Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: February 21, 2012

J. Gregorio
Google
R. Fielding, Ed.
Adobe
M. Hadley
Oracle
M. Nottingham
D. Orchard

Aug 20, 2011

URI Template
draft-gregorio-uritemplate-06

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 reference, 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 in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

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."

This Internet-Draft will expire on February 21, 2012.

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=chien&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, machine-readable instructions 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 string 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.

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}
```

The expansion process for expressions 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
```

A URI Template may be provided in absolute form, as in the examples above, or in relative form. A template MUST be expanded before the resulting reference can be resolved from relative to absolute form.

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 transformed to an IRI by following the process defined in Section 3.2 of [RFC3987].

1.2. Levels and Expression Types

URI Templates are similar to a macro language with a fixed set of macro definitions: the expression type determines the expansion process. The default expression type is simple string expansion, wherein a single named variable is replaced by its value as a string after UTF-8 encoding the characters and then pct-encoding any octets that are not in the unreserved set.

Since most template processors implemented prior to this specification have only implemented the default expression type, we refer to these as Level 1 templates.

Level 1 examples, with variables having values of		
	var := "value"	
	hello := "Hello World!"	
	empty := ""	
	undef := null	

Op	Expression	Expansion	
	Simple string expansion		(Sec 3.2.2)
	{var}	value	
	{hello}	Hello%20World%21	

Level 2 templates add the plus ("+") operator, for expansion of values that are allowed to include reserved characters, and the crosshatch ("#") operator for expansion of fragment identifiers.

Level 2 examples, with variables having values of			
	var := "value"		
	hello := "Hello World!"		
	path := "/foo/bar"		
Op	Expression	Expansion	
+	Reserved string expansion		(Sec 3.2.3)
	{+var}	value	
	{+hello}	Hello%20World!	
	{+path}/here	/foo/bar/here	
	here?ref={+path}	here?ref=/foo/bar	
#	Fragment expansion, crosshatch-prefixed		(Sec 3.2.4)
	X{#var}	X#value	
	X{#hello}	X#Hello%20World!	

Level 3 templates add more complex operators for lists of comma-separated values, dot-prefixed labels, slash-prefixed path segments, semicolon-prefixed path parameters, and the forms-style construction of a query syntax consisting of key=value pairs that are separated by an ampersand character.

Level 3 examples, with variables having values of			
	var := "value"		
	hello := "Hello World!"		
	empty := ""		
	path := "/foo/bar"		
	x := "1024"		
	y := "768"		
Op	Expression	Expansion	
	String expansion with multiple variables		(Sec 3.2.2)
	map?{x,y}	map?1024,768	
	{x,hello,y}	1024,Hello%20World%21,768	

+	Reserved expansion with multiple variables		(Sec 3.2.3)
	{+x,hello,y}	1024,Hello%20World!,768	
	{+path,x}/here	/foo/bar,1024/here	
#	Fragment expansion with multiple variables		(Sec 3.2.4)
	{#x,hello,y}	#1024,Hello%20World!,768	
	{#path,x}/here	#/foo/bar,1024/here	
.	Label expansion, dot-prefixed		(Sec 3.2.5)
	X{.var}	X.value	
	X{.x,y}	X.1024.768	
/	Path segments, slash-prefixed		(Sec 3.2.6)
	{/var}	/value	
	{/var,x}/here	/value/1024/here	
;	Path-style parameters, semicolon-prefixed		(Sec 3.2.7)
	{;x,y}	;x=1024;y=768	
	{;x,y,empty}	;x=1024;y=768;empty	
?	Form-style query, ampersand-separated		(Sec 3.2.8)
	{?x,y}	?x=1024&y=768	
	{?x,y,empty}	?x=1024&y=768&empty=	
&	Form-style query continuation		(Sec 3.2.9)
	?fixed=yes{&x}	?fixed=yes&x=1024	
	{&x,y,empty}	&x=1024&y=768&empty=	

Finally, Level 4 templates add the ability to specify value modifiers as a suffix to the variable name. The prefix modifier (":") indicates that only a limited number of characters from the beginning of the value are used by the expansion. The explode ("*") modifier indicates that the variable is to be treated as a composite value,

consisting of either a list of names or an associative array of (name, value) pairs, that is expanded as if each member were a separate variable.

