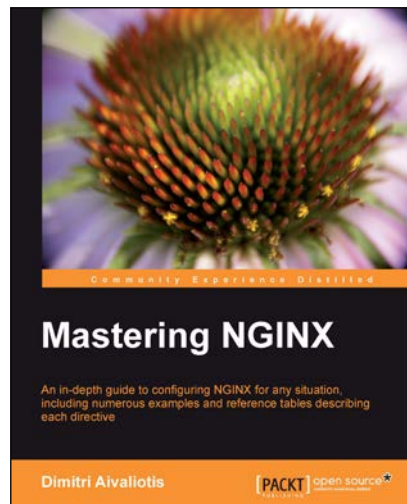


Mastering NGINX

Dimitri Aivaliotis



Chapter No. 3 "Using the Mail Module"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.3 "Using the Mail Module"

A synopsis of the book's content

Information on where to buy this book

About the Author

Dimitri Aivaliotis works as a Systems Architect at a hosting provider in Zurich, Switzerland. His career has taken him from building a Linux-based computer network for a school up through dual-datacenter high-availability infrastructures for banks and online portals. He has spent over a decade solving his customers' problems and discovered NGINX along the way. He uses the software daily to provide web serving, proxying, and media-streaming services to his customers.

Dimitri graduated summa cum laude with a BS in Physics from Rensselaer Polytechnic Institute and received an MS in Management Information Systems at Florida State University.

This is his first book.

For More Information:

www.packtpub.com/mastering-nginx-guide-for-novice-and-advanced-user/book

I would like to thank John Blackwell and Phil Margolis for reading early drafts of the manuscript. Their criticism and tips have helped me greatly and made this a better book.

I would also like to thank the technical reviewers for providing constructive feedback and pointing out errors I have made along the way. Any remaining errors are of course my own.

The team at Packt Publishing has been really supportive in getting this project off the ground. Their faith in me as a writer has bolstered me during the dark times of missed deadlines.

The knowledge and support of the NGINX, Inc. team has been instrumental in filling in the gaps in my understanding of how NGINX works. I could not have written this book without them.

An especially heartfelt thanks goes out to my family. My wife and children have had to cope with my many writing sessions. Their patience during this time is greatly appreciated.

For More Information:

www.packtpub.com/mastering-nginx-guide-for-novice-and-advanced-user/book

Mastering NGINX

NGINX is a high-performance web server designed to use very few system resources. There are many how-to's and example configurations floating around on the Web. This guide will serve to clarify the murky waters of NGINX configuration. In doing so you will learn how to tune NGINX for various situations, what some of the more obscure configuration options do, and how to design a decent configuration to match your needs.

You will no longer feel the need to copy-paste a configuration snippet because you will understand how to construct a configuration file to do exactly what you want it to do. This is a process, and there will be bumps along the way, but with the tips explained in this book you will feel comfortable writing an NGINX configuration file by hand. In case something doesn't work as expected, you will be able to debug the problem yourself or at least be capable of asking for help without feeling like you haven't given it a try yourself.

This book is written in a modular fashion. It is laid out to help you get to the information you need as quickly as possible. Each chapter is pretty much a standalone piece. Feel free to jump in anywhere you feel you need to get more in-depth about a particular topic. If you feel you have missed something major, go back and read the earlier chapters. They are constructed in a way to help you grow your configuration piece-by-piece.

What This Book Covers

Chapter 1, Installing NGINX and Third-Party Modules, teaches you how to install NGINX on your operating system of choice and how to include third-party modules in your installation.

Chapter 2, A Configuration Guide, explains the NGINX configuration file format. You will learn what each of the different contexts are for, how to configure global parameters, and what a location is used for.

Chapter 3, Using the Mail Module, explores NGINX's mail proxy module, detailing all aspects of its configuration. An example authentication service is included in the code for this chapter.

Chapter 4, NGINX as a Reverse Proxy, introduces the concept of a reverse proxy and describes how NGINX fills that role.

Chapter 5, Reverse Proxy Advanced Topics, delves deeper into using NGINX as a reverse proxy to solve scaling issues and performance problems.

Chapter 6, The NGINX HTTP Server, describes how to use the various modules included with NGINX to solve common web serving problems.

For More Information:

www.packtpub.com/mastering-nginx-guide-for-novice-and-advanced-user/book

Chapter 7, NGINX for the Developer, shows how NGINX can be integrated with your application to deliver content to your users more quickly.

Chapter 8, Troubleshooting Techniques, investigates some common configuration problems, how to debug a problem once it arises, and makes some suggestions for performance tuning.

Appendix A, Directive Reference, provides a handy reference for the configuration directives used throughout the book, as well as a selection of others not previously covered.

Appendix B, Rewrite Rule Guide, describes how to use the NGINX rewrite module and describes a few simple steps for converting Apache-style rewrite rules into ones NGINX can process.

Appendix C, Community, introduces you to the online resources available to seek more information.

Appendix D, Persisting Solaris Network Tunings, details what is necessary to persist different network tuning changes under Solaris 10 and above.

