The HTTP parser plugin
PRINTED MANUAL

HTTP parser plugin

© 1999-2024 AGG Software

Printed: 11/2/2024

Publisher

AGG Software

Production

© 1999-2024 AGG Software
http://www.aggsoft.com

Table of Contents

1   Introduction

This parser module mostly works together with Advanced TCP/IP Data Logger and implements very basic HTTP server features. Hypertext Transfer Protocol (HTTP) is a communications protocol for the transfer of information on the Internet. HTTP is a request/response standard between a client and a server. A client is your device, the server is our application. The client making an HTTP request and send any data. Our software parse requests and can export it with many data export modules.

The parser allows you to parse and export variables from a HTTP header and a HTTP request body string and sends answers, if this is necessary. This isn't required to install other HTTP servers and create any scripts for web servers to collect and export data from your devices.

Key features:

- Parsing of a HTTP header;
- Parsing of a query string;
- Detects a request method (POST or GET) and a protocol automatically;
- Allows extract and convert parser items from the string data type to other data types;
- The parser can filter requests;
- Allows you to send responses with code 403, 404 or from a file or a folder.

2   System requirements

The following requirements must be met for "HTTP parser" to be installed:

Operating system: Windows 2000 SP4 and above, including both x86 and x64 workstations and servers. The latest service pack for the corresponding OS is required.

Free disk space: Not less than 5 MB of free disk space is recommended.

Special access requirements: You should log on as a user with Administrator rights in order to install this module.

The main application (core) must be installed, for example, Advanced Serial Data Logger.

3   Installing HTTP parser

1. Close the main application (for example, Advanced Serial Data Logger) if it is running;
2. Copy the program to your hard drive;
3. Run the module installation file with a double click on the file name in Windows Explorer;
4. Follow the instructions of the installation software. Usually, it is enough just to click the "Next" button several times;
5. Start the main application. The name of the module will appear on the "Modules" tab of the "Settings" window if it is successfully installed.

If the module is compatible with the program, its name and version will be displayed in the module list. You can see examples of installed modules on fig.1-2. Some types of modules require additional configuration. To do it, just select a module from the list and click the "Setup" button next to the list. The configuration of the module is described below.

You can see some types of modules on the "Log file" tab. To configure such a module, you should select it from the "File type" list and click the "Advanced" button.
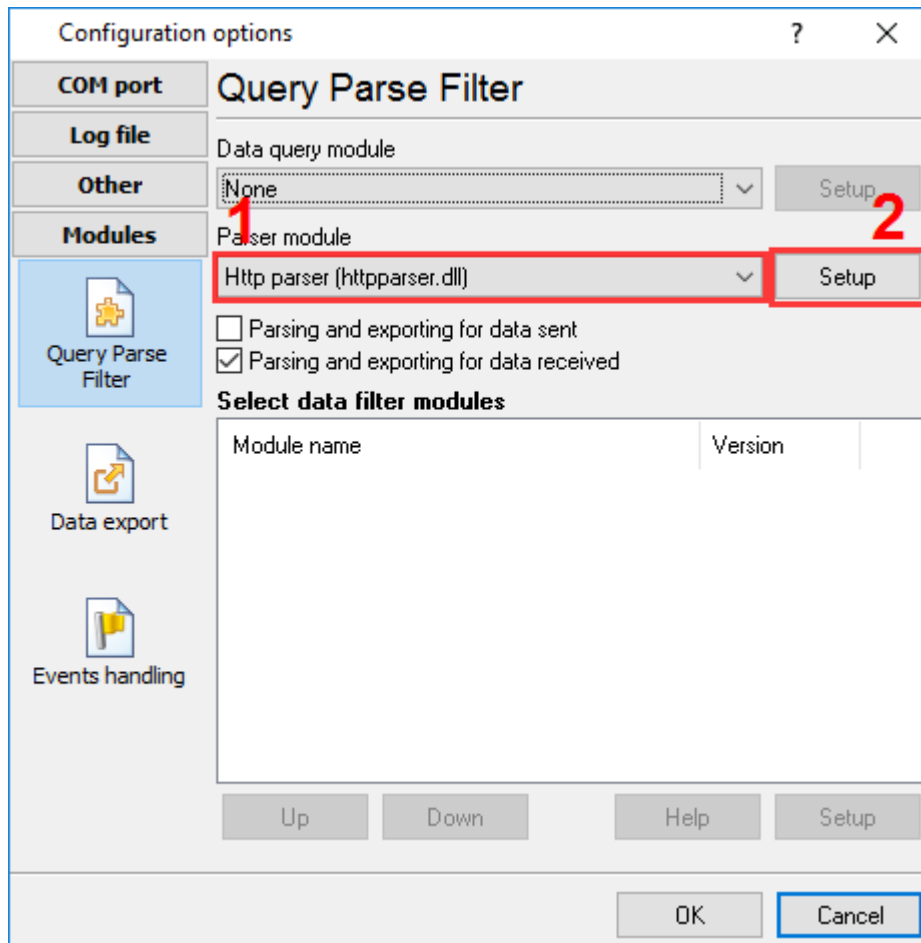
Fig. 1. Example of installed module