Level 4 examples, with variables having values of

```
var   := "value"
hello := "Hello World!"
path  := "/foo/bar"
list  := [ "red", "green", "blue" ]
keys  := [("semi",";"),("dot","."),("comma",",")]
empty_keys  := []
```

Op	Expression	Expansion	
	String expansion with value modifiers		(Sec 3.2.2)
	{var:3}	val	
	{var:30}	value	
	{list}	red,green,blue	
	{list*}	red,green,blue	
	{keys}	semi,%3B,dot,.,comma,%2C	
	{keys*}	semi=%3B,dot=.,comma=%2C	
+	Reserved expansion with value modifiers		(Sec 3.2.3)
	{+path:6}/here	/foo/b/here	
	{+list}	red,green,blue	
	{+list*}	red,green,blue	
	{+keys}	semi,;,dot,.,comma,,	
	{+keys*}	semi=;,dot=.,comma=,	
#	Fragment expansion with value modifiers		(Sec 3.2.4)
	{#path:6}/here	#/foo/b/here	
	{#list}	#red,green,blue	
	{#list*}	#red,green,blue	
	{#keys}	#semi,;,dot,.,comma,,	
	{#keys*}	#semi=;,dot=.,comma=,	
.	Label expansion, dot-prefixed		(Sec 3.2.5)
	X{.var:3}	X.val	
	X{.list}	X.red,green,blue	

		X{.list*}	X.red.green.blue
		X{.keys}	X.semi,%3B,dot,.,comma,%2C
		X{.keys*}	X.semi=%3B.dot=..comma=%2C
/	Path segments, slash-prefixed		(Sec 3.2.6)
		{/var:1,var}	/v/value
		{/list}	/red,green,blue
		{/list*}	/red/green/blue
		{/list*,path:4}	/red/green/blue/%2Ffoo
		{/keys}	/semi,%3B,dot,.,comma,%2C
		{/keys*}	/semi=%3B/dot=./comma=%2C
;	Path-style parameters, semicolon-prefixed		(Sec 3.2.7)
		{;hello:5}	;hello=Hello
		{;list}	;list=red,green,blue
		{;list*}	;red;green;blue
		{;keys}	;keys=semi,%3B,dot,.,comma,%2C
		{;keys*}	;semi=%3B;dot=.;comma=%2C
?	Form-style query, ampersand-separated		(Sec 3.2.8)
		{?var:3}	?var=val
		{?list}	?list=red,green,blue
		{?list*}	?list=red&list=green&list=blue
		{?keys}	?keys=semi,%3B,dot,.,comma,%2C
		{?keys*}	?semi=%3B&dot=.&comma=%2C
&	Form-style query continuation		(Sec 3.2.9)
		{&var:3}	&var=val
		{&list}	&list=red,green,blue
		{&list*}	&list=red&list=green&list=blue
		{&keys}	&keys=semi,%3B,dot,.,comma,%2C
		{&keys*}	&semi=%3B&dot=.&comma=%2C

1.3. Design Considerations

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, while at the same time retaining compatibility with those earlier definitions.

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. The "+" and "#" operators 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 Level 1 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 substrings are often used to partition large spaces of resources, modifiers on variable values provide a way to specify both the substring and the full value string with a single variable name.

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, or at least a means of validating user data-entry for those values.

URI Templates 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.

Some URI Templates can be used in reverse for the purpose of variable matching: comparing the template to a fully formed URI in order to extract the variable parts from that URI and assign them to the named variables. Variable matching only works well if the template expressions are delimited by the beginning or end of the URI or by characters that cannot be part of the expansion, such as reserved characters surrounding a simple string expression. In general, regular expression languages are better suited for variable matching.