For More Information:

www.packtpub.com/mastering-nginx-guide-for-novice-and-advanced-user/book

3

Using the Mail Module

NGINX was designed to not only serve web traffic, but also to provide a means of proxying mail services. In this chapter you will learn how to configure NGINX as a mail proxy for POP3, IMAP, and SMTP services. We will examine running NGINX as a mail proxy server in the following sections:

- Basic proxy service
- Authentication service
- Combining with memcached
- Interpreting log files
- Operating system limits

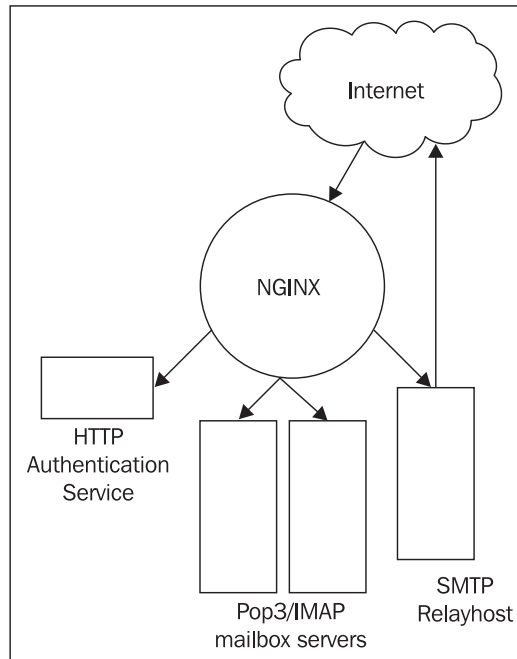
Basic proxy service

The NGINX mail proxy module was originally developed for FastMail. They had a need to provide a single IMAP endpoint for their users, while hosting the actual mail account on one of a number of upstream mail servers. Typical proxying programs of the time used the classic Unix forking model, which meant that a new process was forked for each connection. IMAP has very long-lived connections, which means that these processes would stay around for a very long time. This would then lead to very sluggish proxy servers, as they would have to manage these processes for the lifetime of each connection. NGINX's event-based process model was a better fit for this type of service. As a mail proxy, NGINX is able to direct traffic to any number of mailbox servers where the actual mail account is hosted. This provides the ability to communicate one endpoint to customers, while scaling the number of mailbox servers up with the number of users. Both commercial and open-source mail solutions, such as Atmail and Zimbra, are built around this model.

For More Information:

www.packtpub.com/mastering-nginx-guide-for-novice-and-advanced-user/book

The following diagram will help visualize how this works:



An incoming request will be handled on a per-protocol basis. The mail proxy module may be configured differently for POP3, IMAP, or SMTP. For each protocol, NGINX queries an authentication service with the username and password. If the authentication is successful, the connection is proxied to the mail server indicated in the response from the authentication service. If the authentication was unsuccessful, the client connection is terminated. The authentication service thus determines which clients can use POP3 / IMAP / SMTP services and which mail server they may use. As any number of mail servers may be handled in this way, NGINX can provide a proxy service for all of them through one central gateway.

A proxy acts on behalf of someone or something else. In this case, NGINX is acting on behalf of the mail client, terminating the connection and opening a new one to the upstream server. This means that there is no direct communication between the mail client and the actual mailbox server or SMTP relay host.



If there are any mail rules based on information contained in the client connection, these rules will not work, unless the mail software is able to support an extension, such as XCLIENT for SMTP.

This is an important point in designing an architecture that contains a proxy server — the proxy host will need to be able to support more connections than a typical upstream server. Not as much processing power or memory as a mailbox server would be needed, but the number of persistent connections needs to be taken into account.

POP3 service

The **Post Office Protocol** is an Internet standard protocol used to retrieve mail messages from a mailbox server. The current incarnation of the protocol is Version 3, thus **POP3**. Mail clients will typically retrieve all new messages on a mailbox server in one session, then close the connection. After closing, the mailbox server will delete all messages that have been marked as retrieved.

In order for NGINX to act as a POP3 proxy, some basic directives need to be configured:

```
mail {
    auth_http    localhost:9000/auth;

    server {
        listen    110;
        protocol  pop3;
        proxy     on;
    }
}
```

This configuration snippet enables the mail module and configures it for POP3 service, querying an authentication service running on port 9000 on the same machine. NGINX will listen on port 110 on all local IP addresses, providing a POP3 proxy service. You will notice that we do not configure the actual mail servers here—it is the job of the authentication service to tell NGINX which server a particular client should be connected to.

If your mail server only supports certain capabilities (or you only want to advertise certain capabilities), NGINX is flexible enough to announce these:

```
mail {
    pop3_capabilities    TOP USER;
}
```

Capabilities are a way of advertising support for optional commands. For POP3, the client can request the supported capabilities before or after authentication, so it is important to configure these correctly in NGINX.

You may also specify which authentication methods are supported:

```
mail {
    pop3_auth    apop cram-md5;
}
```

If the APOP authentication method is supported, the authentication service needs to provide NGINX with the user's password in clear text, so that it can generate the MD5 digest.

IMAP service

The **Internet Message Access Protocol** is also an Internet-standard protocol used to retrieve mail messages from a mailbox server. It provides quite a bit of extended functionality over the earlier POP protocol. Typical usage leaves all messages on the server, so that multiple mail clients can access the same mailbox. This also means that there may be many more, persistent connections to an upstream mailbox server from clients using IMAP than those using POP3.

