Pound

version 4.13, 21 August 2024

Sergey Poznyakoff.

Table of Contents

1 Overview

`Pound` is a reverse proxy, load balancer and HTTPS front-end for Web servers. It was developed to provide for even distribution of load between backend httpd servers and to allow for a convenient SSL wrapper for those servers that do not offer it natively.

The core principles of its design are simplicity and safety. `Pound` is a very small program, easily audited for security problems. Normally it runs as a non-privileged user, and can optionally be run in a chrooted environment. With several exceptions, it does not access the hard disk during its runtime. In short, it should pose no security threat to the server it runs at.

The original version of `pound` was written by Robert Segall at Apsis GmbH[1]. In 2018, I added support for newer OpenSSL to the then current version of the program (2.8). This version of `pound`, hosted on `github` was further modified by Rick O'Sullivan and Frank Schmirler, who added WebSocket support.

On April 2020, Apsis started development of `pound` 3.0 – essentially an attempt to rewrite program from scratch, introducing dependencies on some third-party software.

On 2022-09-19, the development and maintenance of `pound` was officially discontinued and Apsis GmbH was dissolved. Following that, I decided to continue development of the program taking my fork as a base. I considered the branch 3.0, which emerged for a short time before the original project was abandoned, to be a failed experiment. To ensure consistent versioning and avoid confusion, my versioning of `pound` started at number 4.0.

[1] `https://web.archive.org/web/20221202094441/https://apsis.ch/`

2 Introduction

The job of a proxy server is to receive incoming HTTP or HTTPS requests, route them to the corresponding web server (*backend*), wait for it to reply and forward the response back to the querying party. If more than one backend is configured to serve requests, the proxy should distribute requests evenly between them, so that each backend gets a share of requests proportional to its capacity.

`Pound` gets information about backends and instructions on HTTP request routing from its configuration file `pound.cfg`. It is located in the *system configuration directory*, which is normally `/etc`[1]. Syntactically, the configuration file is a sequence of *statements* and *sections*, separated by arbitrary amount of empty lines and comments. A *simple statement* occupies a single line and consists of a keyword (*directive*) and one or more values separated by whitespace. A *section* is a compound statement that encloses other statements and sections. Sections begin with a keyword, optionally followed by arguments, and end with a word `End` on a line by itself. All keywords are case-insensitive.

The configuration file defines three kinds of objects: *listeners*, *services*, and *backends*. These are defined as configuration sections.

A *listener* defines IP address (and optionally port), `pound` will be listening on for incoming requests. It can also be regarded as a frontend definition. Listener declarations start with `ListenHTTP` (for plaintext HTTP frontends) or `ListenHTTPS` (for HTTPS frontends) keywords.

Service sections define rules that decide to which backend to route requests received by the listeners. These rules normally involve analysis of the requested URL or HTTP headers. A service may also contain statements that modify requests or responses.

Services are normally declared inside listeners. Thus, when a listener receives a request, it iterates over its services (in the order of their appearance in the configuration file) to find the one that matches the request. If such a service is found, it receives the request and eventually passes it on to a backend.

Services may also be declared outside any listeners, in the global scope. Such services are shared between all listeners. They are tried if none of the services declared within a listener match the incoming request.

Service declarations start with the `Service` keyword.

Backends are objects that actually handle requests and produce responses. Most often these are *regular backends*, which declare IP addresses and ports of servers that are to handle the requests. Backends are defined inside of services, so that the service that matched the request routes it to its backend. If more than one backend is defined within a service, incoming requests will be distributed so that each backend gets its share of the load.

Several special backend types are provided, such as emergency backends, redirects, etc. Only one special backend can be declared for a service, and it cannot be used together with other backend types.

Thus, an average request processing looks as follows. First, a request is received by one of the listeners. The listener then iterates over its services, until it finds one that matches the request. If no such service was found, the listener retries the process with the

[1] The exact location depends on compilation options. When in doubt, examine the output of `pound -V`.

services defined in the global scope. If no matching service is found, a 503 error (‘`Service Unavailable`’) is returned. Otherwise, if the matching service was found, that service passes the request to one of its backends. It may modify the request before that, if it is instructed so by the configuration. Once the backend responds, the service passes the response back to the listener (again, optionally modifying it, if needed), which finally passes it back to the querying party.

3 Usage

When started, `pound` first parses its configuration file. If any errors are detected at this stage, it prints the appropriate diagnostics on the standard error and exits with code 1. Otherwise, if the configuration file is OK, `pound` opens sockets declared in the listener sections, detaches itself from the controlling terminal and starts serving incoming requests. From that moment on, all diagnostic messages are reported via `syslog` (see Chapter 8 [Logging], page 22).

To check whether the configuration file is correct, run `pound` with the `-c` (for *check*) configuration option:

```
pound -c
```

Started this way, `pound` will check the configuration file, report any errors, if found, and exit with status 0 if there are no errors or 1 otherwise. The option `-v` can be used to increase the verbosity level. In particular, it instructs `pound` to print a confirmation message on standard error, if no errors have been encountered (by default it would exit silently in this case).

To use alternative configuration file, supply its full pathname with the `-f` option, e.g.:

```
pound -f /etc/pound/test.cfg
```

If you are experimenting with new configurations, you might want to run `pound` in foreground mode and have it print its diagnostics on the standard error. This is done by the `-e` option. So, for testing purposes, it is quite common to start it this way:

```
pound -e
```

Another option, `-F`, has similar effect, except that it honors logging settings from the configuration file (see Chapter 8 [Logging], page 22), i.e. when used with this option, `pound` will remain in foreground, but will report its messages in accordance with its configuration file.

The following table summarizes all command line options:

`-c`	Check configuration file for syntax error and exit. Exit code indicates whether the configuration is OK (0) or not (1).
`-e`	Start in foreground mode and log to standard error (or standard output, for messages with LOG_DEBUG and LOG_INFO severity levels). This option ignores the `LogLevel` configuration setting (see Chapter 8 [Logging], page 22).
`-F`	Foreground mode. Do not detach from the controlling terminal after startup, but remain in the foreground instead. This overrides the `Daemon` configuration setting (see [Daemon], page 29). The log stream (syslog facility or stderr) requested in the configuration file remains in effect.
`-f` *`file`*	Read configuration from the supplied *file*, instead of from the default location.
`-h`	Print short command line usage summary and exit.
`-p` *`file`*	Sets location of the *PID file*. This is the file where `pound` will write its PID after startup. This option overrides the value set by the `PIDFile` configuration setting (see [PIDFile], page 30).

`-v` Verbose mode. During startup, error messages will be sent to stderr (stdout, for `LOG_DEBUG` and `LOG_INFO` severities). If `pound` is configured to log to syslog, error diagnostics will be duplicated there as well. After startup the configuration settings take effect.

When used with `-c` this option also instructs `pound` to print an extra confirmation message on standard error, if there are no errors in the configuration file.

`-V` Print program version, licensing terms, and configuration flags and exit with status 0. You can use this option, in particular, to get the default values `pound` was built with, such as e.g. configuration file location.

`-W` *`feature`*
`-W no-`*`feature`*

Enable or disable (if prefixed with '`no-`') additional `pound` features. As of version 4.13, the following features are implemented:

`warn-deprecated` [Feature]

When parsing the configuration file, warn if it uses any deprecated statements. This is the default. To suppress deprecation messages, use `-W no-warn-deprecated`.

`dns` [Feature]

Resolve host names found in configuration file and returned in the `Location:` header. This is the default.

You can use `-W no-dns` to disable it, in order to suppress potentially lengthy network host address lookups. Make sure if your configuration file refers to backends only by their IP addresses in this case.

This setting affects also redirection location rewriting: See Section 9.9.6 [Response Modification], page 44.

`include-dir=`*`dir`* [Feature]
`no-include-dir` [Feature]

This controls the *include directory*, i.e. the directory where `pound` looks for relative file names referred to in its configuration file. See [include directory], page 33, for a detailed discussion of this feature.

Using `-W include-dir=`*`dir`* sets the new value of the include directory.

By default, the system configuration directory is used as include directory, so that any relative file names are looked up there. To disable this, use the `-W no-include-dir` option. This means that each relative filename used in arguments to the directives in the configuration file will be looked up in the current working directory. This is useful mainly in testsuite.

4 Simple Proxy

In this chapter we will deploy several simplest proxying configurations to illustrate the concepts introduced above.

Suppose you have an HTTP server running on localhost port 8080, and want to make it accessible from outside. This is achieved by the following configuration file:

```
ListenHTTP
    Address 0.0.0.0
    Port 80
    Service
        Backend
            Address 127.0.0.1
            Port 8080
        End
    End
End
```

This configuration consists of three nested sections: `ListenHTTP`, `Service`, and `Backend`. Each section ends with a keyword `End` on a line by itself.

The first thing that draws attention are `Address` and `Port` statements appearing in both listener and backend sections. In `ListenHTTP` they specify the IP address and port to listen on for incoming requests. Address '`0.0.0.0`' stands for all available IP addresses. In `Backend` section, these keywords specify the address and port of the remote server, where incoming requests are to be forwarded.

The `Service` section has no matching conditions, so it will match all requests.

4.1 Service Selection

To route requests to different servers, multiple services are used. In this case, each service has one or more *matching rules*, i.e. statements that define conditions that a request must match in order to be routed to that particular service. Syntactically, such rules have the form:

```
kw [options] "pattern"
```

where *kw* is a keyword specifying what part of the request is used in comparison, *pattern* is a textual pattern which that part is matched against, and *options* are zero or more flags starting with a dash sign, which define matching algorithm.

Perhaps the most often used condition is `Host`, which compares the value of the HTTP '`Host`' header with the given pattern. By default it uses exact case-insensitive match:

```
Host "example.com"
```

To treat the pattern as a regular expression, use the `-re` option, as in:

```
Host -re ".*\\.example\\.com"
```

Whenever we speak about regular expression we usually mean POSIX extended regular expressions (see Section "POSIX extended regular expressions" in *GNU sed*). However, other regex types can also be used. This is covered in Section 4.1.1 [Regular Expressions], page 9.

Notice the use of double backslashes in the above example. The backslash before each dot is needed to match it literally, while another one protects the first one from being interpreted as an escape character in string (see [Strings], page 27).

Other useful options are `-beg` and `-end`, which enable exact matching at the beginning and end of the value, correspondingly. Thus, the `Host` statement above can be rewritten as:

```
Host -end ".example.com"
```

The set of options available for use in matching statements is uniform. See Table 9.2, for a detailed discussion of available options.

The following configuration snippet illustrates the use of matching rules to select appropriate service (and, correspondingly, backend). It will route all requests for '`www.example.com`' to backend '`192.0.2.1:8080`', and requests for '`admin.example.com`' to '`192.0.2.4:8080`':

```
ListenHTTP
    Address 0.0.0.0
    Port 80

    Service
        Host "www.example.com"
        Backend
            Address 192.0.2.1
            Port 8080
        End
    End

    Service
        Host "admin.example.com"
        Backend
            Address 192.0.2.4
            Port 8080
        End
    End
End
```

Other matching statements use POSIX regexp matching by default. These are:

`Header` Compare HTTP header against a pattern. E.g.

```
Header "Content-Type:[[:space:]]*text/.*"
```

`URL` Match URL:

```
URL "/login/.*&name=.*"
```

`Path` Match the path part of the URL:

```
Path -beg "/login"
```

`Query` Match the query part of the URL.

`QueryParam`
: Match the value of a query parameter. This statement takes two arguments: parameter name and pattern, e.g.:

```
QueryParam "type" "(int)|(bool)"
```

See Section 9.11.1 [Service Selection Statements], page 48, for a detailed description of these and other matching statements.

Multiple matching rules can be used. Unless expressly specified otherwise, they are joined by logical '`and`' operation. For example:

```
Service
    Host "www.example.com"
    URL "^/admin(/.*)?"
    Backend
        Address 192.0.2.4
        Port 8080
    End
End
```

This service will be used for requests directed to host name '`www.example.com`' whose URL begins with '`/admin`', optionally followed by more path components (such as, e.g. '`http://www.example.com/admin/login`').

To select a service that matches one of defined rules (i.e. combine the rules using logical '`or`'), enclose them in `Match OR` block, as in:

```
Match OR
    Host "example.com"
    Host "www.example.com"
End
```

The argument to `Match` can be '`OR`' or '`AND`', specifying logical operation to be used to join the enclosed statements. The argument can be omitted, in which case '`AND`' is implied. `Match` statements can be nested to arbitrary depth, which allows for defining criteria of arbitrary complexity. For example:

```
Service
    Match OR
        Host "admin.example.com"
        Match AND
            Host "www.example.com"
            URL "^/admin(/.*)?"
        End
    End
    Backend
        Address 192.0.2.4
        Port 8080
    End
End
```

4.1.1 Regular Expressions

Request matching directives use POSIX extended regular expressions by default. If `pound` was compiled with `PCRE` or `PCRE2` library, *Perl-compatible regular expressions* can be used instead. This can be done either globally or individually for a given directive.

To change regular expression type globally, use the following directive:

```
RegexType pcre
```

It affects all request matching directives that appear after it in the configuration file, until next `RegexType` directive or end of file, whichever occurs first. To change back to POSIX regular expressions, use `posix` argument:

```
RegexType posix
```

Argument to the `RegexType` directive is case-insensitive.

Regular expression type can also be selected individually for a directive, using `-posix` or `-pcre` flags. For example:

```
Host -pcre -icase "(?<!www\\.)example.org"
```

4.1.2 ACL

Access control lists, or *ACLs*, are special request matching statements that evaluate to true if the request came from one of the predefined IP addresses. Access control lists are defined using the `ACL` section statement. Each line within it defines a single *CIDR* enclosed in double quotes. A CIDR consists of a *network address* (IPv4 or IPv6), optionally followed by slash and *network mask length*, a decimal number in the range [0,32] for IPv4 and [0.64] for IPv6. For example:

```
ACL
    "127.0.0.1/8"
    "192.0.2.0/25"
End
```

Such *anonymous ACLs* can appear anywhere a matching statement is allowed.

If an ACL is intended for use in multiple places of the configuration file, it can be defined as a *named ACL*. In a named ACL declaration, the `ACL` keyword is followed by a symbolic name in double quotes. This name must uniquely identify this ACL among other access control lists. Named ACLs are allowed only in the global (top-level) scope of a configuration file:

```
ACL "secure"
    "127.0.0.1/8"
    "192.0.2.0/25"
End
```

This ACL can then be used in any `Service` appearing after its definition by using the following construct:

```
ACL "secure"
```

Consider for example the following service declaration:

```
Service
    ACL "secure"
    Path -beg "/stat"
    Backend
        ...
    End
End
```

This service will handle requests whose URL starts with '`/stat`', if they came from one of the IP addresses mentioned in the access control list with the name '`secure`'. Effectively, this means that the access to that URL is limited to these IP addresses.

4.2 Request modifications

A service can modify requests before forwarding them to backends. Several statements are provided for that purpose:

SetHeader	Set a HTTP header.
DeleteHeader	Delete a HTTP header.
SetURL	Rewrite the request URL.
SetPath	Rewrite the path part of the URL.
SetQuery	Rewrite the query part of the URL.
SetQueryParam	Set a single query parameter.

For example, the following service declaration will add the header '`X-Resent-By: pound`' to each request:

```
Service
    SetHeader "X-Resent-By: pound"
    Backend
       ...
    End
End
```

Arguments to request modification statements are expanded before actual use. During expansion, references to *parenthesized subexpressions* in matching rules are replaced with their actual values. *Parenthesized subexpression* is a part of a regular expression enclosed in parentheses. It can be referred to in string arguments as '`$n`', where *n* is its ordinal number. Numbers start at one, '`$0`' referring to the entire string that matched.