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
```

1.6. Character Encoding and Unicode Normalization

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".

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].

In spite of the syntax and template expansion process being defined in terms of Unicode codepoints, it should be understood that templates occur in practice as a sequence of characters in whatever form or encoding is suitable for the context in which they occur, whether that be octets embedded in a network protocol element or paint applied to the side of a bus. 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. It is only during the process of template expansion that a string of characters in a URI Template is REQUIRED to be processed as a sequence of Unicode codepoints.

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. The normalization form determines how to consistently encode equivalent strings. In theory, all URI processing implementations, including template processors, should use the same normalization form for generating a URI reference. In practice, they do not. If a value has been provided by the same server as the resource, then it can be assumed that the string is already in the form expected by that server. If a value is provided by a user, such as via a data-entry dialog, then the string SHOULD be normalized as Normalization Form C (NFC: Canonical Decomposition, followed by Canonical Composition) prior to being used in expansions by a template processor.

Likewise, when non-ASCII data that represents readable strings is pct-encoded for use in a URI reference, a template processor MUST first encode the string as UTF-8 [RFC3629] and then pct-encode any octets that are not allowed in a URI reference.

2. 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 )
```

Although templates (and template processor implementations) are described above in terms of four gradual levels, we define the URI-Template syntax in terms of the ABNF for Level 4. A template processor limited to lower level templates MAY exclude the ABNF rules applicable only to higher levels. However, it is RECOMMENDED that all parsers implement the full syntax such that unsupported levels can be properly identified as such to the end user.

2.1. Literals

The characters outside of expressions in a URI Template string are intended to be copied literally to the URI-reference if the character is allowed in a URI (reserved / unreserved / pct-encoded) or, if not allowed, copied to the URI-reference in its UTF-8 pct-encoded form.

```
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 in Section 3 of this specification include:

+ Reserved character strings;

Fragment identifiers prefixed by "#";

. Name labels or extensions prefixed by ".";

/ Path segments prefixed by "/";

; Path parameter key or key=value pairs prefixed by ";";

? Query component beginning with "?" and consisting of key=value pairs separated by "&"; and,

& Continuation of query-style &key=value pairs within a literal query component.

The operator characters equals ("="), comma (","), exclamation ("!"), at-sign ("@"), and pipe ("|") are reserved for future extensions.

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 template processor, and the literal string to use for the name in name=value expansions (aside from when exploding an associative array).

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

A varname MAY contain one or more pct-encoded triplets. These triplets are considered an essential part of the variable name and are not decoded during processing. A varname containing pct-encoded characters is not the same variable as a varname with those same characters decoded. Applications that provide URI Templates are expected to be consistent in their use of pct-encoding within variable names.

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 value in the form of a list of values or an associative array of (name, value) pairs. Such value types are not directly indicated by the template syntax, but do have an impact on the expansion process. A composite value with zero member values is considered undefined.

2.4. Value Modifiers

Each of the variables in a Level 4 template expression can have a modifier indicating either that its expansion is limited to a prefix of the variable's value string or that its expansion is exploded as a composite value in the form of a value list or an associative array of (name, value) pairs.

```
modifier      =  prefix / explode
```

2.4.1. Prefix Values

A prefix modifier indicates that the variable expansion is limited to a prefix of the variable's value string. Prefix modifiers are often used to partition an identifier space hierarchically, as is common in reference indices and hash-based storage. It also serves to limit the expanded value to a maximum number of characters. Prefix modifiers are not applicable to variables that have composite values.

```
prefix        =  ":" max-length
max-length    =  %x31-39 *DIGIT   ; positive integer
```

The max-length is a positive integer that refers to a maximum number of characters from the beginning of the variable's value as a Unicode string. Note that this numbering is in characters, not octets, in order to avoid splitting between the octets of a multi-octet UTF-8 encoded character or within a pct-encoded triplet. If the max-length is greater than the length of the variable's value, then the entire value string is used.

For example,

Given the variable assignments

```
var   := "value"
semi  := ";"
```

Example Template	Expansion
{var}	value
{var:20}	value
{var:3}	val
{semi}	%3B
{semi:2}	%3B

2.4.2. Composite Values

An explode modifier ("*") indicates that the variable represents a composite value that may be substituted in full or partial forms, depending on the variable's type and value set. 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       =  "*"
```

Explode 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*}
```

along with some context that defines what the variable named "address" can include, such as by reference to some other standard for addressing (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
```

The expansion process for exploded variables is dependent on both the

operator being used and whether the composite value is to be treated as a list of values or as an associative array of (name, value) pairs. Structures are processed as if they are an associative array with names corresponding to the fields in the structure definition and "." separators used to indicate name hierarchy in substructures.