To proxy IMAP connections, a configuration similar to the POP3 NGINX snippet used before can be used:

```
mail {
    auth_http    localhost:9000/auth;

    imap_capabilities    IMAP4rev1 UIDPLUS QUOTA;
    imap_auth    login cram-md5;

    server {
        listen 143;
        protocol imap;
        proxy on;
    }
}
```

Note that we did not need to specify the `protocol`, as `imap` is the default value. It is included here for clarity.

The `imap_capabilities` and `imap_auth` directives function similarly to their POP3 counterparts.

SMTP service

The **Simple Mail Transport Protocol** is the Internet-standard protocol for transferring mail messages from one server to another or from a client to a server. Although authentication was not at first conceived for this protocol, SMTP-AUTH is supported as an extension.

As you have seen, the logic of configuring the mail module is fairly straightforward. This holds for SMTP proxying as well:

```
mail {
    auth_http    localhost:9000/auth;

    smtp_capabilities    PIPELINING 8BITMIME DSN;
    smtp_auth    login cram-md5;

    server {
        listen 25;
        protocol smtp;
        proxy on;
    }
}
```

Our proxy server will only advertise the `smtp_capabilities` that we set, otherwise it will only list which authentication mechanisms it accepts, because the list of extensions is sent to the client when it sends the `HELO/EHLO` command. This may be useful when proxying to multiple SMTP servers, each having different capabilities. You could configure NGINX to list only the capabilities that all of these servers have in common. It is important to set these to only the extensions that the SMTP server itself supports.

Due to SMTP-AUTH being an extension to SMTP, and not necessarily supported in every configuration, NGINX is capable of proxying an SMTP connection that does no authentication whatsoever. In this case, only the `HELO`, `MAIL FROM`, and `RCPT TO` parts of the protocol are available to the authentication service for determining which upstream should be chosen for a given client connection. For this setup, ensure that the `smtp_auth` directive is set to `none`.

Using SSL/TLS

If your organization requires mail traffic to be encrypted, or if you yourself want more security in your mail transfers, you can enable NGINX to use TLS to provide POP3 over SSL, IMAP over SSL, or SMTP over SSL. To enable TLS support, either set the `starttls` directive to `on` for STLS/STARTTLS support or set the `ssl` directive to `on` for pure SSL/TLS support and configure the appropriate `ssl_*` directives for your site:

```
mail {
    # allow STLS for POP3 and STARTTLS for IMAP and SMTP
    starttls      on;
    # prefer the server's list of ciphers, so that we may determine
    # security
    ssl_prefer_server_ciphers  on;
    # use only these protocols
    ssl_protocols      TLSv1 SSLv3;
    # use only high encryption cipher suites, excluding those
    # using anonymous DH and MD5, sorted by strength
    ssl_ciphers        HIGH:!ADH:!MD5:@STRENGTH;
    # use a shared SSL session cache, so that all workers can
    # use the same cache
    ssl_session_cache  shared:MAIL:10m;
    # certificate and key for this host
    ssl_certificate     /usr/local/etc/nginx/mail.example.com.crt;
    ssl_certificate_key /usr/local/etc/nginx/mail.example.com.key;
}
```

See https://www.fastmail.fm/help/technology_ssl_vs_tls_starttls.html for a description of the differences between a pure SSL/TLS connection and upgrading a plain connection to an encrypted one with SSL/TLS.



Using OpenSSL to generate an SSL certificate

If you have never generated an SSL certificate before, the following steps will help you create one:

Create a certificate request:

```
$ openssl req -newkey rsa:2048 -nodes -out mail.
example.com.csr -keyout mail.example.com.key
```

This should generate the following output:

```
Generating a 2048 bit RSA private key
.....
.....
....+++
.....+++
writing new private key to 'mail.example.com.key'
-----
You are about to be asked to enter information that will
be incorporated
into your certificate request.
What you are about to enter is what is called a
Distinguished Name or a DN.
There are quite a few fields but you can leave some
blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:CH
State or Province Name (full name) [Some-State]:Zurich
Locality Name (eg, city) []:ZH
Organization Name (eg, company) [Internet Widgits Pty
Ltd]:Example Company
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:mail.
example.com
Email Address []:
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
You can get this Certificate Signing Request (mail.example.com.csr)
signed by a Certificate Authority such as Verisign or GoDaddy, or you
can sign it yourself:
$ openssl x509 -req -days 365 -in mail.example.com.csr
-signkey mail.example.com.key -out mail.example.com.crt
You will see the following response:
Signature ok
subject=/C=CH/ST=Zurich/L=ZH/O=Example Company/CN=mail.
example.com
Getting Private key
```



The signed certificate is shown in the following screenshot.

