HTTP Working Group
INTERNET-DRAFT
<draft-ietf-http-pep-04>
Expires XXXX

D. Connolly, W3C
R. Khare, W3C
H. Frystyk, W3C
14 July, 1997

WD-pep-970714

PEP - an Extension Mechanism for HTTP

W3C Working Draft 26 May 1997

Editor
Henrik Frystyk Nielsen

Authors
Dan Connolly, Rohit Khare, Henrik Frystyk Nielsen

This version:
http://www.w3.org/TR/WD-http-pep-970526

$Id: pep-spec.html,v 1.37 1997/05/29 03:38:59 jigsaw Exp $

Latest Released Version:
http://www.w3.org/TR/WD-http-pep

Previous Version:
http://www.w3.org/TR/WD-http-pep-970428
http://www.w3.org/TR/WD-http-pep-970321
http://www.w3.org/TR/WD-http-pep-970131
http://www.w3.org/TR/WD-http-pep-960820
http://www.w3.org/TR/WD-http-pep-960819
http://www.w3.org/TR/WD-http-pep-960222
http://www.w3.org/TR/WD-http-pep-951122
http://www.w3.org/TR/WD-http-pep-960220

Status of this Document

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

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

To learn the current status of any Internet-Draft, please check the "1id-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories on ftp.is.co.za (Africa), nic.nordu.net (Europe), munnari.oz.au (Pacific Rim), ds.internic.net (US East Coast), or ftp.isi.edu (US West Coast).

This document is also available as a W3C Working Draft. The most recent release is available at http://www.w3.org/TR/WD-http-pep.

Distribution of this document is unlimited. Please send comments to the HTTP working group at `http-wg@cuckoo.hpl.hp.com`. Discussions of the working group are archived at http://www.ics.uci.edu/pub/ietf/http/. The editor maintains background information about PEP at http://www.w3.org/Protocols/PEP/

The contribution of World Wide Web Consortium (W3C) staff time to the HTTP working group is part of the W3C HTTP Activity.

Abstract

HTTP is used increasingly in applications that need more facilities than the standard version of the protocol provides, ranging from distributed authoring, collaboration, and printing, to various remote procedure call mechanisms. The Protocol Extension Protocol (PEP) is an extension mechanism designed to address the tension between private agreement and public specification and to accommodate extension of applications such as HTTP clients, servers, and proxies. The PEP mechanism is designed to associate each extension with a URI[2], and use a few new RFC 822[1] derived header fields to carry the extension identifier and related information between the parties involved in an extended transaction.

This document defines PEP and describes the interactions between PEP and HTTP/1.1[7]. PEP is intended to be compatible with HTTP/1.0[5] inasmuch as HTTP/1.1 is compatible with HTTP/1.0 (see [7], section 19.7). It is proposed that the PEP extension mechanism be included in future versions of HTTP.

The PEP extension mechanism may be applicable to other information exchange not mentioned in this document. It is recommended that readers get acquainted with section 1.4 for a suggested reading of this specification and a list of sections specific for HTTP based applications.

Table of Contents

1 Introduction 3
- 1.1 Requirements 3
- 1.2 Purpose 4
- 1.3 Operational Overview 4
- 1.4 Guide to this Specification 5

2 The PEP Extension Space in HTTP 5

3 Notational Conventions 6
- 3.1 Bag Syntax 6

4 Extension Declarations 6
- 4.1 Mapping Header Fields 7
- 4.2 The Strength of a Declaration 7
- 4.3 End-to-End Extension Declarations 8
- 4.4 Hop-by-Hop Extension Declarations 8

5 Extension Policy Information 8
- 5.1 The Realm of a Policy 9
- 5.2 Policy Expirations 10
- 5.3 Extra Parameters 10
- 5.4 End-to-End Policies 10
- 5.5 Hop-by-Hop Policies 10

6 Publishing an Extension 11

7 Binding HTTP Requests 11
- 7.1 Extending Existing HTTP Methods 12
- 7.2 Adding New HTTP Methods 12

8 HTTP Status Codes 13
- 8.1 420 Bad Extensions 13
- 8.2 421 Bad Mapping 13

9 HTTP Proxy Servers 13
- 9.1 Proxy Servers as End-to-End Recipients 13
  - 9.1.1 Proxy Servers Acting on Behalf of User Agents 13
  - 9.1.2 Proxy Servers Acting on Behalf of Origin Servers 14

9.2 Proxy Servers and Repeated Hop-by-Hop Extensions 14
10 Practical Considerations for HTTP 14
10.1 Interaction with Existing HTTP/1.1 Methods 15
10.2 Interaction with Existing HTTP/1.1 Headers 15
10.3 Server Initiated Extension Declarations 15
11 Security Considerations 16
12 Normative References 16
13 Bibliography: Informative References 17
14 Acknowledgements 17
15 Authors Addresses 18
16 Summary of PEP Interactions 18
17 Examples 19
17.1 Client Queries a Server for DAV 19
17.2 Client Informs a Server about ZipFlate Compression Extension 20
17.3 Server Uses Content-Digest Extension 20
17.4 Server Requires the Client to use Payment Extension 20

1 Introduction

1.1 Requirements

HTTP is a generic request-response protocol, designed to accommodate a variety of applications, from network information exchange and searching to file transfer and repository access to query and forms processing.

Most HTTP transactions are initiated by a user agent issuing a request to be applied to a resource on some origin server, with intermediaries between them in some cases. The origin server replies with a response indicating the result of the transaction.

Semantically, however, an HTTP transaction is between an end user and a principal responsible for the publication of a given resource. The end user is the representative accessing information provided by a publishing party. The publishing party is the one responsible for the service provided at any particular URI, for example, the mapping between the URI and any representation of the resource to which it refers. Exactly who takes the role as end user or publisher is beyond the scope of this document.

HTTP, as is the case for most transaction based information exchange protocols, is used increasingly in applications that need more facilities than the standard version of the protocol provides, from distributed authoring, collaboration and printing, to various remote procedure call mechanisms.

Many extended applications do not require agreement across the whole Internet about the extended facilities; rather, it suffices:

- That conforming peers supporting a particular protocol extension or feature can employ it dynamically with no prior agreement;
- That it is possible for one party having a capability for a new protocol to require that the other party either understand and abide by the new protocol or abort the operation;
- That negotiation of matching capabilities is possible.

