Network Working Group
Request for Comments: 3875
Category: Informational

D. Robinson
K. Coar
The Apache Software Foundation

October 2004

The Common Gateway Interface (CGI) Version 1.1

Status of this Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Copyright Notice

IESG Note

This document is not a candidate for any level of Internet Standard. The IETF disclaims any knowledge of the fitness of this document for any purpose, and in particular notes that it has not had IETF review for such things as security, congestion control or inappropriate interaction with deployed protocols. The RFC Editor has chosen to publish this document at its discretion. Readers of this document should exercise caution in evaluating its value for implementation and deployment.

Abstract

The Common Gateway Interface (CGI) is a simple interface for running external programs, software or gateways under an information server in a platform-independent manner. Currently, the supported information servers are HTTP servers.

The interface has been in use by the World-Wide Web (WWW) since 1993. This specification defines the 'current practice' parameters of the 'CGI/1.1' interface developed and documented at the U.S. National Centre for Supercomputing Applications. This document also defines the use of the CGI/1.1 interface on UNIX(R) and other, similar systems.

Table of Contents

1. Introduction

1.1. Purpose

The Common Gateway Interface (CGI) [22] allows an HTTP [1], [4] server and a CGI script to share responsibility for responding to client requests. The client request comprises a Uniform Resource Identifier (URI) [11], a request method and various ancillary information about the request provided by the transport protocol.

The CGI defines the abstract parameters, known as meta-variables, which describe a client's request. Together with a concrete programmer interface this specifies a platform-independent interface between the script and the HTTP server.

The server is responsible for managing connection, data transfer, transport and network issues related to the client request, whereas the CGI script handles the application issues, such as data access and document processing.

1.2. Requirements

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 BCP 14, RFC 2119 [3].

An implementation is not compliant if it fails to satisfy one or more of the 'must' requirements for the protocols it implements. An implementation that satisfies all of the 'must' and all of the 'should' requirements for its features is said to be 'unconditionally compliant'; one that satisfies all of the 'must' requirements but not all of the 'should' requirements for its features is said to be 'conditionally compliant'.

1.3. Specifications

Not all of the functions and features of the CGI are defined in the main part of this specification. The following phrases are used to describe the features that are not specified:

'system-defined'
: The feature may differ between systems, but must be the same for different implementations using the same system. A system will usually identify a class of operating systems. Some systems are defined in section 7 of this document. New systems may be defined by new specifications without revision of this document.

'implementation-defined'
The behaviour of the feature may vary from implementation to implementation; a particular implementation must document its behaviour.

1.4. Terminology

This specification uses many terms defined in the HTTP/1.1 specification [4]; however, the following terms are used here in a sense which may not accord with their definitions in that document, or with their common meaning.

'meta-variable'
A named parameter which carries information from the server to the script. It is not necessarily a variable in the operating system's environment, although that is the most common implementation.

'script'
The software that is invoked by the server according to this interface. It need not be a standalone program, but could be a dynamically-loaded or shared library, or even a subroutine in the server. It might be a set of statements interpreted at run-time, as the term 'script' is frequently understood, but that is not a requirement and within the context of this specification the term has the broader definition stated.

'server'
The application program that invokes the script in order to service requests from the client.

2. Notational Conventions and Generic Grammar

2.1. Augmented BNF

All of the mechanisms specified in this document are described in both prose and an augmented Backus-Naur Form (BNF) similar to that used by RFC 822 [13]. Unless stated otherwise, the elements are case-sensitive. This augmented BNF contains the following constructs:

name = definition
The name of a rule and its definition are separated by the equals character ('='). Whitespace is only significant in that continuation lines of a definition are indented.

"literal"
Double quotation marks (") surround literal text, except for a literal quotation mark, which is surrounded by angle-brackets ('<' and '>').

rule1 | rule2
Alternative rules are separated by a vertical bar ('|').

(rule1 rule2 rule3)
Elements enclosed in parentheses are treated as a single element.

*rule
A rule preceded by an asterisk ('*') may have zero or more occurrences. The full form is 'n*m rule' indicating at least n and at most m occurrences of the rule. n and m are optional decimal values with default values of 0 and infinity respectively.

[rule]
An element enclosed in square brackets ('[' and ']') is optional, and is equivalent to '*1 rule'.

N rule
A rule preceded by a decimal number represents exactly N occurrences of the rule. It is equivalent to 'N*N rule'.

2.2. Basic Rules

This specification uses a BNF-like grammar defined in terms of characters. Unlike many specifications which define the bytes allowed by a protocol, here each literal in the grammar corresponds to the character it represents. How these characters are represented in terms of bits and bytes within a system are either system-defined or specified in the particular context. The single exception is the rule 'OCTET', defined below.

The following rules are used throughout this specification to describe basic parsing constructs.

```
alpha         = lowalpha | hialpha
lowalpha      = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |
                "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" |
                "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |
                "y" | "z"
hialpha       = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" |
                "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" |
                "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" |
                "Y" | "Z"
```

```
digit         = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
                "8" | "9"
alphanum      = alpha | digit
OCTET         = <any 8-bit byte>
CHAR          = alpha | digit | separator | "!" | "#" | "$" |
                "%" | "&" | "'" | "*" | "+" | "-" | "." | "`" |
                "^" | "_" | "{" | "|" | "}" | "~" | CTL
CTL           = <any control character>
SP            = <space character>
HT            = <horizontal tab character>
NL            = <newline>
LWSP          = SP | HT | NL
separator     = "(" | ")" | "<" | ">" | "@" | "," | ";" | ":" |
                "\" | <"> | "/" | "[" | "]" | "?" | "=" | "{" |
                "}" | SP | HT
token         = 1*<any CHAR except CTLs or separators>
quoted-string = <"> *qdtext <">
qdtext        = <any CHAR except <"> and CTLs but including LWSP>
TEXT          = <any printable character>
```

Note that newline (NL) need not be a single control character, but can be a sequence of control characters. A system MAY define TEXT to be a larger set of characters than <any CHAR excluding CTLs but including LWSP>.

2.3. URL Encoding

Some variables and constructs used here are described as being 'URL-encoded'. This encoding is described in section 2 of RFC 2396 [2]. In a URL-encoded string an escape sequence consists of a percent character ("%") followed by two hexadecimal digits, where the two hexadecimal digits form an octet. An escape sequence represents the graphic character that has the octet as its code within the US-ASCII [9] coded character set, if it exists. Currently there is no provision within the URI syntax to identify which character set non-ASCII codes represent, so CGI handles this issue on an ad-hoc basis.