Please note, though, that a self-signed certificate will generate an error in a client that connects to your server. If you are deploying this certificate on a production server, make sure that you get it signed by a recognized authority.



```
-----BEGIN CERTIFICATE-----
MIIDPDCCAiQCCQDdPKFcY1X35jANBgkqhkiG9w0BAQUFADBGMQswCQYDVQQGEwJD
SDEPMA0GA1UECAwGbnVyaWNoMQswCQYDVQQHDAJaSDEYMBYGA1UECgwPRXhhbXBs
ZSBDb2lwYW55MRkwFwYDVQQDDBBtYWlsLmV4YW1wbGUuY29tMB4XDTEyMDgzMTE0
MjczMloXDTExMDgzMTE0MjczMloYDELMAkGA1UEBhMCQ0gxDzANBgNVBAGtMBlp1
cm1jaDELMAkGA1UEBwwCQkxGDAWBgNVBAoMD0V4YW1wbGUuY29tGFEuTEZMBcG
A1UEAwwQbWpbc5leGFtcGxlLmV4YW1wbGUuY29tGFEuTEZMBcGADggEPADCC
AQoCggEBAN8WUGzQIKR+iuTxlPko/zSR+DbjDYqbMo4PdNvEN46nTFMkktv0sIk
1kf9L2jzVcmUUSZayLp3woDgxRpkpQ5eRpB7yeifszWpJLxfVPTgfXtQkktfPVn
uzOMf70gd2Xt8uI6n0At0DAr8+CxebIprWiwZBXPrWwFFjQvy4/qD7EXs33+xS8
9CMxkGo2FPqCSYE39jN3JtI29YibnZh01NALHRvnqyw3mdzR340mu5WNFjl/NElp
MOyFL7+5wzI4ktgmAo+Mic6JnXC0bSjrL1xZjWfn/STQiYQVzUit4jd1CswtCHw
tv67TRQ3edgvssvzfZlm7QfBbdYGjkUCAwEAATANBgkqhkiG9w0BAQUFAA0CAQEA
TdfdngrMk2w/1KCGbxrg9bVmfKXUSIfpwyt0hG02EtLx83TzajqwtOKhmPh9Q/lc
GZdF1PGscdJ2Bc0eJBUGyt6mevEi2Dg4h727yVvnacnViQvzyLxQgmeCsrDEj4EC
yDzzi4nOI/rddjPeQ0+cMFHz26scsKY0Remzp0yHT8JhK8AF2iOi0LzwaMqxClL
U7lkinHdTAg6nT4WpH05HtSBno8Xco/ujY6xIrShiP0naOd/B4TRCmB96KYhyMdd
Ayr0ZgLqsskKeAlnmUSJA/7zbp1LwHarvUVFpzKed73554lfJ5kpy0ciHrIfyj/2
dM/tjsDVjpe2B/meYBx8Kg==
-----END CERTIFICATE-----
```

Complete mail example

Mail services are often combined on one gateway. The following configuration will enable NGINX to service POP3, IMAP, and SMTP traffic (as well as their encrypted variants) from one authentication service, while offering clients the option to use STLS/STARTTLS on unencrypted ports:

```
events {
    worker_connections 1024;
}

mail {
    server_name mail.example.com;
    auth_http localhost:9000/auth;

    proxy on;

    ssl_prefer_server_ciphers on;
    ssl_protocols TLSv1 SSLv3;
    ssl_ciphers HIGH:!ADH:!MD5:@STRENGTH;
```

For More Information:

www.packtpub.com/mastering-nginx-guide-for-novice-and-advanced-user/book

```
ssl_session_cache    shared:MAIL:10m;
ssl_certificate       /usr/local/etc/nginx/mail.example.com.crt;
ssl_certificate_key   /usr/local/etc/nginx/mail.example.com.key;

pop3_capabilities    TOP USER;
imap_capabilities    IMAP4rev1 UIDPLUS QUOTA;
smtp_capabilities    PIPELINING 8BITMIME DSN;

pop3_auth            apop cram-md5;
imap_auth            login cram-md5;
smtp_auth            login cram-md5;

server {
    listen 25;
    protocol smtp;
    timeout 120000;
}
server {
    listen 465;
    protocol smtp;
    ssl on;
}
server {
    listen 587;
    protocol smtp;
    starttls on;
}
server {
    listen 110;
    protocol pop3;
    starttls on;
}
server {
    listen 995;
    protocol pop3;
    ssl on;
}
server {
    listen 143;
    protocol imap;
    starttls on;
}
server {
    listen 993;
    protocol imap;
    ssl on;
}
}
```

As you can see, we declared the name of this server at the top of the `mail` context. This is because we want each of our mail services to be addressed as `mail.example.com`. Even if the actual hostname of the machine on which NGINX runs is different, and each mail server has its own hostname, we want this proxy to be a single point of reference for our users. This hostname will in turn be used wherever NGINX needs to present its own name, for example, in the initial SMTP server greeting.

The `timeout` directive was used in the `smtp` server context in order to double its default value because we knew this particular upstream SMTP relay host inserted an artificial delay in order to dissuade spammers from trying to send mail via this server.

Authentication service

We have mentioned the authentication service quite a few times in the previous section, but what exactly is the authentication service and what does it do? When a user makes a POP3, IMAP, or SMTP request to NGINX, authenticating the connection is one of the first steps. NGINX does not perform this authentication itself, but rather makes a query to an authentication service that will fulfill the request. NGINX then uses the response from the authentication service to make the connection to the upstream mail server.


This authentication service may be written in any language. It need only conform to the authentication protocol required by NGINX. The protocol is similar to HTTP, so it will be fairly easy for us to write our own authentication service.

NGINX will send the following headers in its request to the authentication service:

- Host
- Auth-Method
- Auth-User
- Auth-Pass
- Auth-Salt
- Auth-Protocol
- Auth-Login-Attempt
- Client-IP
- Client-Host
- Auth-SMTP-Helo
- Auth-SMTP-From
- Auth-SMTP-To

The meaning of each of these headers should be fairly self-explanatory, and not each header will be present in every request. We will go over these as we write our authentication service.