The need for extensibility creates a tension between dynamically extensible applications and public, static specifications.

1.2 Purpose

The Protocol Extension Protocol (PEP) is an extension mechanism designed to accommodate dynamic extension of applications such as HTTP clients and servers by software components; and to address the tension between private agreement and public specification. The kind of extensions capable of being introduced by PEP range from:

- extending a single protocol message;
- introducing new encodings;
- initiating HTTP-derived protocols for new applications; to...
- switching to protocols which, once initiated, run independent of the original protocol stack.

This document defines the protocol extension mechanism referred to as "PEP". The PEP design is the result of analyzing a variety of extensions and extension mechanisms in HTTP and HTTP-like protocols, and the motivation behind them.

The specification also describes the interactions between PEP and HTTP/1.1[7] including scoping rules and cache semantics. PEP is intended to be compatible with HTTP/1.0[5] inasmuch as HTTP/1.1 is compatible with HTTP/1.0 (see section 1.4 and 10) and it is proposed that the PEP extension mechanism be included in future versions of HTTP.

1.3 Operational Overview

PEP is intended to be used as follows:

- Some party designs and specifies an extension; the party assigns the extension an identifier, which is a URI, and makes one or more representations of the extension available at that address (see section 6).
- A party using a PEP compliant agent with an implementation of the extension wishes to use it; the agent declares the use of the extension by referencing its URI in a PEP extension declaration (see section 4).
- Information about extensions can be passed between agents including information of where they can be used and under what conditions (see section 5).

If an extension becomes ubiquitous, it may be incorporated into a new version of the base protocol, hence transitioning from dynamic extension to static specification. In this case, applications can refer to the new version of the base protocol instead of the PEP extension (see section 6).

PEP extension declarations are characterized by the following properties:

- They link features introduced by the extension to the URI identifying the extension, potentially allowing a recipient to interpret the message correctly with no prior agreement.
- They contain a *strength* and a *scope* allowing the sender to define the appropriate action to be taken by the recipient even if it does not understand the semantics of the extension.
- Any agent can generate declarations independent of other agents

The advantage of including the extension identifier is that, at the cost of some extra bytes to spell out the URI, the use of a central registry of extension names is avoided. PEP can also be used to extend applications to support centrally registered extensions, assuming a URI is published as part of the registration (see section 6).

The PEP mechanism is designed to accommodate but does not require dynamic extension of clients, servers, and proxies by software components as follows:

- Clients and servers could be implemented with software component interfaces that allow dynamic installation of extension facilities.
- An implementation compatible with a software component interface supported by the agent could be made available at the URI identifying the extension.

- An agent receiving a message referring to an extension not known by the agent could dereference the extension's identifier and dynamically load support for the extended facility.

The representation and implementation of dynamic extensible software component interfaces is outside the scope of this specification.

1.4 Guide to this Specification

This specification is organized as follows: Section 2 describes how PEP fits into HTTP. This is not required reading but may further the understanding of the specification. Section 3 is an overview of the notational conventions used throughout the specification.

Section 4, 5, and 6 is the core part of the specification describing the generic PEP extension mechanism. Section 7, 8, 9, and 10 describe the interactions between PEP and HTTP/1.1[7].

The generic PEP extension mechanism may be applicable to other information exchange protocols. Such mappings, however, are outside the scope of this specification.

2 The PEP Extension Space in HTTP

PEP is designed to support dynamic extensibility of HTTP methods, headers, and status codes. Before describing in detail how PEP does this, it is constructive to have a look at how methods, headers, and status codes behave in HTTP:

Methods

The method token in an HTTP request indicates the method to be performed on the resource identified by the `Request-URI`. Methods need a priori agreement of semantics and can not be extended dynamically. If an HTTP server does not know a method, it must report an error message (see [7] section 5.1.1). A limitation of the method space is that a request can only contain a single method. Hence, it is not possible to support multiple, simultaneous extensions unless having a multiplicity of methods.

Status Codes

The status code element is a 3-digit integer result code of the attempt to understand and satisfy the request. Status codes are like method tokens in that there can only be a single status code in a response. However, status codes are somewhat easier to extend, as unknown status codes must be treated as the x00 code of that class (see [7] section 6.1.1). For example, a new status code, 223 (My New Code) would default to 200 (OK).

Headers

Header fields can be used to pass information about any of the parties involved in the transaction, the transaction itself, or the resource identified by the Request-URI. The advantage of headers is that the header space is relatively open compared to that of methods and status codes. New headers can be introduced and must be ignored if the recipient does not recognize the header without affecting the outcome of the transaction (see [7] section 7.1).

In order to achieve the desired flexibility, PEP is designed to use the header space for describing extensions and not directly HTTP methods or status codes. Instead, PEP introduces a placeholder in the method space and status code space respectively guaranteeing that all interactions with existing HTTP applications perform according to the PEP specification. The two placeholders are:

- a special `PEP` method and a `PEP-` method prefix which indicates that a request contains one or more PEP extensions that must be adhered to or the transaction aborted (see section 7);
- a special status code 420 (Bad Extensions) that indicates that the policy for accessing the resource was not met and that further information can be found in the response for diagnosing the problem (see section 8.1).

These two placeholders allow for multiple PEP extensions to be deployed simultaneously without overloading the method space or the status code space.

3 Notational Conventions

This specification uses the same notational conventions and basic parsing constructs as RFC 2068[7]. In particular the BNF constructs "`token`", "`quoted-string`", "`field-name`", "`URI`", and "`delta-seconds`" in this document are to be interpreted as described in RFC 2068[7].

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 RFC 2119[9].

PEP does not rely on particular features defined in URLs that cannot potentially be expressed using URNs (see section 6). Therefore, the more generic term URI[2] is used throughout the specification.

3.1 Bag Syntax

The bag element is a recursive structure that uses braces ("{" and "}") to delimit attribute-value pairs that may consist of tokens, quoted-strings, URIs and recursively defined bags. The BNF for the bag syntax is as follows:

```
bag             = "{" bagname *bagitem "}"
bagname         = token
bagitem         = bag
                | token
                | quoted-string
```

The bag semantics are defined by its context and the bag name. The value of a quoted string may be a URI in some cases. Unless explicitly defined otherwise, all tokens within a bag are case-insensitive. Comments as defined by RFC 822[1] indicated by surrounding the comment text with parentheses MUST NOT be used within a bag construct.

