Internet Engineering Task Force (IETF)
Request for Comments: 6570
Category: Standards Track
ISSN: 2070-1721

J. Gregorio
Google
R. Fielding
Adobe
M. Hadley
MITRE
M. Nottingham
Rackspace
D. Orchard
Salesforce.com
March 2012

URI Template

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.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6570.

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 (informally, a "URI space"). 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
```

A URI Template is a compact sequence of characters for describing a range of Uniform Resource Identifiers through variable expansion.

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 the 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 name=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 is expanded before the resulting reference is 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 Internationalized Resource Identifier (IRI) references [RFC3987]. Therefore, a URI Template is 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 pct-encoding any characters not in the set of unreserved URI characters (Section 1.5).

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!"                             |
|                                                                 |
|-----------------------------------------------------------------|
| 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 URI characters (Section 1.5), 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 allow multiple variables per expression, each separated by a comma, and add more complex operators for dot-prefixed labels, slash-prefixed path segments, semicolon-prefixed path parameters, and the form-style construction of a query syntax consisting of name=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 value modifiers as an optional suffix to each variable name. A prefix modifier (":") indicates that only a limited number of characters from the beginning of the value are used by the expansion (Section 2.4.1). An 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 (Section 2.4.2).

```
   .-----------------------------------------------------------------.
   | Level 4 examples, with variables having values of               |
   |                                                                 |
   |             var   := "value"                                    |
   |             hello := "Hello World!"                             |
   |             path  := "/foo/bar"                                 |
   |             list  := ("red", "green", "blue")                   |
   |             keys  := [("semi",";"),("dot","."),("comma",",")]   |
   |                                                                 |
   | 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*}              ;list=red;list=green;list=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 [WSDL], WADL [WADL], and OpenSearch [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. Therefore, the template syntax 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 they 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"
                  ; case-insensitive

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", "character encoding scheme", "code point", "coded character set", "glyph", "non-ASCII", "normalization", "protocol element", and "regular expression" as they are defined in [RFC6365].

The ABNF notation defines its terminal values to be non-negative integers (code points) that are a superset of the US-ASCII coded character set [ASCII]. This specification defines terminal values as code points within the Unicode coded character set [UNIV6].

In spite of the syntax and template expansion process being defined in terms of Unicode code points, 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 glyphs painted on the side of a bus. This specification does not mandate any particular character encoding scheme 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 scheme is defined by that protocol; without such a definition, a URI Template is assumed to be in the same character encoding scheme 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 code points.

The Unicode Standard [UNIV6] 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 as the sequence of pct-encoded triplets corresponding to that character’s encoding in UTF-8 [RFC3629].

```
  literals      =  %x21 / %x23-24 / %x26 / %x28-3B / %x3D / %x3F-5B
                /  %x5D / %x5F / %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-level2 / op-level3 / op-reserve
  op-level2     =  "+" / "#"
  op-level3     =  "." / "/" / ";" / "?" / "&"
  op-reserve    =  "=" / "," / "!" / "@" / "|"
```

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 name or name=value pairs prefixed by ";";
?	Query component beginning with "?" and consisting of name=value pairs separated by "&"; and,
&	Continuation of query-style &name=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 use outside the scope of this specification. For example, a macro language might use these characters to apply macro substitution to a string prior to that string being processed as a URI Template.

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 names are case-sensitive because the name might be expanded within a case-sensitive URI component.

```
  variable-list =  varspec *( "," varspec )
  varspec       =  varname [ modifier-level4 ]
  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 (Section 3.2.1).

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

In Level 4 templates, 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 they do have an impact on the expansion process (Section 3.2.1).

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 are associated with undefined values.

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-level4 =  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 0*3DIGIT   ; positive integer < 10000
```

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-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 is to be treated as a composite value consisting of either a list of values or an associative array of (name, value) pairs. Hence, the expansion process is applied to each member of the composite as if it were listed as a separate variable. This kind of variable specification is significantly less self-documenting than non-exploded variables, since there is less correspondence between the variable name and how the URI reference appears after expansion.

```
explode       =  "*"
```

Since URI Templates do not contain an indication of type or schema, the type for an exploded variable is assumed to be determined by context. For example, the processor might be supplied values in a form that differentiates values as strings, lists, or associative arrays. Likewise, the context in which the template is used (script, mark-up language, Interface Definition Language, etc.) might define rules for associating variable names with types, structures, or schema.

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]). 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.

If a variable has a composite structure and only some of the fields in that structure have defined values, then only the defined pairs are present in the expansion. This can be useful for templates that consist of a large number of potential query terms.

An explode modifier applied to a list variable causes the expansion to iterate over the list’s member values. For path and query parameter expansions, each member value is paired with the variable’s name as a (varname, value) pair. This allows path and query parameters to be repeated for multiple values, as in

```
Given the variable assignments

  year  := ("1965", "2000", "2012")
  dom   := ("example", "com")

Example Template     Expansion

  find{?year*}       find?year=1965&year=2000&year=2012
  www{.dom*}         www.example.com
```

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.