The process of expanding parenthesized subexpressions is called *backreference expansion*.

For example, the following condition:

```
Header "Content-Type: ([^/]+)/(.+)$"
```

has two subexpressions: '`$1`' and '`$2`'. The following fragment uses these values to add two query parameters to the URL:

```
SetQueryParam "type" "$1"
SetQueryParam "subtype" "$2"
```

As a more practical example, the following service rewrites the path to JPEG and GIF images:

```
Service
    Path "/([^/]+\\.(jpg|gif))$"
    SetPath "/images/$1"
    ...
End
```

When several matching statements are used, these forms refer to the last one that matched. Subexpressions in prior statements can be referred to using the '`$i(j)`' construct. Here, *j* is the 0-based number of the statement, counted from the last one upwards. For example, given the following statements:

```
Host -re "www\.(.+)"
Header -icase "^Content-Type: *(.*)"
Path "^/static(/.*)?"
```

'`$1`' refers to the subexpression of `Path`, '`$1(1)`' to that of `Header`, and `$1(2)` to that of `Host`.

String arguments to `Set` statements can also contain *request accessors* – special constructs that are expanded to particular values from the request. Syntactically, a request accessor is '`%[name]`', where *name* denotes the request part to access. For example, `%[url]` expands to entire URL, `%[path]` to the path part of the URL, etc.

Using request accessors, the above example of path modification can be rewritten as:

```
Path "\\.(jpg|gif)$"
SetPath "/images%[path]"
```

See Section 9.3.2 [Request Accessors], page 29, for a detailed discussions of available accessors.

4.3 Conditional branches

Conditional request modifications can be organized in logical branches, each branch being applied only if the request matches certain condition. The `Rewrite` section encloses a set of request matching rules followed by one or more request modification statements, which will be applied if the former match the request. Optional `Else` sub-section, which in turn contains request matching rules and modification statements, will be tried if those rules don't match. Any number of `Else` sub-sections is allowed, each one being tried if the previous ones don't match.

The example below illustrates this concept. This configuration snippet sets different paths depending on the file type and URL used:

```
Service
    Rewrite
        Header "Content-Type:[[:space:]]+image/.*"
        SetPath "/images%[path]"
    Else
        Match AND
            Host "example.org"
            Path "\\.[^.]+$"
        End
        SetPath "/static%[path]"
    Else
        Path "\\.[^.]+$"
        SetPath "/assets%[path]"
    End
    ...
End
```

4.4 Modifying responses

The `rewrite` statement can also be used to modify responses received from backends before passing them back to the querying party. To indicate this intent, the `Rewrite` statement must be followed by the `response` keyword:

```
    Rewrite response
        SetHeader "X-Been-There: pound"
    End
```

When modifying responses, only two request modification statements are allowed: `SetHeader` and `DeleteHeader`. The list of request matching rules is limited as well: `Header` and `StringMatch`, plus `Match` and `Not` conditionals. Notice that these conditionals operate on the response, and not on the request, as in previous chapters. For example:

```
    Rewrite response
        Header "Content-Type:[[:space:]]+text/(.*)"
        SetHeader "X-Text-Type: $1"
    End
```

This will insert an additional `X-Text-Type` header into the response. It will contain the subtype value from the `Content-Type` header of the original response.

4.5 Authentication

Along with access control lists, introduced above (see Section 4.1.2 [ACL], page 9), authentication provides another way to limit access to certain services. `Pound` supports *basic authentication*, as defined in RFC 7617.

This authentication method relies on the presence of the `Authorization` header in the HTTP request. If the header is present, its value specifies the '`Basic`' authorization method and contains credentials (username and password) that match one of the users from the server user database, the request is accepted. Otherwise a 401 ('`Authentication Required`') or 407 ('`Proxy Authentication Required`') response is returned with the `WWW-Authenticate` header requesting basic authentication.

The `BasicAuth` request matching statement verifies if the `Authorization` header is present and provides correct credentials. The statement matches the request if so.

The `BasicAuth` statement takes a single argument, specifying the name of a file containing *user database*. This is a plain-text file created with `htpasswd` or similar utility, i.e. each non-empty line of it must contain username and password hash separated by a colon. Password hash can be one of:

- Password in plain text.

- Hash created by the system `crypt`(3) function.
- Password hashed using SHA1 algorithm and encoded in BASE64. This hash must be prefixed by '`{SHA}`'.
- `Apache`-style '`APR1`' hash.

Password file is read on the first authorization attempt, after which its contents is stored in memory. `Pound` will re-read it if it notices that the file's modification file has changed, so you need not restart the daemon if you do any changes to the file.

Thus, if you put the `BasicAuth` statement in each service that must be accessible to authorized users only, that would do the first and principal part of the basic authentication scheme: access control. There remains second part: returning properly formatted 401 response if the request did not pass authorization. That can be done using a combination of the `Error` internal backend (see Section 4.7 [Error responses], page 14) and response modification techniques described in the previous section.

However, instead of using `BasicAuth` in each service requiring limited access and placing a service generating the 401 response in the end, it is simpler and less error-prone to use the following approach:

Create a service with the following content:

```
Service
    Not BasicAuth "pound/htpasswd"
    Rewrite response
        SetHeader "WWW-Authenticate: Basic realm=\"Restricted access\""
    End
    Error 401
End
```

Replace the file name (`pound/htpasswd`) and realm name ('`Restricted access`') with the actual values.

Make sure that all services that need to be protected by basic authentication are declared after that service. This way, any request that does not convey an `Authentication` header with credentials matching an entry from your password file will match this service, and will be replied to with a properly formatted 401 response, which will prompt the remote user to authenticate himself. On the other hand, authorized requests will not match this service and will eventually be handled by one of the services declared after it.

4.6 Redirects

Apart from regular backends introduced in previous sections, `pound` provides also several *special* or *internal* backends. As their name implies, such backends handle requests and generate responses internally, without forwarding them to any external entities.

One of such internal backends is `Redirect`. It generates responses redirecting the client to another location. The statement takes two arguments: a three-digit HTTP status code to return, and the URL to redirect to:

```
Service
    Redirect 301 "https://www.gnu.org"
End
```

Allowed values for the status code are 301, 302, 303, 307 and 308. This argument is optional: if omitted, 302 is used.

If the URL argument has no path component (as in the example above), then the path (and query, if present) components from the original request will be appended to it. For example, if the original URL were '`http://example.com/software`', the service above would redirect it '`https://www.gnu.org/software`'.

Otherwise, if the path component is present in the URL argument (even if it is a mere '`/`'), then the URL is used as is. For example, the following will drop any path and query components from the URL when redirecting:

```
Redirect 301 "https://www.gnu.org/"
```

The URL argument is subject to backreference expansion and request accessor interpretation (see Section 4.2 [Request modifications], page 10). If any of these are actually used, the above logic is disabled.

String expansions make it possible to implement complex redirects. For example, the following redirect swaps the first two path components of the original URL:

```
Service
    URL "^/([^/]+)/([^/]+)(/.*)?"
    Redirect "http://%[host]/$2/$1$3"
End
```

The following is a standard paradigm for redirecting requests from HTTP to HTTPS:

```
Service
    Redirect 301 "https://%[host]%[url]"
End
```

4.7 Error responses

Another type of internal backends is `Error`, a backend that generates error responses. It is useful, for instance, to provide a custom error status and/or message when no service matches the request. Normally, for such cases `pound` generates a standard 503 response ('`Service Unavailable`') with the built-in error page. You can customize this behavior by using as the last service a `Service` section with `Error` backend. For example:

```
Service
    Error 404 "pound/404.html"
End
```

The first argument specifies the HTTP status code to return. See [Error backend], page 53, for more info.

The second argument is optional. It supplies the name of a file with the error page to return along with the response. The name may be absolute or relative. In the latter case, the file will be looked up in the *include directory*, a special directory for storing pound-specific files. See [include directory], page 33. The file will be read only once, at program startup. If you modify the file and want `pound` to notice changes, you will have to restart it.

If the second argument is not supplied, the error text is determined by the `ErrorFile` ***`code`*** statement in the enclosing listener (where *code* is the HTTP code in question). If it is not supplied, the built-in default text is used. See Section 9.9.3 [Error definitions], page 37.

5 HTTPS

In the previous chapter we have described basic proxying techniques using plain HTTP listener as an example. Now we will discuss how to use HTTPS both for listeners and backends.

To accept HTTPS requests you need to declare `ListenerHTTPS` listener. It is similar to plain `ListenerHTTP` described above, except that it requires a *certificate* to be declared. For example:

```
ListenHTTPS
    Address 0.0.0.0
    Port 443
    Cert "/etc/ssl/priv/example.pem"
    Disable TLSv1
    Ciphers "HIGH:@STRENGTH:!RSA"
End
```

The `Cert` statement supplies the name of the certificate file in PEM format. The file must contain the certificate, intermediate certificates (if necessary), and certificate private key, in that order.

The `Cert` argument can also specify a directory, in which case `pound` will scan that directory, trying to read the certificate from each regular file encountered. It will report an error if unable to load the file, so this directory should contain only certificate files. The order in which certificate files are read is not specified.

Multiple `Cert` statements are allowed. When trying to find the matching certificate, `pound` will stop at the first one whose `CN` matches the requested host name. Thus, the ordering of `Cert` statements is important. Normally they should be placed in most-specific to least-specific order, with wildcard certificates appearing after host-specific ones.

`Cert` directives must precede all other SSL-specific directives.

Another important directive is `Disable`. It disables the use of the specified TLS protocol as well as all protocols older than it. Usually it is used to disable obsolete protocols. For example, the `Disable` statement in the example above disables '`TLSv1`', '`SSLv3`', and '`SSLv2`'.

To further tune the strength of your encryption use the `Ciphers` directive. Its argument is a colon-delimited list of OpenSSL ciphers, as described in See Section "ciphers" in `ciphers(1)`. The cipher selection shown in the example above provides for excellent encryption strength.

5.1 ACME

Automatic Certificate Management Environment (*ACME*), is a protocol for automated deployment of HTTPS certificates. It is perhaps the most often used method for obtaining SSL certificates nowadays. In order to issue certificate for a domain or domains, the protocol verifies that the web server that is requesting a certificate actually owns these domains. This process is based on various *challenge types*.

`Pound` supports HTTP-01[1] challenge type. When issuing a certificate using this challenge type, the ACME client (a program responsible for periodic certificate re-issuing) obtains from the authority a challenge file, and stores it in a predefined *challenge directory*. The authority will then request this file from the webserver using a predefined URL. It is supposed that the server will serve it from the file that has been just written by the agent. If the server returns the file, its claim to own the domain is proved and the certificate is issued.

Configuring `pound` to reply to challenge requests is as simple as putting an `ACME` statement to the `ListenHTTP` section of its configuration file. The statement takes a single argument – name of the challenge directory:

```
ListenHTTP
    Address 0.0.0.0
    Port 80
    ACME "/var/lib/pound/.well-known/acme-challenge"
End
```

Needless to say, your ACME agent and `pound` must agree on this directory location. Configuration of various ACME agents is beyond the scope of this document. Please refer to the documentation of your agent for further details.

5.2 Redirect HTTP to HTTPS

Nowadays it is common to redirect all plain HTTP requests to HTTPS on the same URL. The method of doing so was described in Section 4.6 [Redirects], page 13. As an example, this section shows a working HTTPS configuration with such redirect.

```
ListenHTTP
    Address 0.0.0.0
    Port 80
    Service
        Redirect 301 "https://%[host]%[url]"
    End
End

ListenHTTPS
    Address 0.0.0.0
    Port 443
    Cert "/etc/ssl/priv/example.pem"
    Disable TLSv1
    Service
        Backend
            Address 127.0.0.1
            Port 8080
        End
    End
End
```

[1] `https://letsencrypt.org/docs/challenge-types/#http-01-challenge`

5.3 HTTPS backends

Backends can use HTTPS as well. To inform `pound` that communication with the backend goes over an encrypted channel, use the `HTTPS` keyword. The typical usage is:

```
Backend
    Address 192.0.2.1
    Port 443
    HTTPS
End
```

Notice, that unlike other statements, `HTTPS` is used without arguments.

Additional directives are available for fine-tuning the connection. If used, they must appear after the `HTTPS` directive,

The `Cert` directive specify the client certificate to use when connecting. Use it if the backend requires client authentication.

The `Disable` and `Ciphers` directives are similar to those described when discussing `ListenHTTPS`: the former disables the given TLS protocol and all protocols prior to it, and the latter configures the list of OpenSSL ciphers which the client wishes to use. The actual cipher to use will be selected from this list during negotiation with the backend.

The example below illustrates the use of these directives:

```
Backend
    Address 192.0.2.1
    Port 443
    HTTPS
    Disable TLSv1_1
    Cert "/etc/pound/crt/b1.pem"
    Ciphers "HIGH:!RSA"
End
```

6 Request balancing

When several backends are defined in a service, incoming requests will be distributed among them. This process is called *balancing*. By default, requests are distributed equally. This can be changed by assigning them a *priority* – a decimal number which controls a relative weight of the given backend in the distribution algorithm. The bigger the priority is, the more requests this backend gets from the total flow.

The distribution algorithm is defined by *balancing strategy*. As of version 4.13, `pound` supports two strategies: *weighted random balancing* and *interleaved weighted round robin balancing*.

Weighted Random Balancing

This is the default strategy. Each backend is assigned a numeric priority between 0 and 9, inclusive. The backend to use for each request is determined at random taking into account backend priorities, so that backends with numerically greater priorities have proportionally greater chances of being selected than the ones with lesser priorities.

The share of requests a backend receives can be estimated as:

$$P_i \ / \ S(P)$$

where P_i is the priority of the backend with index i, and $S(P)$ is the sum of all priorities.

Interleaved Weighted Round Robin Balancing

Requests are assigned to each backend in turn. Backend priorities, or weights, are used to control the share of requests received by each backend. The greater the weight, the more requests will be sent to this backend. In general, the share of requests assigned to a backend is calculated by the following relation:

$$(P_i \ + \ 1) \ / \ (N \ + \ S(P))$$

where N is total number of backends, and P_i and $S(P)$ are as discussed above.

Weighted random balancing is used by default. Each backend gets the default priority 5, unless another value is expressly assigned using the `Priority` statement, e.g.:

```
Service
    Backend
        Address 192.168.0.1
        Port 80
        Priority 1
    End
    Backend
        Address 192.168.0.2
        Port 80
        Priority 9
    End
End
```

In this example, backend `192.168.0.2` will receive roughly 9 times more requests than backend `192.168.0.1`.

The balancing strategy to use is defined by the `Balancer` keyword, which can appear either in the global scope or within a `Service` section. Its argument can be one of:

`random`	Use weighted random balancing (default).
`iwrr`	Use interleaved weighted round robin balancing.

The `Balancer` statement appearing in the global scope defines balancing strategy for all services that don't have `Balancer` statement on their own.

6.1 Sessions

Some web applications attempt to introduce state persistence into the stateless HTTP protocol, by defining *sessions* using various mechanisms, such as specially defined headers, cookies, etc. For such applications it is critical that all requests that belong to a single *session* be directed to the same server, i.e. backend. Clearly, this disrupts the balancer logic, and requires that the proxy be able to understand the backend's notion of session.

`Pound` is able to detect and track sessions identified by client address, Basic authentication (user id/password), URL parameter, cookie, HTTP parameter, and HTTP header value.

Session tracking is enabled on a per-service basis by a `Session` section. The section must contain at least the `Type` directive, which specifies what type of session tracking to use, and the `TTL` directive, supplying session idle timeout in seconds.