4   Glossary

Main program - it is the main executable of the application, for example, Advanced Serial Data Logger and asdlog.exe. It allows you to create several configurations with different settings and use different plugins.

Plugin - it is the additional plugin module for the main program. The plugin module extends the functionality of the main program.

Parser - it is the plugin module that processes the data flow, singling out data packets from it, and then variables from data packets. These variables are used in data export modules after that.

Core - see "Main program."

5   Parsing data

The request message consists of the following:

- Request line, such as GET /images/logo.gif HTTP/1.1, which requests the file logo.gif from the /images directory
- Headers, such as Accept-Language: en
- An empty line
- An optional message body

The request line and headers must all end with <CR><LF> (that is, a carriage return followed by a line feed). The empty line must consist of only <CR><LF> and no other whitespace. In the HTTP/1.1 protocol, all headers except Host are optional.

The parser verifies an HTTP message and extracts all fields from a header or message body. The parser doesn't work with body of multipart/mixed POST requests.
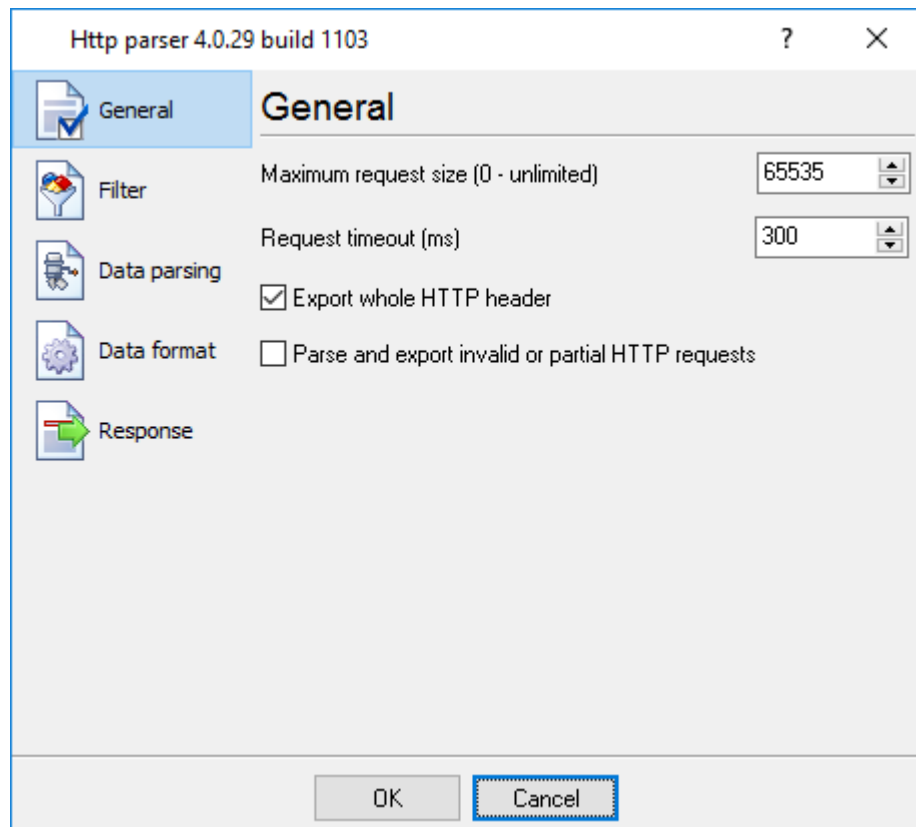

Fig.1. General options

- **Maximum request size** – you can specify the size of the request. This options allows you to protect the server from bad large request messages. If this values is greater than zero, then the parser will parse first N bytes only;

- **Request timeout** – if the parser didn't receive a valid HTTP request from the client within this interval, then the parser may ignore this request and don't export data;
- **Export whole HTTP header** – if this option is on, then the parser will parse and export all header items. Otherwise the parser will export, which you'll define on the "Data parsing" page. If you'll define the variable on the "Data parsing" page, then you can additionally convert header items to any data type;
- **Parse and export invalid or partial HTTP requests** – if this option is on, then the parser will parse and export HTTP requests, which don't match the HTTP specification or received partially.

