DigitalOcean

Subscribe Share Contents

How To Use Apache as a Reverse Proxy with mod_proxy on CentOS 7

3

Posted February 2, 2017 104.4k APACHE LOAD BALANCING CENTOS

By: Mateusz Papiernik

Introduction

A *reverse proxy* is a type of proxy server that takes HTTP(S) requests and transparently distributes them to one or more backend servers. Reverse proxies are useful because many modern web applications process incoming HTTP requests using backend application servers which aren't meant to be accessed by users directly and often only support rudimentary HTTP features.

You can use a reverse proxy to prevent these underlying application servers from being directly accessed. They can also be used to distribute the load from incoming requests to several different application servers, increasing performance at scale and providing fail-safeness. They can fill in the gaps with features the application servers don't offer, such as caching, compression, or SSL encryption too.

In this tutorial, you'll set up Apache as a basic reverse proxy using the `mod_proxy` extension to redirect incoming connections to one or several backend servers running on the same network. This tutorial uses a simple backend written with the with Flask web framework, but you can use any backend server you prefer.

Prerequisites

To follow this tutorial, you will need:

- One CentOS 7 server set up with this initial server setup tutorial, including a sudo non-root user.

- Apache 2 installed on your server by following Step 1 of How To Install Linux, Apache, MySQL, PHP (LAMP) Stack on CentOS 7.
- Optionally, the `nano` text editor installed with `yum install nano`. CentOS comes with the `vi` text editor by default, but `nano` can be more user friendly.

Step 1 — Introducing Necessary Apache Modules

The modules that are needed to use Apache as a reverse proxy include `mod_proxy` itself and several of its add-on modules, which extend its functionality to support different network protocols. Specifically, we will be using:

- `mod_proxy`, the main proxy module Apache module for redirecting connections; it allows Apache to act as a gateway to the underlying application servers.
- `mod_proxy_http`, which adds support for proxying HTTP connections.
- `mod_proxy_balancer` and `mod_lbmethod_byrequests`, which add load balancing features for multiple backend servers.

All four modules are enabled by default on a fresh CentOS 7 installation. You can verify that they are enabled by running:

```
$ httpd -M
```

The command output will list all enabled Apache modules. The four lines you're looking for are the aforementioned module names:

```
Output
. . .
 proxy_module (shared)
. . .
 lbmethod_byrequests_module (shared)
. . .
 proxy_balancer_module (shared)
. . .
 proxy_http_module (shared)
. . .
```

In case the modules are not enabled, you can enable them by opening the `/etc/httpd/conf.modules.d/00-proxy.conf` with `nano`:

```
$ sudo nano /etc/httpd/conf.modules.d/00-proxy.conf
```

and uncommenting lines with necessary modules by removing `#` sign from the line beginnings so the file looks as follows:

/etc/httpd/conf.modules.d/00-proxy.conf

```
. . .
LoadModule proxy_module modules/mod_proxy.so
. . .
LoadModule lbmethod_byrequests_module modules/mod_lbmethod_byrequests.so
. . .
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
. . .
LoadModule proxy_http_module modules/mod_proxy_http.so
. . .
```

To put the changes into effect, save the file and restart Apache.

```
$ sudo systemctl restart httpd
```

Apache is now ready to act as a reverse proxy for HTTP requests. In the next step, we will create two very basic backend servers. These will help us verify if the configuration works properly, but if you already have your own backend application(s), you can skip to Step 3.

Step 2 — Creating Backend Test Servers

Running some simple backend servers is an easy way to test if your Apache configuration is working properly. Here, we'll make two test servers which respond to HTTP requests with printing a line of text. One server will say **Hello world!** and the other will say **Howdy world!**.

> **Note:** In non-test setups, backend servers usually all return the same kind of content. However, for this test in particular, having the two servers return different messages makes it easy to check that the load balancing mechanism uses both.

Flask is a Python microframework for building web applications. We're using Flask to create the test servers because a basic app requires just a few lines of code. You don't need to know Python to set these up, but if you'd like to learn, you can look at these Python tutorials.

Let's install the IUS package repository files first. IUS (Inline with Upstream Stable) is a community project that brings up-to-date versions of select software to CentOS, including Python 3.

```
$ sudo yum -y install https://centos7.iuscommunity.org/ius-release.rpm
```

Then install Python 3 and Pip, the recommended Python package manager.

```
$ sudo yum -y install python35u python35u-pip
```

Use Pip to install Flask.

```
$ sudo pip3.5 install flask
```

Now that all the required components are installed, start by creating a new file that will contain the code for the first backend server in the home directory of the current user.

```
$ nano ~/backend1.py
```

Copy the following code into the file, then save and close it.

