unit> - (optional) if a date value in the filename is incorrect or does not represent the actual date range covered by the file content, this keyword can adjust the time value by a constant offset; this is somewhat complicated and is explained further below **verify this is ok -- used to be just:** shift the generated end date by this amount

The units of delta and shift default to the unit of the code that it modifies. Units may be specified for the codes $m $b $d $j $H $M $S $subsec, and the allowed units are Y, m, d, H, M, or S.

delta example

The duration of a file is assumed to be one unit of the smallest date code in the template. You can change this with the delta modifier. Suppose a collection of files contains data in 6 month chunks, and also each file has a name with year and month values.

```
data_2009_01.dat
data_2009_07.dat
data_2010_01.dat
data_2010_07.dat
```

In order to indicate this 6 month duration for each file, the template would need to be

data_$Y_$(m;delta=6).dat (Note: you can specify the units as months data_$Y_$(m;delta=6m).dat but this is redundant since the units default to months in this case.)

Leaving out the delta data_$Y_$m.dat would indicate monthly files (i.e, files of one month duration).

units example

Modifiers that take a date value (such as the shift and delta modifiers) default to having units of the same type as the date code which they are modifying. So the 12 in $(H;delta=12) refers to 12 hours. Appending a single letter to the value will change the units. So this template $(H;delta=1j) indicates that the filename has an hour field, but the duration covered by the file is one day.

As another example, consider a set of files labeled down to the day, but having data covering an entire month. If a file is created every hour with names

```
data_2009-01-01T00:00:00.txt
data_2009-02-01T00:00:00.txt
data_2009-03-01T00:00:00.txt
data_2009-04-01T00:00:00.txt
```

Any of these templates would describe these files: data_$Y-$m-01.txt data_$Y-$m-$(d;delta=1m).txt data_$Y-$(m;delta=1)-$d.txt

shift example

Sometimes file names contain date elements for a begin time and and end time. Often, the interpretation of the date elements for the end time may be different than for the begin time. As an example, consider a file with the name data_2005132_2005145.txt. The start time of the file is almost certainly the beginning of day 132 in 2005, but the end time is ambiguous. Is it the start or end of day 145? If it is the end of the day, then the shift modifier is needed to communicate that the end time is shifted from the plain date value represented in the file name. The template for files like this one would be: data_Yj_$(Y;end)($j;shift=1).txt. Note that the end modifier is sticky in that once used, it applies to all subsequent fields.

3.2. Y

Numeric four-digit year.

3.3. y

Numeric two-digit year. Modifier:

- start=<4 digit year> - (optional); default=1950; The 4-digit year represents the first of the 100 years that can be represented, e.g., 50 = 1950, 99 = 1999, 00 = 2000, 01 = 2001, ..., 49 = 2049.

3.4. m

Month number, starting with 01. Modifier:

- pad=zero|none|underscore|space - (optional); default=zero

3.5. b

Month name or abbreviated month name. Modifiers:

- fmt=abbrev|full - (optional); default=abbrev
- case=lc|uc|cap - (optional); default=lc

Possibilities:

- $(b;fmt=abbrev;case=lc): jan, feb, ...
- $(b;fmt=abbrev;case=uc): JAN, FEB, ...
- $(b;fmt=abbrev;case=cap): Jan, Feb, ...
- $(b;fmt=full;case=lc): january, february, ...
- $(b;fmt=full;case=cap): JANUARY, FEBRUARY, ...
- $(b;fmt=full;case=uc): January, February, ...

