The JSON HTTP Notation

Specification of the JOHN Language

First Edition

The JOHN Language extends the JavaScript Object Notation syntax by defining a set of keywords and operators with their prescribed behavior.

A JOHN plug-in, meaning a source file written using the JOHN Language, is a valid JSON file: a plugin writer uses the constructs of the language inside `string` literals, as defined in ECMA-404 [1], either as the name of an object member or its value.

In addition to strings, the JOHN Language may employ `number` or `boolean` types for its values, also defined in the standard [1].

The preferred file extension for JOHN Language sources is `.john` although the standard `.json` extension is also valid.

Implementations of the JOHN Language should parse the JSON source file, construct a JOHN data structure in memory (verifying the correct use of its properties in the process), and finally attempt its execution. The latter boils down to sending HTTP requests and managing the objects served as input to those requests and obtained from them. After the series of requests specified in the plug-in (if any) is carried out without issues, the runtime constructs an output object, transforming the results following the format defined in the plug-in.

From here on out, the specification refers to those who create and maintain plug-ins using the JOHN Language as “plug-in authors”, and individuals and organizations running them via a runtime implementation of the specification as “plug-in operators”; since “user” can ambiguously refer to both users of the language, users of the runtime, and users of the complete software solution, it is best to avoid its usage.

Anatomy of a JOHN Plugin

All valid JOHN Language sources must contain a root JSON object with an about member detailing its metadata, a pipeline member holding an array of requests to make, and a result member specifying what data structure to return to the caller.

Listing 1.1: Example of a simple JOHN plug-in

```
{
        "about": {
                "name": "Example Plug-In",
                "version": 1,
                "protocol": "JOHN_Simple_Example_Plugin"
        },
        "parameters": {},
        "pipeline": [
                {
                        "url": "https://www.example.com"
                }
        ],
        "result": "$1"
}
```

1.1 About

The `about` member accepts `About` objects containing metadata about the plug-in intended to aid the plug-in operator in identifying and assigning the correct plug-in to its intended task. This member is mandatory for the JOHN root object. Valid `About` objects must contain:

- A `name` string contains the human-friendly name of the plug-in. Its principal intended usage is identifying the plug-in inside a preference pane in a GUI.
- A `version` number indicates the unique build number of a plug-in. The plug-in operator may use it when upgrading a plug-in to newer versions.
- A `protocol` string uniquely identifies a plug-in among related ones. It enables a plug-in author to upload a group of plug-ins together and have

the receiving party be able to sort them automatically. It is not a globally unique identifier across all possible plug-ins. While it is certainly possible to use a reverse domain notation, it is not required.

1.2 The variable subscript syntax

Each HTTP request described inside the `pipeline` array produces a response including several HTTP headers and a response body decoded according to the `Content-Type` header sent by the source server. Anytime a property of either the `pipeline` or the `result` accepts a `string` value on its right-end side, it is possible to interpolate the value of previous requests inside the string by using a dollar sign `$` followed by the number of the request, starting at 1. When the value referenced by the variable is not a single value (a `string`, `number`, or `boolean`) but a vector or a hierarchy, a subscript syntax is available using a pair of square brackets `[]` and, respectively, an `integer` or a `string` inside them.

Listing 1.2: The response body used in JOHN subscript syntax example

```
{
        "items": [
                { "name": "Luca", "age": 56 },
                { "name": "Mario", "age": 37 },
                { "name": "Anna", "age": 16 },
                { "name": "Luisa", "age": 34 },
                { "name": "Flavio", "age": 73 },
                { "name": "Simona", "age": 89 }
        ]
}
```

For example, if the second request in the pipeline of a JOHN plug-in were to produce a response body of a list of people inside an array, in turn, nested in a property, then, from the third request of the pipeline onwards, it is possible to access the name of the sixth person using `$2[items][6][name]`.

Each node of the output, whatever the root variable or one of its subscripts, is simultaneously capable to answer single-value requests and subscripts: this allows the runtime to augment a single value with extra properties, or to surface complex structures such as XML-like markup.

When the syntax is used to interpolate the value of the subscript with other components of the containing source string, only the single-value, if present, is used. Hierarchical and vector values are ignored.

Customizing the HTTP request

Valid JOHN plug-ins must contain a `pipeline` member accepting an array of `Stage` objects: it can be empty if the desired result of the plug-in is static, and sending any HTTP request is not required.