Data parsing variants

You can find several real-life examples on our site where different data parsing methods are used. It can help you understand how the parser works. https://www.aggsoft.com/serial-data-logger/tutorials.htm

Example:

Please take a look at the example below (it is a typical HTTP request):

```
GET /js/some.js HTTP/1.1
Host: 127.0.0.1:10000
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; rv:1.8.1.17) Gecko/20080829 Firefox/2.0.0.17
Accept: */*
Accept-Language: q=0.8,en-us;q=0.5,en;q=0.3
Accept-Charset: windows-1250,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://127.0.0.1:10000/incl/meta1.txt
```

The data above consists of several strings with different length, all values in the string are delimited by :. In our case, they are:
- **Host** - the string "127.0.0.1:10000";
- **User-Agent** – user agent;
- and so on

The parameter window is located on the "Data parsing" page (fig.2). You may add several variables and specify their parameters and a data type: string, date, time, etc. Each variable in the list may have a different data type. Data types supported by the module are listed below.
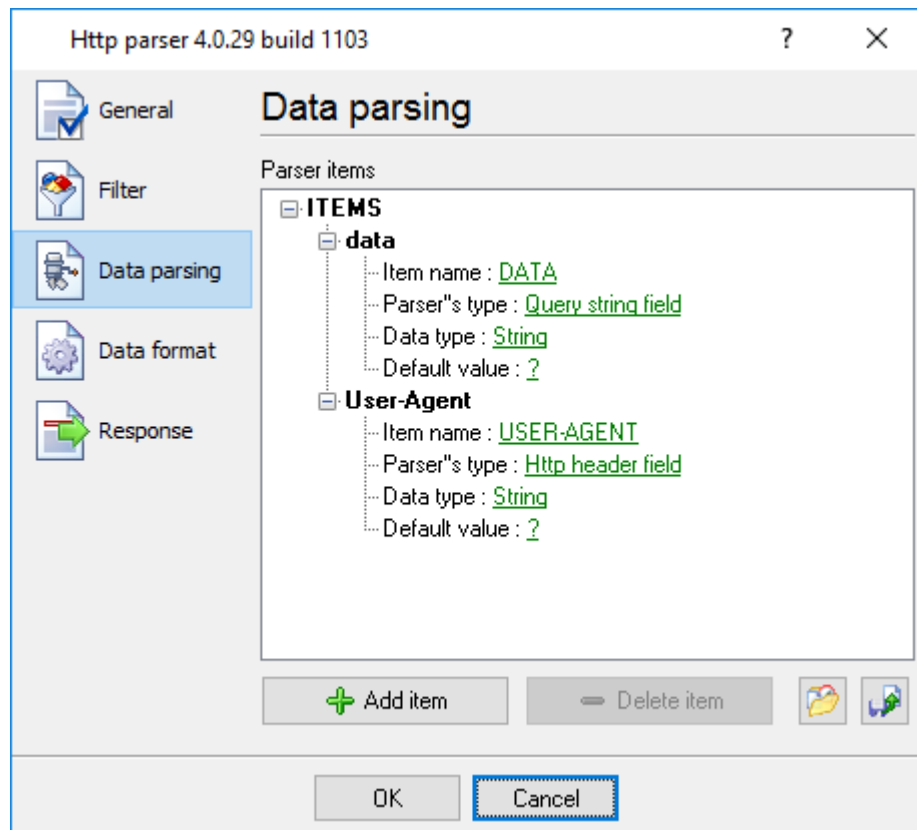
Fig.2. Parsing options

1. Http header field

The parser will extract a field from a header with name, which you'll define in the "Item name" field. Additionally you can specify the data type and default value for this field.

2. Query string field

Same as above, but works with an HTTP request body (query string).