Note that some unsafe (reserved) characters may have different semantics when encoded. The definition of which characters are unsafe depends on the context; see section 2 of RFC 2396 [2], updated by RFC 2732 [7], for an authoritative treatment. These reserved characters are generally used to provide syntactic structure to the character string, for example as field separators. In all cases, the string is first processed with regard to any reserved characters present, and then the resulting data can be URL-decoded by replacing "%" escape sequences by their character values.

To encode a character string, all reserved and forbidden characters are replaced by the corresponding "%" escape sequences. The string can then be used in assembling a URI. The reserved characters will vary from context to context, but will always be drawn from this set:

```
reserved = ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+" | "$" |
           "," | "[" | "]"
```

The last two characters were added by RFC 2732 [7]. In any particular context, a sub-set of these characters will be reserved; the other characters from this set MUST NOT be encoded when a string is URL-encoded in that context. Other basic rules used to describe URI syntax are:

```
hex        = digit | "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b"
             | "c" | "d" | "e" | "f"
escaped    = "%" hex hex
unreserved = alpha | digit | mark
mark       = "-" | "_" | "." | "!" | "~" | "*" | "'" | "(" | ")"
```

3. Invoking the Script

3.1. Server Responsibilities

The server acts as an application gateway. It receives the request from the client, selects a CGI script to handle the request, converts the client request to a CGI request, executes the script and converts the CGI response into a response for the client. When processing the client request, it is responsible for implementing any protocol or transport level authentication and security. The server MAY also function in a 'non-transparent' manner, modifying the request or response in order to provide some additional service, such as media type transformation or protocol reduction.

The server MUST perform translations and protocol conversions on the client request data required by this specification. Furthermore, the server retains its responsibility to the client to conform to the relevant network protocol even if the CGI script fails to conform to this specification.

If the server is applying authentication to the request, then it MUST NOT execute the script unless the request passes all defined access controls.

3.2. Script Selection

The server determines which CGI is script to be executed based on a generic-form URI supplied by the client. This URI includes a hierarchical path with components separated by "/". For any particular request, the server will identify all or a leading part of this path with an individual script, thus placing the script at a particular point in the path hierarchy. The remainder of the path, if any, is a resource or sub-resource identifier to be interpreted by the script.

Information about this split of the path is available to the script in the meta-variables, described below. Support for non-hierarchical URI schemes is outside the scope of this specification.

3.3. The Script-URI

The mapping from client request URI to choice of script is defined by the particular server implementation and its configuration. The server may allow the script to be identified with a set of several different URI path hierarchies, and therefore is permitted to replace the URI by other members of this set during processing and generation of the meta-variables. The server

1. MAY preserve the URI in the particular client request; or

2. it MAY select a canonical URI from the set of possible values for each script; or

3. it can implement any other selection of URI from the set.

From the meta-variables thus generated, a URI, the 'Script-URI', can be constructed. This MUST have the property that if the client had accessed this URI instead, then the script would have been executed with the same values for the SCRIPT_NAME, PATH_INFO and QUERY_STRING meta-variables. The Script-URI has the structure of a generic URI as defined in section 3 of RFC 2396 [2], with the exception that object parameters and fragment identifiers are not permitted. The various components of the Script-URI are defined by some of the meta-variables (see below);

```
script-URI = <scheme> "://" <server-name> ":" <server-port>
             <script-path> <extra-path> "?" <query-string>
```

where <scheme> is found from SERVER_PROTOCOL, <server-name>, <server-port> and <query-string> are the values of the respective meta-variables. The SCRIPT_NAME and PATH_INFO values, URL-encoded with ";", "=" and "?" reserved, give <script-path> and <extra-path>.

See section 4.1.5 for more information about the PATH_INFO meta-variable.

The scheme and the protocol are not identical as the scheme identifies the access method in addition to the application protocol. For example, a resource accessed using Transport Layer Security (TLS) [14] would have a request URI with a scheme of https when using the HTTP protocol [19]. CGI/1.1 provides no generic means for the script to reconstruct this, and therefore the Script-URI as defined includes the base protocol used. However, a script MAY make use of scheme-specific meta-variables to better deduce the URI scheme.

Note that this definition also allows URIs to be constructed which would invoke the script with any permitted values for the path-info or query-string, by modifying the appropriate components.

3.4. Execution

The script is invoked in a system-defined manner. Unless specified otherwise, the file containing the script will be invoked as an executable program. The server prepares the CGI request as described in section 4; this comprises the request meta-variables (immediately available to the script on execution) and request message data. The request data need not be immediately available to the script; the script can be executed before all this data has been received by the server from the client. The response from the script is returned to the server as described in sections 5 and 6.

In the event of an error condition, the server can interrupt or terminate script execution at any time and without warning. That could occur, for example, in the event of a transport failure between the server and the client; so the script SHOULD be prepared to handle abnormal termination.

4. The CGI Request

Information about a request comes from two different sources; the request meta-variables and any associated message-body.

4.1. Request Meta-Variables

Meta-variables contain data about the request passed from the server to the script, and are accessed by the script in a system-defined manner. Meta-variables are identified by case-insensitive names; there cannot be two different variables whose names differ in case only. Here they are shown using a canonical representation of capitals plus underscore ("_"). A particular system can define a different representation.

```
meta-variable-name = "AUTH_TYPE" | "CONTENT_LENGTH" |
                     "CONTENT_TYPE" | "GATEWAY_INTERFACE" |
                     "PATH_INFO" | "PATH_TRANSLATED" |
                     "QUERY_STRING" | "REMOTE_ADDR" |
                     "REMOTE_HOST" | "REMOTE_IDENT" |
                     "REMOTE_USER" | "REQUEST_METHOD" |
                     "SCRIPT_NAME" | "SERVER_NAME" |
                     "SERVER_PORT" | "SERVER_PROTOCOL" |
                     "SERVER_SOFTWARE" | scheme |
                     protocol-var-name | extension-var-name
protocol-var-name  = ( protocol | scheme ) "_" var-name
scheme             = alpha *( alpha | digit | "+" | "-" | "." )
var-name           = token
extension-var-name = token
```

Meta-variables with the same name as a scheme, and names beginning with the name of a protocol or scheme (e.g., HTTP_ACCEPT) are also defined. The number and meaning of these variables may change independently of this specification. (See also section 4.1.18.)