3.6. d

Day of month number, starting with 01. Modifier:

- pad=zero|none|underscore|space - (optional); default=zero

3.7. j

Day of year, starting with 001. Modifier:

- pad=zero|none|underscore|space - (optional); default=zero

Examples:

pad=zero: 2001-001.txt, 2001-002.txt

pad=none: 2001-1.txt, 2001-2.txt

pad=underscore: 2001__1.txt, 2001__2.txt pad=space: 2001 1.txt, 2001 2.txt

3.8. H

Hour of day, from 00 to 23. Modifier:

- pad=zero|none|underscore|space - (optional); default=zero

3.9. M

Integer minute, from 00 to 59. Modifier:

- pad=zero|none|underscore|space - (optional); default=zero

3.10. S

Seconds, either integer or decimal, 00 to 60 (60 is allowed for leap seconds). Modifier:

- pad=zero|none|underscore|space - (optional); default=zero

Generation example: Each file contains 5 seconds of data. Template 2001-01-01-0000$(S;delta=5).txt

```
2001-01-01-000000.txt
2001-01-01-000005.txt
2001-01-01-000010.txt
2001-01-01-000015.txt
2001-01-01-000020.txt
```

Parsing example: Images are taken roughly every five seconds. Template is 2001-01-01-0000$S.txt and list is

```
2001-01-01-000000.png
2001-01-01-000005.png
2001-01-01-000011.png
2001-01-01-000015.png
2001-01-01-000019.png
```

software infers timestamps of each image.

Parsing example: When something interesting happens, data are logged for 10 seconds. Template is 2001-01-01-0000$(S;delta=5).txt

```
2001-01-01-000000.txt
2001-01-01-000031.txt
2001-01-01-000045.txt
```

3.11. periodic

A periodic integer value such as a completely regular orbit number. Modifiers:

- offset=<the integer value of the period beginning at the start time> - **required**
- start=<ISO8601 time value> - **required**
- period=<decimal number followed by basic date code> - **required**; This is assumed to be UTC

date elements which may include leap seconds

Example: Bartels rotation number

```
data_bartels_$(periodic;offset=2285;start=2000-346T00:00;period=27d).txt
```

sample filenames for time range of 2001-01-01 through 2001-03-05

```
data_bartels_2285.txt
data_bartels_2286.txt
data_bartels_2287.txt
data_bartels_2288.txt
```

3.12. enum

URI templates with $(enum) can be used for generation or parsing.

Modifiers:

- values=<value1>,<value2>,<value3>,... - **required**; One or more values separated by a comma character.
- id=<string identifier for values> - (optional); A string that gives a name to the list of values. Does not affect generation or parsing results.

Example: Data from two instruments, A and B, are saved in daily files.

```
2000-01-01-A.dat
2000-01-01-B.dat
2000-01-02-A.dat
2000-01-02-B.dat
```

Template: $Y-$m-$d-$(enum;values=A,B).

To communicate the meaning of the values, the optional id modifier may be used: $Y-$m-$d-$(enum;values=A,B;id=InstrumentName). This field merely helps identify theThere is no change in the generated list.

3.13. hrinterval

URI templates with $(hrinterval) can be used for generation or parsing.

Represents a multi-hour Interval within a day. The name of each period is separated by commas, and the duration of each period defaults to 24 hours divided by the number of periods present. It has these modifiers:

- values=<first name>,<second name>,<third name>,... - **required**
- duration=<number of hours> - (optional); default=24 divided by the number of names

Note that because this date code is more than one letter, and because it has a required modifier, it must be surrounded with parentheses.

Data collections whose files cover intervals that are less than 24 hours are not unusual, and this date code is somewhat more complex than others, so more explanation is provided here using several examples.

File names that describe a multi-hour time interval may contain a single date element representing that multi-hour block of time. For example, the last integer in these filenames (just after the "S") represents a 6 hour interval:

```
http://example.com/CL_QL_ORB2_2011_12_01_S01.gif
http://example.com/CL_QL_ORB2_2011_12_01_S02.gif
http://example.com/CL_QL_ORB2_2011_12_01_S03.gif
http://example.com/CL_QL_ORB2_2011_12_01_S04.gif
http://example.com/CL_QL_ORB2_2011_12_02_S01.gif
http://example.com/CL_QL_ORB2_2011_12_02_S02.gif
http://example.com/CL_QL_ORB2_2011_12_02_S03.gif
http://example.com/CL_QL_ORB2_2011_12_02_S04.gif
```

(This is based on a real-world example.)