4 Extension Declarations

Extension declaration bags are used to indicate the PEP extensions that have been applied to a message. The grammar for an extension declaration is as follows:

```
ext-decl        = "{" req-ext-attr *opt-ext-attr "}"

req-ext-attr    = map
opt-ext-attr    = strength
                | attribute-ext

map             = "{" "map" <"> URI <"> #(header-wildcard) "}"
strength        = "{" "strength" ( "must" | "may" ) "}"
attribute-ext   = bag

header-wildcard = field-name [ ( wildcard ) | ( "=" field-name ) ]
wildcard        = "*"
```

The `map` attribute bag contains the URI identifying the extension and a list of any header field names introduced by the extension (see section 4.1 and 6). If the extension identifier is relative, it is interpreted relative to the base URI of the message as defined by RFC 1808[4].

The `strength` attribute bag indicates whether the recipient MUST or MAY obey the semantics given by the extension or report an error (see section 4.2).

An extension declaration bag (`ext-decl`) can be extended through the use of one or more `attribute-ext` bags. Unrecognized `attribute-ext` bags SHOULD be ignored and MUST NOT be removed by proxies when forwarding the extension declaration (see section 9).

Extension declarations can either be hop-by-hop or end-to-end (see [7], section 13.5.1) depending on the scope of the declaration (see section 4.3 and 4.4). End-to-end declarations MUST be transmitted to the ultimate recipient of the extension declaration. Hop-by-hop declarations are meaningful only for a single transport-level connection.

4.1 Mapping Header Fields

The `header-wildcard` in a `map` attribute bag indicates that any header `field-name` in the message that matches the `header-wildcard` is introduced by the extension declaration. A `header-wildcard` contains a `field-name,` optionally assigned a value or terminated with a `wildcard`.

If a `field-name` is assigned a value then that value is the name of the `header-field` appearing in the message instead of the `field-name` itself. This can be used to distinguish header fields introduced by multiple extension declarations of the same extension in a message.

A `field-name` directly followed by a `wildcard` indicates that all header fields in the message matching the `field-name` value using prefix-matching are introduced by this extension declaration. Linear white space (LWS) MUST NOT be used between the `field-name` and the `wildcard`.

Examples of header-wildcard values are

```
Path, Destination
Path-*
Path=P1, Destination=D1
```

The last example indicates that the extension is using the header fields "Path" and "Destination" but that they in the message are written as "`P1`" and "`D1`". All `header-wildcard` values are case-insensitive including when taking part in prefix-matching. For example, the "`Path-*`" example above matches header fields like "Path-Discovery", "path-length", etc.

Agents SHOULD NOT overload well-known or widely deployed header fields with new semantics unless the new semantics are a superset of the existing semantics so that the header fields still can be interpreted according to the old semantics.

Agents SHOULD NOT reuse already mapped header fields in the same message. If a header field is mapped by multiple extension declarations in the same message, the recipient SHOULD report an error (see section 8.2).

Proxies adding extension declarations to a message MUST make sure that any header fields introduced do not conflict with already mapped header fields (see section 9.2).

4.2 The Strength of a Declaration

The `strength` attribute bag can be used to specify the actions to be taken by the ultimate recipient of the extension declaration. The `strength` value can indicate that

1. the recipient MUST obey the extension declaration or report an error; or
2. the recipient MAY obey the extension declaration or ignore it altogether.

If the strength is "`must`", the ultimate recipient MUST consult and adhere to the rules given by the extension when processing the message or report an error (see section 7 and 8.1).

If the strength is "`may`" the ultimate recipient of the extension MAY consult and adhere to the rules given by the extension when processing the message, or ignore the extension declaration completely. An agent may not be able to distinguish whether the ultimate recipient does not understand an extension referred to by an extension declaration of strength "`may`" or simply ignores the extension declaration.

If no strength attribute is present, the default strength is "`may`".

Not accepting or ignoring an extension declaration is different from not accepting a mapping of header `field-name`s introduced by the `map` attribute bag. If the ultimate recipient cannot accept a mapping, for example if a `field-name` is already mapped by another extension declaration, it SHOULD report an error (see section 8.2).

4.3 End-to-End Extension Declarations

End-to-end declarations MUST be transmitted to the ultimate recipient of the declaration. The `PEP` header field is an end-to-end header field and is defined as follows:

```
pep             = "PEP" ":" 1#ext-decl
```

For example

```
GET / HTTP/1.1
Host: some.host
PEP: {{map "http://www.w3.org/PEP/DAV"}}
```

If multiple end-to-end extensions are declared in the same message, the declarations MUST be listed in the order in which they were applied to the message.

Proxies MAY under certain conditions act as the ultimate recipient of declarations on behalf of user agents and origin servers (see section 9.1).

4.4 Hop-by-Hop Extension Declarations

Hop-by-hop extension declarations are meaningful only for a single transport-level connection. The `C-PEP` header field is a hop-by-hop header field and MUST NOT be communicated by proxies over further connections. The `C-PEP` header has the following grammar:

```
c-pep           = "C-PEP" ":" 1#ext-decl
```

For example

```
GET / HTTP/1.1
Host: some.host
Credentials: Henrik
C-PEP: {{map "http://www.w3.org/PEP/ProxyAuth" Credentials}}
Connection: C-PEP, Credentials
```

In HTTP, the `C-PEP` header field MUST be protected by a `Connection` header by including `C-PEP` as a `Connection` header directive. The directive MUST be handled according to the HTTP/1.1 specification of the `Connection` header (see section 10.2 and [7], section 14.10).

An agent MUST NOT send the `C-PEP` header field to an HTTP/1.0 proxy as it does not obey the HTTP/1.1 rules for parsing the `Connection` header field (see [7], section 19.7.1).

If multiple hop-by-hop extensions are declared in the same message, the extension declarations MUST be listed in the order in which they were applied. Hop-by-hop `C-PEP` declarations MUST be processed before any end-to-end `PEP` declarations.

