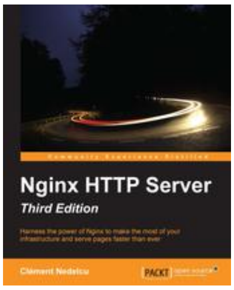


The Rewrite module

This module, in particular, brings much more functionality to Nginx than a simple set of directives. It defines a whole that will be explained throughout this section.

Basically, the purpose of this module (as the name suggests) is to perform URL rewriting. This mechanism allows you containing multiple parameters. For instance, `http://example.com/article.php?id=1234&comment=32` —such uninformative and meaningless for a regular visitor. Instead, links to your website will contain useful information tha page the visitor is about to visit. The URL given in the example becomes `http://website.com/article-1234-32-strengthens.html` . This solution is not only more interesting for your visitors, but also for search engines—URL r **Search Engine Optimization** (**SEO**).

The principle behind this mechanism is simple—it consists of rewriting the URI of the client request after it is receive Once rewritten, the URI is matched against the location blocks in order to find the configuration that should be appli is further detailed in the coming sections.

Reminder on regular expressions

First and foremost, this module requires a certain understanding of **regular expressions**, also known as **regexes** or **r** performed by the `rewrite` directive, which accepts a pattern followed by the replacement URI.

It is a vast topic—entire books are dedicated to explaining the ins and outs of regular expressions. However, the simp about to examine should be more than sufficient to make the most of the mechanism.

Purpose

The first question we must answer is: what is the purpose of regular expressions? To put it simply, the main purpose i characters matches a given pattern. The pattern is written in a particular language that allows the defining of extrem

String	Pattern	Does it match?	Explanation
`hello`	`^hello$`	Yes	The string begins with the character `h` (`^h`), followed by `e` , `l` , `o` (`o$`).
`hell`	`^hello$`	No	The string begins with the character `h` (`^h`), followed by `e` , `l` , with `o` .
`Hello`	`^hello$`	Depends	If the engine performing the match is case-sensitive, the string doesn't ma

This concept becomes a lot more interesting when complex patterns are employed, such as one that validates e-mail `[A-Z0-9.-]+\.[A-Z]{2,4}$` . Programmatically validating if an e-mail address is well-formed would require a grea can be done with a single regular expression in pattern matching.

PCRE syntax

The syntax that Nginx employs originates from the **Perl Compatible Regular Expression** (**PCRE**) library, which (if you **Nginx Configuration**) is a pre-requisite for making your own build, unless you disable the modules that make use of i form of regular expressions, and nearly everything you learn here remains valid for other language variations.

In its simplest form, a pattern is composed of one character, for example, `x` . We can match strings against this pa the pattern `x` ? Yes, `example` contains the character `x` . It can be more than one specific character—the pa character between `a` and `z` , or even a combination of letters and digits: `[a-z0-9]` . In consequence, the p validates the following strings: `hello` and `hell4` but not `hell` or `hell!` .

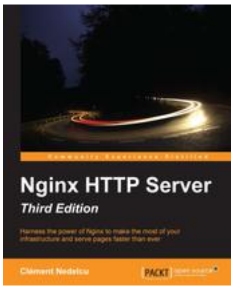
You probably noticed that we employed the brackets `[` and `]` . They are part of what we call **metacharacters** pattern. There are a total of 11 metacharacters, and all play a different role. If you want to create a pattern that actu characters, you need to escape the character with a `\` (backslash).

Metacharacter	Description
`^` Beginning	The entity after this character must be found at the beginning. Example pattern: `^h` Matching strings: `hello` , `h` , `hh` (anything beginning with h) Non-matching strings: `character` , `ssh`
`$` End	The entity before this character must be found at the end. Example pattern: `e$` Matching strings: `sample` , `e` , `file` (anything ending with e) Non-matching strings: `extra` , `shell`
`.` (dot) Any	Matches any character. Example pattern: `hell.` Matching strings: `hello` , `hellx` , `hell5` , `hell!` Non-matching strings: `hell` , `helo`
`[ ]` Set	Matches any character within the specified set. Syntax: `[a-z]` for a range, `[abcd]` for a set, and `[a-z0-9]` for two ranges. Note that if y character in a range, you need to insert it right after `[` or just before `]` . Example pattern: `hell[a-y123-]` Matching strings: `hello` , `hell1` , `hell2` , `hell3` , `hell-` Non-matching strings: `hellz` , `hell4` , `heloo` , `he-llo`
`[^ ]` Negate set	Matches any character that is not within the specified set. Example pattern: `hell[^a-np-z0-9]` Matching strings: `hello` , `hell!` Non-matching strings: `hella` , `hell5`
`|` Alternation	Matches the entity placed either before or after `|` . Example pattern: `hello|welcome` Matching strings: `hello` , `welcome` , `helloes` , `awelcome` Non-matching strings: `hell` , `ellow` , `owelcom`
`( )` Grouping	Groups a set of entities, often used in conjunction with `|` . Also **captures** the matched entities; on. Example pattern: `^(hello|hi) there$` Matching strings: `hello there` , `hi there` . Non-matching strings: `hey there` , `ahoy there`
`\` Escape	Allows you to escape special characters. Example pattern: `Hello\ .` Matching strings: `Hello.` , `Hello. How are you?` , `Hi! Hello...` Non-matching strings: `Hello` , `Hello! how are you?`