2.1 Stage

At a high level, ignoring, for now, advanced directives provided in the JOHN Language, a stage is effectively a single HTTP request and therefore only requires a `url` field for the `string` value of the resource to connect to; by default, JOHN implementations should assume a `GET` method and yield the response body as the stage output, decoded automatically according to its `Content-Type` response header.

It is possible to customize the properties of the HTTP request, as defined in the standard [4], using the following optional properties:

- The `method` property is a case-insensitive `string` value for the HTTP method to use when sending the request. Implementations should uppercase it before use.
- The `headers` property accepts key-value pairs of strings (meaning a JSON object holding only string values) sent as HTTP headers when executing the stage.
- The `query` property also accepts key-value string pairs and acts as a convenience to append the specified query parameters to the URL of the stage. Implementations are not required to guarantee the uniqueness of query parameter names as they get appended to the URL.
- The `authorization` property accepts one of three `string` values: `"basic"`, `"bearer"`, or `"none"`, with the latter one equivalent to providing no prop-

erty at all. This property serves as a convenience for the plug-in operator to specify either the HTTP `Basic` or `Bearer` authentication separately from the plug-in and other parameters. The runtime implementation shall provide the operator with a way to supply a username-password pair or a text token in requesting the execution of a plug-in. When encountering an `authorization` property during execution, the runtime first verifies the presence of the required values and then constructs the appropriate `Authorization` header according to the specification. If the verification fails, the runtime aborts execution.

- The `body` property accepts either a single string value or a key-value dictionary as in headers and query parameters. In the first instance, the string is used as-is for the HTTP request body: if the text represents any encoding, it is on the plug-in author to specify it in a custom `Content-Type` header. In the latter case, by default, the implementation must encode it as a JSON object and set the `Content-Type` header accordingly.

Overriding default policies

So far, two default behaviors of implementations of the JOHN language have been presented without also discussing how to override them:

- The default encoding of key-value pairs in the request body to JSON objects
- After receiving the response to a request, in the absence of other directives in the stage, by default, its body (if present) is decoded according to the `Content-Type` header and stored for interpolation via the subscript syntax.

It is also worth mentioning that the HTTP status code associated with the request is, by default, ignored concerning the processing of the response body. It is possible to change those behaviors by using the following properties of the `Stage` object:

- The `status` property accepts an array of integers representing the valid HTTP status codes for the response to be considered correct. If a stage includes this property and does not contain the returned response's status code, then the JOHN implementation must abort the execution of the plug-in.
- The `yield` property accepts one of three string values `"body"`, `"headers"`, or `"cookies"`. In the latter two instances, the runtime shall make available the key-value pair of either headers or cookies received as part of the response in place of the body when using the variable subscript syntax for the corresponding stage.
- The `encode` property accepts either `"json"` or `"form"` as its string value. The first one is equivalent to having no property at all. The latter one will

encode the body (if provided as a JSON key-value pair object and not a string) using the same encoding as that used for forms submitted from HTML pages: each key and value is percent-encoded to escape reserved entities, the two concatenated with an equal sign; finally, all the pairs are in turn concatenated with each other, separated by an ampersand symbol. Before sending the request, the runtime sets the `Content-Type` header to `"application/x-www-form-urlencoded"`

- The `decode` property acts on the received response body and accepts one of the following values:
  - `"auto"` which is the default behavior and picks one of the other ones based on the `Content-Type` header.
  - `"raw"` which means plain-text; this is the default for when no appropriate `Content-Type` is present.
  - `"json"` or `"form"` which should produce a key-value dictionary accessible via the subscript syntax; they are picked by default respectively for `Content-Type` `"application/json"` and `"application/x-www-form-urlencoded"`.
  - `"xml"` or `"html"` constructing a composite structure where the integer subscript accesses the children of an element and the string subscript accesses its attributes; these are the default behavior of `Content-Type` matching exactly `"text/xml"` or `"application/xml"`, or containing the `"+xml"` suffix for `"xml"`, while values of `Content-Type` matching `"text/html"` or `"application/xhtml+xml"` will default to `"html"`.
  - `"soap"` which works analogously to `"xml"` but unwraps the SOAP [2] envelope to yield its body as the root element, reducing the depth of the subscript used to traverse it; this is the default decoding strategy

for a `Content-Type` header equal to `"application/soap+xml"` or just `"application/soap"`.

