Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 9, 2010

J. Gregorio, Ed.
Google
R. Fielding, Ed.
Day Software
M. Hadley, Ed.
Oracle
M. Nottingham, Ed.
D. Orchard
Mar 08, 2010

URI Template
draft-gregorio-uritemplate-04

Abstract

A URI Template is a compact sequence of characters for describing a range of Uniform Resource Identifiers through variable expansion. This specification defines the URI Template syntax and the process for expanding a URI Template into a URI, along with guidelines for the use of URI Templates on the Internet.

Editorial Note (to be removed by RFC Editor)

To provide feedback on this Internet-Draft, join the W3C URI mailing list (http://lists.w3.org/Archives/Public/uri/) [1].

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on September 9, 2010.

Copyright Notice

Table of Contents

1. Introduction

1.1. Overview

A Uniform Resource Identifier (URI) [RFC3986] is often used to identify a specific resource within a common space of similar resources. For example, personal web spaces are often delegated using a common pattern, such as

```
http://example.com/~fred/
http://example.com/~mark/
```

or a set of dictionary entries might be grouped in a hierarchy by the first letter of the term, as in

```
http://example.com/dictionary/c/cat
http://example.com/dictionary/d/dog
```

or a service interface might be invoked with various user input in a common pattern, as in

```
http://example.com/search?q=cat&lang=en
http://example.com/search?q=dog&lang=fr
```

URI Templates provide a mechanism for abstracting a space of resource identifiers such that the variable parts can be easily identified and described. URI templates can have many uses, including discovery of available services, configuring resource mappings, defining computed links, specifying interfaces, and other forms of programmatic interaction with resources. For example, the above resources could be described by the following URI templates:

```
http://example.com/~{username}/
http://example.com/dictionary/{term:1}/{term}
http://example.com/search{?q,lang}
```

We define the following terms:

- expression - The text between '{' and '}', including the enclosing braces, as defined in Section 2.
- expansion - The string result obtained from a template expression after processing it according to its expression type, list of variable names, and value modifiers, as defined in Section 3.
- template processor - A program or library that, given a URI Template and a set of variables with values, transforms the template string into a URI-reference by parsing the template for expressions and substituting each one with its corresponding expansion.

A URI Template provides both a structural description of a URI space and, when variable values are provided, a simple instruction on how to construct a URI corresponding to those values. A URI Template is transformed into a URI-reference by replacing each delimited expression with its value as defined by the expression type and the values of variables named within the expression. The expression types range from simple value expansion to multiple key=value lists. The expansions are based on the URI generic syntax, allowing an implementation to process any URI Template without knowing the scheme-specific requirements of every possible resulting URI.

A URI Template may be provided in absolute form, as in the examples above, or in relative form if a suitable base URI is defined.

Although the URI syntax is used for the result, the template string is allowed to contain the broader set of characters that can be found in IRI references [RFC3987]. A URI Template is therefore also an IRI template, and the result of template processing can be rendered as an IRI by transforming the pct-encoded sequences to their corresponding Unicode character if the character is not in the reserved set.

1.2. Expression Types

URI Templates are similar to a macro language with a fixed set of macro definitions: the expression type determines the expansion process. For example, the following URI Template includes a form-style parameter expression, as indicated by the "?" operator appearing before the variable names.

```
  http://www.example.com/foo{?query,number}
```

Each template expression describes, in a machine-readable manner, how a URI is to be constructed. In this example, the expansion process for templates beginning with the question-mark ("?") operator follows the same pattern as form-style interfaces on the World Wide Web.

```
  http://www.example.com/foo{?query,number}
                             \_____________/
                                   |
                                   |
          For each defined variable in [ 'query', 'number' ],
          substitute "?" if it is the first substitution or "&"
          thereafter, followed by the variable name, '=', and the
          variable's value.
```

If the variables have the values

```
  query  := "mycelium"
```

```
  number := 100
```

then the expansion of the above URI Template is

```
  http://www.example.com/foo?query=mycelium&number=100
```

Alternatively, if 'query' is undefined, then the expansion would be

```
  http://www.example.com/foo?number=100
```

or if both variables are undefined, then it would be

```
  http://www.example.com/foo
```