We choose Ruby as the language for this authentication service implementation. If you do not currently have Ruby installed, don't worry about doing so now. Ruby as a language is very clear to read, so just try to follow along with the commented code below. Adapting it to your environment and running it is outside the scope of this book. This example will give you a good starting point in writing your own authentication service.

 A good resource to help you get Ruby installed easily is located at <https://rvm.io>.

Let us first examine the request part of the HTTP request/response dialogue.

We first collect the values we need from the headers NGINX sends:

```
# the authentication mechanism
meth = @env['HTTP_AUTH_METHOD']
# the username (login)
user = @env['HTTP_AUTH_USER']
# the password, either in the clear or encrypted,
# depending on the
# authentication mechanism used
pass = @env['HTTP_AUTH_PASS']
# need the salt to encrypt the cleartext password, used for some
# authentication mechanisms, not in our example
salt = @env['HTTP_AUTH_SALT']
# this is the protocol being proxied
proto = @env['HTTP_AUTH_PROTOCOL']
# the number of attempts needs to be an integer
attempt = @env['HTTP_AUTH_LOGIN_ATTEMPT'].to_i
# not used in our implementation, but these are
# here for reference
client = @env['HTTP_CLIENT_IP']
host = @env['HTTP_CLIENT_HOST']
```

What are all these @'s about?



The @ symbol is used in Ruby to denote a class variable. We'll use them in our example to make it easier to pass around variables. In the preceding snippet, we are referencing the environment (@env) as passed into the Rack request. Besides all the HTTP headers that we need, the environment contains additional information relating to how the service is being run.

Now that we know how to handle each of the headers NGINX may send, we need to do something with them and send NGINX a response. The following headers are expected in the response from the authentication service:

- **Auth-Status:** In this header, anything but OK is an error
- **Auth-Server:** This is the IP address to which the connection is proxied
- **Auth-Port:** This is the port to which the connection is proxied
- **Auth-User:** This is the user that will be used to authenticate with the mail server
- **Auth-Pass:** The plaintext password used for APOP
- **Auth-Wait:** How many seconds to wait before another authentication attempt is made
- **Auth-Error-Code:** An alternative error code to return to the client

The three headers used most often are **Auth-Status**, **Auth-Server**, and **Auth-Port**. The presence of these in a response is typically all that is needed for a successful authentication session.

As we will see in the following snippet, additional headers may be used, depending on the situation. The response itself consists of simply emitting the relevant headers with the appropriate values substituted in.

We first check if there have been too many tries:

```
# fail if more than the maximum login attempts are tried
if attempt > @max_attempts
    @res["Auth-Status"] = "Maximum login attempts exceeded"
    return
end
```

Then we return the appropriate headers and set with the values obtained from our authentication mechanism:

```
@res["Auth-Status"] = "OK"
@res["Auth-Server"] = @mailhost
# return the correct port for this protocol
@res["Auth-Port"] = MailAuth::Port[proto]
# if we're using APOP, we need to return the password in
cleartext
if meth == 'apop' && proto == 'pop3'
    @res["Auth-User"] = user
    @res["Auth-Pass"] = pass
end
```

If the authentication check has failed, we need to tell NGINX.

```

        # if authentication was unsuccessful, we return an appropriate
response
        @res["Auth-Status"] = "Invalid login or password"
        # and set the wait time in seconds before the client may make
        # another authentication attempt
        @res["Auth-Wait"] = "3"
        # we can also set the error code to be returned
        # to the SMTP client
        @res["Auth-Error-Code"] = "535 5.7.8"

```

Not every header is required in the response, but as we can see, some are dependent on the status of the authentication query and/or any error condition that may exist.



One interesting use of the Auth-User header is to return a different username than the one given in the request. This can prove useful, for example, when migrating from an older upstream mail server that accepted a username without the domain to a newer upstream mail server that requires the username to have a domain. NGINX will then use this username when connecting to the upstream server.

The authentication database may take any form, from a flat text file, to an LDAP directory, to a relational database. It does not have to necessarily be the same store that your mail service uses to access this information, but should be in sync with that store to prevent any errors due to stale data.

Our example authentication database is a simple hash for this example:

```
@auths = { "test:1234" => '127.0.1.1' }
```

The mechanism used to verify a user is a simple hash lookup:

```

# this simply returns the value looked-up by the 'user:pass' key
if @auths.key?("#{user}:#{pass}")
    @mailhost = @auths["#{user}:#{pass}"]
    return true
# if there is no such key, the method returns false
else
    return false
end

```

Tying these three parts together, we have the complete authentication service:

```
#!/usr/bin/env rackup

# This is a basic HTTP server, conforming to the authentication
protocol
# required by NGINX's mail module.
#
require 'logger'
require 'rack'

module MailAuth

  # setup a protocol-to-port mapping
  Port = {
    'smtp' => '25',
    'pop3' => '110',
    'imap' => '143'
  }

  class Handler

    def initialize
      # setup logging, as a mail service
      @log = Logger.new("| logger -p mail.info")
      # replacing the normal timestamp by the service name and pid
      @log.datetime_format = "nginx_mail_proxy_auth pid: "
      # the "Auth-Server" header must be an IP address
      @mailhost = '127.0.0.1'
      # set a maximum number of login attempts
      @max_attempts = 3
      # our authentication 'database' will just be a fixed hash for
      # this example
      # it should be replaced by a method to connect to LDAP or a
      # database
      @auths = { "test:1234" => '127.0.1.1' }
    end
```