3. Expansion

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

If a template processor encounters an error outside of an expression, such as 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 an error is encountered inside an expression, such as an operator or value modifier that it does not recognize or cannot support, then the expression SHOULD be copied to the result unexpanded, processing of the remainder of the template SHOULD continue, and the location and type of error SHOULD be indicated to the invoking application. In this latter case, the result returned will not be a valid URI reference; it will be an incompletely expanded template string that is only intended for diagnostic use.

3.1. Literal Expansion

If the literal character is allowed anywhere in the URI syntax (unreserved / reserved / pct-encoded), 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.

3.2. 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. Level 1 templates are limited to the default operator (simple string value expansion) and a single variable per expression. Level 2 templates are limited to a single varspec per 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 variable-list. If the first character is not an operator, then the expression type is simple string expansion and the first character is the beginning of the variable-list.

The examples in the subsections below use the following definitions for variable values:

```
dom   := "example.com"
dub   := "me/too"
foo   := "That's right!"
hello := "Hello World!"
half  := "50%"
var   := "value"
who   := "fred"
base  := "http://example.com/home/"
path  := "/foo/bar"
list  := [ "red", "green", "blue" ]
keys  := [("semi",";"),("dot","."),("comma",",")]
v     := "6"
x     := "1024"
y     := "768"
empty := ""
empty_keys  := []
undef := null
```

3.2.1. Variable Expansion

A variable that is undefined has no value and is ignored by the expansion process. A variable defined as a list value is considered undefined if the list contains zero members. A variable defined as an associative array of (name, value) pairs is considered undefined if the array contains zero members or if all member names in the array have undefined values. If all of the variables in an expression are undefined, then the expression's expansion is the empty string.

Variable expansion of a defined, non-empty value results in a substring of allowed URI characters. A template processor MUST encode the value string as UTF-8 and transform each octet that is not in the allowed set into the corresponding pct-encoded triplet. The allowed set depends on the expression type: reserved ("+") and fragment ("#") expansions allow the set of characters in (unreserved / reserved / pct-encoded) to be passed through without pct-encoding,

whereas all other expression types allow only unreserved characters to be passed through without pct-encoding. Note that the percent character ("%") is only allowed as part of a pct-encoded triplet and only for reserved/fragment expansion: in all other cases, a value of "%" MUST be pct-encoded as "%25" by variable expansion.

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). However, if reserved characters or pct-encoded triplets occur in the value, they will be pct-encoded by some expression types and not by others.

For a variable that is a simple string value, expansion consists of appending the encoded value to the result string. The explode modifier has no effect. The prefix modifier limits the expansion to the first max-length characters of the decoded value. If the value contains multibyte UTF-8, care must be taken to avoid splitting the value in mid-character: count each Unicode codepoint as one character.

For a variable that is a list of values, expansion consists of concatenating the defined member string values, encoded as above, with a separator string inserted between those values. A prefix modifier has no effect. If no explode modifier is given, the separator string is a comma (","). If an explode modifier is given, the separator string is defined per operator by the following table, where NUL is the default expression type:

operator:	NUL	+	.	/	;	?	&	#
separator:	","	","	"."	"/"	";"	"&"	"&"	","

For a variable that is an associative array, expansion consists of a list of either "name,value" (without explode modifier) or "name=value" (with explode modifier) pairs, excluding any pairs for which the corresponding value is undefined, with a separator string inserted between defined pairs. The separator string is defined in the same way as for list variables above. Both the name and value strings are encoded in the same way as simple string values.

3.2.2. Simple String Expansion: {var}

Simple string expansion is the default expression type when no operator is given.

For each defined variable in the variable-list, perform variable expansion, as defined in Section 3.2.1, with the allowed characters being those in the unreserved set. If more than one variable has a defined value, append a comma (",") to the result string as a separator between variable expansions.