The requirements on expansion for each aspect of the URI Template grammar are defined in this section. A non-normative algorithm for the expansion process as a whole is provided in Appendix A.

If a template processor encounters a character sequence outside an expression that does not match the <URI-Template> grammar, then processing of the template SHOULD cease, the URI reference result SHOULD contain the expanded part of the template followed by the remainder unexpanded, and the location and type of error SHOULD be indicated to the invoking application.

If an error is encountered in an expression, such as an operator or value modifier that the template processor does not recognize or does not yet support, or a character is found that is not allowed by the <expression> grammar, then the unprocessed parts of 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.

If an error occurs, the result returned might 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 first encoding the character as its sequence of octets in UTF-8 and then encoding each such 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 ("}"). Expressions cannot be nested.

An expression is expanded by determining its 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:

```
count := ("one", "two", "three")
dom   := ("example", "com")
dub   := "me/too"
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 (Section 2.3) has no value and is ignored by the expansion process. 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. As described in Section 1.6, the expansion process is defined in terms of Unicode code points in order to ensure that non-ASCII characters are consistently pct-encoded in the resulting URI reference. One way for a template processor to obtain a consistent expansion is to transcode the value string to UTF-8 (if it is not already in UTF-8) and then transform each octet that is not in the allowed set into the corresponding pct-encoded triplet. Another is to map directly from the value's native character encoding to the set of allowed URI characters, with any remaining disallowed characters mapping to the sequence of pct-encoded triplets that correspond to the octet(s) of that character when encoded as UTF-8 [RFC3629].

The allowed set for a given expansion depends on the expression type: reserved ("+") and fragment ("#") expansions allow the set of characters in the union of (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 character 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. An explode modifier has no effect. A prefix modifier limits the expansion to the first max-length characters of the decoded value. If the value contains multi-octet or pct-encoded characters, care must be taken to avoid splitting the value in mid-character: count each Unicode code point as one character.

For a variable that is an associative array, expansion depends on both the expression type and the presence of an explode modifier. If there is no explode modifier, expansion consists of appending a comma-separated concatenation of each (name, value) pair that has a defined value. If there is an explode modifier, expansion consists of appending each pair that has a defined value as either "name=value" or, if the value is the empty string and the expression type does not indicate form-style parameters (i.e., not a "?" or "&" type), simply "name". Both name and value strings are encoded in the same way as simple string values. A separator string is appended between defined pairs according to the expression type, as defined by the following table:

Type	Separator	
	","	(default)
+	","	
#	","	
.	"."	
/	"/"	
;	";"	
?	"&"	
&	"&"	

For a variable that is a list of values, expansion depends on both the expression type and the presence of an explode modifier. If there is no explode modifier, the expansion consists of a comma-separated concatenation of the defined member string values. If there is an explode modifier and the expression type expands named parameters (";", "?", or "&"), then the list is expanded as if it were an associative array in which each member value is paired with the list’s varname. Otherwise, the value will be expanded as if it were a list of separate variable values, each value separated by the expression type’s associated separator as defined by the table above.

Example Template	Expansion
{count}	one,two,three
{count*}	one,two,three
{/count}	/one,two,three
{/count*}	/one/two/three
{;count}	;count=one,two,three
{;count*}	;count=one;count=two;count=three
{?count}	?count=one,two,three
{?count*}	?count=one&count=two&count=three
{&count*}	&count=one&count=two&count=three

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
{+var}	value
{+hello}	Hello%20World!
{+half}	50%25
{base}index	http%3A%2F%2Fexample.com%2Fhome%2Findex
{+base}index	http://example.com/home/index
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
{#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 "path;property" or "path;name=value".

For each defined variable in the variable-list:

- append ";" to the result string;
- if the variable has a simple string value or no explode modifier is given, then:
  - append the variable name (encoded as if it were a literal string) to the result string;
  - if the variable’s value is not empty, append "=" to the result string;
- 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*}	;list=red;list=green;list=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 the variable has a simple string value or no explode modifier is given, 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 the variable has a simple string value or no explode modifier is given, 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. However, it might be possible to craft unanticipated URIs if an attacker is given control over the template or over the variable values within an expression that allows reserved characters in the expansion. In either case, the security considerations are largely determined by who provides the template, who provides the values to use for variables within the template, in what execution context the expansion occurs (client or server), and where the resulting URIs are used.

This specification does not limit where URI Templates might be used. Current implementations exist within server-side development frameworks and within client-side javascript for computed links or forms.

Within frameworks, templates usually act as guides for where data might occur within later (request-time) URIs in client requests. Hence, the security concerns are not in the templates themselves, but rather in how the server extracts and processes the user-provided data within a normal Web request.