The server MAY set additional implementation-defined extension meta-variables, whose names SHOULD be prefixed with "X_".

This specification does not distinguish between zero-length (NULL) values and missing values. For example, a script cannot distinguish between the two requests http://host/script and http://host/script? as in both cases the QUERY_STRING meta-variable would be NULL.

```
meta-variable-value = "" | 1*<TEXT, CHAR or tokens of value>
```

An optional meta-variable may be omitted (left unset) if its value is NULL. Meta-variable values MUST be considered case-sensitive except as noted otherwise. The representation of the characters in the meta-variables is system-defined; the server MUST convert values to that representation.

4.1.1. AUTH_TYPE

The AUTH_TYPE variable identifies any mechanism used by the server to authenticate the user. It contains a case-insensitive value defined by the client protocol or server implementation.

For HTTP, if the client request required authentication for external access, then the server MUST set the value of this variable from the 'auth-scheme' token in the request Authorization header field.

```
AUTH_TYPE      = "" | auth-scheme
auth-scheme    = "Basic" | "Digest" | extension-auth
extension-auth = token
```

HTTP access authentication schemes are described in RFC 2617 [5].

4.1.2. CONTENT_LENGTH

The CONTENT_LENGTH variable contains the size of the message-body attached to the request, if any, in decimal number of octets. If no data is attached, then NULL (or unset).

```
CONTENT_LENGTH = "" | 1*digit
```

The server MUST set this meta-variable if and only if the request is accompanied by a message-body entity. The CONTENT_LENGTH value must reflect the length of the message-body after the server has removed any transfer-codings or content-codings.

4.1.3. CONTENT_TYPE

If the request includes a message-body, the CONTENT_TYPE variable is set to the Internet Media Type [6] of the message-body.

```
CONTENT_TYPE = "" | media-type
media-type   = type "/" subtype *( ";" parameter )
type         = token
subtype      = token
parameter    = attribute "=" value
attribute    = token
value        = token | quoted-string
```

The type, subtype and parameter attribute names are not case-sensitive. Parameter values may be case sensitive. Media types and their use in HTTP are described section 3.7 of the HTTP/1.1 specification [4].

There is no default value for this variable. If and only if it is unset, then the script MAY attempt to determine the media type from the data received. If the type remains unknown, then the script MAY choose to assume a type of application/octet-stream or it may reject the request with an error (as described in section 6.3.3).

Each media-type defines a set of optional and mandatory parameters. This may include a charset parameter with a case-insensitive value defining the coded character set for the message-body. If the

charset parameter is omitted, then the default value should be derived according to whichever of the following rules is the first to apply:

1. There MAY be a system-defined default charset for some media-types.

2. The default for media-types of type "text" is ISO-8859-1 [4].

3. Any default defined in the media-type specification.

4. The default is US-ASCII.

The server MUST set this meta-variable if an HTTP Content-Type field is present in the client request header. If the server receives a request with an attached entity but no Content-Type header field, it MAY attempt to determine the correct content type, otherwise it should omit this meta-variable.

4.1.4. GATEWAY_INTERFACE

The GATEWAY_INTERFACE variable MUST be set to the dialect of CGI being used by the server to communicate with the script. Syntax:

```
GATEWAY_INTERFACE = "CGI" "/" 1*digit "." 1*digit
```

Note that the major and minor numbers are treated as separate integers and hence each may be incremented higher than a single digit. Thus CGI/2.4 is a lower version than CGI/2.13 which in turn is lower than CGI/12.3. Leading zeros MUST be ignored by the script and MUST NOT be generated by the server.

This document defines the 1.1 version of the CGI interface.

4.1.5. PATH_INFO

The PATH_INFO variable specifies a path to be interpreted by the CGI script. It identifies the resource or sub-resource to be returned by the CGI script, and is derived from the portion of the URI path hierarchy following the part that identifies the script itself. Unlike a URI path, the PATH_INFO is not URL-encoded, and cannot contain path-segment parameters. A PATH_INFO of "/" represents a single void path segment.

```
PATH_INFO = "" | ( "/" path )
path      = lsegment *( "/" lsegment )
lsegment  = *lchar
lchar     = <any TEXT or CTL except "/">
```

The value is considered case-sensitive and the server MUST preserve the case of the path as presented in the request URI. The server MAY impose restrictions and limitations on what values it permits for PATH_INFO, and MAY reject the request with an error if it encounters any values considered objectionable. That MAY include any requests that would result in an encoded "/" being decoded into PATH_INFO, as this might represent a loss of information to the script. Similarly, treatment of non US-ASCII characters in the path is system-defined.

URL-encoded, the PATH_INFO string forms the extra-path component of the Script-URI (see section 3.3) which follows the SCRIPT_NAME part of that path.

4.1.6. PATH_TRANSLATED

The PATH_TRANSLATED variable is derived by taking the PATH_INFO value, parsing it as a local URI in its own right, and performing any virtual-to-physical translation appropriate to map it onto the server's document repository structure. The set of characters permitted in the result is system-defined.

```
PATH_TRANSLATED = *<any character>
```

This is the file location that would be accessed by a request for

```
<scheme> "://" <server-name> ":" <server-port> <extra-path>
```

where <scheme> is the scheme for the original client request and <extra-path> is a URL-encoded version of PATH_INFO, with ";", "=" and "?" reserved. For example, a request such as the following:

```
http://somehost.com/cgi-bin/somescript/this%2eis%2epath%3binfo
```

would result in a PATH_INFO value of

```
/this.is.the.path;info
```

An internal URI is constructed from the scheme, server location and the URL-encoded PATH_INFO:

```
http://somehost.com/this.is.the.path%3binfo
```

This would then be translated to a location in the server's document repository, perhaps a filesystem path something like this:

```
/usr/local/www/htdocs/this.is.the.path;info
```

The value of PATH_TRANSLATED is the result of the translation.

The value is derived in this way irrespective of whether it maps to a valid repository location. The server MUST preserve the case of the extra-path segment unless the underlying repository supports case-insensitive names. If the repository is only case-aware, case-preserving, or case-blind with regard to document names, the server is not required to preserve the case of the original segment through the translation.

The translation algorithm the server uses to derive PATH_TRANSLATED is implementation-defined; CGI scripts which use this variable may suffer limited portability.