3. Fixed position

Allows you to extract data from a fixed position in your HTTP requests. It is useful if you want to extract the method of a HTTP request (POST or GET).

4. Delimited

Allows you to extract values, which are delimited by any character or text. In our case, you can extract header field using the ": " delimiter.

5. None – If you select this type, the parser will not extract a value from the data packet, but will just create a variable with the NULL value.

6. Default – If you select this type, the parser will not single out a value from the data packet, but will just create a variable with the value you specify in the "Default value" field.

7. Regular expression - a widely used data parsing method that uses templates with certain syntax to search for data. The general rules of specifying regular expressions are described in next chapter. In this mode you must specify regular expression text with one subexpression. Result of this subexpression will be used as parsing result.

Data types

- String - String value - Character array with length from 1 to 65535 characters;
- Memo - String value - Character array with length from 1 to 2^32 characters;
- Bytes - Binary value;
- Blob - Binary Large Object field (bytes set);
- Boolean - Logical value (True/False) - 0 or 1;
- Float - Real number - value range: -2.9 x 10^-39 .. 1.7 x 10^38
- Smallint - Signed small integer - value range: 32768..32767;
- Word - Word (unsigned small integer) - value range: 0..65535;
- Integer - Signed integer value: -2147483648..2147483647;
- Date - Date;
- Time - Time;
- DateTime - Date and time.

6 Data format

You can use the following tab to specify the data format for some data types (see figure below).
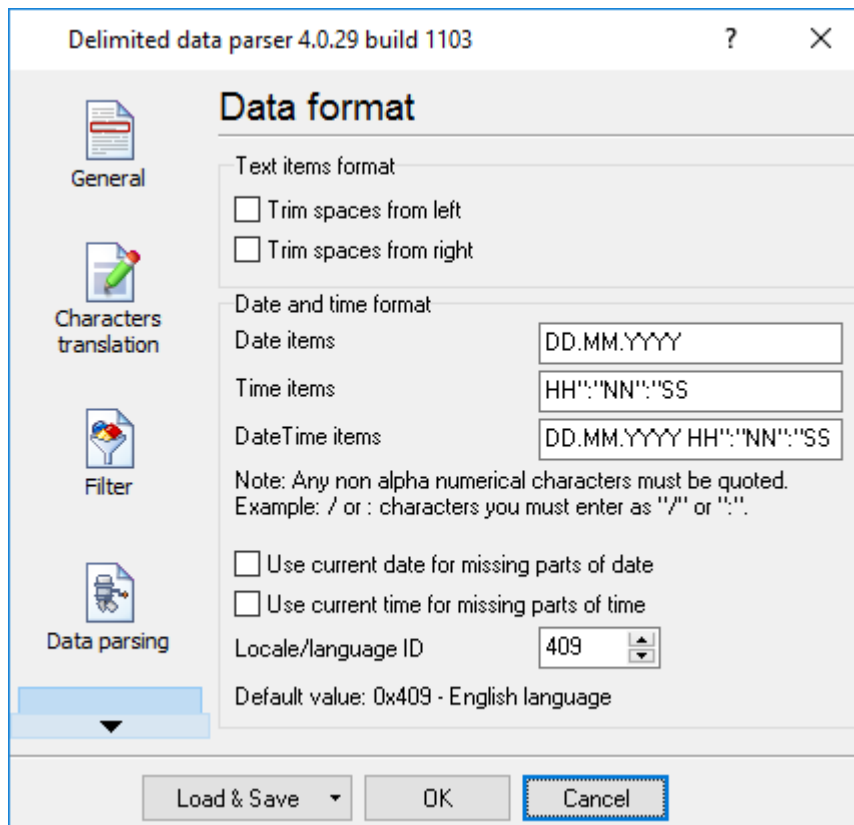
Fig. 2. Data format

Text data format – this group of options allows you to remove leading or trailing spaces in variables of the String type.

Date and time format – it often happens that the date or time format the external device sends does not coincide with the format in which this data is stored in the database. To convert a date into the necessary format, you should specify the format of the received date and time.

The same format is used to specify the date and time as the one that is used in the name of the log file in the main program (for example, Advanced Serial Data Logger). So please read about it in the help file of the main program. Note: the "/" and ":" characters are enclosed in quotation marks in the template (see figure above).