Quantifiers

So far, you are able to express simple patterns with a limited number of characters. Quantifiers allow you to extend t

Quantifier	Description
`*` 0 or more times	The entity preceding `*` must be found 0 or more times. Example pattern: `he*llo` Matching strings: `hllo` , `hello` , `heeeello` Non-matching strings: `hallo` , `ello`



Quantifier	Description
`+` 1 or more times	The entity preceding `+` must be found 1 or more times. Example pattern: `he+llo` Matching strings: `hello` , `heeeello` Non-matching strings: `hllo` , `helo`
`?` 0 or 1 time	The entity preceding `?` must be found 0 or 1 time. Example pattern: `he?llo` Matching strings: `hello` , `hllo` Non-matching strings: `heello` , `heeeello`
`{x}` x times	The entity preceding `{x}` must be found x times. Example pattern: `he{3}llo` Matching strings: `heeello` , `oh heeello there!` Non-matching strings: `hello` , `heello` , `heeeello`
`{x,}` At least x times	The entity preceding `{x,}` must be found at least `x` times. Example pattern: `he{3,}llo` Matching strings: `heeello` , `heeeeeeello` Non-matching strings: `hllo` , `hello` , `heello`
`{x,y}` x to y times	The entity preceding `{x,y}` must be found between `x` and `y` times. Example pattern: `he{2,4}llo` Matching strings: `heello` , `heeello` , `heeeello` Non-matching strings: `hello` , `heeeeello`

As you probably noticed, the `{` and `}` characters in the regular expressions conflict with the block delimiter c syntax language. If you want to write a regular expression pattern that includes curly brackets, you need to place the or double quotes):

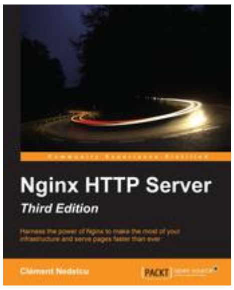
```
rewrite hel{2,}o /hello.php; # invalid
rewrite "hel{2,}o" /hello.php; # valid
rewrite 'hel{2,}o' /hello.php; # valid
```

Captures

One last feature of the regular expression mechanism is the ability to capture sub-expressions. Whatever text is plac `)` is captured and can be used after the matching process. The captured characters become available under the for where N is a numeric index, in order of capture. Alternatively, you can attribute an arbitrary name to each of your ca The variables generated through the captures can be inserted within the directive values. The following are a couple principle:

Pattern	Example of a matching
`^(hello|hi) (sir|mister)$`	`hello sir`
`^(hello (sir))$`	`hello sir`
`^(.*)$`	`nginx rocks`
`^(.{1,3})([0-9]{1,4})([?!]{1,2})$`	`abc1234!?`
Named captures are also supported through the following syntax: `?<name>` . Example: `^/(?<folder>[^/]+)/(?<file>.*)$`	`/admin/doc`

BACK



Search this title...

Bookmarks (0)

When you use a regular expression in Nginx, for example, in the context of a `location` block, the buffers that yo later directives:

```
server {
    server_name website.com;
    location ~* ^/(downloads|files)/(.*)$ {
        add_header Capture1 $1;
        add_header Capture2 $2;
    }
}
```

In the preceding example, the `location` block will match the request URI against a regular expression. A couple would be `/downloads/file.txt`, `/files/archive.zip` , or even `/files/docs/report.doc` . Two parts are either `downloads` or `files` , and `$2` will contain whatever comes after `/downloads/` or `/files/` . directive (syntax: `add_header header_name header_value` , see the **HTTP headers module** section) is employe headers to the client response for the sole purpose of demonstration.