Here the labels "01", "02", "03" and "04" represent 6 hour intervals from 0 to 6 hours, 6 to 12 hours, 12 to 18 hours, and 18 to 24 hours respectively. The template for this data would be http://example.com/CL_QL_ORB2_$Y_$m_$d_S$(hrinterval;values=01,02,03,04).gif The duration of each file is assumed to be 6 hours because there are 4 named intervals, and (24 hrs)/4 is 6 hours.

The date element representing the multi-hour interval need not be numeric. In the following filenames, 'a' indicates hours 0-12 and 'b' represents hours 12-24:

```
http://example.com/summary_images/2002/Nov/DOY314/sdpk2_02314a.gif
http://example.com/summary_images/2002/Nov/DOY314/sdpk2_02314b.gif
http://example.com/summary_images/2002/Nov/DOY315/sdpk2_02315a.gif
http://example.com/summary_images/2002/Nov/DOY315/sdpk2_02315b.gif
```

(Also a real-world example.)

For these files, the template would be: http://example.com/summary_images/$Y/$(b;case=cap)/DOY$j/sdpk2_$y$j$(hrinterval;values=a,b).gif

3.14. subsec

URI templates with $(subsec) can be used for generation or parsing.

The fractional part of the seconds. Modifiers:

- places=<how many places right of the decimal> - **required**
- pad=zero|none|underscore|space - (optional) default = zero

Note that you are likely going to want to use the sparse modifier with this field.

Example: Filename: data_2015-244T17:45:03.123456 Template: data_$Y-$jT$H:$M:$S.$(subsec;places=6).

Note that if this template was used for generation, it would create one filename for every millisecond, which is likely to be an overwhelming number of URIs, most of which are not likely to exist. It is recommended to use the sparse modifier with this field, as in data_$Y-$jT$H:$M:$S.$(subsec;places=6;sparse). This modifier is not meant to prevent generation, but to indicate that many of the generated URIs may not exist. Another way to handle this kind of filename might be to give up on the sub-seconds accuracy altogether, and just use $x for the sub-seconds: data_$Y-$jT$H:$M:$S.$x. The use of $x does prevent URIs from being generated.

3.15. v

URI templates with $v cannot be used for generation.

A version number (of the data) such that only files with the highest version number value should be accepted. Modifiers:

- type=sep|float|int|alpha - (optional); default=sep
- separator=<char> - (optional); default=.
- ge=<str> - (optional); Only look for the highest version among files higher than this version
- lt=<str> - (optional); Only look for the highest version among files less than or equal to this version

Other allowed types are int and alpha (for alphanumeric) where the sort order is the ASCII byte sort order of these types.

Another common version mechanism is the so-called 'dot notation' where the version string is a set of dot-separated numbers, as in 2.0 or 1.0.0. This is referred to as type=sep. With dot-separated notation, you could also get versions like 1.1.13, where this is a higher version than 1.1.7, even though a string sort would put the 1.1.13 first. If the separator is something other than a dot, the modifier separator can be used.

Finally, for restricting the files found to be of at least a certain version (or less than a certain version), the ge or lt modifiers can be used. This would be used when you have a dataset where one set of files (say version 1.x) requires reader A, and version 2.x requires reader B. Then reader A could be paired with a finder that only finds versions that are lt=2.0 and then ge=2.0 would be used for reader B.

Note that the comparison for floats is different than the comparison for sep. (For sep, 3.14 is greater than 3.3; but for float this is not the case.)

3.16. x

URI templates with $x cannot be used for generation.

Wildcard elements are variable elements in a filename that cannot be predicted or known. Thus URIs with this date code in them cannot be fully generated and will require some parsing to see if they match the template.

Modifiers:

- name=<identifier giving a name to this part of the filename> - (optional)
- regex='<pattern string inside single quotes>' - (optional); May not be used with the len modifier
- len=<integer number of characters in the field> - (optional); May not be used with the regex modifier

Example

Consider a dataset that has yearly directories and then filenames with a single digit year in the filename and also a three digit day of year.

```
http://example.com/data/1984/mag_5364.txt
http://example.com/data/1984/mag_5365.txt
http://example.com/data/1984/mag_5366.txt
http://example.com/data/1985/mag_5001.txt
http://example.com/data/1985/mag_5002.txt
http://example.com/data/1985/mag_5003.txt
http://example.com/data/1985/mag_5004.txt
```

There is no date code for a one-digit year, so this can be represented with a wildcard, as in http://example.com/data/$Y/mag_$x$j.txt

The wildcard could be made more specific using the regex modifier: http://example.com/data/$Y /mag_$(x;regex='\d')$j.txt (Recall that \d matches a single digit, 0-9.)