5 Extension Policy Information

Extension Policy bags are used to indicate the extensions that may be applied to a message. Extension policies differ from extension declarations in that the latter is information about which extensions *have been* applied to a message. An extension policy is defined as follows:

```
policy-decl     = "{" req-pol-attr *opt-pol-attr "}"
```

```
req-pol-attr    = id
opt-pol-attr    = for
                | max-age
                | parameters
                | strength
                | attribute-ext

id              = "{" "id" <"> URI <"> "}"
for             = "{" "for" #URI-wildcard "}"
max-age         = "{" "max-age" delta-seconds "}"
parameters      = "{" "params" *bagitem "}"

URI-wildcard    = <"> URI <"> [ wildcard ]
```

The `id` attribute specifies the URI identifying the extension (see section 6). If the extension identifier is relative, it is interpreted relative to the base URI of the message as defined by RFC 1808[4].

The `for` attribute bag specifies which resources, the policy is intended for (see section 5.1) and the `max-age` attribute bag when the information should be considered stale (see section 5.2). The `params` attribute bag can be used to pass additional information about the extension policy (see section 5.3).

The `strength` attribute indicates whether the policy is a requirement or optional for the resource(s) for which it applies (see section 4.2).

An extension policy bag (`policy-decl`) can be extended through the use of one or more `attribute-ext` bags. Unrecognized `attribute-ext` bags SHOULD be ignored and MUST NOT be removed by proxies when forwarding the extension policy (see section 9).

Extension policies can either be hop-by-hop or end-to-end policies (see [7], section 13.5.1) depending on the scope (see section 5.4 and 5.5). End-to-end policies MUST be transmitted to the ultimate recipient of the extension policy. Hop-by-hop policies are meaningful only for a single transport-level connection.

Note: It is expected that extension policies will be integrated with other metadata initiatives like the PICS-NG initiative[11], for example.

5.1 The Realm of a Policy

The `for` attribute bag specifies the resource(s) identified by URI(s) to which the policy applies. A URI followed by a LWS and a wildcard ("*") represents the set of URIs that contains the given URI using prefix matching. A URI with no wildcard means that URI only.

Examples of `URI-wildcard`s are

```
{for "/" *}
{for "http://www.w3.org/pub/" *}
{for "secret/Overview.html"}
```

An empty `for` attribute bag (no `bagitem`s included) indicates that the policy is not applied to any resource. If no `for` attribute bag is present, the default value is the `Request-URI`.

A realm can include any number of resources but note that a single wildcard "*" is not a valid `URI-wildcard` value.

5.2 Policy Expirations

The `max-age` attribute bag can be used to specify a date/time after which the recipient SHOULD consider the policy stale. The `max-age` attribute bag value indicates that the information should no longer be used if the age is greater than the specified time in seconds (see [7] section 13.2.3 for how to calculate the age). A `max-age` attribute bag cannot be used to force the recipient to discard the policy information; its semantics apply only to the caching mechanism of policy information.

5.3 Extra Parameters

The params attribute bag can be used to include additional information about the extension or modifiers on the use of the extension. The `params` values may or may not be case-sensitive, depending on the semantics of the parameter name. The params attribute bag is defined as a generic bag structure, which may be nested. No default parameters are defined.

Note: PEP implementations should pass any parameters to the module or modules handling the particular extension as this may have impact the use of the extension.

5.4 End-to-End Policies

End-to-end policies MUST be transmitted to the ultimate recipient of a message. The `PEP-Info` header field is an end-to-end header and is defines as follows:

```
pep-info         = "PEP-Info" ":" 1#policy-decl
```

For example

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 412
PEP-Info: {{id "http://some.org/payment-extension"}
          {for "/cgi-bin/buy" *}
          {strength must}}

<!doctype html public "-//W3C//DTD HTML 3.2//EN" >
<html> ...
```

Proxies MAY under certain conditions act as the ultimate recipients of extension policies on behalf of user agents and origin servers (see section 9.1).

5.5 Hop-by-Hop Policies

Hop-by-hop policies are meaningful only for a single transport-level connection. The `C-PEP-Info` header field is a hop-by-hop header field and MUST NOT be communicated by proxies over further connections. The `C-PEP-Info` header has the following grammar:

```
c-pep-info       = "C-PEP-Info" ":" 1#policy-decl
```

For example

```
HTTP/1.1 420 Bad Extensions
C-PEP-Info: {{id "http://some.org/provide-stats"}
             {for "/" *}}
Connection: C-PEP-Info
...
```

In HTTP, the `C-PEP-Info` header field MUST be protected by a `Connection` header by including `C-PEP-Info` as a `Connection` header directive. The directive MUST be handled according to the HTTP/1.1 specification of the `Connection` header (see section 10.2 and [7], section 14.10).

An agent MUST NOT send the `C-PEP-Info` header field to an HTTP/1.0 proxy as it does not obey the HTTP/1.1 rules for parsing the `Connection` header field (see [7], section 19.7.1).

6 Publishing an Extension

While the protocol extension definition should be published at the address of the extension identifier, this is not a requirement of this specification. The only absolute requirement is that distinct names be used for distinct semantics. For example, one way to achieve this is to use a *mid*, *cid*, or *uuid* URI. The association between the extension identifier and the specification might be made by distributing a specification, which references the extension identifier.

It is strongly recommended that the integrity and persistence of the extension identifier is maintained and kept unquestioned throughout the lifetime of the extension. Care should be taken not to distribute conflicting specifications that reference the same name. Even when a URI is used to publish extension specifications, care must be taken that the specification made available at that address does not change significantly over time. One agent may associate the identifier with the old semantics, and another might associate it with the new semantics.

The extension definition may be made available in different representations ranging from

- a human-readable specification defining the extension semantics,
- downloadable code which implements the semantics defined by the extension,
- a formal interface description provided by the extension, to
- a machine-readable specification defining the extension semantics.

For example, a software component that implements the specification may reside at the same address as a human-readable specification (distinguished by content negotiation). The human-readable representation serves to document the extension and encourage deployment, while the software component allows clients and servers to be dynamically extended.

7 Binding HTTP Requests

An HTTP request is called a "binding" request if it includes at least one PEP extension declaration of strength "`must`". An HTTP server MUST NOT return a 2xx `status-code` without obeying all extension declaration(s) of strength "`must`" in a binding request. This section describes how the binding request mechanism in PEP interacts with existing HTTP applications.

In [7], section 7.1, it is stated that "Unrecognized header fields SHOULD be ignored by the recipient and MUST be forwarded by proxies." Hence, using a `PEP` or a `C-PEP` extension declaration is not sufficient to evoke the correct behavior from existing HTTP agents in a binding request. However, in [7], section 5.1.1, Method, it is said that "Servers SHOULD return 501 (Not Implemented) if the method is unrecognized or not implemented by the server." A similar statement is made in [5], section 9.5. It is therefore safe to assume that using the method name will produce the correct result from existing HTTP servers and proxies.

PEP uses the HTTP request method name to extend existing HTTP/1.1 methods and to introduce new methods (see section 1.3). In both cases, a binding HTTP request invalidates cached entries as described in [7], section 13.10. Responses to binding requests are not cachable.

7.1 Extending Existing HTTP Methods

The method name of all HTTP/1.1 requests containing a PEP extension declaration of strength "`must`" that semantically extends that method MUST be prefixed by "`PEP-`" (see section 10.1). For example, a client might express the binding rights-management constraints in an HTTP PUT request as follows:

```
PEP-PUT /a-resource HTTP/1.1
PEP: {{map "http://www.w3.org/PEP/rights-management" copyright, contributions}
      {strength must}}
copyright: http://www.w3.org/COPYRIGHT.html
contributions: http://www.w3.org/PATCHES.html
Host: www.w3.org
Content-Length: 1203
Content-Type: text/html

<!doctype html ...
```

The ultimate recipient of a binding HTTP request with the "`PEP-`" prefix on the method name MUST process the request by performing the following actions in the order they occur:

1. Identify all extension declarations (both hop-by-hop and end-to-end) of strength "`must`"; the server MAY ignore declarations of strength "`may`" without affecting the result of the transaction;
2. Evaluate and process the extensions identified in 1) in the order they were declared (see section 4.3 and 4.4) or if the extension declarations do not match the policy for accessing the resource then respond with a 420 (Bad Extensions) `status-code` (see section 8.1);
3. Strip the "`PEP-`" prefix from the method name and process the reminder of the request according to the semantics of the existing HTTP/1.1 method name as defined in [7].

The "`PEP-`" prefix is reserved by PEP and MUST NOT be used by other HTTP extensions.

7.2 Adding New HTTP Methods

The PEP method can be used for all PEP extension declarations of strength "`must`" that do not naturally extend existing HTTP/1.1 methods. Such methods can be address space manipulation extensions like MOVE and COPY, for example:

```
PEP /source.html HTTP/1.1
PEP: {{map "http"//www.w3.org/DAV/MOVE" Destination}
     {strength must}}
Destination: destination.html
Host: some.host
```

The `PEP` method indicates that the semantics of this request are defined by one or more PEP extension declarations of strength "`must`" included in the request. The PEP method does not have any HTTP message semantics besides being a placeholder for PEP extension declarations and hence all other semantics MUST be defined by the declaration(s) included in the request.

The ultimate recipient of a `PEP` request MUST process the request by doing the following:

1. Identify all extension declarations (both hop-by-hop and end-to-end) of strength "must"; the server MAY ignore declarations of strength "`may`" without affecting the result of the transaction;
2. Evaluate and process the extensions identified in 1) in the order they were declared (see section 4.3 and 4.4) or if the extension declarations do not match the policy for accessing the resource then respond with a 420 (Bad Extensions) `status-code` (see section 8.1);

A successful response SHOULD be 200 (OK) if the response includes an entity, 202 (Accepted) if the action has not yet been enacted, or 204 (No Content) if the response is OK but does not include an entity. If no extension declarations have strength "`must`", the response SHOULD be 400 (Bad Request).

The `PEP` method is reserved by PEP and MUST NOT be used by other HTTP extensions.

8 HTTP Status Codes

PEP introduces two new status codes in addition to the ones already defined by HTTP/1.1[7]. Each `Status-Code` is described below, including a description the metainformation required in the response.

8.1 420 Bad Extensions

The policy for accessing the resource has not been met in the request. The response MUST include a `PEP-Info` or a `C-PEP-Info` header field specifying the extensions required by the publishing party for accessing the resource. The server MAY use the `for` attribute bag to indicate whether the policy applies to other resources.

The client MAY repeat the request using the appropriate extension(s). If the initial request already included the extensions requested in the 420 response, then the response indicates that access has been refused for those extension declarations.

If the 420 response contains the same set of extension policies as the prior response, then the client MAY present any entity included in the response to the user, since that entity may include relevant diagnostic information.

Implementers may note the similarity to the way authentication challenges are issued with the 401 (Unauthorized) `status-code` (see [7], section 10.4.2)

8.2 421 Bad Mapping

The mappings indicated by one or more `map` attribute bags in the request were refused. The client MAY repeat the request using a new set of mappings if it believes that it can find a unique set of header fields for which the transaction will succeed.

9 HTTP Proxy Servers

This section describes the role of caching and non-caching proxies and how they interact with PEP extensions. Normally, the ultimate recipient of an end-to-end extension declaration or an end-to-end extension policy is an origin server or a user agent.

In this case, a proxy MUST forward all components of the extension, including declarations, policies, headers, and any methods and status codes defined by this specification.

In other cases, however, intermediate caching and non-caching proxies MAY be authorized to act on behalf of origin servers and/or user agents. How such an agreement is reached between a party representing the proxy and the party on which behalf it can act, is outside the scope of PEP, but for example, the parties may be within the same trust domain.

9.1 Proxy Servers as End-to-End Recipients

9.1.1 Proxy Servers Acting on Behalf of User Agents

In case a proxy is authorized to act as the ultimate recipient on behalf of its proxy clients on end-to-end extensions, it MUST obey the following rules:

- The proxy SHOULD remove the extension declaration(s) and any header fields that are part of these declaration(s) on which it can act authoritatively before forwarding the response to the proxy client;
- it SHOULD issue extension policies for the extensions on which it can act authoritatively as if it was a user agent;
- if an extension declaration added by an HTTP proxy is of strength "`must`", the proxy MUST either prepend the "`PEP-`" method name prefix or use the `PEP` method instead of the method name used in the proxy client request, before forwarding the response to the origin server (see section 7.1).

An example of a proxy acting on behalf of one or more user agents is an elementary school wishing to enforce a certain policy for accessing information on the Internet. The local school proxy can act authoritatively as a retrieval filter on behalf of the pupils instead of having distributed filtering enabled on each of the user agents using the client.

9.1.2 Proxy Servers Acting on Behalf of Origin Servers

In case a proxy is authorized to act as the ultimate recipient on behalf of an origin server on end-to-end extensions, it MUST obey the following rules:

- The proxy SHOULD remove the extension declaration(s) and any header fields that are part of these declaration(s) on which it can act authoritatively before forwarding the request to the origin server;
- it SHOULD issue extension policies for the extensions on which it can act authoritatively as if it was an origin server;
- if an extension declaration added by an HTTP proxy is of strength "`must`" and there are no other extension declarations of strength "`must`" in the request, the proxy MUST remove any "`PEP-`" method name prefix before forwarding the request to the origin server (see section 7.1);
- if a request uses the `PEP` method, the proxy MUST NOT forward the request to the origin server unless the communication between the proxy and the origin server can be completed using an existing HTTP/1.1 method.

An example of a proxy acting on behalf of an origin server is a corporation having a subscription on an on-line journal. All access to the origin server goes through the corporate firewall that runs a caching proxy server. The organization reports to the publisher of the journal on a m onthly basis at which point the subscription is re-evaluated. In the day-to-day access, the proxy has the authority to act authoritatively on behalf of the origin server registering usage of the journal.

9.2 Proxy Servers and Repeated Hop-by-Hop Extensions

If a PEP extension is to be used on parts of a message path, including user agents, origin servers, and proxies, not covered by end-to-end or hop-by-hop extension declarations, it can be defined as a repeated hop-by-hop extension. This can for example be the case for a proxy extension applied to a subset of proxies in a message path.

It is for the designer of the extension to decide whether it can repeat itself on a hop-by-hop basis. In other words, any scope more complex than a hop-by-hop or an end-to-end scope is a property of the extension and is transparent to PEP.

10 Practical Considerations for HTTP

This section describes some practical considerations intended for PEP extended HTTP applications. The issues described may not apply to other information retrieval protocols.

10.1 Interaction with Existing HTTP/1.1 Methods

Extension designers should consider whether an extension is to work with existing HTTP/1.1 methods using the "`PEP-`" method token prefix or with the `PEP` method (see section 7.1 and 7.2). This specification does not provide an absolute rule for when to use the `PEP` method compared to the "`PEP-`" method token prefix except that the "`PEP-`" method token prefix is required in situations where intermediate proxies may act authoritatively on behalf of origin servers or user agents (see section 9.1.1 and 9.1.2). In case the extension can be used with existing methods then it should be considered whether the extension can be used with any of the existing HTTP/1.1 methods or only a subset of them.

Some HTTP/1.1 methods follow the convention of being "safe" to the requester meaning that they should never have the significance of taking an action other than retrieval (see [7], section 9.1). This is for example the case of the `GET` and the `HEAD` method. As PEP extension declarations of strength "`must`" explicitly modify or replace the method name, existing HTTP applications will never be able to mistake a PEP enabled message for any of the existing HTTP messages indicated as being safe.

Some extensions may have the property of "idempotence" in that (aside from error or expiration issues) the side effects of N > 0 identical extended requests is the same as for a single extended request. If this is *not* the case for a PEP extension then it should consider whether it wants to 1) disable itself on repeated requests, and/or 2) inform a user about the behavior of repeating identical requests with this extension.

10.2 Interaction with Existing HTTP/1.1 Headers

Designers of extensions to be used within the HTTP messaging model should consider the interaction with existing HTTP/1.1 headers. Especially, it should be noted that PEP is designed to be compatible with HTTP/1.0[5] inasmuch as HTTP/1.1 is compatible with HTTP/1.0 (see [7], section 19.7).

The `Connection` header as described in [7], section 14.10, allows the sender to specify options that are desired for that particular transport connection only. All PEP hop-by-hop extension declarations and policies along with any header fields introduced by extension declarations MUST be included as `Connection` header directives. PEP applications MUST NOT send any hop-by-hop extension declarations or policies to HTTP/1.0 proxies as they do not obey the rules of HTTP/1.1 for parsing the `Connection` header field (see also [7], section 19.7.1).

The `Upgrade` header, [7], section 14.41, allows the client to specify what additional communication protocols it supports and would like to use if the server finds it appropriate to switch protocols. PEP provides the same functionality but without the need for a central registry of protocol names. PEP compliant agents MAY use the 101 (Switching Protocols) status code to switch to HTTP-based protocols and protocols, which once initiated, run completely independently of HTTP.

The content coding values in the `Content-Encoding` header as described in [7], section 14.12, indicate an encoding transformation that has been applied to an entity. PEP provides the same functionality but without the need for a central registry of content codings. As both content codings and PEP extension declarations are ordered, using both may lead to ambiguous situations. Simultaneous use of both mechanisms is therefore strongly discouraged.