- `"xml-json"` or `"soap-json"` which attempt to parse every text node of the markup as JSON and inserts the decoded object in the structure, available to the subscript syntax; this behavior must be opted in by using the `decode` property and it is never applied automatically.
- `"base64"` which encodes the output as a base64 encoded string, and is the default behavior for `Content-Type` equal to `application/octet-stream` or beginning with either `audio/`, `image/`, `video/` or `font/`.

It follows that JOHN implementations must support decoding for all of them.

3.1 Redirection

While the status code, unless specified otherwise, is ignored when processing the body, it is not when involving `3xx` status codes, as they indicate redirects. By default, the runtime implementation shall attempt no more than five requests when following the URL specified in the `Location` header, and the plug-in author can override this number using the `redirects` property of a `Stage` object, which accepts an integer value. It is possible to set it to 0, and in that case, no redirect following will occur: this is useful if the output desired from the request is precisely the `Location` header.

Manipulating the cookie store

It is not sufficient for JOHN runtime implementations to parse the `Set-Cookie` header returned as part of the response to yield `"cookies"` when desired in a plug-in stage. Implementations must implement the complete HTTP state management mechanism as defined in RFC 6265[3]:

- For the duration of a plug-in execution, a cookie store must be kept, similar to the authorization parameters and the output of the various stages.
- When sending a request, and before applying the headers specified for the stage, they must assemble a standards-compliant `Cookie` header.
- After receiving a response, even in the middle of following redirections, they must parse the `Set-Cookie` headers and change the cookie store by adding, updating, or removing cookies.

4.1 Cookies

The Internet standard prescribes an algorithm to determine what cookies to send based on domain, subdomain, and path [3]. The JOHN Language provides a `cookies` property as part of the directives of stages to manipulate this subset of cookies before executing the request:

- It is possible to assign it a boolean value, where `false` removes all the cookies that would have been sent from the store, while `true` is the default behavior that does no manipulation at all.
- Alternatively, it is possible to assign it a key-value pair where the left-end side is a string containing a particular cookie name, and the right-end side is either a string to set as its value or a boolean to apply the deletion on a per-cookie basis.

Providing and verifying input with parameters and assertions

Complex plug-ins involve multiple stages that rely on the results of the previous ones in the pipeline. If they manipulate data, they might require input information and ways to check along their execution that those values are present and correct.

5.1 Parameters

The variable interpolation syntax starts counting stages from 1 because index 0 points at parameters. Implementations of the JOHN Language must allow the plug-in operator to supply either a string value (or types that are representable losslessly by a string) or any structure traversable via the integer and string subscript. Plug-in authors can explicitly set out the inputs they require via the `parameters` member of the root JOHN object, which accepts a `Parameter` value type either as a single value or recursive key-value tree structure; the `Parameter` can be:

- A boolean type that represents whatever the single-value parameter or the particular key in the tree structure is required for execution or not. In the former case, if the parameter is not provided, then the plug-in should not be executed.
- A string value matching either `"username"`, `"password"`, or `"token"`, which instructs the runtime to substitute the parameter with the respective value, as defined for the `authorization` directive of a Stage.
- Any other string value, that shall be interpreted by the runtime as a default value used for the parameter if no other is provided.

- A JSON object whose keys represent nested parameters of the current parameter, either top-level or another nested one. Implementation should check for the input to contain all the keys that are either set with a `true` requirement or recursively containing a nested object.

5.2 Assertions

If the interpolation of a variable in a string parameter of a stage fails, then the execution is aborted. Plug-in authors can explicitly check for the presence of values at specific variable subscripts before a stage is executed, and even test for its value being included in a set of strings by including the `assert` property in the stage, which is a key-value pair where the key indicates the variable subscript to test for, and the value is an object containing:

- An `exists` property that accepts an optional boolean value that can either check for the variable substitution to succeed or fail.
- A `contained` property that accepts an array of strings and will succeed the assertion if the value resolved by the variable is also a member of the array.

If an assertion fails in the context of a stage, its execution shall stop. If the stage has not run at least once then the whole plug-in should abort execution, as no suitable value for its stage output is available.