Session types are case-insensitive. They are summarized in the table below:

`IP` The `IP` session tracking type instructs `pound` to forward all requests from the same client IP address to the same backend server:

```
Session
    Type IP
    TTL  300
End
```

`Basic` Using this session tracking type, `pound` parses the `Authentication` header of each request. If the header is present, and specifies the '`Basic`' authentication type, user ID is extracted from it. Requests with the same user ID are forwarded to the same backend server.

```
Session
    Type Basic
    TTL  300
End
```

`URL` This tracking scheme uses the value of URL query parameter to define a session. The parameter name is supplied using the `ID` directive:

```
Session
    Type URL
    TTL  300
    ID   "sess"
End
```

In this example, sessions are identified by the '`sess`' parameter, The request URL might look like '`http://example.org?sess=123`'.

`Cookie` The `Cookie` tracking type use a certain cookie to identify sessions. The cookie name is given by the `ID` directive:

```
Session
    Type Cookie
    TTL  300
    ID   "sess"
End
```

`Header` Sessions are identified by the value of HTTP header whose name is given by the `ID` directive, e.g.:

```
Session
    Type Header
    ID   "X-Session"
    TTL  300
End
```

`Parm` This is the least useful scheme. Sessions are identified by HTTP parameter - a string that appears after a semicolon in the URL, such as '`bar`' in '`http://foo.com;bar`'

```
Session
    Type PARM
    TTL  300
End
```

7 Worker model

Each incoming request is processed by a specific *worker*, i.e. a thread in the running program. Total number of running workers is controlled by three configuration parameters. `WorkerMinCount` defines the minimum number of workers that should always be running (5, by default). Another parameter, `WorkerMaxCount` sets the upper limit on the number of running workers (it defaults to 128).

At each given moment, a worker can be in one of two states: *idle* or *active* (processing a request). If an incoming request arrives when all running workers are active, and total number of workers is less than `WorkerMaxCount`, a new thread is started and the new request is handed to it. If the number of active workers has already reached maximum, the new request is added to the *request queue*, where it will wait for a worker to become available to process it.

The third parameter, `WorkerIdleTimeout`, specifies maximum time a thread is allowed to spend in the idle state. If a worker remains idle longer than that and total number of workers is greater than the allotted minimum (`WorkerMinCount`), this idle worker is terminated.

8 Logging

`Pound` can send its diagnostic messages to standard error, syslog, or to both.

Upon startup, while the configuration file is being parsed, the diagnostics goes to the standard error. Once it switches to the operation mode and starts serving requests, diagnostic output is switched to syslog. The syslog facility to use is configured via the `LogFacility` configuration directive. By default, '`daemon`' is used.

When running in foreground mode, the `-e` command line option instructs `pound` to use standard error for logging, thus overriding the settings from the configuration file.

Normally, `pound` is not very loquacious in logging: only errors are reported. Logging of each incoming request can be configured using the `LogLevel` directive. It can be used either in listener scope, in which case it affects only this particular listener, or in global scope, where it affects all listeners that don't configure it on their own. The value of this directive can be either an integer number in range 0 through 5 (inclusive), or a quoted string. Numeric value requests one of the *built-in log formats*. String value refers either to a built-in format name, or to a user-defined format name.

The built-in formats are:

0 null	No request logging at all.
1 regular	For each request, its source address, request line and response status are logged.
2 extended	In addition to the above, the selected service and backend are shown.
3 vhost_combined	Detailed request logging using Apache-style *Combined* log format.
4 combined	Same as above, but without virtual host information.
5 detailed	Same as '`combined`', with additional information about the selected service and backend.

If the string argument to `LogLevel` is not one of the above, it must refer to the name of a *custom format*, defined earlier using the `LogFormat` statement. This statement takes two string arguments: the name to be assigned to the new format, and its definition.

Format definition is a character string composed of ordinary characters (not '`%`'), which are copied unchanged to the resulting log message, and conversion specifications, each of which are replaced by a corresponding piece of information about the request or reply.

Conversion specifications are single characters prefixed with a percent sign. Depending on the specification, an optional *conversion argument* in curly brackets may appear between '`%`' and conversion character.

The following conversion characters are defined:

`%%` [Format specifier]

Replaced with the percent sign.

`%a` [Format specifier]

Originator IP address of the request. If the request contains `X-Forwarded-For` header and `TrustedIP` ACL is defined, the value of the header is consulted to obtain the IP address. The value must be a comma-delimited list of intermediate user-agent IP addresses. To determine the actual user-agent IP, the list is traversed from right to left, until an IP is found that is not listed in `TrustedIP` ACL.

If `X-Forwarded-For` is not present, or `TrustedIP` is not defined, or the above algorithm does not return an IP address, `%a` expands to the actual remote IP address the request came from (same as `%h`).

The `TrustedIP` ACL can be defined in global scope, or in `ListenHTTP` (`ListenHTTPS`) section, or in `Service` section. Most-specific ACL overrides least-specific ones, that is a `TrustedIP` defined in `Service` will be used, if it is defined. If not, the one defined in listener will be used, etc. The syntax of the `TrustedIP` statement is the same as that of `ACL`, i.e.

```
TrustedIP "name"
```

refers to the named ACL *name* (which must be defined earlier, see Section 4.1.2 [ACL], page 9), and

```
TrustedIP
  "cidr0"
  "cidr1"
  ...
End
```

defines the list of trusted IPs in place.

If needed, the `ForwardedHeader` statement may be used to declare the name of the header to use instead of `X-Forwarded-For`. As `TrustedIP`, this statement can appear in global, listener, or in service scope.

`%A` [Format specifier]

Local IP address of the listener.

`%B` [Format specifier]

Size of response in bytes, excluding headers.

`%b` [Format specifier]

Same as '`%B`', but in *CLF* format, i.e. a dash is used when response size is zero.

`%D` [Format specifier]

The time taken to serve the request, in microseconds.

`%h` [Format specifier]

Client IP address of the request.

`%H` [Format specifier]

The request protocol.

`%{`*`hdr`*`}i` [Format specifier]

The contents of '*`hdr`*`:`' header line in the request. Changes made by header modification directives affect this.

`%{`*`hdr`*`}I` [Format specifier]

Same as '`%i`', except that if no such header is present in the request, a dash is substituted.

`%{`*`obj`*`}L` [Format specifier]

Location of the `pound` object that is involved in handling the request. Valid values for *obj* are: '`listener`', '`service`', and '`backend`'.

The location gives position in the configuration file where the object was defined, and is formatted as

```
name:ln1.col1-ln2.col2
```

where *name* is the configuration file name, *ln1* and *col1* are line and column where the object definition begins, *ln2* and *col2* are line and column where it ends. Line and column numbers start with 1.

`%m` [Format specifier]

The request method.

`%{`*`obj`*`}N` [Format specifier]

Name of `pound` object that is involved in handling the request. Valid values for *obj* are: '`listener`', '`service`', and '`backend`'.

`%P` [Format specifier]

Thread ID of the serving thread.

`%q` [Format specifier]

The query string (prepended with a '`?`') if it exists, otherwise an empty string.

`%r` [Format specifier]

First line of request.

`%s` [Format specifier]

Response status code.

`%>s` [Format specifier]

First line of the response.

`%t` [Format specifier]

Time the request was received, in the format '`[18/Sep/2011:19:18:28 -0400]`'. The last number indicates the timezone offset from UTC.

`%{`*`format`*`}t` [Format specifier]

Time the request was received, in the format specified by the argument (see Appendix B [Time and Date Formats], page 70). If the format starts with '`begin:`' (default) the time is taken at the beginning of the request processing. If it starts with '`end:`', it is the time after the response from the backend has been sent back to the requester. In addition to `strftime` formats, the following specifications are recognized:

sec	Number of seconds since the Epoch.
msec	Number of milliseconds since the Epoch.
usec	Number of microseconds since the Epoch.
msec_frac	Millisecond fraction of the time.
usec_frac	Microsecond fraction of the time.

`%T` [Format specifier]

The time taken to process the request, in seconds.

`%{`*`unit`*`}T` [Format specifier]

The time taken to process the request, in a time unit given by *unit*. Valid units are '`ms`' for milliseconds, '`us`' for microseconds, '`s`' for seconds, and '`f`' for seconds with fractional part. Using '`s`' gives the same result as '`%T`' without any format; using '`us`' gives the same result as '`%D`'.

`%u` [Format specifier]

Remote user if the request was authenticated.

`%U` [Format specifier]

The URL path requested. This is affected by request modification directives.

`%v` [Format specifier]

The listener name.

The table below describes the built-in formats in terms of format definitions:

0
null

```
""
```

1
regular

```
"%a %r - %>s"
```

2
extended

```
"%a %r - %>s (%{Host}i/%{service}N -> %{backend}N) %{f}T sec"
```

3
vhost_combined

```
"%{Host}I %a - %u %t \"%r\" %s %b \"%{Referer}i\" \"%{User-Agent}i\""
```

4
combined

```
"%a - %u %t \"%r\" %s %b \"%{Referer}i\" \"%{User-Agent}i\""
```

5
detailed (Split in two lines for readability)

```
"%{Host}I %a - %u %t \"%r\" %s %b \"%{Referer}i\" \"%{User-Agent}i\"
(%{service}N -> %{backend}N) %{f}T sec"
```

9 Configuration

A configuration file provides `pound` with the information necessary for performing its tasks. Some configuration file statements can be overridden from the command line.

9.1 Lexical structure

Lexically, the file contains tokens of three types: keywords, values, and separators. Blanks, tabs, newlines and comments, collectively called *white space* are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent keywords and values.

Comments may appear anywhere where white space may appear in the configuration file. A comment begins with a hash sign (‘`#`’) and continues to the end of the line.

A *keyword* is a sequence of ASCII letters, digits and underscores that begins with an ASCII letter or underscore. Keywords are always case-insensitive.

There are three kinds of *values*: numeric values (or *numbers*), boolean values, quoted strings, and IP addresses.

Numbers A *numeric value* is a sequence of decimal digits.

Booleans A *boolean* is one of the following: ‘`yes`’, ‘`true`’, ‘`on`’ or ‘`1`’, meaning *true*, and ‘`no`’, ‘`false`’, ‘`off`’, ‘`0`’ meaning *false*.

Strings A *quoted string* or *string*, for short, is a sequence of characters enclosed in a pair of double quotes. A backslash (‘\’) appearing within a string acts as an *escape character*: if it is followed by a double-quote or another backslash, it forces the character that follows it to be treated as an ordinary one. For example:

```
"string with \" character"
```

A backslash followed by any character other than ‘`"`’ or ‘\’ is removed and a warning to that effect is output. For example, the following statement:

```
user "r\oot"
```

appearing at line 1 of file `pound.cfg` will result in the following message:

```
pound.cfg:1.8: unrecognized escape character
```

and will be treated as

```
user "root"
```

IP addresses
IP addresses are IPv4 or IPv6 addresses in numeric form, or hostnames.

9.2 Syntax

Syntactically, `pound` configuration is a sequence of statements of two kinds: simple and compound.

A *simple statement* or *directive* consists of a keyword followed by a value, located on a single line. For example:

```
user "proxy"
```

There are some simple statements that don't take any value and thus consist only of a keyword, e.g.

```
HTTPS
```

A *compound statement* or *section* encloses one or more other statements (both simple or compound). It begins with a keyword, optionally followed by a value, both located on a single line (similar to simple directives), followed by any number of subordinate statements, and ends with a keyword `End` on a line by itself. For example:

```
Control
    Socket "/run/pound.sock"
    Mode 660
    ChangeOwner true
End
```

Unless specified otherwise, directives may appear in any order.

9.3 String Expansions

String arguments to some configuration statements undergo several *expansions* before use. The *backreference expansion* replaces special notations in the string called *backreferences* with corresponding parts of the request recently matched against a regular expression. The *request accessor interpretation* inserts some fragments of the request URL into the string.

These expansions are discussed in detail below.

9.3.1 Backreference expansion

Backreference is a construct that refers to a *parenthesized group* within a regular expression matched by one of service matching directives (see Section 9.11.1 [Service Selection Statements], page 48). During backreference expansion, each occurrence of such construct is replaced with the actual value of that parenthesized group.

Syntactically, backreferences can take two forms. The construct `$`*n*, where *n* is a decimal number, refers to *n*th parenthesized subexpression of the most recently matched statement, and the construct `$`*n*`(`*m*`)` refers to *n*th parenthesized subexpression in the *m*th recently matched statement. Numbering of subexpressions starts at 1 (`$0` refers to entire matching string). Numbering of matches starts at 0.

For example, given the following statements

```
Host -re "www\\.(.+)"
Header -re -icase "^Content-Type: *(.*)"
Path "^/static(/.*)?"
```

`$1` refers to the subgroup of `Path`, `$1(1)` refers to that of `Header`, and `$1(2)` to that of `Host`.

Curly braces may be used to avoid incorrectly parsing text fragment that follows the reference as being its part. This is useful if the reference is immediately followed by a decimal digit or opening parenthesis, as in: '`${1}(text)`'.

To insert a literal dollar or percent sign in the string, use '`$$`' or '`$%`', correspondingly.

9.3.2 Request Accessor Interpretation

Request accessor is a syntactical construct of the form:

`%[name]`

where *name* denotes a part of the incoming request to access and square brackets are part of the construct. Accessors are interpreted and replaced with the value of the corresponding part of the request. Some accessors take an argument, which is specified after accessor name and is delimited from it by one or more whitespace characters.

The following accessors are defined:

`url` [Accessor]

Request URL.

`path` [Accessor]

Request path.

`query` [Accessor]

Query part.

`param` *`name`* [Accessor]

The value of the query parameter *name*.

`header` *`name`* [Accessor]

The value of HTTP header *name*.

`host` [Accessor]

Hostname part of the *Host* header. If the latter does not include port number, it is equivalent to `%[header host]`.

`port` [Accessor]

If the value of the `Host` header includes port number, '`%[port]`' expands to port number with the leading colon character. Otherwise, it expands to empty string.

9.4 Global directives

Global directives configure the program operation as a whole. They may appear anywhere at the global scope of the configuration file, although it is customary for them to be at its start.

9.4.1 Runtime directives

`Daemon` *`bool`* [Global directive]

When set to '`true`' (the default), `pound` will detach itself from the controlling terminal after successful parsing of the configuration file and continue operating in the background.

When set to '`false`', `pound` will continue operating in the foreground.

This setting can be overridden by the `-F` and `-e` command line options.

`Group "`*`group_name`*`"` [Global directive]

Sets the group `pound` will run as. If not set, the primary group of the user (as set by the `User` directive) will be used.

`PIDFile "`*`filename`*`"` [Global directive]

Sets the name of the file where to store program PID. This can be also be set from command line, using `-p` command line option (see Chapter 3 [Usage], page 4).

Notice the following:

1. When running with a supervisor, this file holds PID of the supervisor process. Otherwise, it holds PID of the main This means it is always suitable for signalling the program using the traditional `kill ‘cat filename‘` technique.
2. Before shutting down, `pound` removes this file. However, it may fail to do so if it switches to privileges of another user after startup (at least one of `User` or `Group` are set in the configuration file) and the file is stored in a directory whose permissions forbid write access for that user.

`Supervisor `*`bool`* [Global directive]

When running in daemon mode, start a *supervisor* process. This process, in turn, will start main `pound` process and will further monitor it, restarting it if it fails.

The default is `true`.

`RootJail "`*`directory`*`"` [Global directive]

If this directive is present, `pound` will use the system `chroot` call to set the root directory of the process to that specified by *directory*. After that, the program won’t be able to access any files outside that directory.