The server SHOULD set this meta-variable if the request URI includes a path-info component. If PATH_INFO is NULL, then the PATH_TRANSLATED variable MUST be set to NULL (or unset).

4.1.7. QUERY_STRING

The QUERY_STRING variable contains a URL-encoded search or parameter string; it provides information to the CGI script to affect or refine the document to be returned by the script.

The URL syntax for a search string is described in section 3 of RFC 2396 [2]. The QUERY_STRING value is case-sensitive.

```
QUERY_STRING = query-string
query-string = *uric
uric         = reserved | unreserved | escaped
```

When parsing and decoding the query string, the details of the parsing, reserved characters and support for non US-ASCII characters depends on the context. For example, form submission from an HTML document [18] uses application/x-www-form-urlencoded encoding, in which the characters "+", "&" and "=" are reserved, and the ISO 8859-1 encoding may be used for non US-ASCII characters.

The QUERY_STRING value provides the query-string part of the Script-URI. (See section 3.3).

The server MUST set this variable; if the Script-URI does not include a query component, the QUERY_STRING MUST be defined as an empty string ("").

4.1.8. REMOTE_ADDR

The REMOTE_ADDR variable MUST be set to the network address of the client sending the request to the server.

```
REMOTE_ADDR  = hostnumber
hostnumber   = ipv4-address | ipv6-address
ipv4-address = 1*3digit "." 1*3digit "." 1*3digit "." 1*3digit
ipv6-address = hexpart [ ":" ipv4-address ]
hexpart      = hexseq | ( [ hexseq ] "::" [ hexseq ] )
hexseq       = 1*4hex *( ":" 1*4hex )
```

The format of an IPv6 address is described in RFC 3513 [15].

4.1.9. REMOTE_HOST

The REMOTE_HOST variable contains the fully qualified domain name of the client sending the request to the server, if available, otherwise NULL. Fully qualified domain names take the form as described in section 3.5 of RFC 1034 [17] and section 2.1 of RFC 1123 [12]. Domain names are not case sensitive.

```
REMOTE_HOST   = "" | hostname | hostnumber
hostname      = *( domainlabel "." ) toplabel [ "." ]
domainlabel   = alphanum [ *alphahypdigit alphanum ]
toplabel      = alpha [ *alphahypdigit alphanum ]
alphahypdigit = alphanum | "-"
```

The server SHOULD set this variable. If the hostname is not available for performance reasons or otherwise, the server MAY substitute the REMOTE_ADDR value.

4.1.10. REMOTE_IDENT

The REMOTE_IDENT variable MAY be used to provide identity information reported about the connection by an RFC 1413 [20] request to the remote agent, if available. The server may choose not to support this feature, or not to request the data for efficiency reasons, or not to return available identity data.

```
REMOTE_IDENT = *TEXT
```

The data returned may be used for authentication purposes, but the level of trust reposed in it should be minimal.

4.1.11. REMOTE_USER

The REMOTE_USER variable provides a user identification string supplied by client as part of user authentication.

```
REMOTE_USER = *TEXT
```

If the client request required HTTP Authentication [5] (e.g., the AUTH_TYPE meta-variable is set to "Basic" or "Digest"), then the value of the REMOTE_USER meta-variable MUST be set to the user-ID supplied.

4.1.12. REQUEST_METHOD

The REQUEST_METHOD meta-variable MUST be set to the method which should be used by the script to process the request, as described in section 4.3.

```
REQUEST_METHOD   = method
method           = "GET" | "POST" | "HEAD" | extension-method
extension-method = "PUT" | "DELETE" | token
```

The method is case sensitive. The HTTP methods are described in section 5.1.1 of the HTTP/1.0 specification [1] and section 5.1.1 of the HTTP/1.1 specification [4].

4.1.13. SCRIPT_NAME

The SCRIPT_NAME variable MUST be set to a URI path (not URL-encoded) which could identify the CGI script (rather than the script's output). The syntax is the same as for PATH_INFO (section 4.1.5)

```
SCRIPT_NAME = "" | ( "/" path )
```

The leading "/" is not part of the path. It is optional if the path is NULL; however, the variable MUST still be set in that case.

The SCRIPT_NAME string forms some leading part of the path component of the Script-URI derived in some implementation-defined manner. No PATH_INFO segment (see section 4.1.5) is included in the SCRIPT_NAME value.

4.1.14. SERVER_NAME

The SERVER_NAME variable MUST be set to the name of the server host to which the client request is directed. It is a case-insensitive hostname or network address. It forms the host part of the Script-URI.

```
SERVER_NAME = server-name
server-name = hostname | ipv4-address | ( "[" ipv6-address "]" )
```

A deployed server can have more than one possible value for this variable, where several HTTP virtual hosts share the same IP address. In that case, the server would use the contents of the request's Host header field to select the correct virtual host.

4.1.15. SERVER_PORT

The SERVER_PORT variable MUST be set to the TCP/IP port number on which this request is received from the client. This value is used in the port part of the Script-URI.

```
SERVER_PORT = server-port
server-port = 1*digit
```

Note that this variable MUST be set, even if the port is the default port for the scheme and could otherwise be omitted from a URI.

4.1.16. SERVER_PROTOCOL

The SERVER_PROTOCOL variable MUST be set to the name and version of the application protocol used for this CGI request. This MAY differ from the protocol version used by the server in its communication with the client.

```
SERVER_PROTOCOL   = HTTP-Version | "INCLUDED" | extension-version
HTTP-Version      = "HTTP" "/" 1*digit "." 1*digit
extension-version = protocol [ "/" 1*digit "." 1*digit ]
protocol          = token
```

Here, 'protocol' defines the syntax of some of the information passing between the server and the script (the 'protocol-specific' features). It is not case sensitive and is usually presented in upper case. The protocol is not the same as the scheme part of the script URI, which defines the overall access mechanism used by the client to communicate with the server. For example, a request that reaches the script with a protocol of "HTTP" may have used an "https" scheme.

A well-known value for SERVER_PROTOCOL which the server MAY use is "INCLUDED", which signals that the current document is being included as part of a composite document, rather than being the direct target of the client request. The script should treat this as an HTTP/1.0 request.

