Network Working Group
Internet-Draft
Intended status: Informational
Expires: March 27, 2016

R. Fielding
Adobe Systems Incorporated
M. Nottingham

September 24, 2015

The Key HTTP Response Header Field
draft-fielding-http-key-03

Abstract

The 'Key' header field for HTTP responses allows an origin server to describe the secondary cache key ([RFC7234], section 4.1) for a resource, by conveying what is effectively a short algorithm that can be used upon later requests to determine if a stored response is reusable for a given request.

Key has the advantage of avoiding an additional round trip for validation whenever a new request differs slightly, but not significantly, from prior requests.

Key also informs user agents of the request characteristics that might result in different content, which can be useful if the user agent is not sending request header fields in order to reduce the risk of fingerprinting.

Note to Readers

The issues list for this draft can be found at https://github.com/mnot/I-D/labels/key .

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 March 27, 2016.

Table of Contents

1. Introduction 2
1.1. Examples 3
1.2. Notational Conventions 4
2. The "Key" Response Header Field 4
2.1. Relationship with Vary 5
2.2. Calculating a Secondary Cache Key 6
2.2.1. Creating a Header Field Value 8
2.2.2. Failing Parameter Processing 9
2.3. Key Parameters 9
2.3.1. div 9
2.3.2. partition 10
2.3.3. match 11
2.3.4. substr 12
2.3.5. param 13
3. IANA Considerations 15
3.1. Procedure 15
3.2. Registrations 15
4. Security Considerations 15
5. References 16
5.1. Normative References 16
5.2. Informative References 16
Appendix A. Acknowledgements 17
Authors' Addresses 17

1. Introduction

In HTTP caching [RFC7234], the Vary response header field effectively modifies the key used to store and access a response to include information from the request's headers. This "secondary cache key" allows proactive content negotiation [RFC7231] to work with caches.

Vary's operation is generic; it works well when caches understand the semantics of the selecting headers. For example, the Accept-Language request header field has a well-defined syntax for expressing the client's preferences; a cache that understands this header field can select the appropriate response (based upon its Content-Language header field) and serve it to a client, without any knowledge of the underlying resource.

Vary does not work as well when the criteria for selecting a response are specific to the resource. For example, if the nature of the response depends upon the presence or absence of a particular Cookie ([RFC6265]) in a request, Vary doesn't have a mechanism to offer enough fine-grained, resource-specific information to aid a cache's selection of the appropriate response.

This document defines a new response header field, "Key", that allows resources to describe the secondary cache key in a fine-grained, resource-specific manner, leading to improved cache efficiency when responses depend upon such headers.

1.1. Examples

For example, this response header field:

```
Key: cookie;param=_sess;param=ID
```

indicates that the selected response depends upon the "_sess" and "ID" cookie values.

This Key:

```
Key: user-agent;substr=MSIE
```

indicates that there are two possible secondary cache keys for this resource; one for requests whose User-Agent header field contains "MSIE", and another for those that don't.

A more complex example:

```
Key: user-agent;substr=MSIE;Substr="mobile", Cookie;param="ID"
```

indicates that the selected response depends on the presence of two strings in the User-Agent request header field, as well as the value of the "ID" cookie request header field.

1.2. 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 document uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] (including the DQUOTE rule), and the list rule extension defined in [RFC7230], Section 7. It includes by reference the field-name, quoted-string and quoted-pair rules from that document, and the parameter rule from [RFC7231].

2. The "Key" Response Header Field

The "Key" response header field describes the portions of the request that the resource currently uses to select representations.

As such, its semantics are similar to the "Vary" response header field, but it allows more fine-grained description, using "key parameters".

Caches can use this information as part of determining whether a stored response can be used to satisfy a given request. When a cache knows and fully understands the Key header field for a given resource, it MAY ignore the Vary response header field in any stored responses for it.

Additionally, user agents can use Key to discover if additional request header fields might influence the resource's selection of

responses.

The Key field-value is a comma-delimited list of selecting header fields (similar to Vary), with zero to many parameters each, delimited by semicolons. Whitespace is not allowed in the field-value between each field-name and its parameter set.

```
Key = 1#field-name *( ";" parameter )
```

Note that, as per [RFC7231], parameter names are case-insensitive, and parameter values can be double-quoted strings (potentially with ""-escaped characters inside).

The following header fields have the same effect:

```
Vary: Accept-Encoding, Cookie
Key: Accept-Encoding, Cookie
```

However, Key's use of parameters allows:


```
Key: Accept-Encoding, Cookie;param=foo
```

to indicate that the secondary cache key depends upon the Accept-Encoding header field and the "foo" Cookie.