An origin server can explicitly prevent intermediaries from applying a `Content-Encoding` to a resource by using the `no-transform Cache-Control` directive (see [7], section 14.9.4).

10.3 Server Initiated Extension Declarations

PEP extension declarations can be generated by servers as well as clients. If a PEP compliant server sends a response with an extension declaration referring to an extension that modifies the message in such a way that the message can not be decoded without using the extension, and the corresponding request was either

1. received from a client whose version is lower than HTTP/1.1, or

2. received with a `Via` header field indicating that it was forwarded by a proxy whose version is lower than HTTP/1.1,

and the response does not already include an `Expires` header, then the sender SHOULD include an `Expires` header field whose `field-value` is identical to the `field-value` of its `Date` header field(see [7], section 14.12). If all agents in the message path are HTTP/1.1, then the sender SHOULD use the `Cache-Control` header field instead of the `Expires` header field to mark the entity uncachable.

11 Security Considerations

- The `for` parameter allows one party to give information about the extensions used by another party's resources. The parties may provide resources on different servers, or at different addresses on the same server. While distinguishing between the parties responsible for different resources at the same server may be infeasible, clients SHOULD ignore information given by one server about another unless they have reason to trust it, or reason to believe that trusting it will have no significant negative consequences.
- Dynamic installation of extension facilities as described in the introduction involves software written by one party (the provider of the implementation) to be executed under the authority of another (the party operating the host software). This opens the host party to a variety of "Trojan horse" attacks by the provider, or a malicious third party that forges implementations under a provider's name. See, for example RFC2046[6], section 4.5.2 for a discussion of these risks.

12 Normative References

[1] D. H. Crocker. *"Standard for the Format of ARPA Internet Text Messages"*, STD 11, RFC 822, UDEL, August 1982

[2] T. Berners-Lee, *"Universal Resource Identifiers in WWW. A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web"*, RFC 1630, CERN, June 1994.

[3] T. Berners-Lee, L. Masinter, M. McCahill. *"Uniform Resource Locators (URL)"* RFC 1738, CERN, Xerox PARC, University of Minnesota, December 1994.

[4] R. Fielding, *"Relative Uniform Resource Locators"*, RFC 1808, UC Irvine, June 1995.

[5] T. Berners-Lee, R. Fielding, H. Frystyk, *"Hypertext Transfer Protocol -- HTTP/1.0"*, RFC 1945, W3C/MIT, UC Irvine, W3C/MIT, May 1996.

[6] N. Freed, N. Borenstein, *"Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types"*, RFC 2046, Innosoft, First Virtual, November 1996.

[7] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, T. Berners-Lee, *"Hypertext Transfer Protocol -- HTTP/1.1"*, RFC 2068, U.C. Irvine, DEC W3C/MIT, DEC, W3C/MIT, W3C/MIT, January 1997

[8] D. Kristol, L. Montulli, *"HTTP State Management Mechanism"*, RFC 2109, Bell Laboratories Lucent Technologies, Netscape Communications, February 1997

[9] S. Bradner, *"Key words for use in RFCs to Indicate Requirement Levels"*, RFC 2119, Harvard University, March 1997

[10] J. C. Mogul, R. Fielding, J. Gettys, H. Frystyk, *"Use and interpretation of HTTP version numbers"*, Internet Draft RFC 2145, DEC, U.C. Irvine, DEC W3C/MIT, W3C/MIT, HTTP Working Group, May, 1997.

[11] O. Lassila, *"PICS-NG Metadata Model and Label Syntax"*, W3C Working draft, May 1997. This is work in progress

[12] H. Schulzrinne, A. Rao, R. Lanphier, *"Real Time Streaming Protocol (RTSP)"*, Internet Draft draft-ietf-mmusic-rtsp-02, Columbia U./Netscape/Progressive Networks, March 1997. This is work in progress

[13] T. Berners-Lee, R. Fielding, L. Masinter, *"Uniform Resource Locators (URL)"*, Internet Draft draft-fielding-url-syntax-05, W3C/MIT, U.C. Irvine, Xerox Corporation, May 1997. This is work in progress

13 Bibliography: Informative References

[14] D. Eastlake, *"Universal Payment Preamble"*, Internet Draft draft-eastlake-universal-payment-03, CyberCash, March 1996. This is work in progress.

[15] D. M. Kristol, *"A Proposed Extension Mechanism for HTTP"*, Internet Draft draft-kristokl-http-extensions-00, January 1995. Document expired.

[16] JEPI, *"Selecting Payment Mechanisms Over HTTP"*, Internet Draft draft-khare-jepi-uppflow-00, W3C, August 1996. Document expired.

[17] J. Miller et al, *"PICS Label Syntax and Communication Protocols (Version 1.1)"*, W3C Recommendation REC-PICS-labels, W3C, 31 October 1996

[18] Y. Goland et al, *"Extensions for Distributed Authoring and Versioning"*, Internet Draft, draft-jensen-webdav-ext-01, 26 March 1997. This is work in progress.

[19] N. Borenstein, *"A User Agent Configuration Mechanism For Multimedia Mail Format Information"*, RFC 1524 pp. 12, Bellcore, September 1993.

[20] J. Klensin, N. Freed, M. Rose, E. Stefferud, and D. Crocker. *"SMTP Service Extensions"*, RFC 1869, MCI, Innosoft, Dover Beach Consulting, Network Management Associates, Brandenburg Consulting, November 1995.

[21] D. Robinson, *"The WWW Common Gateway Interface Version 1.1"*, Internet Draft draft-robinson-www-interface-01, February 1996. Document expired.

[22] A. Baird-Smith, *"Jigsaw: An object oriented server"*, W3C Note, June 1996

[23] H. Frystyk, *"Libwww Architecture"*, December 1996

[24] R. Thau, *"Design considerations for the Apache Server API"*, Fifth International World Wide Web Conference, May 6-10, 1996, Paris, France

[25] Netscape Corporation, *"The Netscape Server API"*

[26] Microsoft Corporation, *"Internet Server API Documentation"*

[27] Open Market, Inc, *"FastCGI - Restoring All CGI's Good Properties -- and Then Some"*

[28] Spyglass, *"Spyglass MicroServer Application Development Interface"*

[29] J. Franks, *"WN - a Server for the HTTP"*

[30] Roxen, *"Introduction to Roxen Challenger"*

14 Acknowledgements

The PEP protocol is the product of a substantial amount of investigation and collaboration. Dave Kristol did some of the first writing on HTTP extension mechanisms[15]. Jim Miller and Dave Raggett sketched out an initial design, which Rohit Khare wrote up in a number of drafts. Tim Berners-Lee, Anselm Baird-Smith, Eric Prud'hommeaux, Paul Leach and Daniel Dardailler deserve special recognition for their efforts in commenting in the design phase of the protocol. Also thanks to Henning Schulzrinne, Anup Rao and Robert Lanphier for pointing out the generalities of PEP and providing support for integration with RTSP[12].

This specification is a direct reflection of some implementation work: a client implementation in [23] (see the HTPEP module) and a server implementation by Eui-Suk Chung and Anit Chakraborty for the JEPI project.

This document has benefited greatly from the comments of all those participating in the HTTP-WG. In addition to those already mentioned, the following individuals have contributed to this specification:

- Eui-Suk Chung,
- Don Eastlake,
- Roy Fielding,
- Jim Gettys,
- Yaron Goland,
- Phill Hallam-Baker,
- Paul Hoffman,
- Koen Holtman,
- Ora Lassila,
- Larry Masinter, and
- Jim Whitehead

15 Authors Addresses

Dan Connolly
Architecture Domain Lead, World Wide Web Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, U.S.A.
Email: connolly@w3.org

Rohit Khare
Technical Staff, World Wide Web Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, U.S.A.
Email: khare@w3.org

Henrik Frystyk Nielsen
Technical Staff, World Wide Web Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, U.S.A.
Email: frystyk@w3.org

Appendices

16 Summary of PEP Interactions

The following tables summarize the outcome of strength and scope rules in PEP transactions involving PEP compliant and non-PEP compliant HTTP proxies and origin servers. The summary is intended as a guide and index to the text, but is necessarily cryptic and incomplete. This summary should never be used or referenced separately from the complete PEP specification. The tables should be read as follows

Standard processing
: The action taken by an ultimate recipient not understanding or ignoring the extension (see section 4.2)

Extended processing
: The action taken by an ultimate recipient understanding and obeying the extension (see section 4.2)

Forward extension
The action taken by an intermediate party which is not an ultimate recipient (see section 9)

Strip extension
The action taken by an intermediate party which is the ultimate recipient (see section 9)

420 (Bad Extensions)
The response from an ultimate recipient not understanding or not wishing to obey the extension (see section 8.1)

501 (Not Implemented)
The response from an ultimate recipient not understanding the "`PEP`" method or "`PEP-`" method token prefix (see section 6)

Table 1: Origin Server

Scope	Hop-by-hop		End-to-end	
Strength	**Optional (may)**	**Required (must)**	**Optional (may)**	**Required (must)**
PEP not supported	Standard processing	501 (Not Implemented)	Standard processing	501 (Not Implemented)
Extension not supported	Standard processing	420 (Bad Extensions)	Standard processing	420 (Bad Extensions)
Extension supported	Extended processing	Extended processing	Extended processing	Extended processing

Table 2: Proxy Server

Scope	Hop-by-hop		End-to-end	
Strength	**Optional (may)**	**Required (must)**	**Optional (may)**	**Required (must)**
PEP not supported	Strip extension	501 (Not Implemented)	Forward extension	501 (Not Implemented)
Extension not supported	Strip extension	420 (Bad Extensions)	Forward extension	Forward extension
Extension supported	Extended processing and strip	Extended processing and strip	Extended processing and strip	Extended processing and strip

17 Examples

The following examples show various scenarios using PEP in HTTP/1.1 requests and responses. Information not essential for illustrating the examples is left out (referred to as "…")

17.1 Client Queries a Server for DAV

In this example, the purpose of using PEP in the request is to determine whether a server understands and supports the Distributed Authoring and Versioning (DAV) protocol extension[18]. By making the request binding (see section 7), the client forces the server to process the extension declaration and obey the extension or report an error.