Automating pagination with repetition and merging

Usually, HTTP endpoints avoid returning large numbers of items in a single request, preferring a form of paginated access. The JOHN Language provides ways to indicate the runtime to execute a stage more than once and to continuously merge their outputs so that they appear to subsequent pipeline stages as if the remote resource returned them all at once.

6.1 Repetitions

Plug-in authors can express their intent to loop a stage by providing a 'repeat' property inside it:

- Its `start` number value indicates the integer index to start counting at. It is optional and defaults to `0`. It must be a positive number.
- Its `step` number value indicates how many integer units to increment the counter after each iteration. It is optional and defaults to `1`. It must be greater than `0`.
- Its `stop` number value is the integer index after which to conclude the execution of the stage. It must be a positive number as well. This property is also optional and defaults to the stop value provided for the whole plug-in as it is executed, multiplied by the `step` value. Repetition is a negotiation between a plug-in author and its operator, where the former specifies its upper bound of repetitions, and the latter theirs, which should default to no repetition at all. If the former is greater than the latter, the execution of the plug-in is aborted.
- Its `variable` string value is the subscript that is used on the stage to retrieve the iterator, the latter being an object with the following properties:

  - a `chunk` property equivalent to the absolute value of the `step`
  - an `offset` property obtained by summing all the previous chunks with the `start` value
  - a `total` property returning the sum of the `chunk` and the `offset`

  This value is also optional and defaults to the character `"i"`.

After each iteration, by default, a provisional output of the stage is constructed containing an array of the outputs produced by each iteration. The provisional output is augmented with an `Iterator` property retrieved through the string subscript indicated by the `variable` property of the `repeat` directive and appended to the output pipeline for successive iterations of the stage to access. It is possible to escape iteration before reaching the `stop` value by combining the `repeat` directive with the `assert` directive: in the context of repetitions, assertion failures will only conclude the loop instead of aborting the whole plug-in.

6.2 Merger

Plug-in authors can explicitly merge a stage's output with previous ones, similar to how repetition merges the various iterations. It is also possible to customize how the merging works in both occurrences, beyond the simple appending operation into an array. Those preferences are expressed via a `merge` property in a stage, accepting the following object as its value:

- Its `stage` property accepts an array of numbers, representing the indices of the previous stages to merge with the current one. If it contains a greater index than the current one, the runtime must abort the execution of the plug-in. This property is optional if the only source of merging is a `repeat` directive.

- Its `default` and `override` properties accept, respectively, a single value and a list of key-value pairs of `Policy` strings. The key in the pair is the

string subscript that the policy pertains to. When accessing a string subscript, implementations must first check for the iterator variable if present to return that instead, then for the key in the `override` list, and finally use the `default` policy. Single value and integer subscripts always refer to the `default` policy. A `Policy` string can assume one of the following values:

- `"none"`, which is the default, and means that the stages and iterations are not merged at all and are accessed as a vector containing each output in chronological order of appending.
- `"first"` and `"last"` redirect the subscript to the first and the last output that is being merged, respectively.
- `"keep"` redirects the subscript to each page, beginning with the first one, until it resolves any value and it is returned, and if no value is returned, then it behaves like any other non-resolving subscript.
- `"replace"` does the same as `"keep"` but beginning with the last page and going backwards.
- `"append"` which does not resolve any value for single-value subscripts, but in case of integer subscripts, concatenates each merged output that represents an array, and then applies the subscript to the combined array, while in the case of string subscripts, does that for each array that is children of the referenced string subscript in every merged output.

Handling groups of plug-ins together

The JOHN Language does not currently provide constructs allowing for recursion or automatic authentication management. Advanced logic like this, not expressible through the JOHN Language, requires multiple plug-ins executed more than once to accomplish the desired task. JOHN Language implementations must allow plug-ins to share an “execution group” that maintains the same context across them, mainly the cookies. Once in a group, it is possible to mark individual stages with an `id` property, expected to be unique in relation to the group, signaling the runtime to execute the request only once in the group and reuse its response for successive invocations without sending any HTTP request. This property, if present, accepts a string. It is also possible to mark the stage with a `defer` property, accepting an optional boolean indicating whether to skip its execution until all the requests in the group have completed theirs. The affordance to use to provide plug-ins to an execution group and to run the deferred stages is not specified as it is an implementation detail. Combining the `defer` and the `id` properties allows the definition of clean-up requests to run only once, or non-standard authentication procedures that rely on cookies and require explicit login and logout procedures: those would be wasteful or suspicious to repeat for each plug-in execution.