~/backend1.py

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def home():
    return 'Hello world!'
```

The first two lines initialize the Flask framework. There is one function, `home()`, which returns a line of text (`Hello world!`). The `@app.route('/')` line above the `home()` function definition tells Flask to use `home()`'s return value as a response to HTTP requests directed at the `/` root URL of the application.

The second backend server is exactly the same as the first, aside from returning to a different line of text, so start by duplicating the first file.

```
$ cp ~/backend1.py ~/backend2.py
```

Open the newly copied file.

```
$ nano ~/backend2.py
```

Change the message to be returned from **Hello world!** to **Howdy world!**, then save and close the file.

~/backend2.py

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def home():
    return 'Howdy world!'
```

Use the following command to start the first background server on port `8080`. This also redirects Flask's output to `/dev/null` because it would cloud the console output further on.

```
$ FLASK_APP=~/backend1.py flask run --port=8080 >/dev/null 2>&1 &
```

Here, we are preceding the `flask` command by setting `FLASK_APP` environment variable in the same line. Environment variables are a convenient way to pass information into processes that are spawned from the shell. You can learn more about environment variables in How To Read and Set Environmental and Shell Variables on a Linux VPS.

In this case, using an environment variable makes sure the setting applies only to the command being run and will not stay available afterwards, as we will be passing another filename the same way to tell `flask` command to start the second server

Similarly, use this command to start the second server on port `8081`. Note the different value for the `FLASK_APP` environment variable.

```
$ FLASK_APP=~/backend2.py flask run --port=8081 >/dev/null 2>&1 &
```

You can test that the two servers are running using `curl`. Test the first server:

```
$ curl http://127.0.0.1:8080/
```

This will output **Hello world!** in the terminal. Test the second server:

```
$ curl http://127.0.0.1:8081/
```

This will output **Howdy world!** instead.

> **Note**: To close both test servers after you no longer need them, like when you finish this tutorial, you can simply execute `killall flask`.

In the next step, we'll modify Apache's configuration file to enable its use as a reverse proxy.

Step 3 — Modifying the Default Configuration to Enable Reverse Proxy

In this section, we will set up the default Apache virtual host to serve as a reverse proxy for single backend server or an array of load balanced backend servers.

Note: In this tutorial, we're applying the configuration at the virtual host level. On a default installation of Apache, there are no virtual hosts configured. We will be creating a single, default virtual host that will catch all traffic. However, you can use all those configuration fragments in other virtual hosts as well. To learn more about virtual hosts in Apache, you can read this How To Set Up Apache Virtual Hosts on CentOS 7 tutorial.

If your Apache server acts as both HTTP and HTTPS server, your reverse proxy configuration must be placed in both the HTTP and HTTPS virtual hosts. To learn more about SSL with Apache, you can read this How To Create a SSL Certificate on Apache for CentOS 7 tutorial.

Create the new default virtual host by creating a new empty Apache configuration file in `/etc/httpd/conf.d` directory using `nano` or your favorite text editor.

```
$ sudo nano /etc/httpd/conf.d/default-site.conf
```

The first example below explains how to configure the default virtual host to reverse proxy for a single backend server, and the second sets up a load balanced reverse proxy for multiple backend servers.

Example 1 — Reverse Proxying a Single Backend Server

Paste the following contents into the `default-site.conf` file, so your configuration file looks like this:

/etc/httpd/conf.d/default-site.conf

```
<VirtualHost *:80>
    ProxyPreserveHost On

    ProxyPass / http://127.0.0.1:8080/
    ProxyPassReverse / http://127.0.0.1:8080/
</VirtualHost>
```

If you followed along with the example servers in Step 2, use `127.0.0.1:8080` as written in the block above. If you have your own application servers, use their addresses instead.

There are three directives here:

- `ProxyPreserveHost` makes Apache pass the original `Host` header to the backend server. This is useful, as it makes the backend server aware of the address used to access the application.
- `ProxyPass` is the main proxy configuration directive. In this case, it specifies that everything under the root URL (`/`) should be mapped to the backend server at the given address. For example, if Apache gets a request for `/example` , it will connect to `http://your_backend_server/example` and return the response to the original client.
- `ProxyPassReverse` should have the same configuration as `ProxyPass` . It tells Apache to modify the response headers from backend server. This makes sure that if the backend server returns a location redirect header, the client's browser will be redirected to the proxy address and not the backend server address, which would not work as intended.