```
PEP-GET /some.url HTTP/1.1
Host: some.host
PEP: {{map "http://www.w3.org/PEP/DAV"}}

HTTP/1.1 200 OK
PEP-Info: {{id "http://www.w3.org/PEP/DAV"} {for "/Henrik" *}}
...
```

The response shows that the server does understand DAV and that the client can use it on all resources matching the prefix "`/Henrik`" on that server. The policy is informational and other factors like access control may prevent the client from actually using DAV on any of these resources.

PEP does not distinguish between querying about or using an extension – the PEP declaration is identical. Whether it in fact is a query may depend on the request method name and request modifiers.

17.2 Client Informs a Server about ZipFlate Compression Extension

This example shows a client informing a server that it is capable of handling the zipflate compression extension in a response. By issuing an extension policy instead of an extension declaration, the client indicates that the extension is not used in the request.

```
GET /Index HTTP/1.1
Host: some.host
PEP-Info: {{id "http://www.w3.org/PEP/Encoding"}}

HTTP/1.1 200 OK
PEP: {{map "http://www.w3.org/PEP/Encoding"}}
Cache-Control: no-transform
Vary: *
...
```

The response shows that the server knows the extension and decides to use it in the response. It furthermore includes the `no-transform` `cache-control` directive in order to avoid that proxies add their own `content-coding` to the message (see section 10.2) and a `Vary` header field indicating that a cache may not use the response to reply to a subsequent request without revalidation.

In this example, the client could have used an extension declaration of strength "may" instead of an extension policy to achieve the same effect. The request would not have been affected as the compression applies to message bodies and not headers. If the request were to include a message body, however, the difference would be whether the zipflate extension was applied to that body or not.

17.3 Server Uses Content-Digest Extension

This example shows a server applying the Content-Digest extension to a response message indicating that the client may ignore it. The client has not indicated whether it supports the extension or even if it supports PEP.

```
GET /Index HTTP/1.1
Host: some.host

HTTP/1.1 200 OK
PEP: {{map "http://www.w3.org/PEP/Digest" Content-Digest}}
Content-Digest: "a0b1c2d3e4f5g6h7i8j9"
Cache-Control: max-age=3600
...
```

The response is fully cachable and does not require revalidation when replying to subsequent requests.

17.4 Server Requires the Client to use Payment Extension

The last example shows how a server requires a client to use a micro-payment extension in order to access a resource causing an additional roundtrip using the 420 (Bad Extensions) status code (see section 8.1). The first request does not contain any PEP constructs leading to the error message. A non-PEP compliant client will treat this as a 400 (Bad Request) status code and will not be able to fulfill the server's requirement in a second request (see [7], section 10.4.1)

```
GET /Index HTTP/1.1
Host: some.host

420 Bad Extensions
PEP-Info: {{id "http://www.w3.org/PEP/MiniPayment"}
          {params {Price 0.02USD}} {strength must}}
```

```
PEP-GET /Index HTTP/1.1
Host: some.host
PEP: {{map "http://www.w3.org/PEP/MiniPayment" Price} {strength must}}
Price: 0.02USD

HTTP/1.1 200 OK
...
```

The actual price is passed as an extra parameter in the extension policy. The client agrees to the price and issues a new request containing the proper extension declaration. If it did not agree with the price, it could have tried a lower price and depending on the policy of that resource, the server may have responded positively.