Yielding a normalized result

Although services that accomplish the same tasks will gravitate toward similar interfaces, they may feature inconsistencies requiring their conversion into a common representation.

8.1 Result

The `result` property of the root JOHN object allows the plug-in author to describe the structure of the plug-in's output. The value accepted by the property is a recursive key-value array of subscripts on the left-end side and their result on the right-end side, the latter expecting either a `Result` type or another key-value list that implementations must flatten out by concatenating subscripts together; the root `result` property is equivalent to an empty `""` string subscript; it is also possible to chain together multiple subscripts in a single left-end key. The first subscript in a key, whether the only subscript or not, can have its brackets omitted.

8.2 The result subscript syntax

The result subscript syntax is a superset of the one defined for variables: integers or strings between square brackets `[]` indicating array and hierarchy subscripts. To support the output manipulation capabilities, a handful of additions are present:

- The jolly subscript, indicated by the $*$ asterisk symbol, resolves any integer and any string provided at its position of the subscript.
- The range subscript <*< resolves the range of indices indicated at its opposite ends. It is also possible to omit one of them (including the less than sign <), and in that case, they will, respectively, either be replaced with the index `0` or considered as an indication of a minimum floor for the array

subscript, accepting infinite values forward. It is also possible to define a bound with the variable subscript syntax, that will resolve either the single-value if it is convertible to a number, or the index of the vector if the latter is present. The range always includes its defining bounds.

- The pattern subscript supports inserting asterisks in a string subscript, each resolving any number of characters inside them. Testing for the pattern starts at the beginning of the string and proceeds rightward. The testing of string literal components occurs lazily unless they are the last component of the testing pattern, in which case they are tested eagerly from the end. Multiple contiguous asterisks are equivalent to a single one. Implementations must perform the testing in a case-sensitive way.

Catch-all style subscripts, meaning the jolly, pattern, and range subscripts with a non-defined upper bound, cannot be resolved statically as they represent infinite values. In cases where a plug-in's result is transferred to a finite static product, implementations must discard those subscripts. In dynamic contexts, instead, they must provide a mechanism to query the plug-in output of arbitrary subscripts using the JOHN result subscript syntax. Since multiple subscripts might resolve a subscript, implementations must keep track of the specificity of a subscript according to the following rules:

- Exact matches always have the highest value.
- The higher the number of catch-all symbols contained in the pattern subscript, the lower the specificity is. When they are the same, their position in the pattern is considered, with the pattern string with the earliest occurring asterisk $*$ having the lower specificity.
- Ranges with fully defined bounds have equal specificity.

When implementations have to choose between more than one matching result subscript with maximum specificity, the one defined earliest in the `result` direc-

tive is the chosen one. Plug-in authors should not rely on the specificity system provided here for completeness and instead avoid overlap between subscripts when possible.

8.3 The result value mapping

The `Result` type represents what a result subscript key resolves to. It accepts either of the following values:

- A string literal to resolve the subscript with a value unconditionally. Upon access, the following steps must occur:
  - for each catch-all subscript in the left-end side, the JOHN implementation replaces each catch-all symbol on the string with the actual value, starting from the first and going rightward.
  - then the entire string literal is tested as a single variable subscript, which if resolved will incorporate the full output tree (with the single-value, vector, and hierarchy subscripts all reachable) at the corresponding result subscript
  - finally, if the previous step failed to resolve a single variable, the string literal is scanned for variables to interpolate, as defined previously in this document, and their resolved values substituted inside it.

  If the literal contains more catch-all symbols that the left-end side subscript can fill, then no value is returned for the subscript.
- A `Condition` object resolves the subscript only if the specified assertion, as defined previously for stages, is verified. It contains three properties:
  - `assert`, which accepts an `Assertion` object.
  - `result`, which is the `Result` value to choose if the assertion succeeds and it is mandatory for valid `Condition` objects.

- `catch`, which is the `Result` value picked if the assertion test fails instead, and it is optional.