Before chrooting, `pound` makes the necessary preparations to be able to access the files it needs during operation, in particular user databases supplied with the `BasicAuth` statements (see Section 4.5 [Authentication], page 12).

`User "`*`user_name`*`"` [Global directive]

Configures the user `pound` will run as.

9.4.2 Worker Settings

`WorkerMinCount `*`n`* [Global directive]

Sets minimum number of worker threads that must always be running. The default is 5. See Chapter 7 [Worker model], page 21.

`WorkerMaxCount `*`n`* [Global directive]

Sets maximum number of worker threads. The default is 128. See Chapter 7 [Worker model], page 21.

`WorkerIdleTimeout `*`n`* [Global directive]

Sets idle timeout for a worker thread, in seconds. Default is 30 seconds. See Chapter 7 [Worker model], page 21.

`Threads `*`n`* [Global directive]

This statement, retained for backward compatibility with previous versions of pound, is equivalent to:

```
WorkerMinCount n
WorkerMaxCount n
```

9.4.3 Proxy Tuning Directives

`BackendStats` *`bool`* [Global directive]

Whether to enable backend statistics collection. Backend statistics consists of the following values:

1. Total number of requests processed by this backend.
2. Average time per request.
3. Standard deviation of the average time per request.

If enabled, these values are made available via `poundctl` (see [poundctl list], page 58) and telemetry output (see [Metrics], page 54).

`Balancer` *`algo`* [Global directive]

Sets the request balancing algorithm to use. Allowed values for *algo* are:

random Use weighted random balancing algorithm.

iwrr Use interleaved weighted round robin balancing.

See Chapter 6 [Balancer], page 18, for a detailed discussion of these algorithms.

The `Balancer` statement in global scope applies to all `Service` definitions in the file that don't contain `Balancer` definitions of their own.

`HeaderOption` *`opt`* ... [Global directive]

Sets default header addition options. One or more arguments are allowed, each being one of:

`off` Disable additional headers.

`forwarded`
Add `X-For-warded-For`, `X-Forwarded-Proto`, and `X-Forwarded-Port` headers.

`ssl` Pass information about SSL certificates in a set of `X-SSL-*` headers. This will add the following headers:

X-SSL-Cipher
SSL version followed by a slash and active cipher algorithm.

X-SSL-Certificate
The full client certificate (multi-line).

X-SSL-Issuer
Information about the certificate issuer (CA).

X-SSL-Subject
Information about the certificate owner.

X-SSL-notAfter
End of validity date for the certificate.

X-SSL-notBefore
Start of validity date for the certificate.

X-SSL-serial
: Certificate serial number (in decimal).

The default is:

```
HeaderOption forwarded ssl
```

This setting can be overridden for a particular listener using the `HeadOption` within it.

9.4.4 SSL Settings

`SSLEngine "name"` [Global directive]

Use an OpenSSL hardware acceleration card called *name*. Available only if OpenSSL-engine is installed on your system.

`ECDHcurve "name"` [Global directive]

Use the named curve for elliptical curve encryption.

9.4.5 Regular Expression Settings

`RegexType type` [Global directive]

Sets the type of regular expressions to use in request matching statements. Allowed values for *type* are: `posix` and `pcre` (or `perl`), case-insensitive. The latter requires compilation time support.

The selected regular expression type remains in effect for all request matching directives that follow this statement, until next `RegexType` statement or end of the configuration file, whichever occurs first.

Regular expression type can be set individually for a directive, using the `-pcre` or `-posix` option (see Table 9.2).

See Section 4.1.1 [Regular Expressions], page 9, for a detailed discussion.

`IgnoreCase bool` [Global directive]

Ignore case when doing regex matching (default: '`false`'). This directive sets the default for the following service matching directives: `URL`, `Path`, `QueryParam`, `Query`, `StringMatch`, as well as for the `DeleteHeader` modification directive. Its value can be overridden for specific services.

This statement is deprecated and will be removed in future versions. Please, use the `-icase` option to the matching directive instead (see Table 9.2).

9.4.6 ACL Definition

`ACL "name"` [Global directive]

Define a *named access control list*. An *ACL* is a list of network addresses in CIDR notation, one address per line, terminated with an `End` directive on a line by itself. E.g.:

```
ACL "secure"
   "192.0.2.0/26"
   "203.0.113.0/24"
End
```

The `Include` directive is allowed within the ACL section. Named ACLs can be used in `Service` definitions to limit access to services from certain IP addresses only. See Section 4.1.2 [ACL], page 9, for a detailed discussion of this.

9.5 File inclusion

`Include "`*`file`*`"` [Global directive]

Include *file* as if it were part of the configuration file. If *file* is a relative file name, it will be looked in the *include directory* (see below).

This directive is allowed both at topmost level and in any subsections of the configuration file.

`IncludeDir "`*`dir`*`"` [Global directive]

Set the *include directory*, i.e. the directory where `pound` looks for relative file names that appear in other configuration directives: `Include`, `BasicAuth`, `ErrorFile` (or `Err400` through `Err503`), as well as in the argument to `-file` option in service matching directives (see [-file], page 43).

The default value is the system configuration directory as set at compile time (you can check its value in the output of `pound -V`). This initial value can be changed in the command line using the `-W include-dir=`*`dir`* command line option or reset to the current working directory using the `-W no-include-dir` option (see Chapter 3 [Usage], page 4).

9.6 Logging configuration

`LogFacility `*`name`* [Global directive]
`LogFacility -` [Global directive]

Sets the `syslog` facility to use for logging. Allowed names are: '`auth`', '`authpriv`', '`cron`', '`daemon`', '`ftp`', '`kern`', '`lpr`'. '`mail`', '`news`', '`user`'. '`uucp`', and '`local0`' through '`local7`'.

The second form configures *default log destination*. If `pound` runs in foreground, log messages with priority `LOG_DEBUG` and `LOG_INFO` go to stdout, and messages with the remaining priorities are printed to stderr. If `pound` runs as a daemon, log messages go to the syslog facility '`daemon`'.

`LogFormat "`*`name`*`" "`*`format_def`*`"` [Global directive]

Define request logging format. *name* is a string uniquely identifying this format, and *format_def* is the format string definition. See Chapter 8 [Logging], page 22, for a detailed description of format definition syntax.

`LogLevel "`*`name`*`"` [Global directive]
`LogLevel `*`n`* [Global directive]

Specify the format to use to log HTTP requests. *name* is a name of a custom format, defined earlier using the `LogFormat` directive, or one of six built-in format names.

If numeric argument is used, it refers to a built-in format by its number (0 through 5).

See Chapter 8 [Logging], page 22, for a detailed description of HTTP request logging.

`LogTag "string"` [Global directive]

Sets the string to tag syslog messages with. By default, it is the name of the program (more precisely, the name which was used to start it).

`ForwardedHeader "name"` [Global directive]

Defines the name of the HTTP header that carries the list of proxies the request has passed through. Default value is `X-Forwarded-For`. This header is used to determine the originator IP address for logging. See [%a], page 23, for details.

`TrustedIP` [Global directive]

Defines a list of *trusted proxy* IP addresses, which is used to determine the originator IP. See [%a], page 23, for details.

This statement is a special form of `ACL` statement, described below. It can appear in two forms: as a *section* or as a *directive*. When used as a section, it is followed by a list of one or more CIDRs each appearing on a separate line. The `End` keyword terminates the statement, e.g.:

```
TrustedIP
  "127.0.0.1/8"
  "10.16.0.0/16"
End
```

In directive form, this statement takes a single argument, the name of an access control list defined earlier using the `ACL` statement, e.g.

```
TrustedIP "proxy_addresses"
```

`Anonymise` [Global directive]
`Anonymize` [Global directive]

When logging, replace the last byte of client IP addresses with 0.

Default: log the client address in full.

9.7 Control socket settings

`Pound` can be instructed to listen on a UNIX socket for management requests, which will allow you to obtain information about the running instance, change state of configured listeners, services, and backends, etc. These requests are normally issued by the poundctl utility (see Chapter 10 [poundctl], page 57).

Properties of this *control socket* are configured via the `Control` statement. It has two forms: *directive* and *section*.

`Control "filename"` [Global directive]

Create a UNIX socket *filename* and listen on it for management requests. The file will be owned by the user that started `pound` (normally '`root`') and have mode 0600.

In section form, the `Control` statement allows for specifying file mode and, to certain extent, socket file ownership. The section can contain the following statements:

`Socket "filename"` [Control statement]

Specifies the name of the socket file to use. This is the only mandatory statement in the section form.

`Mode` *`octal`* [Control statement]

Sets the mode of the socket file.

`ChangeOwner` *`bool`* [Control statement]

This statement takes effect if at least one of `User` or `Group` global statements is used. When set to `true` it will change the owner of the socket file to that specified by those two statements.

An example of using the `Control` section:

```
Control
    Socket "/run/pound.sock"
    Mode 660
    ChangeOwner true
End
```

9.8 Timeouts

Directives discussed in this section set various timeout values. Their argument is an integer expressing the value in seconds.

`Alive` *`n`* [Global directive]

Specify how often should `pound` check for the status of backend servers marked as *dead* (i.e. inaccessible). It is a good idea to set this as low as possible – it will find resurrected hosts faster. However, if you set it too low it will consume resources.

Default is 30 seconds.

`Client` *`n`* [Global directive]

Specify for how long `pound` will wait for a client request (default: 10 seconds). It will drop the connection if client doesn't send any data within this interval.

This value can be overridden for specific listeners.

`TimeOut` *`n`* [Global directive]

Specify for how long `pound` will wait for the backend to respond (default: 15 seconds).

This value can be overridden for specific backends.

`ConnTO` *`n`* [Global directive]

Specify for how long `pound` will wait for a connection to a backend to be established. Default is the same as the `TimeOut` value.

This value can be overridden for specific backends.

`WSTimeOut` *`n`* [Global directive]

Specify for how long `pound` will wait for data from either backend or client in a connection upgraded to WebSocket protocol. Default is 600 seconds.

This value can be overridden for specific backends.

`Grace` *`n`* [Global directive]

How long should `pound` continue to answer existing connections after a receiving a '`INT`' or '`HUP`' signal (default: 30 seconds). The configured listeners are closed immediately. You can bypass this behaviour by stopping `pound` with a '`TERM`' or '`QUIT`' signal, in which case the program exits without any delay.

9.9 ListenHTTP

The `ListenHTTP` section declares a listener operating in plaintext HTTP mode. The section declaration begins with the keyword `ListenHTTP` optionally followed by a string supplying symbolic name for that listener, e.g.:

```
ListenHTTP "main"
  ...
End
```

The symbolic name can be used in log messages (see [log format], page 22) and in `poundctl` (see Chapter 10 [poundctl], page 57) requests to identify that listener. If the name is not supplied, the listener can be identified by its ordinal number (0-based) in the configuration file.

9.9.1 Listener address

`Address` *`address`* [ListenHTTP directive]

The IP address that `pound` will listen on. This can be a numeric IPv4 or IPv6 address, a symbolic host name that must be resolvable at runtime (unless the `-Wno-dns` option is used), or a full pathname of a UNIX socket. To listen on all available interfaces, use '`0.0.0.0`'.

Either `Address` or `SocketFrom` (see below) must be present in each `ListenHTTP` section.

`Port` *`n`* [ListenHTTP directive]

The port number or service name (as per `/etc/services` that this listener will listen on. This directive must be present, unless the `Address` directive specifies a UNIX socket.

`SocketFrom "`*`pathname`*`"` [ListenHTTP directive]

Read the socket to listen on from the UNIX socket supplied by *pathname*. If this parameter is supplied, neither `Address` nor `Port` may be used. This parameter is intended for use in `pound` testsuite.

9.9.2 Listener-specific limits

`Client` *`n`* [ListenHTTP directive]

Specify for how long `pound` will wait for a client request (default: 10 seconds). It will drop the connection if client doesn't send any data within this interval.

This statement overrides the global timeout value (see Section 9.8 [Timeouts], page 35) for this particular listener.

`MaxRequest` *`n`* [ListenHTTP directive]

Limits the maximum allowed size of incoming requests. A request bigger than that will be responded with status 413.

By default, there is no limit on the request size.

`MaxURI` *`n`* [ListenHTTP directive]

Limits the maximum allowed length of incoming request URI. A request with an URI longer than that will be responded with status 414.

By default, there is no limit on the URI length.

`CheckURL "pattern"` [ListenHTTP directive]

Define a pattern that must be matched by each request sent to this listener. A request that does not match will be returned a 501 status.

`xHTTP n` [ListenHTTP directive]

Defines which HTTP method are accepted. The possible values are:

0	Accept only standard HTTP methods: `GET`, `POST`, `HEAD`. This is the default.
1	Allow also extended HTTP methods: `PUT`, `PATCH`, `DELETE`.
2	Additionally allow standard WebDAV methods: `LOCK`, `UNLOCK`, `PROPFIND`, `PROPPATCH`, `SEARCH`, `MKCOL`, `MOVE`, `COPY`, `OPTIONS`, `TRACE`, `MKACTIVITY`, `CHECKOUT`, `MERGE`, `REPORT`.
3	Additionally allow MS extension WebDAV methods: `SUBSCRIBE`, `UNSUBSCRIBE`, `NOTIFY`, `BPROPFIND`, `BPROPPATCH`, `POLL`, `BMOVE`, `BCOPY`, `BDELETE`, `CONNECT`.

9.9.3 Error definitions

When `pound` returns an error status, it uses built-in error-specific description code and status page template. These values can be customized using the `ErrorFile` statement.

`ErrorFile code "filename"` [ListenHTTP directive]

Read HTML page for HTTP status code *code* from file *filename*.

The *code* argument is a three-digit HTTP response status, and *filename* is the name of a file which supplies text of the error page to be returned. The file is read once, at program startup.

For compatibility with `pound` versions up to 4.11, the following statement is also recognized:

```
Errnnn "filename"
```

where *nnn* is a three-digit HTTP status code. This statement is entirely equivalent to

```
ErrorFile nnn "filename"
```

`Pound` produces only a subset of all possible status codes, so not all *nnn* codes are allowed. The discussion below lists available HTTP codes, along with the error description and default error page text.

`400` [HTTP status]

‘`Bad Request`’

> Your browser (or proxy) sent a request that this server could not understand.

`401` [HTTP status]

‘`Unauthorized`’

> This server could not verify that you are authorized to access the document requested. Either you supplied the wrong credentials (e.g., bad

password), or your browser doesn't understand how to supply the credentials required.

`403` [HTTP status]
‘`Forbidden`’

You don't have permission to access this resource. It is either read-protected or not readable by the server.

`404` [HTTP status]
‘`Not Found`’

The requested URL was not found on this server.

`405` [HTTP status]
‘`Method Not Allowed`’

The request method is not supported for the requested resource.

`413` [HTTP status]
‘`Payload Too Large`’

The request content is larger than the proxy server is able to process.

`414` [HTTP status]
‘`URI Too Long`’

The length of the requested URL exceeds the capacity limit for this server.

`500` [HTTP status]
‘`Internal Server Error`’

The server encountered an internal error and was unable to complete your request.

`501` [HTTP status]
‘`Not Implemented`’

The server does not support the action requested.

`503` [HTTP status]
‘`Service Unavailable`’

The server is temporarily unable to service your request due to maintenance downtime or capacity problems. Please try again later.

9.9.4 Listener logging

Following statements are similar to the ones described in Section 9.6 [Logging configuration], page 33, but apply only to the listener they appear in.