If some part is missing in the received date and time (for example, year), you can take the missing part from the current date and time:

Use the current date for missing parts;
Use the current time for missing parts;

Sometimes devices send dates with month names (for example, "Jan 10, 2024"), and this name can be in a language (in this example, it is in English) different from the language of your operating system. The language ID a date is sent in is specified in the "**Date language ID**" field in this case.

7   Filter

The filter is used to ignore some data packets that you do not want to export with the help of other modules.
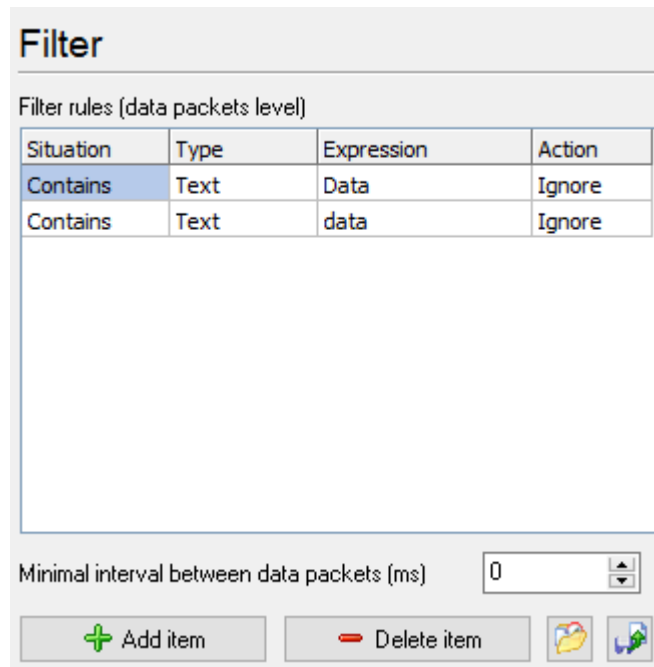

Fig. 3. Filtering rules

You should define one or several filtering rules. If some condition is true, the program does what is specified in the "Action" field with the matching packet.

Action types

- **Ignore** – the current data packet will be ignored and will not be exported;
- **Parse** – the current data packet will be parsed and exported.

Several condition types can be specified in the "Status" field.

Rule status types

- **Disabled** – this rule is disabled and not used for filtering purposes;
- **Contains** – this rules checks whether the string/expression from the "Expression" field is present in the data packet;
- **Does not contain** – this rule checks whether the string/expression from the "Expression" field is absent in the data packet.

Expressions types – Expressions in the "Expression" field can be of 2 types:

- **Text** – the module will search the data packet for a string specified in the "Expression" field. The search is case-sensitive.
- **Regular expression** – the module will use the regular expression specified in the "Expression" field in its search. The search is case-sensitive.

8   Syntax of Regular Expressions

Introduction

Regular Expressions are a widely-used method of specifying patterns of text to search for. Special metacharacters allow you to specify, for instance, that a particular string you are looking for occurs at the beginning or end of a line, or contains n recurrences of a certain character.

Regular expressions look ugly for novices, but they are a very simple, handy, and powerful tool.

Let's start our learning trip!

Simple matches

Any single character matches itself unless it is a metacharacter with a special meaning described below.

A series of characters matches that series of characters in the target string, so the pattern "bluh" would match "bluh" in the target string. Quite simple, eh?

If you want to use metacharacters or escape sequences or literally, you need to 'escape' them by the "\" backslash character. For instance, the "^" metacharacter matches the beginning of the string, but "\^" matches the "^" character and "\\" matches "\" and so on.

Examples:

`foobar`	*matches the 'foobar' string.*
`\^FooBarPtr`	*matches '^FooBarPtr'.*

Escape sequences

Characters may be specified using an escape sequence syntax, much like that used in C and Perl: "\n" matches a newline, "\t" a tab, etc. More generally, \xnn, where nn is a string of hexadecimal digits, matches the character whose ASCII value is nn. If you need wide (Unicode) character code, you can use '\x{nnnn}', where 'nnnn' - one or more hexadecimal digits.