Also, consider these files (from a real-world example) that have three kinds of id codes in the filenames. The first two are 5 digit number and the other is a string that is always one of "in" or "on" or "is" or "os".

```
http://example.com/data/2008/2008_03_04/46564/fa_k0_ees_46564_in.gif
http://example.com/data/2008/2008_03_04/46564/fa_k0_ees_46564_on.gif
http://example.com/data/2008/2008_03_04/46565/fa_k0_ees_46565_in.gif
http://example.com/data/2008/2008_03_04/46565/fa_k0_ees_46565_on.gif
http://example.com/data/2008/2008_03_04/46566/fa_k0_ees_46566_in.gif
http://example.com/data/2008/2008_03_04/46566/fa_k0_ees_46566_on.gif
```

The following template ignores the unpredictable elements in the file names: http://example.com /data/$Y/$Y_$m_$d/$x/fa_k0_dcf_$x_$x.gif

If you want to make sure that the wildcards match more closely (to prevent potential unrelated files from matching) you could use the regex modifier to indicate a regular expression pattern for the wildcard elements. http://example.com/data/$Y/$Y_$m_$d/$(x;regexp='\d{1,5}')/fa_k0_dcf_$(x; regexp='\d{1,5}')_$(x;regexp='(?:in|on|is|os)').gif

3.17. ver

Indicates the version of the URI template specification to use for interpreting the template. If not given, $(ver;n=1) is assumed. Modifier:

- n=<version number of the template spec> - **required**

Example: $(ver;n=1)

4. Time Range Rules

There are two types of filenames that are supported by the standard: those with just a begin time or those that include both the begin time and end time in the filename. The rules for finding the time range for both cases are given below.

4.1. Only a begin time in filename

The default time width is one unit of the smallest time code in the URI. For example, data_$Y$m$d.txt is assumed to have a duration of 1 day. If the timerange is 2000-01-01/2000-01-03, the file list is

```
data_2000_01_01.txt
data_2000_01_02.txt
```

If the time width is different than the default, it can be manually specified using the delta modifier. When URIs are parsed, the duration will be set to the time duration given by the value of the delta modifier. For example, the template

```
data_$Y_$m_$d_$(H;delta=6).txt
```

could be used when filenames have only hour values of 00, 06, 12, 18. If the timerange is 2000-01-01/2000-01-03, the file list is

```
data_2000_01_01_00.txt
data_2000_01_01_06.txt
data_2000_01_01_12.txt
data_2000_01_01_18.txt
data_2000_01_02_00.txt
data_2000_01_02_06.txt
data_2000_01_02_12.txt
data_2000_01_02_18.txt
```

Note that the delta applies to the entire begin time. Also, the units of the delta default to those of the time field on which the delta is being applied, although other units may be used.

For example, in the template

```
data_$(Y;delta=6H)_$m_$d_$H.txt
```

the 6H for the delta applies to the entire start date, so only hours 00, 06, 12, 18 are present. If the timerange is 2000-01-01/2000-01-03, the file list is the same as that in the previous example:

```
data_2000_01_01_00.txt
data_2000_01_01_06.txt
data_2000_01_01_12.txt
data_2000_01_01_18.txt
data_2000_01_02_00.txt
data_2000_01_02_06.txt
data_2000_01_02_12.txt
data_2000_01_02_18.txt
```

The hour, minute, second, and also the month date fields have a natural defaults for their starting points (hours start at a day boundary, minutes at an hour boundary, seconds start at the minute boundary, and months start at the year boundary), so that the start of the phase is always clear when interpreting delta modifiers for these date elements.