To put these changes into effect, restart Apache.

```
$ sudo systemctl restart httpd
```

Now, if you access `http://your_server_ip` in a web browser, you will see your backend server response instead of standard Apache welcome page. If you followed Step 2, this means you'll see **Hellow world!**.

Example 2 — Load Balancing Across Multiple Backend Servers

If you have multiple backend servers, a good way to distribute the traffic across them when proxying is to use load balancing features of `mod_proxy`.

Replace all the contents within the `VirtualHost` block with the following, so your configuration file looks like this:

/etc/httpd/conf.d/default-site.conf

```
<VirtualHost *:80>
<Proxy balancer://mycluster>
    BalancerMember http://127.0.0.1:8080
    BalancerMember http://127.0.0.1:8081
</Proxy>

    ProxyPreserveHost On

    ProxyPass / balancer://mycluster/
    ProxyPassReverse / balancer://mycluster/
</VirtualHost>
```

The configuration is similar to the previous one, but instead of specifying a single backend server directly, we've used an additional `Proxy` block to define multiple servers. The block is named `balancer://mycluster` (the name can be freely altered) and consists of one or more `BalancerMember`s, which specify the underlying backend server addresses. The `ProxyPass` and `ProxyPassReverse` directives use the load balancer pool named `mycluster` instead of a specific server.

If you followed along with the example servers in Step 2, use `127.0.0.1:8080` and `127.0.0.1:8081` for the `BalancerMember` directives, as written in the block above. If you have your own application servers, use their addresses instead.

To put these changes into effect, restart Apache.

```
$ sudo systemctl restart httpd
```

If you access `http://your_server_ip` in a web browser, you will see your backend servers' responses instead of the standard Apache page. If you followed Step 2, refreshing the page multiple times should show **Hello world!** and **Howdy world!**, meaning the reverse proxy worked and is load balancing between both servers.

Conclusion

You now know how to set up Apache as a reverse proxy to one or many underlying application servers. `mod_proxy` can be used effectively to configure reverse proxy to application servers written in a vast array of languages and technologies, such as Python and Django or Ruby and Ruby on Rails. It can be also used to balance traffic between multiple backend servers for sites with lots of traffic or to provide high availability through multiple servers, or to provide secure SSL support to backend servers not supporting SSL natively.

While `mod_proxy` with `mod_proxy_http` is the perhaps most commonly used combination of modules, there are several others that support different network protocols. We didn't use them here, but some other popular modules include:

- `mod_proxy_ftp` for FTP.
- `mod_proxy_connect` for SSL tunneling.
- `mod_proxy_ajp` for AJP (Apache JServ Protocol), like Tomcat-based backends.
- `mod_proxy_wstunnel` for web sockets.

To learn more about `mod_proxy`, you can read the official Apache `mod_proxy` documentation.

By: Mateusz Papiernik

Editor:
Hazel Virdó

Related Tutorials

How To Migrate your Apache Configuration from 2.2 to 2.4 Syntax.

How To Get Started With mod_pagespeed with Apache on a CentOS and Fedora Cloud Server

How To Use the .htaccess File

How To Set Up Mod_Rewrite (page 2)

How To Create a Custom 404 Page in Apache

3 Comments

Leave a comment...

Log In to Comment

ryanaklein *April 16, 2017*

0 This is a great article. For some reason, I was getting logs that said:

```
[Sun Apr 16 18:14:12.083870 2017] [proxy:error] [pid 17345] (13)Permission denied: AH
[Sun Apr 16 18:14:12.083986 2017] [proxy:error] [pid 17345] AH00959: ap_proxy_connect
[Sun Apr 16 18:14:12.084082 2017] [proxy_http:error] [pid 17345] [client 192.168.1.14
```

Solution at the bottom of this page

---

rraavvii *February 8, 2018*

0 Very well explained, thanks alot!!!!

---

lvvloten *May 24, 2018*

0 Note that on CentOS7, with SELinux enabled (as it should be!), Apache needs to be allowed explicitly to connect to a backend server on the network (i.e., not on localhost):

```
# /usr/sbin/setsebool -P httpd_can_network_connect 1
```