The following table summarizes each type of template expression by its associated operator and cross-references the section of this document that defines the operator and its specific expansion process. The example expansions are based on the following variables and values:

```
   var   := "value";
   hello := "Hello World!";
   undef := null;
   empty := "";
   list  := [ "val1", "val2", "val3" ];
   keys  := [("key1", "val1"), ("key2", "val2")];
   path  := "/foo/bar"
   x     := "1024";
   y     := "768";
```

Sec	Op	Description	
		Expression	Expansion
3.3		Simple expansion with comma-separated values	
		{var}	value
		{hello}	Hello%20World%21
		{path}/here	%2Ffoo%2Fbar/here
		{x,y}	1024,768
		{var=default}	value
		{undef=default}	default
		{list}	val1,val2,val3
		{list*}	val1,val2,val3
		{list+}	list.val1,list.val2,list.val3
		{keys}	key1,val1,key2,val2
		{keys*}	key1,val1,key2,val2
		{keys+}	keys.key1,val1,keys.key2,val2

3.4	+	Reserved expansion with comma-separated values	
		{+var}	value
		{+hello}	Hello%20World!
		{+path}/here	/foo/bar/here
		{+path,x}/here	/foo/bar,1024/here
		{+path}{x}/here	/foo/bar1024/here
		{+empty}/here	/here
		{+undef}/here	/here
		{+list}	val1,val2,val3
		{+list*}	val1,val2,val3
		{+list+}	list.val1,list.val2,list.val3
		{+keys}	key1,val1,key2,val2
		{+keys*}	key1,val1,key2,val2
		{+keys+}	keys.key1,val1,keys.key2,val2
3.5	;	Path-style parameters, semicolon-prefixed	
		{;x,y}	;x=1024;y=768
		{;x,y,empty}	;x=1024;y=768;empty
		{;x,y,undef}	;x=1024;y=768
		{;list}	;val1,val2,val3
		{;list*}	;val1;val2;val3
		{;list+}	;list=val1;list=val2;list=val3
		{;keys}	;key1,val1,key2,val2
		{;keys*}	;key1=val1;key2=val2
		{;keys+}	;keys.key1=val1;keys.key2=val2
3.6	?	Form-style parameters, ampersand-separated	
		{?x,y}	?x=1024&y=768
		{?x,y,empty}	?x=1024&y=768&empty=
		{?x,y,undef}	?x=1024&y=768
		{?list}	?list=val1,val2,val3
		{?list*}	?val1&val2&val3
		{?list+}	?list=val1&list=val2&list=val3
		{?keys}	?keys=key1,val1,key2,val2
		{?keys*}	?key1=val1&key2=val2
		{?keys+}	?keys.key1=val1&keys.key2=val2
3.7	/	Hierarchical path segments, slash-separated	
		{/var}	/value

			{/var,empty}	/value/
			{/var,undef}	/value
			{/list}	/val1,val2,val3
			{/list*}	/val1/val2/val3
			{/list*,x}	/val1/val2/val3/1024
			{/list+}	/list.val1/list.val2/list.val3
			{/keys}	/key1,val1,key2,val2
			{/keys*}	/key1/val1/key2/val2
			{/keys+}	/keys.key1/val1/keys.key2/val2
3.8	.	Label expansion, dot-prefixed		
			X{.var}	X.value
			X{.empty}	X.
			X{.undef}	X
			X{.list}	X.val1,val2,val3
			X{.list*}	X.val1.val2.val3
			X{.list*,x}	X.val1.val2.val3.1024
			X{.list+}	X.list.val1.list.val2.list.val3
			X{.keys}	X.key1,val1,key2,val2
			X{.keys*}	X.key1.val1.key2.val2
			X{.keys+}	X.keys.key1.val1.keys.key2.val2

1.3. Design Considerations

The URI Template syntax has been designed to carefully balance the need for a powerful expansion mechanism with the need for ease of implementation. The syntax is designed to be trivial to parse while at the same time providing enough flexibility to express many common template scenarios. Implementations are able to parse the template and perform the expansions in a single pass.