Internal requests

Nginx differentiates external and internal requests. External requests directly originate from the client; the URI is th `location` blocks:

```
server {
    server_name website.com;
    location = /document.html {
        deny all; # example directive
    }
}
```

A client request to `http://website.com/document.html` would directly fall into the `location` block.

As opposed to this, internal requests are triggered by Nginx via specific directives. Among the directives offered by t are several directives capable of producing internal requests: `error_page` , `index` , `rewrite` , `try_file` `d_after_body` (from the Addition module), the `include` SSI command, and more.

There are two different types of internal requests:

- **Internal redirects:** Nginx redirects the client requests internally. The URI is changed, and the request may the `location` block and become eligible for different settings. The most common case of internal redirects is v directive, which allows you to rewrite the request URI.
- **Sub-requests**: These are additional requests that are triggered internally to generate content that is complem simple example would be with the Addition module. The `add_after_body` directive allows you to specify a after the original one, the resulting content being appended to the body of the original request. The SSI modul requests to insert content with the `include` SSI command.

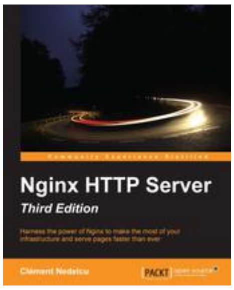
error_page

Detailed in the module directives of the Nginx HTTP Core module, `error_page` allows you to define the server b code occurs. The simplest form is that of affecting a URI to an error code:

```
server {
    server_name website.com;
    error_page 403 /errors/forbidden.html;
    error_page 404 /errors/not_found.html;
}
```

When a client attempts to access a URI that triggers one of these errors (such as loading a document or a file that do resulting in a 404 error), Nginx is supposed to serve the page associated with the error code. In fact, it does not just s actually initiates a completely new request based on the new URI.

BACK



Search this title...

Bookmarks (0)

Consequently, you can end up falling back on a different configuration, like in the following example:

```
server {
    server_name website.com;
    root /var/www/vhosts/website.com/httpdocs/;
    error_page 404 /errors/404.html;
    location /errors/ {
        alias /var/www/common/errors/;
        internal;
    }
}
```

When a client attempts to load a document that does not exist, they will initially receive a `404` error. We employ to specify that `404` errors should create an internal redirect to `/errors/404.html`. As a result, a new reques URI `/errors/404.html`. This URI falls under the location block `/errors/`, so the corresponding configuratic

Note

Logs can prove to be particularly useful when working with redirects and URL rewrites. Be aware that information on internal rec you set the `error_log` directive to `debug`. You can also get it to show up at the `notice` level, under the conditio `on;` wherever you need it.

A raw but trimmed excerpt from the debug log summarizes the mechanism:

```
->http request line: "GET /page.html HTTP/1.1"
->http uri: "/page.html"
->test location: "/errors/"
->using configuration ""
->http filename: "/var/www/vhosts/website.com/httpdocs/page.html"
-> open() "/var/www/vhosts/website.com/httpdocs/page.html" failed (2: No such file or directory), client: 12
->http finalize request: 404, "/page.html?" 1
->http special response: 404, "/page.html?"
->internal redirect: "/errors/404.html?"
->test location: "/errors/"
->using configuration "/errors/"
->http filename: "/var/www/common/errors/404.html"
->http finalize request: 0, "/errors/404.html?" 1
```

Note that the use of the `internal` directive in the `location` block forbids clients from accessing the `/err` can thus only be accessed through an internal redirect.

The mechanism is the same for the `index` directive (detailed further on in the Index module)—if no file path is pr Nginx will attempt to serve the specified index page by triggering an internal redirect.

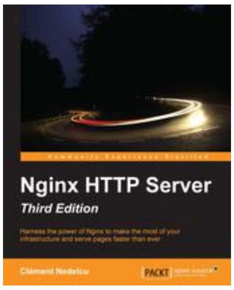
Rewrite

While the previous directive, `error_page`, is not actually a part of the Rewrite module, detailing its functionality the way Nginx handles client requests.

Similarly to how the `error_page` directive redirects to another location, rewriting the URI with the `rewrite` redirect:

```
server {
    server_name website.com;
    root /var/www/vhosts/website.com/httpdocs/;
    location /storage/ {
        internal;
        alias /var/www/storage/;
    }
    location /documents/ {
        rewrite ^/documents/(.*)$ /storage/$1;
    }
}
```

BACK



Search this title...