Since, from a JSON perspective, a key-value list of `Results` and a `Condition` object are syntactically the same, implementations must first check for the reserved keyword `"result"` to be present inside it: result subscript literals containing only the `"result"` keyword are not allowed, as they will map to `Condition` objects and their other properties discarded.

Example usage of advanced JOHN features

The Dropbox indexing plug-in of the University Cloud Suite showcases assertions, merging of stages with and without repetition, and result normalization.

Listing 9.1: Indexing a Dropbox drive for the University Cloud Suite

```
{
   "about": { /* Omitted for brevity */ },
   "pipeline": [
      {
         "url": "https://api.dropboxapi.com/2/files/list_folder",
         "method": "POST",
         "authorization": "bearer",
         "body": { /* See Dropbox API */ },
         "status": [ 200 ]
      },
      {
         "assert": {
            "$2[has_more]": { "contained": ["true"] }
         },
         "repeat": { "stop": 150 },
         "merge": {
            "stage": [1],
            "default": "append",
            "override": {
               "has_more": "last",
               "cursor": "last"
            }
         },
         "url": "https://api.dropboxapi.com/2/files/list_folder/
            ↪ continue",
         "method": "POST",
         "authorization": "bearer",
         "body": {
            "cursor": "$2[cursor]"
         },
         "status": [ 200 ]
      }
   ],
   "result": {
      "cursor": "$2[cursor]",
      "items[0<*<$2[entries]]": {
         "id": "$2[entries][*][id]",
```

```
37 |         "name": "$2[entries][*][name]",
38 |         "iso8601": "$2[entries][*][server_modified]",
39 |         "bytes": "$2[entries][*][size]",
40 |         "path": {
41 |           "full": "$2[entries][*][path_display]"
42 |         },
43 |         "kind": {
44 |           "assert": {
45 |             "$2[entries][*][.tag]": { "contained": ["file"] }
46 |           },
47 |           "result": "file",
48 |           "catch": {
49 |             "assert": {
50 |               "$2[entries][*][.tag]": { "contained": ["folder"] }
51 |             },
52 |             "result": "folder",
53 |             "catch": "unknown"
54 |           }
55 |         }
56 |       }
57 |   }
58 |}
```

Quick reference of the JOHN Language syntax

For the convenience of plug-in authors, a reference of all the possible JOHN commands is provided here.

Listing 10.1: The JOHN Language syntax

```
{
    "about": {
        "name": String,
        "version": Int,
        "protocol": String
    },
    "parameters": String | true | false | "username" | "password" | "token" | [String: ...],
    "pipeline": [
        {
            "id": String,
            "defer": Bool,
            "assert": [String: {
                "exists": Bool,
                "contained": [String]
            }]
            "repeat": {
                "start": Int
                "step": Int
                "stop": Int
                "variable": String
            }
            "merge": {
                "stage": [Int],
                "default": "first" | "keep" | "append" | "replace" | "last" | "none",
                "override": [String: "first" | "keep" | ...]
            }
            "authorization": "basic" | "bearer" | "none",
            "redirects": Int,
            "url": String,
            "method": String,
            "status": [Int],
            "headers": [String: String],
            "query": [String: String],
            "cookies": true | false | [String: true | false | String ],
```

```
            "body": String | [String: String],
            "yield": "headers" | "body" | "cookies",
            "encode": "json" | "form",
            "decode": "auto" | "raw" | "json" | "form" | "xml" | "
                ↪ xml-json" | "soap" | "soap-json" | "html" | "
                ↪ base64"
        }
    ],
    "result": String | {
        "assert": [String: {
            "exists": Bool,
            "contained": [String]
        }],
        "result": ...,
        "catch": ...
    } | [String: ...]
}
```

References

[1] ECMA-404 The JSON data interchange syntax 2nd edition, December 2017. https://www.ecma-international.org/publications-and-standards/standards/ecma-404/.

[2] Nilo Mitra, Ericsson Yves Lafon, W3C . SOAP Version 1.2 Part 0: Primer (Second Edition) . https://www.w3.org/TR/2007/REC-soap12-part0-20070427/, April 2007.

[3] Adam Barth. HTTP State Management Mechanism. RFC 6265, April 2011.

[4] Henrik Nielsen, Jeffrey Mogul, Larry M Masinter, Roy T. Fielding, Jim Gettys, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.