Templates are simple and readable when used with common examples because the single-character operators match the URI generic syntax delimiters. The operator's associated delimiter (";", "?", "/", and ".") is omitted when none of the listed variables are defined. Likewise, the expansion process for ";" (path-style parameters) will omit the "=" when the variable value is empty, whereas the process for "?" (form-style parameters) will not omit the "=" when the value is empty. Multiple variables and list values have their values joined with "," if there is no predefined joining mechanism for the operator. Only one operator, plus ("+"), will substitute unencoded reserved characters found inside the variable values; the other operators will pct-encode reserved characters found in the variable values prior to expansion.

The most common cases for URI spaces can be described with simple URI Template expressions. If we were only concerned with URI generation, then the template syntax could be limited to just simple variable expansion, since more complex forms could be generated by changing the variable values. However, URI Templates have the additional goal of describing the layout of identifiers in terms of preexisting data values. The template syntax therefore includes operators that reflect how resource identifiers are commonly allocated. Likewise, since prefix and suffix substrings are often used to partition large spaces of resources, modifiers on variable values provide a way to specify those substrings.

Mechanisms similar to URI Templates have been defined within several specifications, including WSDL, WADL and OpenSearch. This specification extends and formally defines the syntax so that URI Templates can be used consistently across multiple Internet applications and within Internet message fields.

1.4. Limitations

Since a URI Template describes a superset of the identifiers, there is no implication that every possible expansion for each delimited variable expression corresponds to a URI of an existing resource. Our expectation is that an application constructing URIs according to the template will be provided with an appropriate set of values for the variables being substituted and will be able to cope with any errors that might occur when the resulting URI is used for name resolution or access.

URI Template expressions are not URIs: they do not identify an abstract or physical resource, they are not parsed as URIs, and should not be used in places where a URI would be expected unless the template expressions will be expanded by a template processor prior to use. Distinct field, element, or attribute names should be used to differentiate protocol elements that carry a URI Template from those that expect a URI-reference.

1.5. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234]. The following ABNF rules are imported from the normative references [RFC5234], [RFC3986], and [RFC3987].

```
   ALPHA          =  %x41-5A / %x61-7A   ; A-Z / a-z
   DIGIT          =  %x30-39             ; 0-9
   HEXDIG         =  DIGIT / "A" / "B" / "C" / "D" / "E" / "F"

   pct-encoded    =  "%" HEXDIG HEXDIG
   unreserved     =  ALPHA / DIGIT / "-" / "." / "_" / "~"
   reserved       =  gen-delims / sub-delims
   gen-delims     =  ":" / "/" / "?" / "#" / "[" / "]" / "@"
   sub-delims     =  "!" / "$" / "&" / "'" / "(" / ")"
                  /  "*" / "+" / "," / ";" / "="

   ucschar        =  %xA0-D7FF / %xF900-FDCF / %xFDF0-FFEF
                  /  %x10000-1FFFD / %x20000-2FFFD / %x30000-3FFFD
                  /  %x40000-4FFFD / %x50000-5FFFD / %x60000-6FFFD
                  /  %x70000-7FFFD / %x80000-8FFFD / %x90000-9FFFD
                  /  %xA0000-AFFFD / %xB0000-BFFFD / %xC0000-CFFFD
                  /  %xD0000-DFFFD / %xE1000-EFFFD

   iprivate       =  %xE000-F8FF / %xF0000-FFFFD / %x100000-10FFFD
```

This specification uses the terms "character" and "coded character set" in accordance with the definitions provided in [RFC2978], and "character encoding" in place of what [RFC2978] refers to as a "charset".

A URI Template is defined as a sequence of characters and therefore has the same issues as URIs with regard to codepoints and character sets. That is, URI Template characters are frequently encoded as octets for transport or presentation. This specification does not mandate any particular character encoding for mapping between URI Template characters and the octets used to store or transmit those characters. When a URI Template appears in a protocol element, the character encoding is defined by that protocol; without such a definition, a URI Template is assumed to be in the same character encoding as the surrounding text.

A URI Template and its associated variable values are converted to a normal form of UTF-8 [RFC3629] prior to template processing, as defined in Section 3.1.