Example Template	Expansion
{var}	value
{hello}	Hello%20World%21
{half}	50%25
O{empty}X	OX
O{undef}X	OX
{x,y}	1024,768
{x,hello,y}	1024,Hello%20World%21,768
?{x,empty}	?1024,
?{x,undef}	?1024
?{undef,y}	?768
{var:3}	val
{var:30}	value
{list}	red,green,blue
{list*}	red,green,blue
{keys}	semi,%3B,dot,.,comma,%2C
{keys*}	semi=%3B,dot=.,comma=%2C

3.2.3. Reserved expansion: {+var}

Reserved expansion, as indicated by the plus ("+") operator for Level 2 and above templates, is identical to simple string expansion except that the substituted values may also contain pct-encoded triplets and characters in the reserved set.

For each defined variable in the variable-list, perform variable expansion, as defined in Section 3.2.1, with the allowed characters being those in the set (unreserved / reserved / pct-encoded). If more than one variable has a defined value, append a comma (",") to the result string as a separator between variable expansions.

Example Template	Expansion
{foo}	That%27s%20right%21
{+foo}	That%27s%20right!
{+half}	50%25
{base}index	http%3A%2F%2Fexample.com%2Fhome%2Findex
{+base}index	http://example.com/home/index
{+var}	value
{+hello}	Hello%20World!
O{+empty}X	OX
O{+undef}X	OX
{+path}/here	/foo/bar/here
here?ref={+path}	here?ref=/foo/bar
up{+path}{var}/here	up/foo/barvalue/here
{+x,hello,y}	1024,Hello%20World!,768
{+path,x}/here	/foo/bar,1024/here
{+path:6}/here	/foo/b/here
{+list}	red,green,blue
{+list*}	red,green,blue
{+keys}	semi,;,dot,.,comma,,
{+keys*}	semi=;,dot=.,comma=,

3.2.4. Fragment expansion: {#var}

Fragment expansion, as indicated by the crosshatch ("#") operator for Level 2 and above templates, is identical to reserved expansion except that a crosshatch character (fragment delimiter) is appended first to the result string if any of the variables are defined.

Example Template	Expansion
{#foo}	#That%27s%20right!
{#var}	#value
{#hello}	#Hello%20World!
{#half}	#50%25
foo{#empty}	foo#
foo{#undef}	foo
{#x,hello,y}	#1024,Hello%20World!,768
{#path,x}/here	#/foo/bar,1024/here
{#path:6}/here	#/foo/b/here
{#list}	#red,green,blue
{#list*}	#red,green,blue
{#keys}	#semi,;,dot,.,comma,,
{#keys*}	#semi=;,dot=.,comma=,

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

Label expansion, as indicated by the dot (".") operator for Level 3 and above templates, is useful for describing URI spaces with varying domain names or path selectors (e.g., filename extensions).

For each defined variable in the variable-list, append "." to the result string and then perform variable expansion, as defined in Section 3.2.1, with the allowed characters being those in the unreserved set.

Since "." is in the unreserved set, a value that contains a "." has the effect of adding multiple labels.

Example Template	Expansion
{.who}	.fred
{.who,who}	.fred.fred
{.half,who}	.50%25.fred
www{.dom}	www.example.com
X{.var}	X.value
X{.empty}	X.
X{.undef}	X
X{.var:3}	X.val
X{.list}	X.red,green,blue
X{.list*}	X.red.green.blue
X{.keys}	X.semi,%3B,dot,.,comma,%2C
X{.keys*}	X.semi=%3B.dot=..comma=%2C
X{.empty_keys}	X
X{.empty_keys*}	X

3.2.6. Path segment expansion: {/var}

Path segment expansion, as indicated by the slash ("/") operator in Level 3 and above templates, is useful for describing URI path hierarchies.

For each defined variable in the variable-list, append "/" to the result string and then perform variable expansion, as defined in Section 3.2.1, with the allowed characters being those in the unreserved set.

Note that the expansion process for path segment expansion is identical to that of label expansion aside from the substitution of "/" instead of ".". However, unlike ".", a "/" is a reserved character and will be pct-encoded if found in a value.