Within client-side implementations, a URI Template has many of the same properties as HTML forms, except limited to URI characters and possibly included in HTTP header field values instead of just message body content. Care ought to be taken to ensure that potentially dangerous URI reference strings, such as those beginning with "javascript:", do not appear in the expansion unless both the template and the values are provided by a trusted source.

Other security considerations are the same as those for URIs, as described in Section 7 of [RFC3986].

5. Acknowledgments

The following people made contributions to this specification: Mike Burrows, Michaeljohn Clement, DeWitt Clinton, John Cowan, Stephen Farrell, Robbie Gates, Vijay K. Gurbani, Peter Johanson, Murray S. Kucherawy, James H. Manger, Tom Petch, Marc Portier, Pete Resnick, James Snell, and Jiankang Yao.

6. References

6.1. 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.

[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.

[RFC6365] Hoffman, P. and J. Klensin, "Terminology Used in Internationalization in the IETF", BCP 166, RFC 6365, September 2011.

[UNIV6] The Unicode Consortium, "The Unicode Standard, Version 6.0.0", (Mountain View, CA: The Unicode Consortium, 2011. ISBN 978-1-936213-01-6), <http://www.unicode.org/versions/Unicode6.0.0/>.

[UTR15] Davis, M. and M. Duerst, "Unicode Normalization Forms", Unicode Standard Annex # 15, April 2003, <http://www.unicode.org/unicode/reports/tr15/tr15-23.html>.

6.2. Informative References

[OpenSearch] Clinton, D., "OpenSearch 1.1", Draft 5, December 2011, <http://www.opensearch.org/Specifications/OpenSearch>.

[UPU-S42] Universal Postal Union, "International Postal Address Components and Templates", UPU S42-1, November 2002, <http://www.upu.int/en/activities/addressing/standards.html>.

[WADL] Hadley, M., "Web Application Description Language", World Wide Web Consortium Member Submission SUBM-wadl-20090831, August 2009, <http://www.w3.org/Submission/2009/SUBM-wadl-20090831/>.

[WSDL] Weerawarana, S., Moreau, J., Ryman, A., and R. Chinnici, "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language", World Wide Web Consortium Recommendation REC-wsdl20-20070626, June 2007, <http://www.w3.org/TR/2007/REC-wsdl20-20070626>.

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 non-normative 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 "{", an error is indicated by the presence of a non-literals character other than "{", 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 or key name when no explode modifier is given. The entry for "ifemp" is a string to append to the name if its corresponding value is empty. 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 union of (unreserved / reserved / pct-encoding) will be encoded; and, for both cases, each disallowed character is first encoded as its sequence of octets in UTF-8 and then each such 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 a variable name and optional modifier from the expression by scanning the variable-list until a character not in the varname set is found or the end of the expression is reached.

- If it is the end of the expression and the varname is empty, go back to scan the remainder of the template.

- If it is not the end of the expression and the last character found indicates a modifier ("*" or ":"), remember that modifier. If it is an explode ("*"), scan the next character. If it is a prefix (":"), continue scanning the next one to four characters for the max-length represented as a decimal integer and then, if it is still not the end of the expression, scan the next character.

- If it is not the end of the expression and the last character found is not a comma (","), append "{", the stored operator (if any), the scanned varname and modifier, the remaining 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.

Lookup the value for the scanned variable name, and then

- If the varname is unknown or corresponds to a variable with an undefined value (Section 2.3), 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 this variable’s value is a string, 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 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 pct-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 code point;

  - otherwise, append the value to the result string after pct-encoding any characters that are not in the allow set.

- else if no explode modifier is given, 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; and

  - if this variable’s value is a list, append each defined list member to the result string, after pct-encoding any characters that are not in the allow set, with a comma (",") appended to the result between each defined list member;

  - if this variable’s value is an associative array or any other form of paired (name, value) structure, append each pair with a defined value to the result string as "name,value", after pct-encoding any characters that are not in the allow set, with a comma (",") appended to the result between each defined pair.

- else if an explode modifier is given, then
  - if named is true, then for each defined list member or array (name, value) pair with a defined value, do:
    - if this is not the first defined member/value, append the sep string to the result string;
    - if this is a list, append the varname to the result string using the same encoding process as for literals;
    - if this is a pair, append the name to the result string using the same encoding process as for literals;
    - if the member/value is empty, append the ifemp string to the result string; otherwise, append "=" and the member/value to the result string, after pct-encoding any member/value characters that are not in the allow set.
  - else if named is false, then
    - if this is a list, append each defined list member to the result string, after pct-encoding any characters that are not in the allow set, with the sep string appended to the result between each defined list member.
    - if this is an array of (name, value) pairs, append each pair with a defined value to the result string as "name=value", after pct-encoding any characters that are not in the allow set, with the sep string appended to the result between each defined pair.

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
Adobe Systems Incorporated

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

Marc Hadley
The MITRE Corporation

EMail: mhadley@mitre.org
URI: http://mitre.org/

Mark Nottingham
Rackspace

EMail: mnot@mnot.net
URI: http://www.mnot.net/

David Orchard
Salesforce.com

EMail: orchard@pacificspirit.com
URI: http://www.pacificspirit.com/