4.1.17. SERVER_SOFTWARE

The SERVER_SOFTWARE meta-variable MUST be set to the name and version of the information server software making the CGI request (and running the gateway). It SHOULD be the same as the server description reported to the client, if any.

```
SERVER_SOFTWARE = 1*( product | comment )
product         = token [ "/" product-version ]
product-version = token
comment         = "(" *( ctext | comment ) ")"
ctext           = <any TEXT excluding "(" and ")">
```

4.1.18. Protocol-Specific Meta-Variables

The server SHOULD set meta-variables specific to the protocol and scheme for the request. Interpretation of protocol-specific variables depends on the protocol version in SERVER_PROTOCOL. The server MAY set a meta-variable with the name of the scheme to a non-NULL value if the scheme is not the same as the protocol. The presence of such a variable indicates to a script which scheme is used by the request.

Meta-variables with names beginning with "HTTP_" contain values read from the client request header fields, if the protocol used is HTTP. The HTTP header field name is converted to upper case, has all occurrences of "-" replaced with "_" and has "HTTP_" prepended to give the meta-variable name. The header data can be presented as sent by the client, or can be rewritten in ways which do not change its semantics. If multiple header fields with the same field-name are received then the server MUST rewrite them as a single value having the same semantics. Similarly, a header field that spans multiple lines MUST be merged onto a single line. The server MUST, if necessary, change the representation of the data (for example, the character set) to be appropriate for a CGI meta-variable.

The server is not required to create meta-variables for all the header fields that it receives. In particular, it SHOULD remove any header fields carrying authentication information, such as 'Authorization'; or that are available to the script in other variables, such as 'Content-Length' and 'Content-Type'. The server MAY remove header fields that relate solely to client-side communication issues, such as 'Connection'.

4.2. Request Message-Body

Request data is accessed by the script in a system-defined method; unless defined otherwise, this will be by reading the 'standard input' file descriptor or file handle.

```
Request-Data   = [ request-body ] [ extension-data ]
request-body   = <CONTENT_LENGTH>OCTET
extension-data = *OCTET
```

A request-body is supplied with the request if the CONTENT_LENGTH is not NULL. The server MUST make at least that many bytes available for the script to read. The server MAY signal an end-of-file condition after CONTENT_LENGTH bytes have been read or it MAY supply extension data. Therefore, the script MUST NOT attempt to read more than CONTENT_LENGTH bytes, even if more data is available. However, it is not obliged to read any of the data.

For non-parsed header (NPH) scripts (section 5), the server SHOULD attempt to ensure that the data supplied to the script is precisely as supplied by the client and is unaltered by the server.

As transfer-codings are not supported on the request-body, the server MUST remove any such codings from the message-body, and recalculate the CONTENT_LENGTH. If this is not possible (for example, because of large buffering requirements), the server SHOULD reject the client request. It MAY also remove content-codings from the message-body.

4.3. Request Methods

The Request Method, as supplied in the REQUEST_METHOD meta-variable, identifies the processing method to be applied by the script in producing a response. The script author can choose to implement the methods most appropriate for the particular application. If the script receives a request with a method it does not support it SHOULD reject it with an error (see section 6.3.3).

4.3.1. GET

The GET method indicates that the script should produce a document based on the meta-variable values. By convention, the GET method is 'safe' and 'idempotent' and SHOULD NOT have the significance of taking an action other than producing a document.

The meaning of the GET method may be modified and refined by protocol-specific meta-variables.

4.3.2. POST

The POST method is used to request the script perform processing and produce a document based on the data in the request message-body, in addition to meta-variable values. A common use is form submission in HTML [18], intended to initiate processing by the script that has a permanent affect, such a change in a database.

The script MUST check the value of the CONTENT_LENGTH variable before reading the attached message-body, and SHOULD check the CONTENT_TYPE value before processing it.

4.3.3. HEAD

The HEAD method requests the script to do sufficient processing to return the response header fields, without providing a response message-body. The script MUST NOT provide a response message-body for a HEAD request. If it does, then the server MUST discard the message-body when reading the response from the script.

4.3.4. Protocol-Specific Methods

The script MAY implement any protocol-specific method, such as HTTP/1.1 PUT and DELETE; it SHOULD check the value of SERVER_PROTOCOL when doing so.

The server MAY decide that some methods are not appropriate or permitted for a script, and may handle the methods itself or return an error to the client.

4.4. The Script Command Line

Some systems support a method for supplying an array of strings to the CGI script. This is only used in the case of an 'indexed' HTTP query, which is identified by a 'GET' or 'HEAD' request with a URI query string that does not contain any unencoded "=" characters. For such a request, the server SHOULD treat the query-string as a search-string and parse it into words, using the rules

```
search-string = search-word *( "+" search-word )
search-word   = 1*schar
schar         = unreserved | escaped | xreserved
xreserved     = ";" | "/" | "?" | ":" | "@" | "&" | "=" | "," |
                "$"
```

After parsing, each search-word is URL-decoded, optionally encoded in a system-defined manner and then added to the command line argument list.

If the server cannot create any part of the argument list, then the server MUST NOT generate any command line information. For example, the number of arguments may be greater than operating system or server limits, or one of the words may not be representable as an argument.

The script SHOULD check to see if the QUERY_STRING value contains an unencoded "=" character, and SHOULD NOT use the command line arguments if it does.

5. NPH Scripts

5.1. Identification

The server MAY support NPH (Non-Parsed Header) scripts; these are scripts to which the server passes all responsibility for response processing.

This specification provides no mechanism for an NPH script to be identified on the basis of its output data alone. By convention, therefore, any particular script can only ever provide output of one type (NPH or CGI) and hence the script itself is described as an 'NPH script'. A server with NPH support MUST provide an implementation-defined mechanism for identifying NPH scripts, perhaps based on the name or location of the script.

5.2. NPH Response

There MUST be a system-defined method for the script to send data back to the server or client; a script MUST always return some data. Unless defined otherwise, this will be the same as for conventional CGI scripts.

Currently, NPH scripts are only defined for HTTP client requests. An (HTTP) NPH script MUST return a complete HTTP response message, currently described in section 6 of the HTTP specifications [1], [4]. The script MUST use the SERVER_PROTOCOL variable to determine the appropriate format for a response. It MUST also take account of any generic or protocol-specific meta-variables in the request as might be mandated by the particular protocol specification.

The server MUST ensure that the script output is sent to the client unmodified. Note that this requires the script to use the correct character set (US-ASCII [9] and ISO 8859-1 [10] for HTTP) in the header fields. The server SHOULD attempt to ensure that the script output is sent directly to the client, with minimal internal and no transport-visible buffering.