Bookmarks (0)

A client query to `http://website.com/documents/file.txt` initially matches the second `location` block
However, the block contains a rewrite instruction that transforms the URI from `/documents/file.txt` to `/st`
transformation reinitializes the process—the new URI is matched against the `location` blocks. This time, the firs
on `/storage/`) matches the URI (`/storage/file.txt`).

Again, a quick peek at the debug log details the mechanism:

```
->http request line: "GET /documents/file.txt HTTP/1.1"
->http uri: "/documents/file.txt"
->test location: "/storage/"
->test location: "/documents/"
->using configuration "/documents/"
->http script regex: "^/documents/(.*)$"
->"^/documents/(.*)$" matches "/documents/file.txt", client: 127.0.0.1, server: website.com, request: "GET /
->rewritten data: "/storage/file.txt", args: "", client: 127.0.0.1, server: website.com, request: "GET /docu
->test location: "/storage/"
->using configuration "/storage/"
->http filename: "/var/www/storage/file.txt"
->HTTP/1.1 200 OK
->http output filter "/storage/test.txt?"
```

Infinite loops

With all the different syntaxes and directives, you could easily get confused. Worse—you might get Nginx confused.
your rewrite rules are redundant, and cause internal redirects to loop infinitely:

```
server {
    server_name website.com;
    location /documents/ {
        rewrite ^(.*)$ /documents/$1;
    }
}
```

You thought you were doing well, but this configuration actually triggers internal redirects `/documents/anything`
`s/anything` . Moreover, since the location patterns are re-evaluated after an internal redirect, `/documents//do`
`/documents//documents//documents/anything` .

Here is the corresponding excerpt from the debug log:

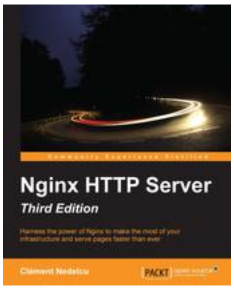
```
->test location: "/documents/"
->using configuration "/documents/"
->rewritten data: "/documents//documents/file.txt", [...]
->test location: "/documents/"
->using configuration "/documents/"
->rewritten data: "/documents//documents//documents/file.txt" [...]
->test location: "/documents/"
->using configuration "/documents/"
->rewritten data: -
>"/documents//documents//documents//documents/file.txt" [...]
->[...]
```

You probably wonder if this goes on indefinitely—the answer is no. The number of cycles is restricted to 10. You are
redirects. Anything past this limit and Nginx will produce a `500 Internal Server Error` .

Server Side Includes

A potential source of sub-requests is the **Server Side Include** (**SSI**) module. The purpose of SSI is for the server to par
the response to the client in a fashion somewhat similar to PHP or other preprocessors.

Within a regular HTML file (for example), you are offered the possibility of inserting tags corresponding to the comm



```
<html>
<head>
  <!--# include file="header.html" -->
</head>
<body>
  <!--# include file="body.html" -->
</body>
</html>
```

(Nginx processes these two commands; in this case, it reads the contents of `header.html` and `body.html` an
source, which is then sent to the client.

Several commands are at your disposal; they are detailed in the SSI module section in this chapter. The one we are in
e command for including a file into another file:

```
<!--# include virtual="/footer.php?id=123" -->
```

The specified file is not just opened and read from a static location. Instead, a whole subrequest is processed by Ngin
is inserted instead of the `include` tag.

Conditional structure

The Rewrite module introduces a new set of directives and blocks among which is the `if` conditional structure:

```
server {
    if ($request_method = POST) {
      […]
    }
}
```

This allows you to apply a configuration according to the specified condition. If the condition is true, the configuratio

The following table describes the various syntaxes accepted when forming a condition:

Operator	Description

None	The condition is true if the specified variable or data is not equal to an empty string or a string st
	`if ($string) {`
	`  […]`
	`}`

`=` , `!=`	The condition is true if the argument preceding the `=` symbol is equal to the argument follow
	can be read as "if the `request_method` is equal to `POST` , then apply the configuration":
	`if ($request_method = POST) {`
	`  […]`
	`}`
	The `!=` operator does the opposite: "if the request method is not equal to `GET` , then apply
	`if ($request_method != GET) {`
	`  […]`
	`}`