After the preceding setup and module initialization, we tell Rack which requests we would like to have handled and define a `get` method to respond to requests from NGINX.

```
def call(env)
  # our headers are contained in the environment
  @env = env
```

```

    # set up the request and response objects
    @req = Rack::Request.new(env)
    @res = Rack::Response.new
    # pass control to the method named after the HTTP verb
    # with which we're called
    self.send(@req.request_method.downcase)
    # come back here to finish the response when done
    @res.finish
end

def get
  # the authentication mechanism
  meth = @env['HTTP_AUTH_METHOD']
  # the username (login)
  user = @env['HTTP_AUTH_USER']
  # the password, either in the clear or encrypted, depending on
  # the authentication mechanism used
  pass = @env['HTTP_AUTH_PASS']
  # need the salt to encrypt the cleartext password, used for some
  # authentication mechanisms, not in our example
  salt = @env['HTTP_AUTH_SALT']
  # this is the protocol being proxied
  proto = @env['HTTP_AUTH_PROTOCOL']
  # the number of attempts needs to be an integer
  attempt = @env['HTTP_AUTH_LOGIN_ATTEMPT'].to_i
  # not used in our implementation, but these are here for
  # reference
  client = @env['HTTP_CLIENT_IP']
  host = @env['HTTP_CLIENT_HOST']

  # fail if more than the maximum login attempts are tried
  if attempt > @max_attempts
    @res["Auth-Status"] = "Maximum login attempts exceeded"
    return
  end

  # for the special case where no authentication is done
  # on smtp transactions, the following is in nginx.conf:
  #   smtp_auth none;
  # may want to setup a lookup table to steer certain senders
  # to particular SMTP servers
  if meth == 'none' && proto == 'smtp'
    helo = @env['HTTP_AUTH_SMTP_HELO']

```

```
# want to get just the address from these two here
from = @env['HTTP_AUTH_SMTP_FROM'].split(/: /)[1]
to = @env['HTTP_AUTH_SMTP_TO'].split(/: /)[1]
@res["Auth-Status"] = "OK"
@res["Auth-Server"] = @mailhost
# return the correct port for this protocol
@res["Auth-Port"] = MailAuth::Port[proto]
@log.info("a mail from #{from} on #{helo} for #{to}")
# try to authenticate using the headers provided
elsif auth(user, pass)
  @res["Auth-Status"] = "OK"
  @res["Auth-Server"] = @mailhost
  # return the correct port for this protocol
  @res["Auth-Port"] = MailAuth::Port[proto]
  # if we're using APOP, we need to return the password in
  cleartext
  if meth == 'apop' && proto == 'pop3'
    @res["Auth-User"] = user
    @res["Auth-Pass"] = pass
  end
  @log.info("+ #{user} from #{client}")
# the authentication attempt has failed
else
  # if authentication was unsuccessful, we return an appropriate
  response
  @res["Auth-Status"] = "Invalid login or password"
  # and set the wait time in seconds before the client may make
  # another authentication attempt
  @res["Auth-Wait"] = "3"
  # we can also set the error code to be returned to the SMTP
  client
  @res["Auth-Error-Code"] = "535 5.7.8"
  @log.info("! #{user} from #{client}")
end

end
```

The next section is declared `private` so that only this class may use the methods declared afterwards. The `auth` method is the workhorse of the authentication service, checking the username and password for validity. The `method_missing` method is there to handle invalid methods, responding with a `Not Found` error message:

```
private

# our authentication method, adapt to fit your environment
def auth(user, pass)
  # this simply returns the value looked-up by the 'user:pass' key
  if @auths.key?("#{user}:#{pass}")
    @mailhost = @auths["#{user}:#{pass}"]
    return @mailhost
  # if there is no such key, the method returns false
  else
    return false
  end
end

# just in case some other process tries to access the service
# and sends something other than a GET
def method_missing(env)
  @res.status = 404
end

end # class MailAuthHandler
end # module MailAuth
```

This last section sets up the server to run and routes the `/auth` URI to the proper handler:

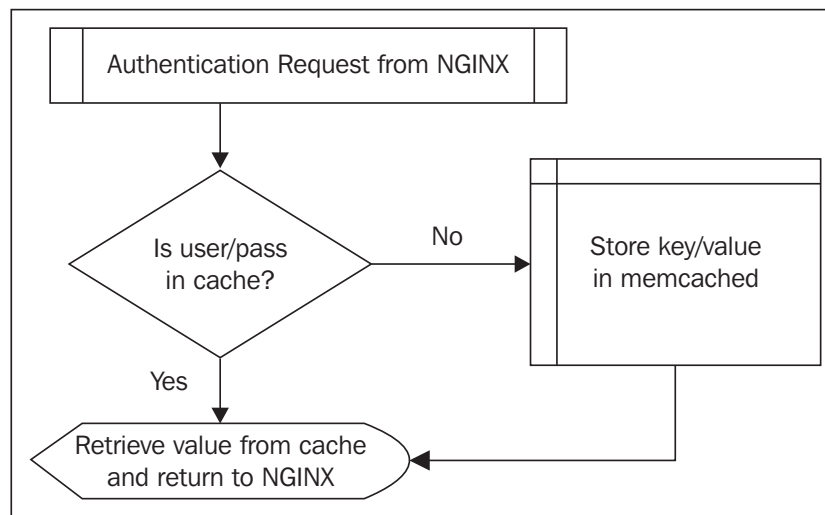
```
# setup Rack middleware
use Rack::ShowStatus
# map the /auth URI to our authentication handler
map "/auth" do
  run MailAuth::Handler.new
end
```

This listing may be saved as a file, `nginx_mail_proxy_auth.ru`, and called with a `-p <port>` parameter to tell it on which port it should run. For more options and more information about the Rack web server interface, visit <http://rack.github.com>.