`\xnn`	*char with hex code nn*
`\x{nnnn}`	*char with hex code nnnn (one byte for plain text and two bytes for Unicode)*
`\t`	*tab (HT/TAB), same as \x09*
`\n`	*newline (NL), same as \x0a*
`\r`	*car.return (CR), same as \x0d*
`\f`	*form feed (FF), same as \x0c*
`\a`	*alarm (bell) (BEL), same as \x07*
`\e`	*escape (ESC), same as \x1b*

Examples:

`foo\x20bar`	*matches 'foo bar' (note the space in the middle).*
`\tfoobar`	*matches 'foobar' preceded by the tab character.*

Character classes

You can specify a **character class** by enclosing a list of characters in [], which will match any **one** character from the list.

If the first character after the "[" is "^", the class matches any character **not** in the list.

Examples:

`foob[aeiou]r`	*finds strings 'foobar', 'foober' etc. but not 'foobbr', 'foobcr' etc.*
`foob[^aeiou]r`	*find strings 'foobbr', 'foobcr' etc. but not 'foobar', 'foober' etc.*

Within a list, the "-" character is used to specify a **range**, so that a-z represents all characters between "a" and "z", inclusive.

If you want "-" itself to be a member of a class, put it at the start or end of the list, or escape it with a backslash. If you want ']' you may place it at the start of the list or escape it with a backslash.

Examples:

`[-az]`	*matches 'a', 'z' and '-'*
`[az-]`	*matches 'a', 'z' and '-'*
`[a\-z]`	*matches 'a', 'z' and '-'*
`[a-z]`	*matches all twenty six small characters from 'a' to 'z'*

`[\n-\x0D]` *matches any of #10,#11,#12,#13.*
`[\d-t]` *matches any digit, '-' or 't'.*
`[]-a]` *matches any char from ']'..'a'.*

Metacharacters

Metacharacters are special characters which are the essence of Regular Expressions. There are different types of metacharacters, described below.

Metacharacters - line separators

   `^`   *start of a line*
   `$`   *end of a line*
   `\A`   *start of a text*
   `\Z`   *end of a text*
   `.`   *any character in a line*

Examples:
`^foobar` *matches the 'foobar' string only if it's at the beginning of a line*
`foobar$` *matches the 'foobar' string only if it's at the end of a line*
`^foobar$` *matches the 'foobar' string only if it's the only string in a line*
`foob.r` *matches strings like 'foobar', 'foobbr', 'foob1r' and so on*

By default, the "^" metacharacter is only guaranteed to match at the beginning of the input string/text, the "$" metacharacter only at the end. Embedded line separators will not be matched by "^" or "$".
You may, however, wish to treat a string as a multi-line buffer, such that the "^" will match after any line separator within the string, and "$" will match before any line separator.

The "." metacharacter by default matches any character.

Note that "^.*$" (an empty line pattern) does not match the empty string within the sequence \x0D\x0A, but matches the empty string within the sequence \x0A\x0D.

Metacharacters - predefined classes

   `\w`   *an alphanumeric character (including "_")*
   `\W`   *a non-alphanumeric*
   `\d`   *a numeric character*
   `\D`   *a non-numeric*
   `\s`   *any space (same as [\t\n\r\f])*
   `\S`   *a non space*

You may use \w, \d, and \s within custom **character classes**.

Examples:
`foob\dr` *matches strings like 'foob1r', "foob6r' and so on but not 'foobar', 'foobbr' and so on*
`foob[\w\s]r` *matches strings like 'foobar', 'foob r', 'foobbr' and so on but not 'foob1r', 'foob=r' and so on*

Metacharacters - iterators

Any item of a regular expression may be followed by another type of metacharacters - **iterators**. Using this type of metacharacters, you can specify the number of occurrences of the previous character, **metacharacter**, or **sub-expression**.

`*`	*zero or more ("greedy"), similar to {0,}*
`+`	*one or more ("greedy"), similar to {1,}*
`?`	*zero or one ("greedy"), similar to {0,1}*
`{n}`	*exactly n times*
`{n,}`	*at least n times ("greedy")*
`{n,m}`	*at least n but not more than m times ("greedy")*
`*?`	*zero or more ("non-greedy"), similar to {0,}?*
`+?`	*one or more ("non-greedy"), similar to {1,}?*
`??`	*zero or one ("non-greedy"), similar to {0,1}?*
`{n,}?`	*at least n times ("non-greedy")*
`{n,m}?`	*at least n but not more than m times ("non-greedy")*

So, digits in curly brackets of the form {n,m}, specify the minimum number of times to match the item n and the maximum m. The form {n} is equivalent to {n,n} and matches exactly n times. The form {n,} matches n or more times. There is no limit to the size of n or m, but large numbers will chew up more memory and slow down an execution time of a regular expression.