One important difference between Vary and Key is how they are applied. Vary is specified to be specific to the response it occurs within, whereas Key is specific to the resource (as identified by the request URL) it is associated with. The most recent key you receive for a given resource is applicable to all responses from that resource.

This difference allows more efficient implementation (and reflects practices that many caches use in implementing Vary already).

This specification defines a selection of Key parameters to address common use cases such as selection upon individual Cookie header fields, User-Agent substrings and numerical ranges. Future parameters may define further capabilities.

2.1. Relationship with Vary

Origin servers SHOULD still send Vary when using Key, to ensure backwards compatibility.

For example,

```
Vary: User-Agent
Key: User-Agent;substr="mozilla"
```

Note that, in some cases, it may be better to explicitly use "Vary: *" if clients and caches don't have any practical way to use the Vary header field's value. For example,

```
Vary: *
Key: Cookie;param="ID"
```

Except when Vary: * is used, the set of headers used in Key SHOULD reflect the same request header fields as Vary does, even if they don't have parameters. For example,

```
Vary: Accept-Encoding, User-Agent
Key: Accept-Encoding, User-Agent;substr="mozilla"
```

Here, Accept-Encoding is included in Key without parameters; caches MAY treat these as they do values in the Vary header, relying upon knowledge of their generic semantics to select an appropriate response.

2.2. Calculating a Secondary Cache Key

When used by a cache to determine whether a stored response can be used to satisfy a presented request, each field-name in Key identifies a potential request header, just as with the Vary response header field.

However, each of these can have zero to many key parameters that change how the response selection process (as defined in [RFC7234], Section 4.3)) works.

In particular, when a cache fully implements this specification, it creates a secondary cache key for every request by following the instructions in the Key header field, ignoring the Vary header for this purpose.

Then, when a new request is presented, the secondary cache key generated for that request can be compared to the stored one to find the appropriate response, to determine if it can be selected.

To generate a secondary cache key for a given request (including that which is stored with a response) using Key, the following steps are taken:

1. If the Key header field is not present on the most recent cacheable (as per [RFC7234], Section 3)) response seen for the resource, abort this algorithm (i.e., fall back to using Vary to determine the secondary cache key).

2. Let "key_value" be the most recently seen Key header field value for the resource, as the result of Creating a Header Field Value (Section 2.2.1).

3. Let "secondary_key" be an empty string.

4. Create "key_list" by splitting "key_value" on "," characters.

5. For "key_item" in "key_list":

    1. Remove any leading and trailing WSP from "key_item".

    2. If "key_item" does not contain a ";" character, fail parameter processing (Section 2.2.2) and skip to the next "key_item".

    3. Let "field_name" be the string before the first ";" character in "key_item".

    4. Let "field_value" be the result of Creating a Header Field Value (Section 2.2.1) with "field_name" as the "target_field_name" and the request header list as "header_list".

    5. Let "parameters" be the string after the first ";" character in "key_item".

6. Create "param_list" by splitting "parameters" on ";" characters, excepting ";" characters within quoted strings, as per [RFC7230] Section 3.2.6.

7. For "parameter" in "param_list":

    1. If "parameter" does not contain a "=", fail parameter processing (Section 2.2.2) and skip to the next "key_item".

    2. Let "param_name" be the string before the first "=" character in "parameter", case-normalized to lowercase.

    3. If "param_name" does not identify a Key parameter processing algorithm that is implemented, fail parameter processing (Section 2.2.2) and skip to the next "key_item".

    4. Let "param_value" be the string after the first "=" character in "parameter".

    5. If the first and last characters of "param_value" are both DQUOTE:

        1. Remove the first and last characters of "param_value".

        2. Replace quoted-pairs within "param_value" with the octet following the backslash, as per [RFC7230] Section 3.2.6.

    6. If "param_value" does not conform to the syntax defined for it by the parameter definition, fail parameter processing Section 2.2.2 and skip to the next "key_item".

    7. Run the identified processing algorithm on "field_value" with the "param_value", and append the result to "secondary_key". If parameter processing fails Section 2.2.2, skip to the next "key_item".

8. Append a separator character (e.g., NULL) to

"secondary_key".

6. Return "secondary_key".

Note that this specification does not require that exact algorithm to be implemented. However, implementations' observable behavior MUST be identical to running it. This includes parameter processing algorithms; implementations MAY use different internal artefacts for secondary cache keys, as long as the results are the same.

Likewise, while the secondary cache key associated with both stored and presented requests is required to use the most recently seen Key header field for the resource in question, this can be achieved using a variety of implementation strategies, including (but not limited to):

- Generating a new secondary cache key for every stored response associated with the resource upon each request.