Combining with memcached

Depending on the frequency of clients accessing the mail services on your proxy and how many resources are available to the authentication service, you may want to introduce a caching layer into the setup. To this end, we will integrate `memcached` as an in-memory store for authentication information.

NGINX can look up a key in `memcached`, but only in the context of a location in the `http` module. Therefore, we will have to implement our own caching layer outside of NGINX.



As the flowchart shows, we will first check whether or not this username/password combination is already in the cache. If not, we will query our authentication store for the information and place the key/value pair into the cache. If it is, we can retrieve this information directly from the cache.



Zimbra has created a memcache module for NGINX that takes care of this directly within the context of NGINX. To date, though, this code has not been integrated into the official NGINX sources.

The following code will extend our original authentication service by implementing a caching layer (admittedly, a little overkill for our implementation, but this is to provide a basis for working with a networked authentication database):

```
# gem install memcached (depends on libsassl2 and gettext libraries)
require 'memcached'

# set this to the IP address/port where you have memcached running
@cache = Memcached.new("localhost:11211")

def get_cache_value(user, pass)
  resp = ''
  begin
    # first, let's see if our key is already in the cache
    resp = @cache.get("#{user}:#{pass}")
  rescue Memcached::NotFound
    # it's not in the cache, so let's call the auth method
    resp = auth(user, pass)
    # and now store the response in the cache, keyed on 'user:pass'
    @cache.set("#{user}:#{pass}", resp)
  end
  # explicitly returning the response to the caller
  return resp
end
```

In order to use this code, you will of course have to install and run memcached. There should be a pre-built package for your operating system:

- Linux (deb-based)
`sudo apt-get install memcached`
- Linux (rpm-based)
`sudo yum install memcached`
- FreeBSD
`sudo pkg_add -r memcached`

Memcached is configured simply by passing parameters to the binary when running it. There is no configuration file that is read directly, although your operating system and/or packaging manager may provide a file that is parsed to make passing these parameters easier.

The most important parameters for `memcached` are as follows:

- `-l`: This parameter specifies the address(es) on which `memcached` will listen (default is `all`). It is important to note that for the greatest security, `memcached` shouldn't listen on an address that is reachable from the Internet because there is no authentication.
- `-m`: This parameter specifies the amount of RAM to use for the cache (in megabytes).
- `-c`: This parameter specifies the maximum number of simultaneous connections (default is 1024).
- `-p`: This parameter specifies the port on which `memcached` will listen (default is 11211).

Setting these to reasonable values will be all you need to do to get `memcached` up and running.

Now, by substituting the `elseif auth(user, pass)` with `elseif get_cache_value(user, pass)` in our `nginx_mail_proxy_auth.ru` service, you should have an authentication service running with a caching layer, to help serve as many requests as quickly as possible.

Interpreting log files

Log files provide some of the best clues as to what is going on when a system doesn't act as expected. Depending on the verbosity level configured and whether or not NGINX was compiled with debugging support (`--enable-debug`), the log files will help you understand what is going on in a particular session.

Each line in the error log corresponds to a particular log level, configured using the `error_log` directive. The different levels are `debug`, `info`, `notice`, `warn`, `error`, `crit`, `alert`, and `emerg`, in order of increasing severity. Configuring a particular level will include messages for all of the more severe levels above it. The default log level is `error`.

In the context of the `mail` module, we would typically want to configure a log level of `info`, so that we can get as much information about a particular session as possible without having to configure debug logging. Debug logging in this case would be useful only for following function entry points, or seeing what password was used for a particular connection.



Since mail is extremely dependent upon a correctly-functioning DNS, many errors can be traced back to invalid DNS entries or expired cache information. If you believe you may have a case that could be explained by a name resolution error, you can get NGINX to tell you what IP address a particular hostname is resolved to by configuring debug logging. Unfortunately, this requires a recompile if your nginx binary was not initially compiled with debugging support.

A typical proxy connection is logged as in the following example of a POP3 session.

First, the client establishes a connection to the proxy:

```
<timestamp> [info] <worker pid>#0: *<connection id> client <ip
address> connected to 0.0.0.0:110
```

Then, once the client has completed a successful login, a statement listing all relevant connection information is logged:

```
<timestamp> [info] <worker pid>#0: *<connection id> client logged
in, client: <ip address>, server: 0.0.0.0:110, login: "<username>",
upstream: <upstream ip>:<upstream port>, [<client ip>:<client port>-
<local ip>:110] <=> [<local ip>:<high port>-<upstream ip>:<upstream
port>]
```

You will notice that the section before the double arrows <=> relates to the client-to-proxy side of the connection, whereas the section after the double arrows describes the proxy-to-upstream part of the connection. This information is again repeated once the session is terminated:

```
<timestamp> [info] <worker pid>#0: *<connection id> proxied session
done, client: <ip address>, server: 0.0.0.0:110, login: "<username>",
upstream: <upstream ip>:<upstream port>, [<client ip>:<client port>-
<local ip>:110] <=> [<local ip>:<high port>-<upstream ip>:<upstream
port>]
```

In this way, we see which ports are in use on all sides of the connection, to help debug any potential problems or to perhaps correlate the log entry with what may appear in a firewall log.