If a curly bracket occurs in any other context, it is treated as a regular character.

Examples:

`foob.*r`	*matches strings like 'foobar', 'foobalkjdflkj9r' and 'foobr'*
`foob.+r`	*matches strings like 'foobar', 'foobalkjdflkj9r', but not 'foobr'*
`foob.?r`	*matches strings like 'foobar', 'foobbr' and 'foobr', but not 'foobalkj9r'*
`fooba{2}r`	*matches the string 'foobaar'*
`fooba{2,}r`	*matches strings like 'foobaar', 'foobaaar', 'foobaaaar' etc.*
`fooba{2,3}r`	*matches strings like 'foobaar', or 'foobaaar', but not 'foobaaaar'*

A little explanation about "greediness". "Greedy" takes as many as possible, "non-greedy" takes as few as possible. For example, 'b+' and 'b*' applied to string 'abbbbc' return 'bbbb', 'b+?' returns 'b', 'b*?' returns empty string, 'b{2,3}?' returns 'bb', 'b{2,3}' returns 'bbb'.

Metacharacters - alternatives

You can specify a series of **alternatives** for a pattern using "|" to separate them, so that fee|fie|foe will match any of "fee", "fie", or "foe" in the target string (as would f(e|i|o)e). The first alternative includes everything from the last pattern delimiter ("(", "[", or the beginning of the pattern) up to the first "|", and the last alternative contains everything from the last "|" to the next pattern delimiter. For this reason, it's common practice to include alternatives in parentheses, to minimize confusion about where they start and end.
Alternatives are tried from left to right, so the first alternative found for which the entire expression matches, is the one that is chosen. This means that alternatives are not necessarily greedy. For example: when matching foo|foot against "barefoot", only the "foo" part will match, as that is the first alternative tried, and it successfully matches the target string. (This might not seem important, but it is important when you are capturing matched text using parentheses.)
Also remember that "|" is interpreted as a literal within square brackets, so if you write [fee|fie|foe], you're really only matching [feio|].

Examples:

`foo(bar|foo)`   *matches strings 'foobar' or 'foofoo'.*

Metacharacters - subexpressions

The bracketing construct (...) may also be used to define a sub-expressions of the regular expression.

Sub-expressions are numbered based on the left to right order of their opening parenthesis. The first sub-expression has the number '1'.

Examples:

`(foobar){8,10}`   *matches strings which contain 8, 9 or 10 instances of 'foobar'*

`foob([0-9]|a+)r` *matches 'foob0r', 'foob1r' , 'foobar', 'foobaar', 'foobaar', etc.*

Metacharacters - backreferences

Metacharacters \1 through \9 are interpreted as backreferences. \<n> matches previously matched **subexpression** #<n>.

Examples:

`(.)\1+`   *matches 'aaaa' and 'cc'.*

`(.+)\1+`   *also matches 'abab' and '123123'*

([' "] ?) (| d+) | 1   matches '"13" (in double quotes), or '4' (in single quotes) or 77 (without quotes) etc

Modifiers

Modifiers are used for changing the behaviour of the parser.

There are many ways to set up modifiers.
These modifiers may be embedded within the regular expression itself using the (?...) construct.

i
   Do case-insensitive pattern matching (using installed in your system locale settings).
m
   Treat string as multiple lines that change "^" and "$" from matching at only the very start or end of the string to the start or end of any line anywhere within the string.
s
   Treat string as a single line that change "." to match any character whatsoever, even a line separators, which it normally would not match.
g
   Non-standard modifier. It switches off all following operators into non-greedy mode (by default this modifier is On). So, if modifier /g is Off then '+' works as '+?', '*' as '*?' and so on.
x
   Extend your pattern's legibility by permitting white-space and comments (see explanation below).

The modifier /x itself needs a little more explanation. It tells the parser to ignore whitespace that is neither backslashed nor within a character class. You can use this to break up your regular

expression into (slightly) more readable parts. The # character is also treated as a metacharacter introducing a comment, for example:

```
(
(abc) # comment 1
   |   # You can use spaces to format a regular expression. - parser ignores it
(efg) # comment 2
)
```

It also means that if you want real whitespace or # characters in the pattern (outside a character class, where they are unaffected by /x), that you'll either have to escape them or encode them using octal or hex escapes. These features go a long way towards making regular expressions text more readable.