`LogLevel "`*`name`*`"` [ListenHTTP directive]
`LogLevel `*`n`* [ListenHTTP directive]

Specify the format to use to log HTTP requests. *name* is a name of a custom format, defined earlier using the `LogFormat` directive, or one of six built-in format names.

If numeric argument is used, it refers to a built-in format by its number (0 through 5).

See Chapter 8 [Logging], page 22, for a detailed description of HTTP request logging.

`ForwardedHeader "`*`name`*`"` [ListenHTTP directive]

Defines the name of the HTTP header that carries the list of proxies the request has passed through. Default value is `X-Forwarded-For`. This header is used to determine the originator IP address for logging. See [%a], page 23, for details.

`TrustedIP` [ListenerHTTP directive]

Defines a list of *trusted proxy* IP addresses, which is used to determine the originator IP. See [%a], page 23, for details.

9.9.5 Request Modification

The statements discussed in this subsection modify incoming requests prior to passing them to the backend. These same set of statements can also be used in `Service` section (see Section 9.11 [Service], page 47). When appearing in both sections, the directive from `ListenHTTP` (`ListenHTTPS`) section are applied first, followed by directives from the `Service` section. Directives from the same section are applied in order of their appearance.

`RewriteDestination `*`bool`* [ListenerHTTP directive]

If set to '`true`', the `Destination:` request header will be changed to point to the backend with the correct protocol.

`SetURL "`*`url`*`"` [ListenerHTTP directive]

Set the URL of the incoming request to *url*.

`SetPath "`*`value`*`"` [ListenerHTTP directive]

Set the path part of the URL to the given string.

`SetQuery "`*`value`*`"` [ListenerHTTP directive]

Set the query part of the URL to the given string. *Value* must be a valid query with the special characters properly encoded using percent encoding.

`SetQueryParam "`*`name`*`" "`*`value`*`"` [ListenerHTTP directive]

Set the query parameter *name* to the *value*. Value must be properly encoded if it contains reserved characters.

`SetHeader "`*`name: value`*`"` [ListenerHTTP directive]
`HeaderAdd "`*`name: value`*`"` [ListenerHTTP directive]

`AddHeader "`*`name: value`*`"` [ListenerHTTP directive]

Sets the HTTP header. If the header *name* already exists, it will be overwritten. Otherwise, new header will be added to the end of the header list.

The `HeaderAdd` and `AddHeader` forms are retained for backward compatibility with earlier `pound` versions. You are advised against using them.

`DeleteHeader [`*`options`*`] "`*`pattern`*`"` [ListenerHTTP directive]

Remove from the request all headers matching *pattern*. The `HeaderRemove` and `HeadRemove` forms are retained for backward compatibility with earlier `pound` versions. You are advised against using them.

By default, *pattern* is treated as extended POSIX regular expression. The *options* argument can be used to alter this. It consists of zero or more option flags from the following list:

Flag	**Meaning**
`-beg`	Exact match at the beginning of string (prefix match).
`-case`	Case-sensitive comparison.
`-contain`	Delete each header where "*pattern*" is a substring.
`-end`	Exact match at the end of string (suffix match).
`-exact`	Use exact string match.
`-icase`	Case-insensitive comparison.
`-pcre`	Use Perl-compatible regular expression. see Section 4.1.1 [Regular Expressions], page 9.
`-perl`	Same as `-pcre`.
`-posix`	Use POSIX extended regular expression. see Section 4.1.1 [Regular Expressions], page 9.
`-re`	Use regular expression match. This assumes the default regular expression type, as set by the `RegexType` directive (see Section 4.1.1 [Regular Expressions], page 9).

Table 9.1: Header matching flags for `DeleteHeader` directive

The following options are mutually exclusive: `-beg`, `-contain`, `-end`, `-exact`, `-pcre` (`-perl`), `-posix`, `-re`. If more than one of these are used, the last one takes effect.

`HeaderRemove "`*`pattern`*`"` [ListenerHTTP directive]
`HeadRemove "`*`pattern`*`"` [ListenerHTTP directive]

These are obsolete keywords, equivalent to

```
DeleteHeader -icase "pattern"
```

9.9.5.1 The `rewrite` statement

The `Rewrite` block statement associates one or more header modification directives discussed above with *request matching directives*, so that request modification takes place only when the request matches certain conditions.

Syntactically, a `Rewrite` section is:

```
  Rewrite [ request ]
    conditional_directives...
    modification_directives...
[ Else
    conditional_directives...
    modification_directives... ]
  End
```

where *conditional_directives* represents one or more *request conditionals* described below and *modification_directives* stands for one or more header modification directives. The `Else` part is optional; any number of `Else` blocks can be supplied, thus providing for conditional branching.

The `Rewrite` statement is processed sequentially until a branch is found whose *conditional_directives* yield '`true`', or `End` is encountered. If a matching branch is found, its *modification_directives* are applied to the request.

Request matching directives or *request conditionals* are special statements that, being applied to a HTTP request, yield '`true`' or '`false`' depending on whether the request satisfies the condition described in the directive. The following conditionals are available:

`ACL "`*`name`*`"` [Request Conditional]

Returns '`true`' if the source IP matches one of the CIDRs from the named access control list *name*. The ACL itself must have been defined earlier (see Section 9.4.6 [ACL definition], page 32).

See Section 4.1.2 [ACL], page 9, for a detailed discussion.

`ACL` [Request Conditional]

This statement defines an unnamed ACL to match the source IP against. This line must be followed by one or more lines defining CIDRs, as described in Section 9.4.6 [ACL definition], page 32. The ACL definition is finished with an `End` keyword on a line by itself.

Semantically, this statement is equivalent to the named ACL reference described above.

See Section 4.1.2 [ACL], page 9, for a detailed discussion.

`BasicAuth "`*`filename`*`"` [Request Conditional]

Evaluates to '`true`', if the incoming request passes basic authorization as described in RFC 7617. *Filename* is the name of a plain text file containing usernames and passwords, created with `htpasswd` or similar utility. Unless the name starts with a slash, it is taken relative to the `IncludeDir` directory (see [include directory], page 33). The file is cached in the memory on the first authorization attempt, so that further authorizations do not result in disk operations. The file will be re-scanned if `pound` notices that its modification time has changed.

See Section 4.5 [Authentication], page 12.

`Header [`*`options`*`] "`*`pattern`*`"` [Request Conditional]

Yields '`true`', if the request contains at least one header matching the given *pattern*. By default, *pattern* is treated as case-insensitive POSIX extended regular expression. This can be changed by *options*, described below.

`Host [`*`options`*`] "`*`hostname`*`"` [Request Conditional]

Evaluates to '`true`', if the `Host` header matches *hostname*. In the absence of *options*, case-insensitive exact match is assumed, i.e. this construct is equivalent to

```
Header "Host:[[:space:]]*qhost"
```

where *qhost* is the *hostname* argument in quoted form, i.e. with all characters that have special meaning in regular expressions escaped.

See Table 9.2, for a detailed discussion of *options* and their effect on matching.

This statement is provided to facilitate handling of *virtual hosts*. See Section 4.1 [Service selection], page 6, for details.

`Path [`*`options`*`] "`*`pattern`*`"` [Request Conditional]

Returns '`true`', if the path part of the incoming request matches *pattern*.

`Query [`*`options`*`] "`*`pattern`*`"` [Request Conditional]

Returns '`true`', if the query part of the incoming request matches *pattern*. The argument must be properly percent-encoded, if it contains whitespace or other non-printable characters.

`QueryParam "`*`name`*`" [`*`options`*`] "`*`pattern`*`"` [Request Conditional]

Returns '`true`', if the value of the query parameter *name* matches *pattern*.

See Table 9.2, for a detailed discussion of *options* and their effect on matching.

`StringMatch "`*`string`*`" [`*`options`*`] "`*`pattern`*`"` [Request Conditional]

Expands *string* as described in Section 9.3 [String Expansions], page 28, and matches the resulting value against *pattern*.

`URL [`*`options`*`] "`*`pattern`*`"` [Request Conditional]

Matches URL of the request. *Pattern* is treated as case-sensitive extended regular expression, unless instructed otherwise by *options* (see below).

In these directives, *options* is a whitespace-delimited list of zero or more flags from the following table:

Flag	**Meaning**
`-beg`	Exact match at the beginning of string (prefix match).
`-case`	Case-sensitive comparison.
`-contain`	Match if *pattern* is a substring of the original value.
`-end`	Exact match at the end of string (suffix match).
`-exact`	Use exact string match.
`-file`	Treat *pattern* as the name of a file to read patterns from. If the name is relative, it will be looked up in the [include directory], page 33. Patterns are read from the file line by line. Leading and trailing whitespace is removed. Empty lines and comments (lines starting with `#`) are ignored.
`-icase`	Case-insensitive comparison.
`-pcre`	Use Perl-compatible regular expression. see Section 4.1.1 [Regular Expressions], page 9.
`-perl`	Same as `-pcre`.
`-posix`	Use POSIX extended regular expression. see Section 4.1.1 [Regular Expressions], page 9.
`-re`	Use regular expression match. This assumes the default regular expression type, as set by the `RegexType` directive (see Section 4.1.1 [Regular Expressions], page 9).

Table 9.2: Conditional directive flags

The following options are mutually exclusive: `-beg`, `-contain`, `-end`, `-exact`, `-pcre` (`-perl`), `-posix`, `-re`. If more than one of these are used, the last one takes effect.

Placing the keyword `Not` before a header matching directive reverts its meaning. For example, the following will match any request whose URL does not begin with `/static/`:

```
Not URL -beg "/static/"
```

The `Match` block statement can be used to join multiple header matching directives. Its syntax is:

```
Match op
  ...
End
```

where ... stands for any number of matching directives, and *op* is a boolean operation: `AND` or `OR` (case-insensitive). For example, the statement

```
Match OR
  Host "www.example.net"
  Path -beg "/ssl"
End
```

will match if the request `Host` header has the value '`www.example.net`', or the path part of its URL starts with '`/ssl`'. In contrast, the statement below:

```
Match AND
  Host "www.example.net"
  Path -beg "/ssl"
End
```

will match only if both these conditions are met. As a syntactical short-cut, two or more matching statements appearing outside of `Match` are joined by an implicit logical `AND`, so that the latter example is equivalent to:

```
Host "www.example.net"
Path -beg "/ssl"
```

The `Match` statement, like any other matching directive, can be prefixed with `Not`, which reverts its meaning.

9.9.6 Response Modification

`RewriteLocation` *n* [ListenerHTTP directive]

This statement controls whether `Location:` and `Content-location:` headers in HTTP responses are modified before sending them back to the client.

If *n* is 0, both headers are left intact.

If *n* is 1, the headers are changed as follows. If they point to the backend itself or to the listener (but with the wrong protocol), the request host name will be used instead. This is the default.

If *n* is 2, do the same, but compare only the backend address; this is useful for redirecting a request to an HTTPS listener on the same server as the HTTP listener.

To check whether the location points to the listener or to the backend, its hostname part is resolved and the obtained IP address (or addresses) are compared with that of listener or backend. This process is affected by the `dns` feature setting (see [dns], page 5). If it is disabled (`-W no-dns` option is given), no resolving takes place. In this case the location is deemed to point to the listener if its hostname part matches that of the incoming request. For backends, the hostname is compared with the value of the `ServerName` setting of that backend (see [ServerName], page 53), if any.

9.9.6.1 The `Rewrite response` statement.

A special form of the `Rewrite` statement is provided for modifying headers in the response obtained from a regular backend or generated with a `Error` backend, before sending them back to the requesting server:

```
    Rewrite response
      conditional_directives...
      modification_directives...
  [ Else
      conditional_directives...
      modification_directives... ]
    End
```

The conditional directives allowed for use in this statement are:

`Header [options] "pattern"` [Rewrite response conditional]
: Returns '`true`', if the response contains at least one header matching the given *pattern*.

`StringMatch "string" [options] "pattern"` [Rewrite response conditional]
: Expands *string* as described in Section 9.3 [String Expansions], page 28, and matches the resulting value against *pattern*.

Both conditionals treat their *pattern* argument as case-insensitive POSIX extended regular expression. See Table 9.2, for a discussion of available *options*.

The following *response modification* directives are defined:

`DeleteHeader [options] "pattern"` [Response modification]
: Remove matching headers from the response. By default, *pattern* is treated as extended POSIX regular expression. Use *options* to alter this behavior. See Table 9.1, for a list of available options.

`SetHeader "name: value"` [Response modification]
: Sets the HTTP response header. Argument undergoes string expansion (See Section 9.3 [String Expansions], page 28). If the header *name* already exists, it will be overwritten. Otherwise, new header will be added to the end of the header list.

9.9.7 Service definitions

The `Service` section defines rules that decide to which backend to route requests received by that listener. Any number of `Service` section can be present. When a request is received, the listener iterates over all services in the order of their appearance in the configuration and applies the section rules to the request. If the rules match the request, the request is forwarded to the backend defined in that section.

See Section 9.11 [Service], page 47, for a detailed discussion of the `Service` statement.

`ACME dir` [ListenHTTP statement]
: This statement defines a special service with a built-in backend for handling '`ACME`' challenge requests. See Section 5.1 [ACME], page 15, for a detailed discussion of its use.

: The *dir* argument defines the directory where to look for challenge files.

9.10 ListenHTTPS

The `ListenHTTPS` section defines a listener that operates in HTTPS. The section declaration begins with the keyword `ListenHTTPS` optionally followed by a string supplying symbolic name for that listener:

```
ListenHTTPS "main"
  ...
End
```

The purpose of the symbolic name is the same as in `ListenHTTP` statement. All keywords defined for `ListenHTTP` can be used for `ListenHTTPS` as well. See Section 9.9 [ListenHTTP], page 36, for a detailed discussion of these.

Statements specific for this section are:

`Cert "`*`filename`*`"` [ListenHTTPS]

Specifies the server *certificate*. *Filename* is either a certificate file name, or the name of a directory containing certificate files.

A *certificate file* is a file containing the certificate, possibly a certificate chain and the signature for this server, in that order.

This directive is mandatory within `ListenHTTPS`.

Multiple `Cert` directives are allowed. If multiple directives are used, the first one is the default certificate, with additional certificates used if the client requests them.

The ordering of the directives is important: the first certificate where the CN matches the client request will be used, so put your directives in the most-specific-to-least specific order (i.e. wildcard certificates after host-specific certificates).

`Cert` directives must precede all other SSL-specific directives.

`ClientCert` *`mode depth`* [ListenHTTPS]

Specifies whether the listener must ask for the client's HTTPS certificate. Allowed values for *mode* are:

0. Never ask for the certificate (the default).
1. Ask for the client certificate.
2. Ask and fail, if no certificate was presented.
3. Ask but do not verify.

Depth is the depth of verification for a client certificate (up to 9). The default depth limit is 9, allowing for the peer certificate and additional 9 CA certificates that must be verified.

`Disable` *`proto`* [ListenHTTPS]

Disable the SSL protocol *proto* and all lower protocols as well. Allowed values for *proto* are: `SSLv2`, `SSLv3`, `TLSv1`, `TLSv1_1`, `TLSv1_2`.

For example:

```
Disable TLSv1
```

This disables SSLv2, SSLv3 and TLSv1, thus allowing only TLSv1_1 and TLSv1_2.

`Ciphers "cipher_list"` [ListenHTTPS]

This is the list of ciphers that will be accepted by the SSL connection; it is a string in the same format as in `OpenSSL ciphers` and `SSL_CTX_set_cipher_list` functions.

`SSLHonorCipherOrder bool` [ListenHTTPS]

If set `true`, the server will broadcast a preference to use ciphers in the order supplied in the `Ciphers` directive. If the value is `false`, the server will accept any cipher from the `Ciphers` list. Default value is `false`.