Other log entries at the `info` level pertain to timeouts or invalid commands/responses sent by either the client or upstream.

Entries at the `warn` log level are typically configuration errors:

```
<timestamp> [warn] <worker pid>#0: *<connection id> "starttls"
directive conflicts with "ssl on"
```

Many errors that are reported at the error log level are indicative of problems with the authentication service. You will notice the text `while in http auth state` in the following entries. This shows where in the connection state the error has occurred:

```
<timestamp> [error] <worker pid>#0: *<connection id> auth http server
127.0.0.1:9000 timed out while in http auth state, client: <client
ip>, server: 0.0.0.0:25
<timestamp> [error] <worker pid>#0: *<connection id> auth http server
127.0.0.1:9000 sent invalid response while in http auth state, client:
<client ip>, server: 0.0.0.0:25
```

If the authentication query is not successfully answered for any reason, the connection is terminated. NGINX doesn't know to which upstream the client should be proxied, and thereby closes the connection with an `Internal server error` with the protocol-specific response code.

Depending on whether or not the username is present, the information will appear in the log file. Here's an entry from an authenticated SMTP connection:

```
<timestamp> [error] <worker pid>#0: *<connection id> auth http server
127.0.0.1:9000 did not send server or port while in http auth state,
client: <client ip>, server: 0.0.0.0:25, login: "<login>"
```

Note the previous two entries are missing in the `login` information.

An alert log level event will indicate that NGINX was not able to set a parameter as expected, but will otherwise operate normally.

Any log entry at the `emerg` level, however, will prevent NGINX from starting: either the situation has to be corrected or the configuration must be changed. If NGINX is already running, it will not restart any worker process until the change has been made:

```
<timestamp> [error] <worker pid>#0: *<connection id> no "http_auth" is
defined for server in /opt/nginx/conf/nginx.conf:32
```

Here we need to define an authentication service using the `http_auth` directive.

Operating system limits

You may run into a situation in which NGINX does not perform as you expect. Either connections are being dropped or warning messages are printed in the log file. This is when it is important to know what limits your operating system may place on NGINX and how to tune them to get the best performance out of your server.

The area in which a mail proxy is most likely to run into problems is a connection limit. To understand what this means, you first have to know how NGINX handles client connections. The NGINX master process starts a number of workers, each of which runs as a separate process. Each process is able to handle a fixed number of connections, set by the `worker_connections` directive. For each proxied connection, NGINX opens a new connection to the mail server. Each of these connections requires a file descriptor and per mail server IP/port combination, a new TCP port from the ephemeral port range (see the following explanation).

Depending on your operating system, the maximum number of open file descriptors is tunable in a resource file or by sending a signal to a resource-management daemon. You can see what the current value is set to by entering the following command at the prompt:

```
ulimit -n
```

If by your calculations, this limit is too low, or you see a message in your error log that `worker_connections` exceed open file resource limit, you'll know that you need to increase this value. First tune the maximum number of open file descriptors at the operating system level, either for just the user that NGINX runs as or globally. Then, set the `worker_rlimit_nofile` directive to the new value in the main context of the `nginx.conf` file. Sending `nginx` a configuration reload signal (HUP) will then be enough to raise this limit without restarting the main process.

If you observe a connection limit due to exhaustion of available TCP ports, you will need to increase the ephemeral port range. This is the range of TCP ports which your operating system maintains for outgoing connections. It can default to as few as 5000, but is typically set to a range of 16384 ports. A good description of how to increase this range for various operating systems is provided at http://www.ncftpd.com/ncftpd/doc/misc/ephemeral_ports.html.

Summary

In this chapter, we have seen how NGINX can be configured to proxy POP3, IMAP, and SMTP connections. Each protocol may be configured separately, announcing support for various capabilities in the upstream server. Encrypting mail traffic is possible by using TLS and providing the server with an appropriate SSL certificate.

The authentication service is fundamental to the functioning of the `mail` module, as no proxying can be done without it. We have detailed an example of such an authentication service, outlining the requirements of both what is expected in the request and how the response should be formed. With this as a foundation, you should be able to write an authentication service that fits your environment.

Understanding how to interpret log files is one of the most useful skills a system administrator can develop. NGINX gives fairly detailed log entries, although some may be a bit cryptic. Knowing where to place the various entries within the context of a single connection and seeing the state NGINX is in at that time is helpful to deciphering the entry.

NGINX, like any other piece of software, runs within the context of an operating system. It is therefore extremely useful to know how to increase any limits the OS may place on NGINX. If it is not possible to increase the limits any further, then an architectural solution must be found by either multiplying the number of servers on which NGINX runs, or using some other technique to reduce the number of connections a single instance must handle.

In the next chapter, we see how to configure NGINX to proxy HTTP connections.

Where to buy this book

You can buy Mastering NGINX from the Packt Publishing website:

<http://www.packtpub.com/mastering-nginx-guide-for-novice-and-advanced-user/book>.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our [shipping policy](#).

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.



www.PacktPub.com

For More Information:

www.packtpub.com/mastering-nginx-guide-for-novice-and-advanced-user/book