Example Template	Expansion
{/who}	/fred
{/who,who}	/fred/fred
{/half,who}	/50%25/fred
{/who,dub}	/fred/me%2Ftoo
{/var}	/value
{/var,empty}	/value/
{/var,undef}	/value
{/var,x}/here	/value/1024/here
{/var:1,var}	/v/value
{/list}	/red,green,blue
{/list*}	/red/green/blue
{/list*,path:4}	/red/green/blue/%2Ffoo
{/keys}	/semi,%3B,dot,.,comma,%2C
{/keys*}	/semi=%3B/dot=./comma=%2C

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

Path-style parameter expansion, as indicated by the semicolon (";") operator in Level 3 and above templates, is useful for describing URI path parameters, such as "pathname;property" or "pathname;key=value".

For each defined variable in the variable-list:

- append ";" to the result string;
- if no explode modifier is present or the variable does not have a composite value, append the variable name (encoded as if it were a literal string) to the result string and, if the variable's value is not empty, append "=" to the result string; and,
- perform variable expansion, as defined in Section 3.2.1, with the allowed characters being those in the unreserved set.

Example Template	Expansion
{;who}	;who=fred
{;half}	;half=50%25
{;empty}	;empty
{;v,empty,who}	;v=6;empty;who=fred
{;v,bar,who}	;v=6;who=fred
{;x,y}	;x=1024;y=768
{;x,y,empty}	;x=1024;y=768;empty
{;x,y,undef}	;x=1024;y=768
{;hello:5}	;hello=Hello
{;list}	;list=red,green,blue
{;list*}	;red;green;blue
{;keys}	;keys=semi,%3B,dot,.,comma,%2C
{;keys*}	;semi=%3B;dot=.;comma=%2C

3.2.8. Form-style query expansion: {?var}

Form-style query expansion, as indicated by the question-mark ("?") operator in Level 3 and above templates, is useful for describing an entire optional query component.

For each defined variable in the variable-list:
- append "?" to the result string if this is the first defined value or append "&" thereafter;
- if no explode modifier is present or the variable does not have a composite value, append the variable name (encoded as if it were a literal string) and an equals character ("=") to the result string; and,
- perform variable expansion, as defined in Section 3.2.1, with the allowed characters being those in the unreserved set.

Example Template	Expansion
{?who}	?who=fred
{?half}	?half=50%25
{?x,y}	?x=1024&y=768
{?x,y,empty}	?x=1024&y=768&empty=
{?x,y,undef}	?x=1024&y=768
{?var:3}	?var=val
{?list}	?list=red,green,blue
{?list*}	?list=red&list=green&list=blue
{?keys}	?keys=semi,%3B,dot,.,comma,%2C
{?keys*}	?semi=%3B&dot=.&comma=%2C

3.2.9. Form-style query continuation: {&var}

Form-style query continuation, as indicated by the ampersand ("&") operator in Level 3 and above templates, is useful for describing optional &name=value pairs in a template that already contains a literal query component with fixed parameters.

For each defined variable in the variable-list:
- append "&" to the result string;
- if no explode modifier is present or the variable does not have a composite value, append the variable name (encoded as if it were a literal string) and an equals character ("=") to the result string; and,
- perform variable expansion, as defined in Section 3.2.1, with the allowed characters being those in the unreserved set.

Example Template	Expansion
{&who}	&who=fred
{&half}	&half=50%25
?fixed=yes{&x}	?fixed=yes&x=1024
{&x,y,empty}	&x=1024&y=768&empty=
{&x,y,undef}	&x=1024&y=768
{&var:3}	&var=val
{&list}	&list=red,green,blue
{&list*}	&list=red&list=green&list=blue
{&keys}	&keys=semi,%3B,dot,.,comma,%2C
{&keys*}	&semi=%3B&dot=.&comma=%2C

4. Security Considerations

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

5. IANA Considerations

No IANA actions are required by this document.

6. Acknowledgments

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

7. 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. Implementation Hints

The normative sections on expansion describe each operator with a separate expansion process for the sake of descriptive clarity. In actual implementations, we expect the expressions to be processed left-to-right using a common algorithm that has only minor variations in process per operator. This appendix describes one such algorithm.

Initialize an empty result string and its non-error state.

Scan the template and copy literals to the result string (as in Section 3.1) until an expression is indicated by a "{" or the template ends. When it ends, return the result string and its current error or non-error state.