The ABNF notation defines its terminal values to be non-negative integers (codepoints) that are a superset of the US-ASCII coded character set [ASCII]. This specification defines terminal values as codepoints within the Unicode coded character set [UNIV4]. Thus, a string of characters in a URI Template is assumed to be transformed into its corresponding sequence of Unicode codepoints prior to testing for a match with the URI Template grammar.

2. URI Template Syntax

A URI Template is a string of printable Unicode characters that contains zero or more embedded variable expressions, each expression being delimited by a matching pair of braces ('{', '}').

```
URI-Template  = *( literals / expression )
```

2.1. Literals

The characters outside of expressions in a URI Template string are intended to be translated literally to the URI-reference.

```
literals      =  %x21 / %x23-24 / %x26 / %x28-3B / %x3D / %x3F-5B
              /  %x5D-5F / %x61-7A / %x7E / ucschar / iprivate
              /  pct-encoded
                   ; any Unicode character except: CTL, SP,
                   ;  DQUOTE, "'", "%" (aside from pct-encoded),
                   ;  "<", ">", "\", "^", "`", "{", "|", "}"
```

2.2. Expressions

Template expressions are the parameterized parts of a URI Template. Each expression contains an optional operator, which defines the expression type and its corresponding expansion process, followed by a comma-separated list of variable specifiers (variable names and optional value modifiers). If no operator is provided, the expression defaults to simple variable expansion of unreserved values.

```
expression    =  "{" [ operator ] variable-list "}"
operator      =  "+" / "." / "/" / ";" / "?" / op-reserve
op-reserve    =  "|" / "!" / "@"
                     ; reserved for local use: "$" / "(" / ")"
```

The operator characters have been chosen to reflect each of their roles as reserved characters in the URI generic syntax. The operators defined by this specification include: plus ("+") for substituting values that may contain reserved characters; dot (".") for substituting values as a sequence of name labels prefixed by "."; slash ("/") for substituting values as a sequence of path segments separated by "/"; semicolon (";") for substituting key=value pairs as path parameters prefixed by ";"; and, question-mark ("?") for substituting a query component beginning with "?" and consisting of key=value pairs separated by "&". These operators will be described in detail in Section 3.

The operator characters pipe ("|"), exclamation ("!"), and at-sign

("@") are reserved for future extensions. A processor that unexpectedly encounters such an extension operator SHOULD pass the expression through unexpanded and MAY also indicate a warning to the invoking application.

The expression syntax specifically excludes use of the dollar ("$") and parentheses ["(" and ")"] characters so that they remain available for local language extensions outside the scope of this specification.

2.3. Variables

After the operator (if any), each expression contains a list of one or more comma-separated variable specifiers (varspec). The variable names serve multiple purposes: documentation for what kinds of values are expected, identifiers for associating values within a URI Template processor, and the string to use for each key on key=value expansions.

```
variable-list =  varspec *( "," varspec )
varspec       =  varname [ modifier ] [ "=" default ]
varname       =  varchar *( varchar / "." )
varchar       =  ALPHA / DIGIT / "_" / ucschar / iprivate
              /  pct-encoded
```

An expression MAY reference variables that are unknown to the template processor or whose value is set to a special "undefined" value, such as undef or null. Such undefined variables are given special treatment by the expansion process.

A variable value that is a string of length zero is not considered undefined; it has the defined value of an empty string.

A variable may have a composite or structured value, such as a list of values, an associative array of (key, value) pairs, or a structure of components defined by some separate schema. Such value types are not directly indicated by the template syntax, but do have an impact on the expansion process. A composite or structured value with zero member values is considered undefined.

If a variable appears more than once in an expression or within multiple expressions of a URI Template, the value of that variable MUST remain static throughout the expansion process (i.e., the variable must have the same value for the purpose of calculating each expansion).