Unless the implementation defines otherwise, the script MUST NOT indicate in its response that the client can send further requests over the same connection.

6. CGI Response

6.1. Response Handling

A script MUST always provide a non-empty response, and so there is a system-defined method for it to send this data back to the server. Unless defined otherwise, this will be via the 'standard output' file descriptor.

The script MUST check the REQUEST_METHOD variable when processing the request and preparing its response.

The server MAY implement a timeout period within which data must be received from the script. If a server implementation defines such a timeout and receives no data from a script within the timeout period, the server MAY terminate the script process.

6.2. Response Types

The response comprises a message-header and a message-body, separated by a blank line. The message-header contains one or more header fields. The body may be NULL.

```
generic-response = 1*header-field NL [ response-body ]
```

The script MUST return one of either a document response, a local redirect response or a client redirect (with optional document) response. In the response definitions below, the order of header fields in a response is not significant (despite appearing so in the BNF). The header fields are defined in section 6.3.

```
CGI-Response = document-response | local-redir-response |
               client-redir-response | client-redirdoc-response
```

6.2.1. Document Response

The CGI script can return a document to the user in a document response, with an optional error code indicating the success status of the response.

```
document-response = Content-Type [ Status ] *other-field NL
                    response-body
```

The script MUST return a Content-Type header field. A Status header field is optional, and status 200 'OK' is assumed if it is omitted. The server MUST make any appropriate modifications to the script's output to ensure that the response to the client complies with the response protocol version.

6.2.2. Local Redirect Response

The CGI script can return a URI path and query-string ('local-pathquery') for a local resource in a Location header field. This indicates to the server that it should reprocess the request using the path specified.

```
local-redir-response = local-Location NL
```

The script MUST NOT return any other header fields or a message-body, and the server MUST generate the response that it would have produced in response to a request containing the URL

```
scheme "://" server-name ":" server-port local-pathquery
```

6.2.3. Client Redirect Response

The CGI script can return an absolute URI path in a Location header field, to indicate to the client that it should reprocess the request using the URI specified.

```
client-redir-response = client-Location *extension-field NL
```

The script MUST not provide any other header fields, except for server-defined CGI extension fields. For an HTTP client request, the server MUST generate a 302 'Found' HTTP response message.

6.2.4. Client Redirect Response with Document

The CGI script can return an absolute URI path in a Location header field together with an attached document, to indicate to the client that it should reprocess the request using the URI specified.

```
client-redirdoc-response = client-Location Status Content-Type
                           *other-field NL response-body
```

The Status header field MUST be supplied and MUST contain a status value of 302 'Found', or it MAY contain an extension-code, that is, another valid status code that means client redirection. The server MUST make any appropriate modifications to the script's output to ensure that the response to the client complies with the response protocol version.

6.3. Response Header Fields

The response header fields are either CGI or extension header fields to be interpreted by the server, or protocol-specific header fields to be included in the response returned to the client. At least one CGI field MUST be supplied; each CGI field MUST NOT appear more than once in the response. The response header fields have the syntax:

```
header-field    = CGI-field | other-field
CGI-field       = Content-Type | Location | Status
other-field     = protocol-field | extension-field
protocol-field  = generic-field
extension-field = generic-field
generic-field   = field-name ":" [ field-value ] NL
field-name      = token
field-value     = *( field-content | LWSP )
field-content   = *( token | separator | quoted-string )
```

The field-name is not case sensitive. A NULL field value is equivalent to a field not being sent. Note that each header field in a CGI-Response MUST be specified on a single line; CGI/1.1 does not support continuation lines. Whitespace is permitted between the ":" and the field-value (but not between the field-name and the ":"), and also between tokens in the field-value.

6.3.1. Content-Type

The Content-Type response field sets the Internet Media Type [6] of the entity body.

```
Content-Type = "Content-Type:" media-type NL
```

If an entity body is returned, the script MUST supply a Content-Type field in the response. If it fails to do so, the server SHOULD NOT attempt to determine the correct content type. The value SHOULD be sent unmodified to the client, except for any charset parameter changes.

Unless it is otherwise system-defined, the default charset assumed by the client for text media-types is ISO-8859-1 if the protocol is HTTP and US-ASCII otherwise. Hence the script SHOULD include a charset parameter. See section 3.4.1 of the HTTP/1.1 specification [4] for a discussion of this issue.

6.3.2. Location

The Location header field is used to specify to the server that the script is returning a reference to a document rather than an actual document (see sections 6.2.3 and 6.2.4). It is either an absolute URI (optionally with a fragment identifier), indicating that the client is to fetch the referenced document, or a local URI path (optionally with a query string), indicating that the server is to fetch the referenced document and return it to the client as the response.

```
Location       = local-Location | client-Location
client-Location = "Location:" fragment-URI NL
local-Location  = "Location:" local-pathquery NL
fragment-URI    = absoluteURI [ "#" fragment ]
fragment        = *uric
local-pathquery = abs-path [ "?" query-string ]
abs-path        = "/" path-segments
path-segments   = segment *( "/" segment )
segment         = *pchar
pchar           = unreserved | escaped | extra
extra           = ":" | "@" | "&" | "=" | "+" | "$" | ","
```

The syntax of an absoluteURI is incorporated into this document from that specified in RFC 2396 [2] and RFC 2732 [7]. A valid absoluteURI always starts with the name of scheme followed by ":"; scheme names start with a letter and continue with alphanumerics, "+", "-" or ".". The local URI path and query must be an absolute path, and not a relative path or NULL, and hence must start with a "/".

Note that any message-body attached to the request (such as for a POST request) may not be available to the resource that is the target of the redirect.

6.3.3. Status

The Status header field contains a 3-digit integer result code that indicates the level of success of the script's attempt to handle the request.

```
Status         = "Status:" status-code SP reason-phrase NL
status-code    = "200" | "302" | "400" | "501" | extension-code
extension-code = 3digit
reason-phrase  = *TEXT
```

Status code 200 'OK' indicates success, and is the default value assumed for a document response. Status code 302 'Found' is used with a Location header field and response message-body. Status code

400 'Bad Request' may be used for an unknown request format, such as a missing CONTENT_TYPE. Status code 501 'Not Implemented' may be returned by a script if it receives an unsupported REQUEST_METHOD.