Operator	Description

`~` , `~*` , `!~` , `!~*`	The condition is true if the argument preceding the `~` symbol matches the regular expressior
	`if ($request_filename ~ "\.txt$") {`
	`  […]`
	`}`
	`~` is case-sensitive, `~*` is case-insensitive. Use the `!` symbol to negate the matching:
	`if ($request_filename !~* "\.php$") {`
	`  […]`
	`}`
	Note that you can insert the capture buffers in the regular expression:
	`if ($uri ~ "^/search/(.*)$") {`
	`  set $query $1;`
	`  rewrite ^ http://google.com/search?q=$query;`
	`}`

`-f` , `!-f`	Tests the existence of the specified file:
	`if (-f $request_filename) {`
	`  […] # if the file exists`
	`}`
	Use `!-f` to test the non-existence of the file:
	`if (!-f $request_filename) {`
	`  […] # if the file does not exist`
	`}`

`-d` , `!-d`	Similar to the `-f` operator, is used for testing the existence of a directory.
`-e` , `!-e`	Similar to the `-f` operator, is used for testing the existence of a file, directory, or symbolic lin
`-x` , `!-x`	Similar to the `-f` operator, is used for testing whether a file exists and is executable.

As of version 1.8, there is no `else` or `else if` -like instruction. However, other directives allowing you to co sequencing are available.

You might wonder: what are the advantages of using a `location` block over an `if` block? Indeed, in the follo the same effect:

```
if ($uri ~ /search/) {
  […]
}
location ~ /search/ {
    […]
}
```

As a matter of fact, the main difference lies within the directives that can be employed within either block—some ca and some can't; on the contrary, almost all the directives are authorized within a `location` block, as you probabl so far. In general, it's best to only insert the directives from the Rewrite module within an `if` block, as other dire intended for such usage.

Directives

(

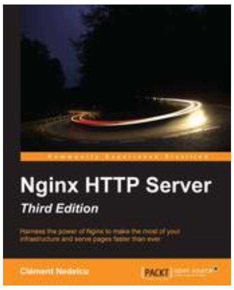
The Rewrite module provides you with a set of directives that do more than just rewriting a URI. The following table with the context in which they can be employed:

Directive	Description

`rewrite`
Context:
`server` ,
`location`
`if`
	As discussed previously, the `rewrite` directive allows you to rewrite the URI of the current reque of the said request.
	Syntax: `rewrite regexp replacement [flag];`
	,Where `regexp` is the regular expression that the URI should match in order for the replacement t
	Flag may take one of the following values:
	- `last` : The current rewrite rule should be the last to be applied. After its application, the Nginx, and a `location` block is searched for. However, further rewrite instructions will l
	- `break` : The current rewrite rule is applied, but Nginx does not initiate a new request for restart the search for matching `location` blocks). All further rewrite directives are igno
	- `redirect` : Returns a `302 Moved temporarily` HTTP response, with the replacemen `ocation` header.
	- `permanent` : Returns a `301 Moved permanently` HTTP response, with the replaceme `location` header.
	- If you specify a URI beginning with `http://` as the replacement URI, Nginx will automat flag.
	- Note that the request URI processed by the directive is a relative URI: It does not contain tl a request such as `http://website.com/documents/page.html` , the request URI is `/d`
	- Is decoded: The URI corresponding to a request such as `http://website.com/my%20page` `e.html` (in the encoded URI, %20 indicates a white space character).
	- Does not contain arguments: For a request such as `http://website.com/page.php?id=1` `age.php` . When rewriting the URI, you don't need to consider including the arguments in does it for you. If you want Nginx not to include the arguments after the rewritten URI, you at the end of the replacement URI: `rewrite ^/search/(.*)$ /search.php?q=$1?` .
	- Examples:
	`rewrite ^/search/(.*)$ /search.php?q=$1;`
	`rewrite ^/search/(.*)$ /search.php?q=$1?;`
	`rewrite ^ http://website.com;`
	`rewrite ^ http://website.com permanent;`

`break`
Context:
`server` ,
`location`
`if`
	The `break` directive is used to prevent further rewrite directives. Past this point, the URI is fixed .
	Example:
	`if (-f $uri) {`
	`  break; # break if the file exists`
	`}`
	`if ($uri ~ ^/search/(.*)$) {`
	`  set $query $1;`
	`  rewrite ^ /search.php?q=$query?;`
	`}`
	This example rewrites `/search/anything` -like queries to `/search.php?q=anything` . Howeve (such as `/search/index.html`), the break instruction prevents Nginx from rewriting the URI.

BACK



Search this title...