2.4. Value Modifiers

Any of the variables can have a modifier indicating that its value is exploded into components or is limited to a prefix, suffix, or the remainder of a prefix or suffix of the variable value.

```
modifier      =  explode / partial
```

2.4.1. Component Values

The explode modifiers ("*" and "+") indicate that the variable represents a composite value that may be substituted in full or partial forms, depending on the variable's type or schema. Since URI Templates do not contain an indication of type or schema, this is assumed to be determined by context. An example context is a mark-up element or header field that contains one attribute that is a template and one or more other attributes that define the schema applicable to variables found in the template. Likewise, a typed programming language might differentiate variables as strings, lists, associative arrays, or structures.

```
explode       =  ( "*" / "+" )
```

The primary difference between the two explode modifiers is that an asterisk ("*") indicates that just the component names and values are included in the expansion, whereas the plus ("+") indicates that each component name is prefixed with the given variable name and a period (".), thereby enabling multiple variables with the same component names to be disambiguated.

Component modifiers improve brevity in the URI Template syntax. For example, a resource that provides a geographic map for a given street address might accept a hundred permutations on fields for address input, including partial addresses (e.g., just the city or postal code). Such a resource could be described as a template with each and every address component listed in order, or with a far more simple template that makes use of an explode modifier, as in

```
/mapper{?address*}
```

or

```
/directions{?from+,to+}
```

along with some context that defines each variable (address, from, and to) as adhering to a given addressing standard (e.g., UPU S42 or AS/NZS 4819:2003). A recipient aware of the schema can then provide appropriate expansions, such as:

```
/mapper?city=Newport%20Beach&state=CA
/directions?from.zipcode=92660&to.zipcode=90210
```

The expansion process for variables, as defined in Section 3, is dependent on both the operator being used and, if one of the explode modifiers is used, the type and schema of the variable being substituted.

2.4.2. Prefix and Suffix Values

Prefix and suffix modifiers are often used to partition an identifier space hierarchically, as is common in reference indices and hash-based storage, or to limit the substituted value to a maximum number of characters.

```
partial      =  ( substring / remainder ) offset
substring    =  ":"
remainder    =  "^"
offset       =  [ from-end ] 1*DIGIT
from-end     =  "-"
```

The offset refers to a maximum number of characters from either the beginning (prefix) or end (suffix) of the variable's value as a Unicode string. Note that this numbering is in characters, not octets, in order to avoid substituting improperly encoded values due to splitting a multi-octet UTF-8 encoded character or a pct-encoded triplet.

A substring modifier requires that only the indicated prefix or suffix be used in the expansion. A remainder modifier requires that only the remainder of the value, excluding the indicated prefix or suffix, be used in the expansion. If the offset is greater than the length of the variable's value, then the entire string is used for a substring and the empty string is used for a remainder.

The following examples illustrate how modifiers work with the different variable types. More complex examples are provided in Section 4.

```
Given the variable assignments:
  var   := "value";
  name  := [ "Fred", "Wilma", "Pebbles" ];

Example Template     Expansion

  {var}              value
  {var:20}           value
  {var:3}            val
  {var^3}            ue
  {var:-3}           lue
  {var^-3}           va

  {?name}            ?name=Fred,Wilma,Pebbles
  {?name:1}          ?name=F
```

2.5. Value Defaults

Any of the variables may also be supplied with a default value to be used when a template processor determines the variable to be undefined. The default value is limited to the unreserved and pct-encoded characters of a URI-reference, since our intention is for the default to be presented in the exact form that it would appear in the resulting URI. The default is not affected by the variable modifiers; it is assumed that the default string provided in the expression already reflects any necessary substring or remainder processing.

```
default       =  *( unreserved / pct-encoded )
```

