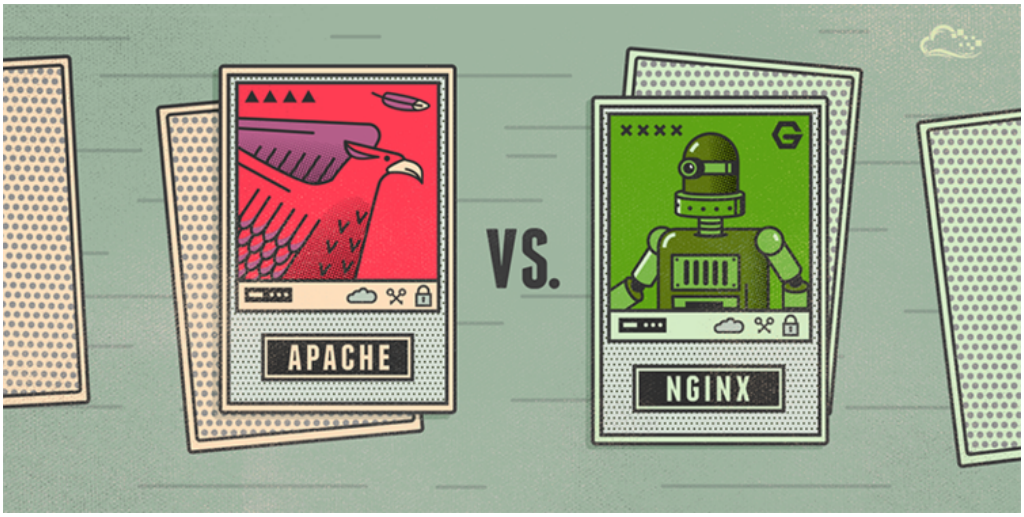
Justin Ellingwood Subscribe Share

Mark as Complete



Apache vs Nginx: Practical Considerations

243

Posted January 28, 2015 522.2k APACHE NGINX

Introduction

Apache and Nginx are the two most common open source web servers in the world. Together, they are responsible for serving over 50% of traffic on the internet. Both solutions are capable of handling diverse workloads and working with other software to provide a complete web stack.

While Apache and Nginx share many qualities, they should not be thought of as entirely interchangeable. Each excels in its own way and it is important to understand the situations where you may need to reevaluate your web server of choice. This article will be devoted to a discussion of how each server stacks up in various areas.

General Overview

Before we dive into the differences between Apache and Nginx, let's take a quick look at the background of these two projects and their general characteristics.

Apache

The Apache HTTP Server was created by Robert McCool in 1995 and has been developed under the direction of the Apache Software Foundation since 1999. Since the HTTP web server is the foundation's original project and is by far their most popular piece of software, it is often referred to simply as "Apache".

The Apache web server has been the most popular server on the internet since 1996. Because of this popularity, Apache benefits from great documentation and integrated support from other software projects.

Apache is often chosen by administrators for its flexibility, power, and widespread support. It is extensible through a dynamically loadable module system and can process a large number of interpreted languages without connecting out to separate software.

Nginx

In 2002, Igor Sysoev began work on Nginx as an answer to the C10K problem, which was a challenge for web servers to begin handling ten thousand concurrent connections as a requirement for the modern web. The initial public release was made in 2004, meeting this goal by relying on an asynchronous, events-driven architecture.

Nginx has grown in popularity since its release due to its light-weight resource utilization and its ability to scale easily on minimal hardware. Nginx excels at serving static content quickly and is designed to pass dynamic requests off to other software that is better suited for those purposes.

Nginx is often selected by administrators for its resource efficiency and responsiveness under load. Advocates welcome Nginx's focus on core web server and proxy features.

Connection Handling Architecture

One big difference between Apache and Nginx is the actual way that they handle connections and traffic. This provides perhaps the most significant difference in the way that they respond to different traffic conditions.

Apache

Apache provides a variety of multi-processing modules (Apache calls these MPMs) that dictate how client requests are handled. Basically, this allows administrators to swap out its connection handling architecture easily. These are:

- **mpm_prefork**: This processing module spawns processes with a single thread each to handle request. Each child can handle a single connection at a time. As long as the number of requests is fewer than the number of processes, this MPM is very fast. However, performance degrades quickly after the requests surpass the number of processes, so this is not a good choice in many scenarios. Each process has a significant impact on RAM consumption, so this MPM is difficult to scale effectively. This may still be a good choice though if used in conjunction with other components that are not built with threads in mind. For instance, PHP is not thread-safe, so this MPM is recommended as the only safe way of working with `mod_php`, the Apache module for processing these files.
- **mpm_worker**: This module spawns processes that can each manage multiple threads. Each of these threads can handle a single connection. Threads are much more efficient than processes, which means that this MPM scales better than the prefork MPM. Since there are more threads than processes, this also means that new connections can immediately take a free thread instead of having to wait for a free process.
- **mpm_event**: This module is similar to the worker module in most situations, but is optimized to handle keep-alive connections. When using the worker MPM, a connection will hold a thread regardless of whether a request is actively being made for as long as the connection is kept alive. The event MPM handles keep alive connections by setting aside dedicated threads for handling keep alive connections and passing active requests off to other threads. This keeps the module from getting bogged down by keep-alive requests, allowing for faster execution. This was marked stable with the release of Apache 2.4.