Bookmarks (0)

Directive	Description

`return`
Context: `server` , `location` , `if`
	Interrupts the processing of the request, and returns the specified HTTP status code or specified tex
	Syntax: `return code | text;`
	Where the code is one of the following status codes: `204` , `400` , `402` to `406` , `408` , `41(`
	, `500` to `504` . In addition, you may use the Nginx-specific code `444` in order to return a `HTT`
	further response header or body. Alternatively, you may also specify a raw text value that will be retu
	response body. This comes in handy when testing whether your request URIs fall within particular lo
	Example:
	`if ($uri ~ ^/admin/) {`
	`  return 403;`
	`  # the instruction below is NOT executed`
	`  # as Nginx already completed the request`
	`  rewrite ^ http://website.com;`
	`}`

`set`
Context: `server` , `location` , `if`
	Initializes or redefines a variable. Note that some variables cannot be redefined, for example, you ar
	Syntax: `set $variable value;`
	Examples:
	`set $var1 "some text";`
	`if ($var1 ~ ^(.*) (.*)$) {`
	`  set $var2 $1$2; #concatenation`
	`  rewrite ^ http://website.com/$var2;`
	`}`

`uninitialized_variable_warn` Context: `http` , `server` , `location` , `if`	If set to `on` , Nginx will issue log messages when the configuration employs a variable that has not Syntax: `on` or `off` `uninitialized_variable_warn on;`
`rewrite_log` Context: `http` , `server` , `location` , `if`	If set to `on` , Nginx will issue log messages for every operation performed by the rewrite engine at `error_log` directive). Syntax: `on` or `off` Default value: `off` `rewrite_log off;`

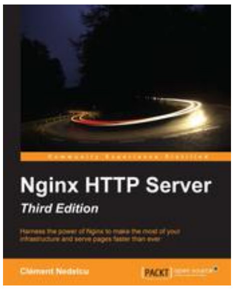
Common rewrite rules

Here is a set of rewrite rules that satisfy the basic needs of the dynamic websites that wish to beautify their page linl mechanism. You will obviously need to adjust these rules according to your particular situation, as every website is d

Performing a search

This rewrite rule is intended for search queries. Search keywords are included in the URL.

Input URI	`http://website.com/search/some-search-keywords`
Rewritten URI	`http://website.com/search.php?q=some-search-keywords`
Rewrite rule	`rewrite ^/search/(.*)$ /search.php?q=$1?;`



Search this title...

Bookmarks (0)

User profile page

Most dynamic websites that allow the visitors to register, offer a profile view page. URLs of this form, containing bot
can be employed.

Input URI	`http://website.com/user/31/James`
Rewritten URI	`http://website.com/user.php?id=31&name=James`
Rewrite rule	`rewrite ^/user/([0-9]+)/(.+)$ /user.php?id=$1&name=$2?;`

Multiple parameters

Some websites may use different syntaxes for the argument string, for example, separating non-named arguments w

Input URI	`http://website.com/index.php/param1/param2/param3`
Rewritten URI	`http://website.com/index.php?p1=param1&p2=param2&p3=param3`
Rewrite rule	`rewrite ^/index.php/(.*)/(.*)/(.*)$ /index.php?p1=$1&p2=$2&p3=$3?;`

Wikipedia-like

Many websites have now adopted the URL style introduced by Wikipedia: a prefix folder, followed by an article nam

Input URI	`http:// website.com/wiki/Some_keyword`
Rewritten URI	`http://website.com/wiki/index.php?title=Some_keyword`
Rewrite rule	`rewrite ^/wiki/(.*)$ /wiki/index.php?title=$1?;`

News website article

This URL structure is often employed by news websites, as the URLs contain indications to the articles' contents. It i
followed by a slash, then a list of keywords. The keywords can usually be ignored and excluded from the rewritten U

Input URI	`http://website.com/33526/us-economy-strengthens`
Rewritten URI	`http://website.com/article.php?id=33526`
Rewrite rule	`rewrite ^/([0-9]+)/.*$ /article.php?id=$1?;`

Discussion board

Modern bulletin boards now mostly use **pretty URLs**. The following example shows how to create a **topic view** URL
identifier and the starting post. Once again, keywords are ignored:

Input URI	`http://website.com/topic-1234-50-some-keywords.html`
Rewritten URI	`http://website.com/viewtopic.php?topic=1234&start=50`
Rewrite rule	`rewrite ^/topic-([0-9]+)-([0-9]+)-(.*)\.html$ /viewtopic.php?topic=$1&start`

Next Section (/mapt/book/networking-and-servers/9781785280337/4/ch04lvl1sec25/s

Related titles