The following examples illustrate how default values work with different variable types. More complex examples are provided in Section 4.

```
Given the variable assignments:
  var   := "value";
  empty := "";
  undef := null;
  name  := [ "Fred", "Wilma", "Pebbles" ];
  favs  := [("color","red"), ("volume","high")];
  empty_list := [];
  empty_keys := [];

Example Template        Expansion

  {var=default}         value
  {undef=default}       default
```

```
x{empty}y             xy
x{empty=_}y           xy
x{undef}y             xy
x{undef=_}y           x_y

x{empty_list}y        xy
x{empty_list=_}y      xy
x{empty_list*}y       xy
x{empty_list*=_}y     x_y
x{empty_list+}y       xy
x{empty_list+=_}y     xempty_list._y

x{empty_keys}y        xy
x{empty_keys=_}y      xy
x{empty_keys*}y       xy
x{empty_keys*=_}y     x_y
x{empty_keys+}y       xy
x{empty_keys+=_}y     xempty_keys._y

x{?name=none}         x?name=Fred,Wilma,Pebbles
x{?favs=none}         x?favs=color,red,volume,high
x{?favs*=none}        x?color=red&volume=high
x{?favs+=none}        x?favs.color=red&favs.volume=high

x{?undef}             x
x{?undef=none}        x?undef=none
x{?empty}             x?empty=
x{?empty=none}        x?empty=

x{?empty_list}        x
x{?empty_list=none}   x?empty_list=none
x{?empty_list*}       x
x{?empty_list*=none}  x?none
x{?empty_list+}       x
x{?empty_list+=none}  x?empty_list.none

x{?empty_keys}        x
x{?empty_keys=none}   x?empty_keys=none
x{?empty_keys*}       x
x{?empty_keys*=none}  x?none
x{?empty_keys+}       x
x{?empty_keys+=none}  x?empty_keys.none

x{;name=none}         x;name=Fred,Wilma,Pebbles
x{;favs=none}         x;favs=color,red,volume,high
x{;favs*=none}        x;color=red;volume=high
x{;favs+=none}        x;favs.color=red;favs.volume=high
```

```
x{;undef}             x
x{;undef=none}        x;undef=none
x{;empty}             x;empty
x{;empty=none}        x;empty

x{;empty_list}        x
x{;empty_list=none}   x;empty_list=none
x{;empty_list*}       x
x{;empty_list*=none}  x;none
x{;empty_list+}       x
x{;empty_list+=none}  x;empty_list.none

x{;empty_keys}        x
x{;empty_keys=none}   x;empty_keys=none
x{;empty_keys*}       x
x{;empty_keys*=none}  x;none
x{;empty_keys+}       x
x{;empty_keys+=none}  x;empty_keys.none

x{/name=none}         x/Fred,Wilma,Pebbles
x{/name*=none}        x/Fred/Wilma/Pebbles
x{/name+=none}        x/name.Fred/name.Wilma/name.Pebbles
x{/favs=none}         x/color,red,volume,high
x{/favs*=none}        x/color/red/volume/high
x{/favs+=none}        x/favs.color/red/favs.volume/high

x{/undef}             x
x{/undef=none}        x/none
x{/empty}             x/
x{/empty=none}        x/

x{/empty_list}        x
x{/empty_list=none}   x/none
x{/empty_list*}       x
x{/empty_list*=none}  x/none
x{/empty_list+}       x
x{/empty_list+=none}  x/empty_list.none

x{/empty_keys}        x
x{/empty_keys=none}   x/none
x{/empty_keys*}       x
x{/empty_keys*=none}  x/none
x{/empty_keys+}       x
x{/empty_keys+=none}  x/empty_keys.none
```

3. URI Template Expansion

The process of URI Template expansion is to scan the template string from beginning to end, copying literal characters as-is and replacing each expression with the result of applying the expression's operator to the value of each variable named in the expression.

If a template processor encounters an error, such as an operator that it does not understand or a character sequence that does not match the <URI-Template> grammar, then processing of the template SHOULD cease, the URI-reference result SHOULD be undefined, and the location and type of error SHOULD be indicated to the invoking application.

If a template processor encounters a warning, such as the use of an operator character reserved for future extension, then the processing of the template SHOULD NOT cease, and the location and type of warning SHOULD be indicated to the invoking application.

3.1. Unicode normalization

The Unicode Standard [UNIV4] defines various equivalences between sequences of characters for various purposes. Unicode Standard Annex #15 [UTR15] defines various Normalization Forms for these equivalences, in particular Normalization Form KC (NFKC: Compatibility Decomposition followed by Canonical Composition). Since different Normalized Form unicode strings will have different UTF-8 representations, the only way to guarantee that template processors will produce the same URI is to require a common Normalized Form.

The string values for the URI Template and template variables MUST be in NFKC and encoded as UTF-8 [RFC3629] prior to use in the template expansion process (US-ASCII is a proper subset of NFKC UTF-8). The remaining sections defining the expansion process assume strings are in NFKC UTF-8.