`SSLAllowClientRenegotiation mode` [ListenHTTPS]

If *mode* is 0, client initiated renegotiation will be disabled. This will mitigate DoS exploits based on client renegotiation, regardless of the patch status of clients and servers related to *Secure renegotiation*. If *mode* is 1, secure renegotiation is supported. If *mode* value is 2, insecure renegotiation is supported.

The default value is 0.

`CAlist "filename"` [ListenHTTPS]

Set the list of trusted CA's for this server. The *filename* is the name of a file containing a sequence of CA certificates (in PEM format). The names of the defined CA certificates will be sent to the client on connection.

`VerifyList "filename"` [ListenHTTPS]

Set the certificate authority list. The *filename* is the name of a file with CA root certificates, in PEM format.

Please note, that there is an important difference between the `CAlist` and the `VerifyList`. The `CAlist` tells the client (browser) which client certificates it should send. The `VerifyList` defines which CAs are actually used for the verification of the returned certificate.

`CRLlist "filename"` [ListenHTTPS]

Set the *Certificate Revocation List* file. *Filename* is the name of a file that contains the CRLs (in PEM format).

`NoHTTPS11 mode` [ListenHTTPS]

Behave like an `HTTP/1.0` server for HTTPS clients. If *mode* is 0, always conform to HTTPS/1.1. If it is 1, do not allow multiple requests on SSL connections. If the value is 2 (default), disable multiple requests on SSL connections only for MSIE clients.

9.11 Service

The `Service` statements define backends to use and conditions a request should satisfy in order to be routed to these backends. These statements can appear both within `ListenHTTP` (`ListenHTTPS`) sections and outside of them. When processing an incoming request, the listener will first try to match it against services defined within it. If none of these services matches the request, it will try to match it against services defined in the top level. If a matching service is found, it will be used to process the request. Otherwise a 503 (*Service Unavailable*) response will be returned.

A service is defined by a section statement that begins with the `Section` keyword, followed by service definition statements and terminated by `End` on a line by itself:

```
Service "name"
  ...
End
```

Optional *name* argument assigns a symbolic name to the service. That name is used to identify the service in diagnostic and access log messages (see [log format], page 22), metric output (see [Metrics], page 54), and in `poundctl` requests (see Chapter 10 [poundctl], page 57). In the absence of an assigned *name*, the ordinal number of the service in the enclosing section is used as its identifier. Service numbers start at 0.

Following subsections discuss statements that can be used in `Service` sections.

9.11.1 Service Selection Statements

Service selection statements define conditions an incoming request must satisfy in order to be handled by this service.

`ACL "name"` [Service Conditional]

Returns '`true`' if the source IP of the request matches one of the CIDRs from the named access control list *name*. The ACL itself must have been defined earlier (see Section 9.4.6 [ACL definition], page 32).

See Section 4.1.2 [ACL], page 9, for a detailed discussion.

`ACL` [Service Conditional]

This statement defines an unnamed ACL to match the source IP against. This line must be followed by one or more lines defining CIDRs, as described in Section 9.4.6 [ACL definition], page 32. The ACL definition is finished with an `End` keyword on a line by itself.

Semantically, this statement is equivalent to the named ACL reference described above.

See Section 4.1.2 [ACL], page 9, for a detailed discussion.

`BasicAuth "filename"` [Service Conditional]

Evaluates to '`true`', if the incoming request passes basic authorization as described in RFC 7617. *Filename* is the name of a plain text file containing usernames and passwords, created with `htpasswd` or similar utility. Unless the name starts with a slash, it is taken relative to the `IncludeDir` directory (see [include directory], page 33). The file is cached in the memory on the first authorization attempt, so that further authorizations do not result in disk operations. The file will be rescanned if `pound` notices that its modification time has changed.

See Section 4.5 [Authentication], page 12.

`Header [options] "pattern"` [Service Conditional]

Yields '`true`', if the request contains at least one header matching the given *pattern*. By default, *pattern* is treated as case-insensitive POSIX extended regular expression. This can be changed by *options*, described below.

`Host [options] "hostname"` [Service Conditional]

Evaluates to '`true`', if the `Host` header matches *hostname*. In the absence of *options*, case-insensitive exact match is assumed, i.e. this construct is equivalent to

```
Header "Host:[[:space:]]*qhost"
```

where *qhost* is the *hostname* argument in quoted form, i.e. with all characters that have special meaning in regular expressions escaped.

See Table 9.2, for a detailed discussion of *options* and their effect on matching.

This statement is provided to facilitate handling of *virtual hosts*. See Section 4.1 [Service selection], page 6, for details.

`Path [options] "pattern"` [Service Conditional]

Returns '`true`', if the path part of the incoming request matches *pattern*.

`Query [options] "pattern"` [Service Conditional]

Returns '`true`', if the query part of the incoming request matches *pattern*. The argument must be properly percent-encoded, if it contains whitespace or other non-printable characters.

`QueryParam "name" [options] "pattern"` [Service Conditional]

Returns '`true`', if the value of the query parameter *name* matches *pattern*.

See Table 9.2, for a detailed discussion of *options* and their effect on matching.

`StringMatch "string" [options] "pattern"` [Service Conditional]

Expands *string* as described in Section 9.3 [String Expansions], page 28, and matches the resulting value against *pattern*.

`URL [options] "pattern"` [Service Conditional]

Matches URL of the request. *Pattern* is treated as case-sensitive extended regular expression, unless instructed otherwise by *options* (see below).

The *options* argument in the directives discussed above defines the comparison algorithm used. It consists of one or more flags described in Table 9.2.

Placing the keyword `Not` before a header matching directive reverts its meaning. For example, the following will match any request whose URL does not begin with `/static/`:

```
Not URL -beg "/static/"
```

The `Match` block statement can be used to join multiple header matching directives. Its syntax is:

```
Match op
  ...
End
```

where ... stands for any number of matching directives, and *op* is a boolean operation: `AND` or `OR` (case-insensitive). See [Match in service statement], page 8, for a detailed discussion with examples.

9.11.2 Request and Response Modification

These statements modify incoming requests prior to passing them to the backend. A similar set of statements can be used in listeners (see Section 9.9.5 [Request Modification], page 39). In case both the listener and service contain request modification statements, those from the listener are applied first, followed by the ones from the service.

`SetURL "`*`url`*`"` [Service directive]

Set the URL of the incoming request to *url*.

`SetPath "`*`value`*`"` [Service directive]

Set the path part of the URL to the given string.

`SetQuery "`*`value`*`"` [Service directive]

Set the query part of the URL to the given string. *Value* must be a valid query with the special characters properly encoded using percent encoding.

`SetQueryParam "`*`name`*`" "`*`value`*`"` [Service directive]

Set the query parameter *name* to the *value*. The value must be properly encoded if it contains reserved characters.

`SetHeader "`*`name: value`*`"` [Service directive]
`HeaderAdd "`*`name: value`*`"` [Service directive]
`AddHeader "`*`name: value`*`"` [Service directive]

Sets the HTTP header. If the header *name* already exists, it will be overwritten. Otherwise, new header will be added to the end of the header list.

The `HeaderAdd` and `AddHeader` forms are retained for backward compatibility with earlier `pound` versions. You are advised against using them.

`DeleteHeader [`*`options`*`] "`*`pattern`*`"` [Service directive]
`HeaderRemove [`*`options`*`] "`*`pattern`*`"` [Service directive]
`HeadRemove [`*`options`*`] "`*`pattern`*`"` [Service directive]

Remove from the request all headers matching *pattern*. The `HeaderRemove` and `HeadRemove` forms are retained for backward compatibility with earlier `pound` versions. You are advised against using them.

By default, *pattern* is treated as extended POSIX regular expression. The *options* argument can be used to alter this. It consists of zero or more option flags, described in Table 9.1.

`Rewrite` [*request* | *response*] ... *End* [Service statement]

This block statement associates one or more header modification directives discussed above with *request matching directives*, so that request modification takes place only when the request matches certain conditions.

By default `Rewrite` statements apply to incoming requests. The subject of rewriting can also be specified explicitly after the `Rewrite` keyword.

See Section 9.9.5.1 [Rewrite], page 41, for a detailed discussion of this statement.

See Section 4.3 [Conditional branches], page 11, for an in-depth discussion with examples.

See Section 4.4 [Modifying responses], page 12, for a discussion of the use of this statement to modify responses.

9.11.3 Service Logging

`ForwardedHeader "name"` [Service directive]

Defines the name of the HTTP header that carries the list of proxies the request has passed through. Default value is `X-Forwarded-For`. This header is used to determine the originator IP address for logging. See [%a], page 23, for details.

`TrustedIP` [Service directive]

Defines a list of *trusted proxy* IP addresses, which is used to determine the originator IP.

See [%a], page 23, for a detailed discussion.

This statement is a special form of `ACL` statement, described in Section 4.1.2 [ACL], page 9. It can appear in two forms: as a *section* or as a *directive*. When used as a section, it is followed by a list of one or more CIDRs each appearing on a separate line. The `End` keyword terminates the statement, e.g.:

```
TrustedIP
  "127.0.0.1/8"
  "10.16.0.0/16"
End
```

In directive form, this statement takes a single argument, the name of an access control list defined earlier using the `ACL` statement, e.g.

```
TrustedIP "proxy_addresses"
```

`LogSuppress class [class...]` [Service directive]

Suppresses HTTP logging for requests that resulted in status codes from the specified classes. Valid classes are:

info 1	'`1xx`' response codes.
success 2	'`2xx`' response codes.
redirect 3	'`3xx`' response codes.
clterr 4	'`4xx`' response codes.
srverr 5	'`5xx`' response codes.
all	All response codes.

This statement is designed for services that receive a constant stream of similar HTTP requests from a controlled set of IP addresses, such as e.g. Openmetric services. See [Metrics], page 54, for an example.

9.11.4 Backends

9.11.4.1 Backend

The `Backend` section defines a regular backend. The overall syntax, as for any section statement, is:

```
Backend [ "name" ]
  ...
End
```

Optional *name* argument assigns a symbolic name to the service. That name is used to identify the backend in diagnostic and access log messages (see [log format], page 22), metric output (see [Metrics], page 54), and in `poundctl` requests (see Chapter 10 [poundctl], page 57). In the absence of an assigned *name*, the ordinal (0-based) number of the backend in the enclosing `Service` is used as its identifier.

The following statements can be used in a `Backend` section:

`Address` *`IP`* [Backend directive]

IP address or host name of the backend server. If the name cannot be resolved to a valid address, `pound` will assume that it represents a path to a Unix-domain socket.

This directive is mandatory.

`Port` *`n`* [Backend directive]

Sets the port number to connect to. This directive must be present if the `Address` statement contains an IP address.

`Disabled` *`bool`* [Backend directive]

Mark this backend as disabled.

Backends can be enabled or disabled at runtime using the `poundctl` utility (see Section 10.1 [poundctl commands], page 58).

Note: not to be confused with the `Disable` statement, described below.

`Priority` *`n`* [Backend directive]

Sets numeric priority for this backend. Priorities are used to control probability of receiving a request for handling in case of multiple backends. See Chapter 6 [Balancer], page 18, for a detailed discussion.

Allowed values for *n* depend on the balancing algorithm in use. For random balancing, allowed values are 0 to 9. For IWRR, allowed values are between 0 and 100.

The following three directives set various timeout parameters for backend operations:

`TimeOut` *`n`* [Backend directive]

Sets the *response timeout*, i.e. time to wait for a response from the backend (in seconds). Default is 15.

`ConnTO` *`n`* [Backend directive]

Sets *connection timeout*, i.e. time to wait for establishing connection with the backend (in seconds).

`WSTimeOut` *`n`* [Backend directive]

Idle timeout for WebSocket operations, in seconds. Default value: 600 (10 minutes).

Backend servers can use HTTPS as well as plaintext HTTP. The following directives configure HTTPS backends:

`HTTPS` [Backend directive]
: This directive indicates that the remote server speaks HTTPS.

`ServerName "`*`name`*`"` [Backend directive]
: This directive specifies the name to use for server name identification (*SNI*). It also rewrites the `Host:` header for this particular backend. This means you don't have to use `SetHeader` in addition to it.

`Cert "`*`filename`*`"` [Backend directive]
: This specifies the certificate that `pound` will use as a client. The *filename* is the name of a file containing the certificate, possibly a certificate chain and the signature.

`Ciphers "`*`cipherlist`*`"` [Backend directive]
: This is the list of ciphers that will be accepted by the SSL connection with the backend (for HTTPS backends); it is a string in the same format as used by the OpenSSL functions `ciphers` and `SSL_CTX_set_cipher_list`.

`Disable` *`proto`* [Backend directive]
: Disable the SSL protocol *proto* and all earlier protocols. Allowed values for *proto* are: `SSLv2`, `SSLv3`, `TLSv1`, `TLSv1_1`, `TLSv1_2`.

  Note: not to be confused with the `Disabled` statement, described above.

9.11.4.2 Globally Defined Backends

The `Backend` section described above can also be used at the topmost level of the configuration file. Use this if you plan to use same backend in several different services.

When used globally the `Backend` keyword must always be followed by the backend name in double-quotes. The assigned name must be unique among all global backends.

To include a globally defined backend in a service, use `UseBackend` or `Backend` keywords.

`UseBackend "`*`name`*`"` [Service directive]
: Use globally-defined backend *name* in this service. The backend itself may be defined in global scope before or after the `Service` section that uses it.

The `UseBackend` keyword adds the backend to the service exactly as it was defined. However, it may sometimes be necessary to alter its priority and state. To do so, use the `Backend` section. If the name argument specifies a globally-defined backend, the `Backend` section can contain only the `Priority` and `Disable` statements.

9.11.4.3 Special Backends

Special backends are backends that don't rely on an external server to handle the response, but instead are served by `pound` itself.

`Error` *`status`* [*`file`*] [Service directive]
: Return a particular HTTP status.

  The *status* argument supplies the HTTP status code to return.

Optional *file* argument is the name of a disk file with the error page content. If not supplied, the text is determined as usual: first the `ErrorFile` ***`status`*** statement from the enclosing listener is consulted. If it is not present, the default error page is used.

This directive is useful in a catch-all service, which outputs an error page if no service matching the incoming request was found. See Section 4.7 [Error responses], page 14, for a discussion.

`Redirect` [*`code`*] "*`url`*" [Service directive]

Declares a special backend that responds to each request with a redirect response.

Optional *code* can be one of: 301, 302 (the default), 303, 307, or 308.

The *url* argument specifies the URL to redirect the request to. Before use it is expanded as described in Section 9.3 [String Expansions], page 28.

For compatibility with previous `pound` versions, if no '`$`*`n`*' references are found in *url*, the following logic is used: if it is a "pure" host (i.e. with no path) then the client will be redirected to that host, with the original request path appended. If the *url* does contain a path (even if it is just a '`/`'), then the request path is ignored.

See Section 4.6 [Redirects], page 13, for a detailed discussion of this backend and its use.

`Metrics` [Service directive]

This directive defines a special backend that generates Openmetric telemetry output on the given URL. Example usage:

```
Service
    URL "/metrics"
    Metrics
End
```

To control access to the telemetry endpoint, use the `ACL` statement (see Section 4.1.2 [ACL], page 9).

The `LogSuppress` directive (see [LogSuppress], page 51) is often used in openmetric services to suppress logging of served HTTP requests:

```
Service
    URL "/metrics"
    Metrics
    ACL "secure"
    LogSuppress success
End
```

The metrics output is discussed in Appendix A [Metric Families], page 67.