How to change modifiers

```
(?imsxr-imsxr)
```
You may use it in a regular expression to change modifiers on-the-fly. If this construction is inlined into subexpression, it affects only into this subexpression.

Examples:

`(?i)New-York`	*matches 'New-york' and 'New-York'*
`(?i)New-(?-i)York`	*matches 'New-York' but not 'New-york'*
`(?i)(New-)?York`	*matches 'New-york' and 'new-york'*
`((?i)New-)?York`	*matches 'New-York', but not 'new-york*

```
(?#text)
```
It is a comment. The text inside brackets will be ignored. Note that the parser closes the comment as soon as it sees the ")" character, so there is no way to put a literal ")" in the comment.

9   Response

Upon receiving the request, the module may send back a status line, such as "HTTP/1.1 200 OK", and a message of its own, the body of which is perhaps the requested file, an error message, or some other information. **Content type** specifies the Internet media type of the data conveyed by the http message.
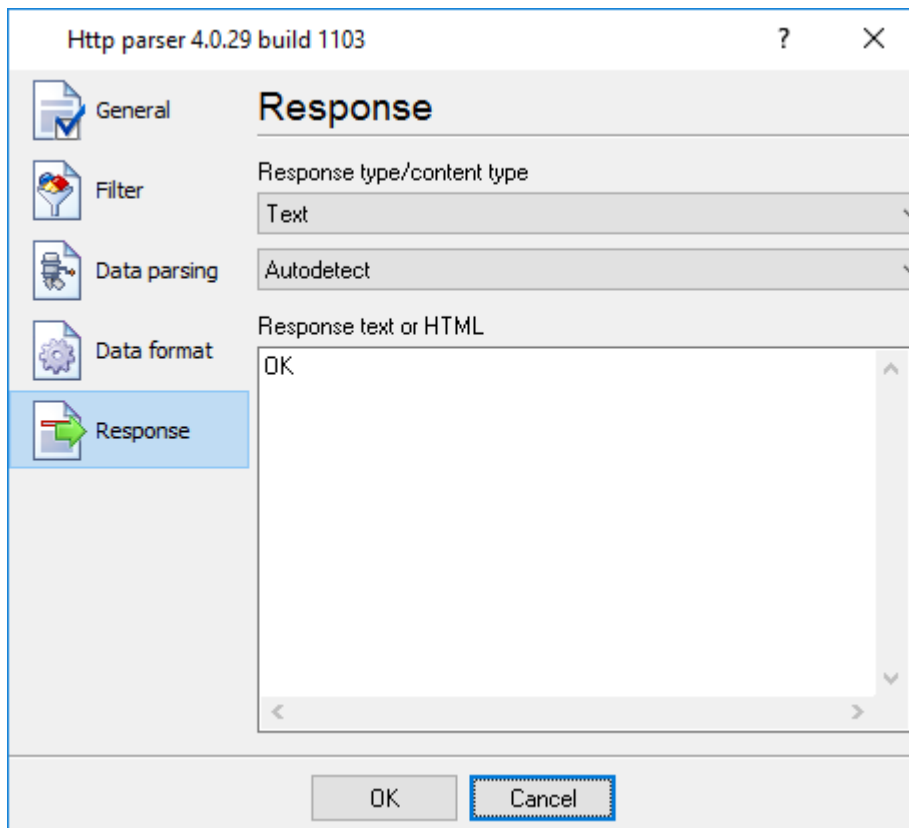
Fig.5. Response options

The module allows supports following response types:

- **Disabled** - the module will not send responses at all. Please, note, that most HTTP clients will wait any response;
- **403** - the module will always send a response message with the 403 (Forbidden) status code;
- **404** - the module will always send a response message with the 404 (Not Found) status code;
- **Text** - you may define any plain text or HTML in the corresponding field. The module will send it with the 200 (OK) status code;
- **From file** - the module may read response from a text or binary file and send it with the 200 (OK) status code. The module will ignore a file name, which is specified in the HTTP request. If the file doesn't exist then the module will return the 404 message;
- **From root folder** - the module extracts a file name from the HTTP request and search this file in the specified folder. If the file exists, then module send it with the 200 (OK) status code. Otherwise the module will return the 404 message. We recommend to use the "**Autodetect**" content type for this response type;