Other valid status codes are listed in section 6.1.1 of the HTTP specifications [1], [4], and also the IANA HTTP Status Code Registry [8] and MAY be used in addition to or instead of the ones listed above. The script SHOULD check the value of SERVER_PROTOCOL before using HTTP/1.1 status codes. The script MAY reject with error 405 'Method Not Allowed' HTTP/1.1 requests made using a method it does not support.

Note that returning an error status code does not have to mean an error condition with the script itself. For example, a script that is invoked as an error handler by the server should return the code appropriate to the server's error condition.

The reason-phrase is a textual description of the error to be returned to the client for human consumption.

6.3.4. Protocol-Specific Header Fields

The script MAY return any other header fields that relate to the response message defined by the specification for the SERVER_PROTOCOL (HTTP/1.0 [1] or HTTP/1.1 [4]). The server MUST translate the header data from the CGI header syntax to the HTTP header syntax if these differ. For example, the character sequence for newline (such as UNIX's US-ASCII LF) used by CGI scripts may not be the same as that used by HTTP (US-ASCII CR followed by LF).

The script MUST NOT return any header fields that relate to client-side communication issues and could affect the server's ability to send the response to the client. The server MAY remove any such header fields returned by the client. It SHOULD resolve any conflicts between header fields returned by the script and header fields that it would otherwise send itself.

6.3.5. Extension Header Fields

There may be additional implementation-defined CGI header fields, whose field names SHOULD begin with "X-CGI-". The server MAY ignore (and delete) any unrecognised header fields with names beginning "X-CGI-" that are received from the script.

6.4. Response Message-Body

The response message-body is an attached document to be returned to the client by the server. The server MUST read all the data provided by the script, until the script signals the end of the message-body by way of an end-of-file condition. The message-body SHOULD be sent unmodified to the client, except for HEAD requests or any required transfer-codings, content-codings or charset conversions.

```
response-body = *OCTET
```

7. System Specifications

7.1. AmigaDOS

Meta-Variables
: Meta-variables are passed to the script in identically named environment variables. These are accessed by the DOS library routine GetVar(). The flags argument SHOULD be 0. Case is ignored, but upper case is recommended for compatibility with case-sensitive systems.

The current working directory
: The current working directory for the script is set to the directory containing the script.

Character set
: The US-ASCII character set [9] is used for the definition of meta-variables, header fields and values; the newline (NL) sequence is LF; servers SHOULD also accept CR LF as a newline.

7.2. UNIX

For UNIX compatible operating systems, the following are defined:

Meta-Variables
: Meta-variables are passed to the script in identically named environment variables. These are accessed by the C library routine getenv() or variable environ.

The command line
: This is accessed using the argc and argv arguments to main(). The words have any characters which are 'active' in the Bourne shell escaped with a backslash.

The current working directory
: The current working directory for the script SHOULD be set to the directory containing the script.

Character set
The US-ASCII character set [9], excluding NUL, is used for the definition of meta-variables, header fields and CHAR values; TEXT values use ISO-8859-1. The PATH_TRANSLATED value can contain any 8-bit byte except NUL. The newline (NL) sequence is LF; servers should also accept CR LF as a newline.

7.3. EBCDIC/POSIX

For POSIX compatible operating systems using the EBCDIC character set, the following are defined:

Meta-Variables
Meta-variables are passed to the script in identically named environment variables. These are accessed by the C library routine getenv().

The command line
This is accessed using the argc and argv arguments to main(). The words have any characters which are 'active' in the Bourne shell escaped with a backslash.

The current working directory
The current working directory for the script SHOULD be set to the directory containing the script.

Character set
The IBM1047 character set [21], excluding NUL, is used for the definition of meta-variables, header fields, values, TEXT strings and the PATH_TRANSLATED value. The newline (NL) sequence is LF; servers should also accept CR LF as a newline.

media-type charset default
The default charset value for text (and other implementation-defined) media types is IBM1047.

8. Implementation

8.1. Recommendations for Servers

Although the server and the CGI script need not be consistent in their handling of URL paths (client URLs and the PATH_INFO data, respectively), server authors may wish to impose consistency. So the server implementation should specify its behaviour for the following cases:

1. define any restrictions on allowed path segments, in particular whether non-terminal NULL segments are permitted;

2. define the behaviour for "." or ".." path segments; i.e., whether they are prohibited, treated as ordinary path segments or interpreted in accordance with the relative URL specification [2];

3. define any limits of the implementation, including limits on path or search string lengths, and limits on the volume of header fields the server will parse.

8.2. Recommendations for Scripts

If the script does not intend processing the PATH_INFO data, then it should reject the request with 404 Not Found if PATH_INFO is not NULL.

If the output of a form is being processed, check that CONTENT_TYPE is "application/x-www-form-urlencoded" [18] or "multipart/form-data" [16]. If CONTENT_TYPE is blank, the script can reject the request with a 415 'Unsupported Media Type' error, where supported by the protocol.

When parsing PATH_INFO, PATH_TRANSLATED or SCRIPT_NAME the script should be careful of void path segments ("//") and special path segments ("." and ".."). They should either be removed from the path before use in OS system calls, or the request should be rejected with 404 'Not Found'.

When returning header fields, the script should try to send the CGI header fields as soon as possible, and should send them before any HTTP header fields. This may help reduce the server's memory requirements.

Script authors should be aware that the REMOTE_ADDR and REMOTE_HOST meta-variables (see sections 4.1.8 and 4.1.9) may not identify the ultimate source of the request. They identify the client for the immediate request to the server; that client may be a proxy, gateway, or other intermediary acting on behalf of the actual source client.

9. Security Considerations

9.1. Safe Methods

As discussed in the security considerations of the HTTP specifications [1], [4], the convention has been established that the GET and HEAD methods should be 'safe' and 'idempotent' (repeated requests have the same effect as a single request). See section 9.1 of RFC 2616 [4] for a full discussion.

9.2. Header Fields Containing Sensitive Information

Some HTTP header fields may carry sensitive information which the server should not pass on to the script unless explicitly configured to do so. For example, if the server protects the script by using the Basic authentication scheme, then the client will send an Authorization header field containing a username and password. The server validates this information and so it should not pass on the password via the HTTP_AUTHORIZATION meta-variable without careful consideration. This also applies to the Proxy-Authorization header field and the corresponding HTTP_PROXY_AUTHORIZATION meta-variable.