`Emergency` ... *End* [Service directive]

Defines an *emergency backend*, which will be used only if all other backends become unavailable. The following directives are available for use within the `Emergency` section: `Address`, `Port`, `TimeOut`, `WSTimeOut`, `ConnTO`, `HTTPS`, `Cert`, `Ciphers`, `Disable`, `ServerName`, These are discussed in Section 9.11.4.1 [Backend], page 52.

9.11.5 Session

`Session` ... *End* [Service directive]

Defines how a service deals with possible HTTP sessions. Once a session is identified, `pound` will attempt to send all requests within that session to the same backend server.

See Section 6.1 [Sessions], page 19, for a detailed discussion of HTTP sessions and their handling.

The following directives are available for use in `Session` section.

`Type` *`type`* [Session directive]

Defines the expected type of sessions to handle. Allowed values for *type* are:

`IP`	A session is defined by the source IP. All requests coming from the same IP are considered to be in the same session. The IP address is defined by the `ID` statement (see below).
`BASIC`	A session is defined by the `Authentication` HTTP header. If the header is present, and specifies the '`Basic`' authentication type, user ID is extracted from it.
`URL`	A session is identified by the value of a particular query parameter. The name of the parameter is given by the `ID` statement.
`PARM`	Sessions are identified by HTTP parameter - a string that appears after a semicolon in the URL, such as '`bar`' in '`http://foo.com;bar`'.
`COOKIE`	Sessions are identified by the value of an HTTP cookie, whose name is given by the `ID` directive.
`HEADER`	Sessions are identified by the value of HTTP header whose name is given by the `ID` directive.

`ID "`*`name`*`"` [Session directive]

Specifies the *session identifier*: IP address (for `Type IP`), query parameter name (for `Type URL`), cookie name (for `Type COOKIE`), or header name (for `Type HEADER`).

`TTL` *`n`* [Session directive]

How long can a session be idle (in seconds). A session that has been idle for longer than the specified number of seconds will be discarded. This directive is mandatory.

9.11.6 Other Statements

`Disabled` *`bool`* [Service directive]

If `true`, mark this service as disabled. Disabled services are not used for request processing. A service can be enabled or disabled at runtime using the `poundctl` utility (see Section 10.1 [poundctl commands], page 58).

`Balancer` *`algo`* [Service directive]

Sets the request balancing algorithm to use. Allowed values for *algo* are:

random	Use weighted random balancing algorithm.
iwrr	Use interleaved weighted round robin balancing.

See Chapter 6 [Balancer], page 18, for a detailed discussion of these algorithms.

This statement overrides the global `Balancer` statement (see Section 9.4 [Global directives], page 29).

`IgnoreCase` *`bool`* [Service directive]

Ignore case when doing regex matching (default: '`false`'). This directive sets the default for the following service matching directives: `URL`, `Path`, `QueryParam`, `Query`, `StringMatch`, as well as for the `DeleteHeader` modification directive.

This statement is deprecated and will be removed in future versions. Please, use the `-icase` option to the matching directive instead (see Table 9.2).

10 poundctl

The `poundctl` command displays status of various objects of the running instance and allows you to change some of them.

The program communicates with the running `pound` daemon via a UNIX socket. For this to work, `pound` configuration file must contain a `Control` statement (see [Control statement], page 34). When started, `poundctl` opens the default `pound.cfg` file, looks up for this statement and then uses the pathname defined in it as the control socket file.

This behavior can be altered in two ways. First, if the configuration file is in a non-standard location, the pathname of this file can be given to the program using the `-f` command line option. Secondly, the socket name can be supplied in the command line explicitly, using the `-s` option.

The program invocation syntax is:

```
poundctl [options] command object [arg]
```

Here, *options* are command line options, *command* is a command verb that instructs `poundctl` what to do, *object* identifies the `pound` object to operate upon (see [objects], page 2), and optional *arg* supplies argument to the command verb.

Pound objects identifiers are formed in a path-like fashion:

```
/listener/service/backend
```

where:

listener Symbolic name of the listener or its ordinal number in the configuration. If referring to a globally-defined service, or to a backend in such a service, a dash is used.

service Symbolic name or ordinal number of the service located in that listener.

backend Ordinal number of backend in the service.

Depending on the command, either '*/backend*' or both '*/service/backend*' may be omitted.

For example, the following command will disable backend 2 in service 1 of listener 0:

```
poundctl disable /0/1/2
```

Assuming listener 0 is named '`web`', this example can also be written as:

```
poundctl disable /web/1/2
```

The following command disables the listener 0 itself:

```
poundctl disable /0
```

A dash in place of *listener* refers to the global scope. Thus, the following disables service 1 defined in the global scope of `pound.cfg`:

```
poundctl disable /-/1
```

10.1 `poundctl` commands

`list` */L/S/B* [poundctl]
`list` */L/S* [poundctl]
`list` */L* [poundctl]
`list` [poundctl]

Lists status of the given object and its subordinates. Without argument, shows all listeners and underlying objects.

`enable` */L/S/B* [poundctl]
`enable` */L/S* [poundctl]
`enable` */L* [poundctl]
`on` */L/S/B* [poundctl]
`on` */L/S* [poundctl]
`on` */L* [poundctl]

Enables listener, service, or backend.

`disable` */L/S/B* [poundctl]
`disable` */L/S* [poundctl]
`disable` */L* [poundctl]
`off` */L/S/B* [poundctl]
`off` */L/S* [poundctl]
`off` */L* [poundctl]

Disables listener, service, or backend.

`delete` */L/S key* [poundctl]

Delete the session with the given key. Notice that backend may not be specified.

`add` */L/S/B key* [poundctl]

Add a session with the given *key*.

10.2 `poundctl` options

The following options are understood by `poundctl`:

`-f` *file*	Read `pound` configuration from *file*, instead of the default configuration file.
`-i` *n*	Sets indentation level for JSON output to *n* columns.
`-j`	Use JSON output format.
`-h`	Shows a short help output and exits.
`-s` *socket*	Sets pathname of the control socket.
`-T` *file*	Sets the name of the template file to use.
`-t` *name*	Defines the name of the template to use, instead of the '`default`'.
`-V`	Prints program version, compilation settings, and exits.
`-v`	Increases output verbosity level.

10.3 poundctl template

Information received from the pound daemon is formatted as a JSON object. To produce human-readable output, poundctl uses a *template*, i.e. a text written in a domain-specific language expressly designed for that purpose. The template language complies, in general, with the specification in https://pkg.go.dev/text/template. See Section 10.3.1 [Template syntax], page 59, for a detailed description.

Templates are stored in template files, which are looked up in the template search path. The path is a column-delimited list of directories or file names. To locate the template file, the path is scanned left-to right. If an element is a regular file name (or a hard or symbolic link to a regular file), poundctl tries to open that file. If an element is a directory name, the program tries to open the file poundctl.tmpl in that directory. If opening succeeds, further scanning stops and templates are read from that file.

The default template path is

```
~/.poundctl.tmpl:datadir/pound
```

where *datadir* stands for the program data directory[1]. That is, the file .poundctl.tmpl in the user home directory is searched first, then the file poundctl.tmpl (without the leading dot) is looked up in the program data directory.

The default search path can be changed by setting the environment variable POUND_TMPL_PATH.

To examine the default value of the search path, use the -V command line option.

The template file to use can be requested from the command line using the -t option. In this case, template search path in not searched and the supplied file is used verbatim.

Unless instructed otherwise, poundctl uses the template 'default'. You can request another template name using the -T command line option.

The default poundctl.tmpl file defines two templates: 'default' and 'xml'.

10.3.1 Template syntax

The syntax of poundctl templates is modelled after and mostly conforming to the specifications of the golang template module[2].

Templates are executed by applying them to a JSON object. Annotations in a template refer to attributes of the object to control execution and derive values to be displayed. Execution of the template walks the structure and sets the cursor, represented by a period (called *dot*), to the value at the current location in the object as execution proceeds.

The input text for a template is as ASCII text is arbitrary format.

Actions (data evaluations or control structures) are delimited by '{{' and '}}'; all text outside actions is copied to the output verbatim.

To aid in formatting template source code, if '{{' is followed immediately by a minus sign and white space, all trailing white space is trimmed from the immediately preceding text. Similarly, if '}}' is preceded by white space and a minus sign, all leading white space is trimmed from the immediately following text. Notice that the presence of the whitespace in these trim markers is mandatory: '{{- 3}}' trims the immediately preceding text and outputs '3', while "'{{-3}}' parses as an action containing the number '-3'.

[1] It is determined at compile time. Normally it is /usr/share/pound or /usr/local/share/pound.

[2] https://pkg.go.dev/text/template

10.3.1.1 Actions

Here is the list of actions. *Arguments* and *pipelines* are evaluations of data, defined in detail in the sections that follow.

`{{ }}` Empty action is discarded. It may be useful to trim the preceding or following whitespace, as in

```
{{- -}}
```

`{{/* a comment */}}`
Comments are discarded. They may span multiple lines of text. Comments do not nest and must start immediately after the opening delimiter (with optional dash and whitespace in between). A comment may be followed by any action described below.

Comments may be used to control trailing and leading whitespace as well:

```
{{- a comment trimming the surrounding whitespace -}}
```

`{{` *`pipeline`* `}}`
The *pipeline* is evaluated, and the default textual representation of its value is copied to the output.

`{{if` *`pipeline`* `}}` *`T1`* `{{end}}`
If the value of the *pipeline* is empty, no output is generated; otherwise, *T1* is executed. The empty values are `null`, `false`, numeric 0, empty string ('`""`'), array ('`[]`'), or object ('`{}`'). Dot is unaffected.

`{{if` *`pipeline`* `}}` *`T1`* `{{else}}` *`T0`* `{{end}}`
If the value of the pipeline is empty, *T0* is executed; otherwise, *T1* is executed. Dot is unaffected.

`{{if` *`pipeline`* `}}` *`T1`* `{{else if` *`pipeline`* `}}` *`T2`* `{{else}}` *`T0`* `{{end}}`
A shortcut to simplify writing the if-else chains. Equivalent to (newlines added for readability):

```
{{if pipeline }}
  T1
{{else -}}
 {{if pipeline }}
   T2
 {{else}}
   T0
 {{end}}
{{end}}
```

`{{range` *`pipeline`* `}}` *`T1`* `{{end}}`
The value of *pipeline* must be an object or array. If it is of length zero, nothing is output. Otherwise, dot is set to the successive elements of the array or object and *T1* is executed. For objects, the elements will be visited in sorted key order.

`{{range` *`pipeline`* `}}` *`T1`* `{{else}}` *`T0`* `{{end}}`
Same as above, except that if the value of the *pipeline* is of length zero, *T0* is executed with dot unaffected.

Within the `{{range}}` action, the following two keywords may appear:

`{{break}}`
: The innermost '`{{range` *`pipeline`*`}}`' loop is ended early, stopping the current iteration and bypassing all remaining iterations.

`{{continue}}`
: The current iteration of the innermost '`{{range` *`pipeline`*`}}`' loop is stopped, and the loop starts the next iteration.

`{{define "`*`name`*`"}}` *`text`* `{{end}}`
: The *text* is collected and stored for the further use as template with the given *name*. It can be invoked using the '`{{template}}`' action (see below).

`{{template "`*`name`*`"}}`
: The template with the specified *name* (see the '`{{define}}`' above) is executed with dot set to `null`.

`{{template "`*`name`*`"` *`value`* `}}`
: The template with the specified *name* (see the '`{{define}}`' above) is executed with dot set to *value*.

`{{block "`*`name`*`"` *`pipeline`* `}}` *`T1`* `{{end}}`
: A block is shorthand for defining a template and then executing it in place:

```
{{define "name"}} T1 {{end}}
{{template "name" pipeline}}
```

`{{with` *`pipeline`* `}}` *`T1`* `{{end}}`
: If the value of the *pipeline* is empty, no output is generated; otherwise, dot is set to the value of the *pipeline* and *T1* is executed.

`{{with` *`pipeline`* `}}` *`T1`* `{{else}}` *`T0`* `{{end}}`
: Same as above, but if the value of the *pipeline* is empty, *T0* is executed with dot unchanged.

10.3.1.2 Arguments

An *argument* is a simple value, i.e. any of the following:

- Numeric value (integer or floating point)
- Boolean value: `true` or `false`.
- Quoted string.
- A dot ('`.`') This represents the cursor value.
- Attribute: '`.`*`attr`*' This is the value of the attribute *attr* in the current value (dot). Attribute references can be nested, as in '`.Attr.Xattr.Yattr`'.
- A variable reference: '`$`*`var`*'. Here, *var* is the name of the variable defined in the `range` action. See Section 10.3.3 [Variables], page 63, below.
- Function call in parentheses, for grouping.

10.3.2 Pipelines

A *pipeline* is a series of one or more *commands* delimited by pipe sign (‘`|`’). Each *command* is either an argument or a *function call*, in form:

```
func arg1 arg2...
```

where *func* is the name of one of the built-in functions discussed below.

Pipelines are executed from left to right, with the result of the previous command implicitly added to the list of arguments of each subsequent command. For example, the pipeline

```
.attr | eq $x
```

is equivalent to

```
eq $x .attr
```

i.e. it calls the built-in function `eq` with two arguments: the value of the variable ‘`x`’ and attribute ‘`attr`’ of the cursor value.

The following built-in functions are defined:

`and` *A1 A2* [Template built-in]

Evaluates to `true` if pipelines *A1* and *A2* both evaluate to `true`. Notice, that there is no boolean shortcut evaluation: both pipelines are evaluated prior to calling `and`.

`or` *A1 A2* [Template built-in]

Evaluates to `true` if at least one of the pipelines *A1* and *A2* evaluates to `true`. Notice, that there is no boolean shortcut evaluation: both pipelines are evaluated prior to calling `or`.

`index` *A1 A2...* [Template built-in]

Returns the result of indexing its first argument by the following arguments. Thus, if ‘`.`’ is an array, then:

```
index . 5
```

evaluates to its fifth element (‘`.[5]`’).

`len` *A1* [Template built-in]

Returns the integer length of its argument.

`not` *A1* [Template built-in]

Returns `true` if its argument evaluates to `false`.

`eq` *A1 A2* [Template built-in]

Returns `true` if both its arguments are equal. This applies only if both *A1* and *A2* are numeric or if they both are strings.

`ne` *A1 A2* [Template built-in]

Returns `true` if its arguments (both must be numeric or strings) are not equal.

`lt` *A1 A2* [Template built-in]

Returns `true` if *A1* is numerically less than *A2*.

`le` *A1 A2* [Template built-in]

Returns `true` if *A1* is numerically less than or equal to *A2*.

`gt` *A1 A2* [Template built-in]

Returns `true` if *A1* is numerically greater than *A2*.

`ge` *A1 A2* [Template built-in]

Returns `true` if *A1* is numerically greater than or equal to *A2*.

`even` *A1* [Template built-in]

Returns `true` if *A1*, which must evaluate to an integer value, is divisible by 2.

`printf` *FMT A1...* [Template built-in]

Implements the `printf` function. *FMT* must evaluate to string. Rest of arguments is interpreted according to the conversion specifications in *FMT*. The result is a formatted string.

In addition to the standard conversion specifications, the '`%v`' specifier is implemented: it formats its argument in the best way, depending on its actual type.

`typeof` *A1* [Template built-in]

Evaluates to the type of its argument, one of: `null`, `bool`, `number`, `integer`, `string`, `array`, and `object`.

`exists` *A1 A2* [Template built-in]

A1 must evaluate to an object and *A2* to string. The function evaluates to `true` if the attribute *A2* is present in *A1*.

`add` *A1 A2...* [Template built-in]

Returns the sum of its arguments.