But for year and day fields, the starting point of the delta is arbitrary and therefore must be stated explicitly using the phasestart modifier in order for generation to be possible. Year and day fields with a delta and no phasestart modifier can only be used for only parsing of URIs.

Example: Data files of three-day duration starting on a specific day:

```
data_$Y_$m_$(d;delta=3;phasestart=2013-01-01).txt
```

Given a date range of 2013-01-01 to 2013-01-08 results in

```
data_2013_01_01.txt
data_2013_01_04.txt
data_2013_01_07.txt
```

NOTE: Do not confuse the phasestart with the start time of the dataset. They might be the same, but they do not have to coincide. In the example above, the phasestart is just the beginning of ANY of the three day intervals.

4.2. Begin and end time in filename

If the filename has both a begin and end date in the filename, then when parsing the filename, these two dates give the time range for that URI.

When generating URIs of this type, the end time must be inferred from the start time. The default delta to apply to the start time is assumed to be one unit of the smallest date field. This can be overridden by specifying a delta modifier on the begin time.

If date elements are not present (e.g., only a year and day of year are given, with nothing in the file name for hours, minutes, seconds), then those values are assumed to be zero. This is an issue if the end time is meant to be inclusive, and a shift modifier would be needed to add time to the end time to make it inclusive.

Examples:

Data files with a one-day duration:

- Filename: ace_mag_2005_001_to_2005_002.cdf

  The default assumptions about the date elements in this file lead to the conclusion that this file contains data from just one day, namely day 1 of 2005. In other words, the end time in the file name is not inclusive so that the data in the file go right up to but do not include day 2 of 2005.

- Template: ace_mag_$Y_$j_to_$(Y;end)_$j.cdf

  The template does not need a delta modifier because the default delta is one unit of the smallest date field in the name, which is one day in this case, and that is correct.

Data files with a 4-hour duration

- Filename: ace_mag_2005_001_00_to_2005_001_04.cdf

  The start time is at hour 00 and the stop time is at (the beginning of) hour 04. This is a 4 hour duration. Therefore the delta must be specified because otherwise the default duration of 1 hour would be used, which is incorrect.

- Template: ace_mag_$Y_$j_$(H;delta=4)_to_$(Y;end)_$j_$H.cdf

The delta on the begin time indicates the offset to be applied to get the end time

Data files with a 4-day duration

- Filename: ace_mag_2005_001_to_2005_005.cdf

  The start time is on day 001 of 2005 and the end time is at (the beginning of) day 005, giving a 4 day duration. The delta must be specified because otherwise the duration would default to one unit of smallest time field, which would be 1 day, and that is not correct.

- Template: ace_mag_$Y_$(j;delta=4;phasestart=2005-01-09)_to_$(Y;end)_$j_$H.cdf

  Because this delta is being applied to a day-of-year field, this field must also include the phasestart, because there is no common phase for a day field. Notice also that the value for the phasestart modifier can be any valid starting date for any file in the collection. 2005-01-09 is the start date of the third file in year 2005. The phase start value is only used for generation of URIs -- it is not needed just to parse the URI. Also notice that at year boundaries, this pattern would break down (the files would have to use some other duration scheme for the last file of the year).

End time is shifted

In some cases, the end time in the filename is intended to be inclusive. For example, consider the case where the file data_2009_001_2009_002.dat contains two days of data -- the end day is inclusive (the next file in the list would be data_2009_003_2009_004.dat). To parse or to generate these filenames, a shift modifier is needed on the end time.

The shift modifier requires a value: the amount by which the associated date value is shifted. It is important to understand what is being shifted. The file contents have an actual duration of 2 days in the example, and this value need to be shifted by -1 day to get the date value that goes in the filename, so the modifier value would be negative, as in shift=-1d. Thus the modifier value answers the question: "How much time is added to the actual file duration to get the time stamp value that goes in the file name?"

So the full template for the above example would be data_$Y_$j_$(Y;end)_$(j;duration=2;shift=-1;phasestart=2009-001).dat. This template would enable generation of these file names. If this were a real naming scheme, it would break down at the year boundary, so if the files had different names there, the template scheme would not be able to represent those file names.