- Caching the secondary cache key with the stored request/response pair and re-generating it when the Key header field is observed to change.

- Caching the secondary cache key with the stored response and invalidating the stored response(s) when the Key header field is observed to change.

2.2.1. Creating a Header Field Value

Given a header field name "target_field_name" and "header_list", a list of ("field_name", "field_value") tuples:

1. Let "target_field_values" be an empty list.

2. For each ("field_name", "field_value") tuple in "header_list":

    1. If "field_name" does not match "target_field_name", skip to the next tuple.

    2. Strip leading and trailing WSP from "field_value" and append it to "target_field_values".

3. If "target_field_values" is empty, return an empty string.

4. Return the concatenation of "target_field_values", separating each with "," characters.

2.2.2. Failing Parameter Processing

In some cases, a key parameter cannot determine a secondary cache key corresponding to its nominated header field value. When this happens, Key processing needs to fail safely, so that the correct behavior is observed.

When this happens, implementations MUST either behave as if the Key header was not present, or assure that the nominated header fields being compared match, as per [RFC7234], Section 4.1.

2.3. Key Parameters

A Key parameter associates a name with a specific processing algorithm that takes two inputs; a HTTP header value "header_value" (as described in Section 2.2.1), and "parameter_value", a string that indicates how the identified header should be processed.

The set of key parameters (and their associated processing algorithms) is extensible; see Section 3. This document defines the following key parameters:

2.3.1. div

The "div" parameter normalizes positive integer header values into groups by dividing them by a configured value.

Its value's syntax is:

```
div    = 1*DIGIT
```

To process a set of header fields against a div parameter, follow these steps (or their equivalent):

1. If "parameter_value" is "0", fail parameter processing Section 2.2.2.

2. If "header_value" is the empty string, return "none".

3. If "header_value" contains a ",", remove it and all subsequent characters.

4. Remove all WSP characters from "header_value".

5. If "header_value" does not match the div ABNF rule, fail parameter processing (Section 2.2.2).

6. Return the quotient of "header_value" / "parameter_value" (omitting the modulus).

For example, the Key:

Key: Bar;div=5

indicates that the "Bar" header's field value should be partitioned into groups of 5. Thus, the following field values would be considered the same (because, divided by 5, they all result in 1):

Bar: 1
Bar: 3 , 42
Bar: 4, 1

whereas these would be considered to be in a different group (because, divided by 5, they all result in 2);

Bar: 12
Bar: 10
Bar: 14, 1

2.3.2. partition

The "partition" parameter normalizes positive numeric header values into pre-defined segments.

Its value's syntax is:

```
partition = [ segment ] *( ":" [ segment ] )
segment   = [ 0*DIGIT "." ] 1*DIGIT
```

To process a set of header fields against a partition parameter, follow these steps (or their equivalent):

1. If "header_value" is the empty string, return "none".

2. If "header_value" contains a ",", remove it and all subsequent characters.

3. Remove all WSP characters from "header_value".

4. If "header_value" does not match the segment ABNF rule, fail parameter processing (Section 2.2.2).

5. Let "segment_id" be 0.

6. Create a list "segment_list" by splitting "parameter_value" on ":" characters.

7. For each "segment_value" in "segment_list":

    1. If "header_value" is less than "segment_value" when they are numerically compared, skip to step 7.

    2. Increment "segment_id" by 1.

8. Return "segment_id".

For example, the Key:

Key: Foo;partition=20:30:40

indicates that the "Foo" header's field value should be divided into four segments:

- less than 20
- 20 to less than 30
- 30 to less than 40
- forty or greater

Thus, the following headers would all be normalized to the first segment:

Foo: 1
Foo: 0

```
Foo: 4, 54
Foo: 19.9
```

whereas the following would fall into the second segment:

```
Foo: 20
Foo: 29.999
Foo:  24   , 10
```

2.3.3. match

The "match" parameter is used to determine if an exact value occurs in a list of header values. It is case-sensitive.

Its value's syntax is:


```
match  = ( token / quoted-string )
```

To process a set of header fields against a match parameter, follow these steps (or their equivalent):

1. If "header_value" is the empty string, return "none".
2. Create "header_list" by splitting "header_value" on "," characters.
3. For each "header_item" in "header_list":
    1. Remove leading and trailing WSP characters in "header_item".
    2. If the value of "header_item" is character-for-character identical to "parameter_value", return "1".
4. Return "0".

For example, the Key:

```
Key: Baz;match="charlie"
```

Would return "1" for the following header field values:

```
Baz: charlie
Baz: foo, charlie
Baz: bar, charlie     , abc
```

and "0" for these:

```
Baz: theodore
Baz: joe, sam
Baz: "charlie"
Baz: Charlie
Baz: cha rlie
Baz: charlie2
```

2.3.4. substr

The "substr" parameter is used to determine if a value occurs as a substring of an item in a list of header values. It is case-sensitive.