3.2. Literal expansion

If the literal character is allowed anywhere in the URI syntax (unreserved / reserved), then it is copied directly to the result string. Otherwise, the pct-encoded equivalent of the literal character is copied to the result string by encoding the character in UTF-8 (a sequence of octets) and then encoding each octet as a pct-encoded triplet.

3.3. Expression expansion

Each expression is indicated by an opening brace ("{") character and continues until the next closing brace ("}"). The expression is expanded by determining the expression type and then following that type's expansion process for each comma-separated varspec in the expression.

The expression type is determined by looking at the first character after the opening brace. If the character is an operator, then remember the expression type associated with that operator for later expansion decisions and skip to the next character for the varspec list. If the first character is not an operator, then the expression type is simple expansion and the first character is the beginning of the varspec list.

If the expression does not contain any varspec, as in "{}" or "{,}", then a template processor SHOULD copy that invalid expression to the result string, continue processing the remainder of the template, and indicate that an error occurred to the caller.

If the template contains an opening brace without a corresponding closing brace (the template ends in mid-expression), then a processor SHOULD attempt to process the template as if it ended in a closing brace and indicate that an error occurred to the caller.

3.4. Variable and modifier expansion

A variable that is undefined has no value and thus is excluded from the expansion. A variable defined as composite or component values is undefined if it contains zero members or all of its components are undefined. If all of the variables are undefined, then the expression's expansion is the empty string.

A variable defined as a single value is expanded by converting its value to a NFKC UTF-8 string, replacing any character within the string that is not in the unreserved set with its corresponding sequence of pct-encoded octets, applying any prefix or suffix modifier (Section 2.4.2), and then appending the result to the URI-reference.

A variable defined as a list of values is substituted as a string of comma-separated single values when no explode modifier is given. If the "*" modifier is used, then each value is separated by the default delimiter for the expression type. If the "+" operator is used, then the variable name is prepended to the expansion list as if it were the initial value in the list. If a partial modifier is indicated, the modifier is applied to the combined string of values. The list

expansion is then appended to the result string.

A variable defined as an associative array is expanded as a list of alternating key and value pairs, excluding any keys for which the corresponding value is undefined. If no explode modifier is used, then the list is substituted as comma-separated single values. If the "*" modifier is given, then the list is delimited as key=value pairs according to the default delimiters defined by the expression type. If the "+" modifier is used, the values are substituted as in the "*" case, except that each key name is prefixed by the variable name and a ".", as in "name.key=value".

When a variable containing component values is given without an explode modifier, the value of each defined component is substituted, separated by a comma (",") character, in the order indicated by the variable's schema or, if the schema is unknown, in the order provided by the variable's value. A structure of component values is expanded as a list of the component values in the order implied by a preorder (depth-first) traversal of that structure, excluding any components that are undefined.

When an explode modifier is used with an operator that substitutes variables as key=value pairs, the key is determined as follows. If the modifier is an asterisk ("*"), then each "key" is the name of the component. If the modifier is a plus ("+"), then each key is the variable name followed by a period (".") and the component name. In both cases, if the component names have a hierarchical structure, then the component subnames are also appended to the key, each separated by a period.

When an explode modifier is used with the hierarchical ("/") operator, the slash delimiter is substituted before each defined component's value if the modifier is "*", or before each conjunction of component name and value (e.g., "name.value") if the modifier is "+".

3.5. Simple expansion: {var}

The default expression type when no operator is given is simple expansion: the value of each defined variable is substituted in the order given, modified as indicated by the optional modifiers, with each value separated by a comma character (","). A variable that is undefined has no value and thus is excluded from the expansion. If all of the variables are undefined, then the expansion is the empty string.

For example,

```
  foo := "fred"

  "{foo}"        -> "fred"
  "{foo,foo}"    -> "fred,fred"
  "{bar,foo}"    -> "fred"
  "{bar=wilma}"  -> "wilma"
```

3.6. Reserved expansion: {+var}

Reserved expansion is identical to simple expansion except that the substituted values may contain characters in the reserved set.