Note that the units for the duration and shift are not indicated, but default to days, since they are being used on a date field representing a day value.

Also note that if you just wanted to parse these filenames, you would not need the duration or phasestart modifiers -- just the shift would work, as in data_$Y_$j_$(Y;end)_$(j;shift=-1).dat. As long as the end time was always inclusive, this would work for all the files in the collection.

4.3. Filenames with missing date information

A filename may have only fine-grained date elements (perhaps days and hours), but lack the larger time fields (such as the month and year) that give context to the finer date elements. Consider these files: Given this list of URIs:

```
data_001.txt
data_002.txt
```

The coarse-grained, context date values can be hard-coded into the template so that a complete date range can be identified when parsing the URI. This template data_$(j;Y=2004).txt disambiguates the date by providing a context year for the day of year values. (Evidently, the users of this data all know that it was from 2004.) The URI template is then able to correctly parse the URIs and obtain a fully defined date range. Also, if you wanted to generate URIs given this template and a date range, the generation mechanism would know to only generate files for 2004.

Allowed context date elements are: Y m d j H M S

These context date elements are applied to the entire date value, and so can appear on any field of that date value, i.e., the year context can be set on any field of the begin date, and likewise for the end date.

The following fields can have their context specified this way: m d j H M S

The year is the highest, so there is no context for it!

Examples:

- Set the context year to 2004:

  data_$(j;Y=2004)_$H$M.txt

- Same as the above example where the context date can be set on any of the start date fields

  data_$j_$H$(M;Y=2004).txt

- Set the context year to be 2004 and the day of year to be 365

  data_H(M;Y=2004;j=365).txt

5. Appendix

5.1. Extending the standard

If you want to add your own codes for URIs that are internal to your application, this is possible. These URIs will of course break the standard and should not "escape" outside your application. You must use codes that are not being used by the spec, and they should be preferably multi-character codes.

5.2. Reserved Codes

These codes are already in use: **Y y m b d j H M S v x**

These field codes are not implemented now, but they might be added to a future version of the spec:

$a - weekday name

- modifiers

    - (optional) fmt=abbrev|full (default=abbrev to three letters)
    - (optional) case=lc|uc|cap (default=lc)

$I - hour of day, from 0 to 12

- modifier
    - (optional) pad=zero|none|space|underscore (default=zero (means two places))

$p - AM or PM

- modifier
    - (optional) case=lc|uc (default=uc)

$N - number of nanoseconds

- modifier
    - None.

Might add this later - an option to the wildcard code:

$x (this is the already the code for a wildcard)

- modifier
- (optional) datecode=<valid date code> (this wildcard element in the filename is actually a date element)

The Q code is reserved for extensibility. This code may eventually be used to represent an extended code not supported by the standard. Thus $(Q;ext=b) would indicate an application specific external code b. The idea is that a standard interpreter of the templates would treat all Q codes as wildcards. The Q code would also need a required qualifier to indicate the application or namespace of the extended code. Thus the full extended code would look like $(Q;ext=b;app=org.myorg.myproject).

5.3. JavaCC Parsing Grammar

A parsing grammar for JavaCC has been created that tokenizes and parses template URIs for version 1 of the specification. An entire example application is available here uri-templates-2015-10-09.tgz, and the JavaCC parsing tree file is here uri_template.jjt. This mechanism does only syntax checking -- it does not verify all the rules and constraints that apply to each field. It just checks that the field are well-formed. The sample application includes several sample templates and also provides additional Java classes for separating a URI template into parts so that the field names (each with modifiers and modifier values) can be obtained in a list.

For information about the JavaCC parsing engine, see the JavaCC page (https://java.net/projects/javacc) (or this (http://javacc.java.net/) page. A good book about it is here (http://generatingparserswithjavacc.com/) . There is also a decent JavaCC plugin for Eclipse (http://eclipse-javacc.sourceforge.net/) .

See this application uri-templates-2015-10-09.tgz for a context in which to use the above JavaCC parsing tree.