- If an expression is found, scan the template to the next "}" and extract the characters in between the braces.
- If the template ends before a "}", then append the "{" and extracted characters to the result string and return with an error status indicating the expression is malformed.

Examine the first character of the extracted expression for an operator.

- If the expression ended (i.e., is "{}"), an operator is found that is unknown or unimplemented, or the character is not in the varchar set (Section 2.3), then append "{", the extracted

expression, and "}" to the result string, remember that the result is in an error state, and then go back to scan the remainder of the template.

- If a known and implemented operator is found, store the operator and skip to the next character to begin the varspec-list.
- Otherwise, store the operator as NUL (simple string expansion).

Use the following value table to determine the processing behavior by expression type operator. The entry for "first" is the string to append to the result first if any of the expression's variables are defined. The entry for "sep" is the separator to append to the result before any second (or subsequent) defined variable expansion. The entry for "named" is a boolean for whether or not the expansion includes the variable name when no explode modifier is given. The entry for "ifemp" is a string to append to the name when the expansion includes the variable (or key) name and that variable (or key) has an empty value. The entry for "allow" indicates what characters to allow unencoded within the value expansion: (U) means any character not in the unreserved set will be encoded; (U+R) means any character not in the (unreserved / reserved / pct-encoding) set will be encoded; and, for both cases, disallowed characters are encoded as UTF-8 (a sequence of octets) and then each octet is encoded as a pct-encoded triplet.

	NUL	+	.	/	;	?	&	#
first	""	""	"."	"/"	";"	"?"	"&"	"#"
sep	","	","	"."	"/"	";"	"&"	"&"	","
named	false	false	false	false	true	true	true	false
ifemp	""	""	""	""	""	"="	"="	""
allow	U	U+R	U	U	U	U	U	U+R

With the above table in mind, process the variable-list as follows:

For each varspec, extract the varname and optional modifier, lookup the value for that variable, and then:

- If the varname is unknown or corresponds to a variable with an undefined value (Section 3.2.1), then skip to the next varspec.
- If this is the first defined variable for this expression, append the first string for this expression type to the result string and remember that it has been done. Otherwise, append the sep string to the result string.
- If an explode modifier is present, then
  - If the variable is a list, then append each defined list member to the result string, after encoding any characters that are not in the allow set, with the sep string appended to the

result between each defined list member.

* If the variable is an associative array of (name, value) pairs, then append each pair with a defined value to the result string as "name=value", after encoding any characters that are not in the allow set, with the sep string appended to the result between each defined pair.
* Otherwise (the variable is a string):
  + If named is true, append the varname to the result string using the same encoding process as for literals, and
    - if the value is empty, append the ifemp string to the result string and skip to the next varspec;
    - otherwise, append "=" to the result string.
  + Append the value to the result string after encoding any characters that are not in the allow set.

o If an explode modifier is not present, then
* If named is true, append the varname to the result string using the same encoding process as for literals, and
  + if the value is empty, append the ifemp string to the result string and skip to the next varspec;
  + otherwise, append "=" to the result string.
* If the variable is a list, append each defined list member to the result string, after encoding any characters that are not in the allow set, with a comma (",") appended to the result between each defined list member.
* If the variable is an associative array of (name, value) pairs, append each pair with a defined value to the result string as "name,value", after encoding any characters that are not in the allow set, with a comma (",") appended to the result between each defined pair.
* Otherwise (the variable is a string), then
  + if a prefix modifier is present and the prefix length is less than the value string length in number of Unicode characters, append that number of characters from the beginning of the value string to the result string, after encoding any characters that are not in the allow set, while taking care not to split multi-octet or pct-encoded triplet characters that represent a single Unicode codepoint;
  + otherwise, append the value to the result string after encoding any characters that are not in the allow set.

When the variable-list for this expression is exhausted, go back to scan the remainder of the template.

Authors' Addresses

Joe Gregorio
Google

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

Roy T. Fielding (editor)
Adobe Systems Incorporated

Email: fielding@gbiv.com
URI: http://roy.gbiv.com/

Marc Hadley
Oracle

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

Mark Nottingham

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

David Orchard

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