`sub` *A1 A2* [Template built-in]

Returns the difference `A1 - A2`.

`mul` *A1 A2* [Template built-in]

Multiplies *A1* by *A2*.

`div` *A1 A2* [Template built-in]

Divides *A1* by *A2*.

10.3.3 Variables

Variables (referred to as `$name`) can be defined in `range` and `with` actions. For `range`, the syntax is:

```
{{range $index, $element = pipeline }} T1 {{end}}
```

where *index* and *element* are arbitrary variable names. When executing this action, during each iteration *$index* and *$element* are set to the index (attribute name) and value of each successive element. Dot remains unaffected.

For `with`, the syntax is:

```
{{with $var = pipeline }} T1 {{end}}
```

Pipeline is evaluated, its result is assigned to *$var* and the *T1* block is executed with dot unchanged.

A variable's scope extends to the `end` action of the control structure (`with` or `range`) in which it is declared. This includes any nested statements that may appear in between.

10.3.4 Input object

Depending on the request issued by `poundctl`, the invoked template can receive as its argument (*dot*) an object of the following types: *full listing*, *listener*, *service*, or *backend*.

Since there is no explicit indication of the object type being passed, templates normally use heuristics based on the presence or absence of certain attribute to deduce the object type in question. The recommended approach is described in the following pseudo-code fragment:

```
{{if exists . "listeners" }}
  {{/* This is a full listing, as requested by poundctl list. */}}
  ...
{{else if exists . "services"}}
  {{/* Single listener, as requested by poundctl list /L.
       Notice that this attribute is present in the full listing as
       well, so presence of "listeners" should be checked first. */}}
  ...
{{else if exists . "backends"}}
  {{/* Single service, as requested by poundctl list /L/S. */}}
  ...
{{else}}
  {{/* Backend listing (poundctl list /L/I/B) */}}
  ...
{{end}}
```

Structures of each object are discussed in subsections that follow.

10.3.4.1 Full listing

A full listing contains the following attributes:

`listeners`
: An array of *listener* objects. See below for a description.

`services`
: An array of *service* objects, representing services defined in the global scope of the `pound` configuration file.

`pid`
: PID of the running `pound` daemon.

`version`
: `Pound` version number (string).

`workers`
: Workers statistics. This is a JSON object with the following attributes:

`active`	Number of active threads.
`count`	Number of threads currently running.
`max`	Maximum number of threads.
`min`	Minimum number of threads.
`timeout`	Thread idle timeout.

`queue_len`
: Number of incoming HTTP requests in the queue (integer).

`timestamp`
: Current time on the server, formatted as ISO 8601 date-time with microsecond precision, e.g.: '`2023-01-05T22:43:18.071559`'.

10.3.4.2 Listener

A *listener* object represents a single HTTP or HTTPS listener in `pound` configuration. It has the following attributes:

`address` Address of this listener. A string formatted as '*`ip:port`*'. for IPv4 and IPv6 addresses or containing a socket file name, for UNIX sockets.

`protocol` Protocol used: either '`http`' or '`https`'.

`services` Array of *service* objects representing services defined in this listener. See below for the definition of a *service* object.

`enabled` Boolean. Whether this listener is enabled or not.

`nohttps11`
: Value of the `NoHTTPS11` configuration statement for this listener (see Section 9.10 [ListenHTTPS], page 46). One of: 0, 1, 2.

10.3.4.3 Service

A *service* object describes a single service.

`name` Symbolic name of this service.

`enabled` Boolean. Whether this service is enabled or not.

`tot_pri` Sum of priority values of active backends in this service.

`abs_pri` Sum of priority values of all defined backends in this service.

`session_type`
: Name of the session handling algorithm for this service. One of: '`IP`', '`BASIC`', '`URL`', '`PARM`', '`COOKIE`', '`HEADER`'.

`sessions` List of active sessions in this service. Each session is represented as object with the following attributes:

  `key` Session key (string).

  `backend` Ordinal number of the backend assigned to handle requests with this session.

  `expire` Expiration time of this session, formatted as '`1970-01-01T00:00:00.000000`' (with microsecond precision).

`backends` List of *backends* defined for this service.

`emergency`
: Emergency *backend* object, or `null` if no such backend is defined.

10.3.4.4 Backend

The following attributes are always present in each *backend* object:

`alive`	Whether or not this backend is alive.
`conn_to`	Connection timeout for this backend (seconds).
`enabled`	Whether or not this backend is enabled.
`io_to`	I/O timeout for this backend (seconds).
`priority`	Priority value assigned to this backend.
`protocol`	Protocol used by this backend: either '`http`' or '`https`'.
`type`	Backend type. One of: '`acme`', '`backend`', '`control`', '`redirect`'.
`ws_to`	Websocket timeout (seconds).

Depending on the backend type, the following attributes may be present:

`acme` An object of the following structure:

- `path` Directory where ACME challenges are stored.

`backend` Object:

- `address` Backend address.

`redirect` Object:

- `url` URL to redirect to.
- `code` HTTP code for redirection responses. One of: 301, 302, 307.
- `redir_req`
  Boolean: whether to append the original request path to the resulting location.

If backend statistics is enabled (see [BackendStats], page 31), the `stats` object will be present, with the following attributes:

`request_count`	Total number of requests processed by this backend.
`request_time_avg`	Average time per request, in nanoseconds.
`request_time_stddev`	Standard deviation of the above.

Appendix A Metric Families

This appendix describes metric families returned in the output of *openmetrics* `pound` backends (see [Metrics], page 54).

`gauge pound_workers` [Metric family]

Number of pound workers (see Chapter 7 [Worker model], page 21). Indexed by types:

‘`active`’	Number of workers currently active.
‘`count`’	Number of workers running (both idle and active).
‘`min`’	Minimum number of workers as set by the `WorkerMinCount` configuration directive (see Section 9.4 [Global directives], page 29).
‘`max`’	Maximum number of workers as set by the `WorkerMaxCount` configuration directive (see Section 9.4 [Global directives], page 29).

Example:

```
pound_workers{type="active"} 2
pound_workers{type="count"} 5
pound_workers{type="max"} 128
pound_workers{type="min"} 5
```

`stateset pound_listener_enabled` [Metric family]

State of a listener: enabled/disabled. Indexed by the listener ordinal number.

```
pound_listener_enabled{listener="0"} 1
pound_listener_enabled{listener="1"} 0
pound_listener_enabled{listener="2"} 1
```

`info pound_listener_info` [Metric family]

Description of a listener. Each instance contains the following indices:

‘`listener`’	Listener ordinal number.
‘`name`’	Listener name, as set in the `ListenHTTP` or `ListenHTTPS` statement (see Section 9.9 [ListenHTTP], page 36).
‘`address`’	Listener address. For INET family, it is formatted as ‘*`IP`*:*`PORT`*’, for UNIX sockets, it is the pathname of the socket.
‘`protocol`’	Either ‘`http`’ or ‘`https`’.

The value of this metrics is always ‘`1`’.

```
pound_listener_info{listener="0",name="",address="/run/pound.sock",protocol="http
pound_listener_info{listener="1",name="plain",address="0.0.0.0:80",protocol="http
pound_listener_info{listener="2",name="tls",address="0.0.0.0:443",protocol="https
```

`info pound_service_info` [Metric family]

Description of a service. Indices:

`listener`	Listener ordinal number. This index is absent for globally defined services.

`service` Index of the service in listener (or in global configuration, for globally defined services).

`name` Service name as set in the `Service` definition (see Section 9.11 [Service], page 47).

```
pound_service_info{listener="0",service="0",name=""} 1
pound_service_info{listener="1",service="0",name=""} 1
pound_service_info{listener="1",service="1",name="redirect"} 1
pound_service_info{listener="2",service="0",name="metrics"} 1
pound_service_info{listener="2",service="1",name="web"} 1
pound_service_info{service="0",name="fallback"} 1
```

`stateset pound_service_enabled` [Metric family]

State of a particular service.

```
pound_service_enabled{listener="0",service="0"} 1
pound_service_enabled{listener="1",service="0"} 1
pound_service_enabled{listener="2",service="0"} 1
pound_service_enabled{service="0"} 1
```

`gauge pound_service_pri` [Metric family]

Service priority value. This is the sum of priorities of all backends defined in the service (see Chapter 6 [Balancer], page 18). Indexes:

`listener` Listener ordinal number. This index is absent for globally defined services.

`service` Index of the service in listener (or in global configuration, for globally defined services).

`entity` If '`total`', the metrics contains the sum of priorities of all currently active backends, if '`absolute`', the sum of priorities of all backends, both active and inactive.

```
pound_service_pri{listener="0",service="0",entity="total"} 10
pound_service_pri{listener="0",service="0",entity="absolute"} 15
pound_service_pri{service="0",entity="total"} 1
pound_service_pri{service="0",entity="absolute"} 1
```

`gauge pound_backends` [Metric family]

Number of backends per service: total, alive, enabled, and active (both alive and enabled). Indices:

`listener` Listener ordinal number. This index is absent for globally defined services.

`service` Index of the service in listener (or in global configuration, for globally defined services).

`state` Backend state: '`total`', '`alive`', '`enabled`', or '`active`'.

Example:

```
pound_backends{listener="0",service="0",state="total"} 5
pound_backends{listener="0",service="0",state="enabled"} 3
pound_backends{listener="0",service="0",state="alive"} 3
pound_backends{service="0",state="total"} 1
pound_backends{service="0",state="enabled"} 1
pound_backends{service="0",state="alive"} 1
```

`stateset pound_backend_state` [Metric family]

State of each backend. Indices:

`listener`	Listener ordinal number. This index is absent for globally defined services.
`service`	Index of the service in listener (or in global configuration, for globally defined services).
`backend`	Index of the backend in service.
`state`	'`enabled`': whether the backend is enabled or not. '`alive`': whether the backend is alive or not.

Example:

```
pound_backend_state{listener="0",service="0",backend="0",state="alive"} 1
pound_backend_state{listener="0",service="0",backend="0",state="enabled"} 1
pound_backend_state{listener="0",service="0",backend="1",state="alive"} 1
pound_backend_state{listener="0",service="0",backend="1",state="enabled"} 0
```

`gauge pound_backend_requests` [Metric family]

Number of requests processed by backend. This metrics is available only if backend statistics is enabled (see [BackendStats], page 31).

Example:

```
pound_backend_requests{listener="0",service="0",backend="0"} 40587
pound_backend_requests{listener="1",service="0",backend="0"} 13858
```

`gauge pound_backend_request_time_avg_nanoseconds` [Metric family]

Average time per request spent in backend (nanoseconds). This metrics is available only if backend statistics is enabled (see [BackendStats], page 31).

```
pound_backend_request_time_avg_nanoseconds{listener="0",service="0",backend="0"}
pound_backend_request_time_avg_nanoseconds{listener="1",service="2",backend="0"}
```

`gauge pound_backend_request_stddev_nanoseconds` [Metric family]

Standard deviation of the average time per request. This metrics is available only if backend statistics is enabled (see [BackendStats], page 31).

```
pound_backend_request_stddev_nanoseconds{listener="0",service="0",backend="0"} 0
pound_backend_request_stddev_nanoseconds{listener="1",service="2",backend="0"} 59
```

Appendix B Time and Date Formats

This appendix documents the time format specifications understood by the '`%{`*`format`*`}t`' log format conversion. Essentially, it is a reproduction of the man page for GNU `strftime` function.

Ordinary characters placed in the format string are reproduced without conversion. Conversion specifiers are introduced by a '`%`' character, and are replaced as follows:

%a	The abbreviated weekday name according to the current locale.
%A	The full weekday name according to the current locale.
%b	The abbreviated month name according to the current locale.
%B	The full month name according to the current locale.
%c	The preferred date and time representation for the current locale.
%C	The century number (year/100) as a 2-digit integer.
%d	The day of the month as a decimal number (range 01 to 31).
%D	Equivalent to '`%m/%d/%y`'.
%e	Like '`%d`', the day of the month as a decimal number, but a leading zero is replaced by a space.
%E	Modifier: use alternative format, see below (see [conversion specs], page 72).
%F	Equivalent to '`%Y-%m-%d`' (the ISO 8601 date format).
%G	The ISO 8601 year with century as a decimal number. The 4-digit year corresponding to the ISO week number (see '`%V`'). This has the same format and value as '`%y`', except that if the ISO week number belongs to the previous or next year, that year is used instead.
%g	Like '`%G`', but without century, i.e., with a 2-digit year (00-99).
%h	Equivalent to '`%b`'.

%H	The hour as a decimal number using a 24-hour clock (range 00 to 23).
%I	The hour as a decimal number using a 12-hour clock (range 01 to 12).
%j	The day of the year as a decimal number (range 001 to 366).
%k	The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are preceded by a blank. (See also '`%H`'.)
%l	The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are preceded by a blank. (See also '`%I`'.)
%m	The month as a decimal number (range 01 to 12).
%M	The minute as a decimal number (range 00 to 59).
%n	A newline character.
%O	Modifier: use alternative format, see below (see [conversion specs], page 72).
%p	Either '`AM`' or '`PM`' according to the given time value, or the corresponding strings for the current locale. Noon is treated as '`pm`' and midnight as '`am`'.
%P	Like '`%p`' but in lowercase: '`am`' or '`pm`' or a corresponding string for the current locale.
%r	The time in '`a.m.`' or '`p.m.`' notation. In the POSIX locale this is equivalent to '`%I:%M:%S %p`'.
%R	The time in 24-hour notation ('`%H:%M`'). For a version including the seconds, see '`%T`' below.
%s	The number of seconds since the Epoch, i.e., since 1970-01-01 00:00:00 UTC.
%S	The second as a decimal number (range 00 to 61).
%t	A tab character.
%T	The time in 24-hour notation ('`%H:%M:%S`').

%u	The day of the week as a decimal, range 1 to 7, Monday being 1. See also '`%w`'.
%U	The week number of the current year as a decimal number, range 00 to 53, starting with the first Sunday as the first day of week 01. See also '`%V`' and '`%W`'.
%V	The ISO 8601:1988 week number of the current year as a decimal number, range 01 to 53, where week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week. See also '`%U`' and '`%W`'.
%w	The day of the week as a decimal, range 0 to 6, Sunday being 0. See also '`%u`'.
%W	The week number of the current year as a decimal number, range 00 to 53, starting with the first Monday as the first day of week 01.
%x	The preferred date representation for the current locale without the time.
%X	The preferred time representation for the current locale without the date.
%y	The year as a decimal number without a century (range 00 to 99).
%Y	The year as a decimal number including the century.
%z	The time-zone as hour offset from GMT. Required to emit RFC822-conformant dates (using '`%a, %d %b %Y %H:%M:%S %z`')
%Z	The time zone or name or abbreviation.
%+	The date and time in *date(1)* format.
%%	A literal '`%`' character.

Some conversion specifiers can be modified by preceding them by the '`E`' or '`O`' modifier to indicate that an alternative format should be used. If the alternative format or specification does not exist for the current locale, the behaviour will be as if the unmodified conversion specification were used. The Single Unix Specification mentions '`%Ec`', '`%EC`', '`%Ex`', '`%EX`', '`%Ry`', '`%EY`', '`%Od`', '`%Oe`', '`%OH`', '`%OI`', '`%Om`', '`%OM`', '`%OS`', '`%Ou`', '`%OU`', '`%OV`', '`%Ow`', '`%OW`', '`%Oy`', where the effect of the '`O`' modifier is to use alternative numeric symbols (say, roman numerals), and that of the '`E`' modifier is to use a locale-dependent alternative representation.

Appendix C GNU Free Documentation License

ADDENDUM: How to use this License for your documents

Index