9.3. Data Privacy

Confidential data in a request should be placed in a message-body as part of a POST request, and not placed in the URI or message headers. On some systems, the environment used to pass meta-variables to a script may be visible to other scripts or users. In addition, many existing servers, proxies and clients will permanently record the URI where it might be visible to third parties.

9.4. Information Security Model

For a client connection using TLS, the security model applies between the client and the server, and not between the client and the script. It is the server's responsibility to handle the TLS session, and thus it is the server which is authenticated to the client, not the CGI script.

This specification provides no mechanism for the script to authenticate the server which invoked it. There is no enforced integrity on the CGI request and response messages.

9.5. Script Interference with the Server

The most common implementation of CGI invokes the script as a child process using the same user and group as the server process. It should therefore be ensured that the script cannot interfere with the server process, its configuration, documents or log files.

If the script is executed by calling a function linked in to the server software (either at compile-time or run-time) then precautions should be taken to protect the core memory of the server, or to ensure that untrusted code cannot be executed.

9.6. Data Length and Buffering Considerations

This specification places no limits on the length of the message-body presented to the script. The script should not assume that statically allocated buffers of any size are sufficient to contain the entire submission at one time. Use of a fixed length buffer without careful overflow checking may result in an attacker exploiting 'stack-smashing' or 'stack-overflow' vulnerabilities of the operating system. The script may spool large submissions to disk or other buffering media, but a rapid succession of large submissions may result in denial of service conditions. If the CONTENT_LENGTH of a message-body is larger than resource considerations allow, scripts should respond with an error status appropriate for the protocol version; potentially applicable status codes include 503 'Service Unavailable' (HTTP/1.0 and HTTP/1.1), 413 'Request Entity Too Large' (HTTP/1.1), and 414 'Request-URI Too Large' (HTTP/1.1).

Similar considerations apply to the server's handling of the CGI response from the script. There is no limit on the length of the header or message-body returned by the script; the server should not assume that statically allocated buffers of any size are sufficient to contain the entire response.

9.7. Stateless Processing

The stateless nature of the Web makes each script execution and resource retrieval independent of all others even when multiple requests constitute a single conceptual Web transaction. Because of this, a script should not make any assumptions about the context of the user-agent submitting a request. In particular, scripts should examine data obtained from the client and verify that they are valid, both in form and content, before allowing them to be used for sensitive purposes such as input to other applications, commands, or operating system services. These uses include (but are not limited to) system call arguments, database writes, dynamically evaluated source code, and input to billing or other secure processes. It is important that applications be protected from invalid input regardless of whether the invalidity is the result of user error, logic error, or malicious action.

Authors of scripts involved in multi-request transactions should be particularly cautious about validating the state information; undesirable effects may result from the substitution of dangerous values for portions of the submission which might otherwise be presumed safe. Subversion of this type occurs when alterations are made to data from a prior stage of the transaction that were not meant to be controlled by the client (e.g., hidden HTML form elements, cookies, embedded URLs, etc.).

9.8. Relative Paths

The server should be careful of ".." path segments in the request URI. These should be removed or resolved in the request URI before it is split into the script-path and extra-path. Alternatively, when the extra-path is used to find the PATH_TRANSLATED, care should be taken to avoid the path resolution from providing translated paths outside an expected path hierarchy.

9.9. Non-parsed Header Output

If a script returns a non-parsed header output, to be interpreted by the client in its native protocol, then the script must address all security considerations relating to that protocol.

10. Acknowledgements

This work is based on the original CGI interface that arose out of discussions on the 'www-talk' mailing list. In particular, Rob McCool, John Franks, Ari Luotonen, George Phillips and Tony Sanders deserve special recognition for their efforts in defining and implementing the early versions of this interface.

This document has also greatly benefited from the comments and suggestions made Chris Adie, Dave Kristol and Mike Meyer; also David Morris, Jeremy Madea, Patrick McManus, Adam Donahue, Ross Patterson and Harald Alvestrand.

11. References

11.1 Normative References

[1] Berners-Lee, T., Fielding, R. and H. Frystyk, "Hypertext Transfer Protocol -- HTTP/1.0", RFC 1945, May 1996.

[2] Berners-Lee, T., Fielding, R. and L. Masinter, "Uniform Resource Identifiers (URI) : Generic Syntax", RFC 2396, August 1998.

[3] Bradner, S., "Key words for use in RFCs to Indicate Requirements Levels", BCP 14, RFC 2119, March 1997.

[4] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[5] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999.

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

[7] Hinden, R., Carpenter, B., and L. Masinter, "Format for Literal IPv6 Addresses in URL's", RFC 2732, December 1999.

[8] "HTTP Status Code Registry", http://www.iana.org/assignments/http-status-codes, IANA.

[9] "Information Systems -- Coded Character Sets -- 7-bit American Standard Code for Information Interchange (7-Bit ASCII)", ANSI INCITS.4-1986 (R2002).

[10] "Information technology -- 8-bit single-byte coded graphic character sets -- Part 1: Latin alphabet No. 1", ISO/IEC 8859-1:1998.

11.2. Informative References

[11] Berners-Lee, T., "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, June 1994.

[12] Braden, R., Ed., "Requirements for Internet Hosts -- Application and Support", STD 3, RFC 1123, October 1989.

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

[14] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.

[15] Hinden R. and S. Deering, "Internet Protocol Version 6 (IPv6) Addressing Architecture", RFC 3513, April 2003.

[16] Masinter, L., "Returning Values from Forms: multipart/form-data", RFC 2388, August 1998.

[17] Mockapetris, P., "Domain Names - Concepts and Facilities", STD 13, RFC 1034, November 1987.

[18] Raggett, D., Le Hors, A., and I. Jacobs, Eds., "HTML 4.01 Specification", W3C Recommendation December 1999, http://www.w3.org/TR/html401/.

[19] Rescola, E. "HTTP Over TLS", RFC 2818, May 2000.

[20] St. Johns, M., "Identification Protocol", RFC 1413, February 1993.

[21] IBM National Language Support Reference Manual Volume 2, SE09-8002-01, March 1990.

[22] "The Common Gateway Interface", http://hoohoo.ncsa.uiuc.edu/cgi/, NCSA, University of Illinois.

12. Authors' Addresses

David Robinson
The Apache Software Foundation

EMail: drtr@apache.org

Ken A. L. Coar
The Apache Software Foundation

EMail: coar@apache.org

13. Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.