Its value's syntax is:

```
substr  = ( token / quoted-string )
```


To process a set of header fields against a substr parameter, follow these steps (or their equivalent):

1. If "header_value" is the empty string, return "none".

2. Create "header_list" by splitting "header_value" on "," characters.

3. For each "header_item" in "header_list":

    1. Remove leading and trailing WSP characters in "header_item".

    2. If the value of "parameter_value" is character-for-character present as a substring of "header_value", return "1".

4. Return "0".

For example, the Key:

Key: Abc;substr=bennet

Would return "1" for the following header field values:

Abc: bennet
Abc: foo, bennet
Abc: abennet00
Abc: bar, 99bennet , abc
Abc: "bennet"

and "0" for these:

Abc: theodore
Abc: joe, sam
Abc: Bennet
Abc: Ben net

2.3.5. param

The "param" parameter considers the request header field as a list of key=value parameters, and uses the nominated key's value as the secondary cache key.

Its value's syntax is:

```
param  = ( token / quoted-string )
```

To process a list of header fields against a param parameter, follow these steps (or their equivalent):

1. Let "header_list" be an empty list.

2. Create "header_list_tmp1" by splitting header_value on "," characters.

3. For each "header_item_tmp1" in "header_list_tmp1":

    1. Create "header_list_tmp2" by splitting "header_item_tmp1" on ";" characters.

    2. For each "header_item_tmp2" in "header_list_tmp2":

1. Remove leading and trailing WSP from "header_item_tmp2".

2. Append "header_item_tmp2" to header_list.

4. For each "header_item" in "header_list":

1. If the "=" character does not occur within "header_item", skip to the next "header_item".

2. Let "item_name" be the string occurring before the first "=" character in "header_item".

3. If "item_name" does not case-insensitively match "parameter_value", skip to the next "header_item".

4. Return the string occurring after the first "=" character in "header_item".

5. Return the empty string.

Note that steps 2 and 3 accommodate semicolon-separated values, so that it can be used with the Cookie request header field.

For example, the Key:

Key: Def;param=liam

The following headers would return the string (surrounded in single quotes) indicated:

```
Def: liam=123           // '123'
Def: mno=456            // ''
Def:                    // ''
Def: abc=123; liam=890  // '890'
Def: liam="678"         // '"678"'
```


3. IANA Considerations

This specification defines the HTTP Key Parameter Registry, maintained at http://www.iana.org/assignments/http-parameters/http-

parameters.xhtml#key .

3.1. Procedure

Key Parameter registrations MUST include the following fields:

- Parameter Name: [name]
- Reference: [Pointer to specification text]

Values to be added to this namespace require IETF Review (see Section 4.1 of [RFC5226]) and MUST conform to the purpose of content coding defined in this section.

3.2. Registrations

This specification makes the following entries in the HTTP Key Parameter Registry:

Parameter Name	Reference
div	Section 2.3.1
partition	Section 2.3.2
match	Section 2.3.3
substr	Section 2.3.4
param	Section 2.3.5

4. Security Considerations

Because Key is an alternative to Vary, it is possible for caches to behave differently based upon whether they implement Key. Likewise, because support for any one Key parameter is not required, it is possible for different implementations of Key to behave differently. In both cases, an attacker might be able to exploit these differences.

This risk is mitigated by the requirement to fall back to Vary when unsupported parameters are encountered, coupled with the requirement that servers that use Key also include a relevant Vary header.

An attacker with the ability to inject response headers might be able to perform a cache poisoning attack that tailors a response to a

specific user (e.g., by Keying to a Cookie that's specific to them). While the attack is still possible without Key, the ability to tailor is new.

When implemented, Key might result in a larger number of stored responses for a given resource in caches; this, in turn, might be used to create an attack upon the cache itself. Good cache replacement algorithms and denial of service monitoring in cache implementations are reasonable mitigations against this risk.

5. References

5.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <http://www.rfc-editor.org/info/rfc5234>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <http://www.rfc-editor.org/info/rfc7230>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <http://www.rfc-editor.org/info/rfc7231>.

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <http://www.rfc-editor.org/info/rfc7234>.

5.2. Informative References

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <http://www.rfc-editor.org/info/rfc6265>.

Appendix A. Acknowledgements

Thanks to Ilya Grigorik, Amos Jeffries and Yoav Weiss for their feedback.

Authors' Addresses

Roy T. Fielding
Adobe Systems Incorporated

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

Mark Nottingham

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