For example,

```
  foo  := "That's right!"

  "{foo}"        -> "That%27s%20right%21"
  "{+foo}"       -> "That%27s%20right!"

  base := "http://example.com/home/"

  "{base}index"  -> "http%3A%2F%2Fexample.com%2Fhome%2Findex"
  "{+base}index" -> "http://example.com/home/index"
```

The same expansion process is followed as in Section 3.5 except that, instead of replacing any character within each value string that is not in the unreserved set with its corresponding sequence of pct-encoded octets, replace any character within each value string that is not in the set of unreserved or reserved characters with its corresponding sequence of pct-encoded octets.

3.7. Path-style parameter expansion: {;var}

TBD.

3.8. Form-style parameter expansion: {?var}

TBD.

3.9. Hierarchical path expansion: {/var}

TBD.

3.10. Label expansion with dot-prefix: {.var}

TBD.

4. Examples

TBD.

5. Security Considerations

A URI Template does not contain active or executable content. Other security considerations are the same as those for URIs, see section 7 of [RFC3986].

6. IANA Considerations

No IANA actions are required by this document.

7. Acknowledgments

The following people made significant contributions to this specification: Mike Burrows, Michaeljohn Clement, DeWitt Clinton, John Cowan, James H. Manger, and James Snell.

8. Normative References

[ASCII] American National Standards Institute, "Coded Character Set - 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2978] Freed, N. and J. Postel, "IANA Charset Registration Procedures", BCP 19, RFC 2978, October 2000.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC3987] Duerst, M. and M. Suignard, "Internationalized Resource Identifiers (IRIs)", RFC 3987, January 2005.

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

[UNIV4] The Unicode Consortium, "The Unicode Standard, Version 4.0.1, defined by: The Unicode Standard, Version 4.0 (Reading, MA, Addison-Wesley, 2003. ISBN 0-321-18578-1), as amended by Unicode 4.0.1 (http://www.unicode.org/versions/Unicode4.0.1/)", March 2004.

[UTR15] Davis, M. and M. Duerst, "Unicode Normalization Forms", Unicode Standard Annex # 15, April 2003.

[1] <http://lists.w3.org/Archives/Public/uri/>

Appendix A. Example URI Template Parser

Parsing a valid URI Template expression does not require building a parser from the given ABNF. Instead, the set of allowed characters in each part of URI Template expression has been chosen to avoid complex parsing, and breaking an expression into its component parts can be achieved by a series of splits of the character string.

Here is example Python code that parses a URI Template expression and returns the operator, argument, and variables as a tuple. The variables are returned as a dictionary of variable names mapped to their default values. If no default is given then the name maps to None.

TBD.

Appendix B. Revision History (to be removed by RFC Editor)

04 - Changed the operator syntax to a single character that is analogous to its reserved role within the URI generic syntax, resulting in templates that are far more readable for the common cases. Added value modifiers for prefix and suffix expansion. Added explode modifiers to allow expansion of complex variables and lists according to (external) variable types or schema. Replaced use of "expansion" with "expression", since expansion is traditionally used to refer to the result after expanding a macro (not the macro itself). Made applicable to any hypertext reference string, such that the process for template expansion also includes transforming

the surrounding string into a proper URI-reference rather than assuming it is already in absolute URI form. Rewrote the text accordingly.

03 - Added more examples. Introduced error conditions and defined their handling. Changed listjoin to list. Changed -append to -suffix, and allowed -prefix and -suffix to accept list variables. Clarified the handling of unicode.

02 - Added operators and came up with coherent percent-encoding and reserved character story. Added large examples section which is extracted and tested against the implementation.

01

00 - Initial Revision.

Authors' Addresses

Joe Gregorio (editor)
Google

Email: joe@bitworking.org
URI: http://bitworking.org/

Roy T. Fielding (editor)
Day Software

Email: fielding@day.com
URI: http://www.day.com/

Marc Hadley (editor)
Oracle

Email: Marc.Hadley@oracle.com
URI: http://oracle.com/

Mark Nottingham (editor)

Email: mnot@pobox.com
URI: http://mnot.net/

David Orchard

URI: http://www.pacificspirit.com/