As you can see, Apache provides a flexible architecture for choosing different connection and request handling algorithms. The choices provided are mainly a function of the server's evolution and the increasing need for concurrency as the internet landscape has changed.

Nginx

Nginx came onto the scene after Apache, with more awareness of the concurrency problems that would face sites at scale. Leveraging this knowledge, Nginx was designed from the ground up to use an asynchronous, non-blocking, event-driven connection handling algorithm.

Nginx spawns worker processes, each of which can handle thousands of connections. The worker processes accomplish this by implementing a fast looping mechanism that continuously checks for and processes events. Decoupling actual work from connections allows each worker to concern itself with a connection only when a new event has been triggered.

Each of the connections handled by the worker are placed within the event loop where they exist with other connections. Within the loop, events are processed asynchronously, allowing work to be handled in a non-blocking manner. When the connection closes, it is removed from the loop.

This style of connection processing allows Nginx to scale incredibly far with limited resources. Since the server is single-threaded and processes are not spawned to handle each new connection, the memory and CPU usage tends to stay relatively consistent, even at times of heavy load.

Static vs Dynamic Content

In terms of real world use-cases, one of the most common comparisons between Apache and Nginx is the way in which each server handles requests for static and dynamic content.

Apache

Apache servers can handle static content using its conventional file-based methods. The performance of these operations is mainly a function of the MPM methods described above.

Apache can also process dynamic content by embedding a processor of the language in question into each of its worker instances. This allows it to execute dynamic content within the web server itself without having to rely on external components. These dynamic processors can be enabled through the use of dynamically loadable modules.

Apache's ability to handle dynamic content internally means that configuration of dynamic processing tends to be simpler. Communication does not need to be coordinated with an additional piece of software and modules can easily be swapped out if the content requirements change.

Nginx

Nginx does not have any ability to process dynamic content natively. To handle PHP and other requests for dynamic content, Nginx must pass to an external processor for execution and wait for the rendered content to be sent back. The results can then be relayed to the client.

For administrators, this means that communication must be configured between Nginx and the processor over one of the protocols Nginx knows how to speak (http, FastCGI, SCGI, uWSGI, memcache). This can complicate things slightly, especially when trying to anticipate the number of connections to allow, as an additional connection will be used for each call to the processor.

However, this method has some advantages as well. Since the dynamic interpreter is not embedded in the worker process, its overhead will only be present for dynamic content. Static content can be served in a straight-forward manner and the interpreter will only be contacted when needed. Apache can also function in this manner, but doing so removes the benefits in the previous section.

Distributed vs Centralized Configuration

For administrators, one of the most readily apparent differences between these two pieces of software is

whether directory-level configuration is permitted within the content directories.

Apache

Apache includes an option to allow additional configuration on a per-directory basis by inspecting and interpreting directives in hidden files within the content directories themselves. These files are known as `.htaccess` files.

Since these files reside within the content directories themselves, when handling a request, Apache checks each component of the path to the requested file for an `.htaccess` file and applies the directives found within. This effectively allows decentralized configuration of the web server, which is often used for implementing URL rewrites, access restrictions, authorization and authentication, even caching policies.

While the above examples can all be configured in the main Apache configuration file, `.htaccess` files have some important advantages. First, since these are interpreted each time they are found along a request path, they are implemented immediately without reloading the server. Second, it makes it possible to allow non-privileged users to control certain aspects of their own web content without giving them control over the entire configuration file.

This provides an easy way for certain web software, like content management systems, to configure their environment without providing access to the central configuration file. This is also used by shared hosting providers to retain control of the main configuration while giving clients control over their specific directories.

Nginx

Nginx does not interpret `.htaccess` files, nor does it provide any mechanism for evaluating per-directory configuration outside of the main configuration file. This may be less flexible than the Apache model, but it does have its own advantages.

The most notable improvement over the `.htaccess` system of directory-level configuration is increased performance. For a typical Apache setup that may allow `.htaccess` in any directory, the server will check for these files in *each* of the parent directories leading up to the requested file, for each request. If one or more `.htaccess` files are found during this search, they must be read and interpreted. By not allowing directory overrides, Nginx can serve requests faster by
doing a single directory lookup and file read for each request (assuming that the file is found in the conventional directory structure).

Another advantage is security related. Distributing directory-level configuration access also distributes the responsibility of security to individual users, who may not be trusted to handle this task well. Ensuring that the administrator maintains control over the entire web server can prevent some security missteps that may occur when access is given to other parties.

Keep in mind that it is possible to turn off `.htaccess` interpretation in Apache if these concerns resonate with you.

File vs URI-Based Interpretation

How the web server interprets requests and maps them to actual resources on the system is another area where these two servers differ.

Apache

Apache provides the ability to interpret a request as a physical resource on the filesystem or as a URI location that may need a more abstract evaluation. In general, for the former Apache uses `<Directory>`

or `<Files>` blocks, while it utilizes `<Location>` blocks for more abstract resources.

Because Apache was designed from the ground up as a web server, the default is usually to interpret requests as filesystem resources. It begins by taking the document root and appending the portion of the request following the host and port number to try to find an actual file. Basically, the filesystem hierarchy is represented on the web as the available document tree.

Apache provides a number of alternatives for when the request does not match the underlying filesystem. For instance, an `Alias` directive can be used to map to an alternative location. Using `<Location>` blocks is a method of working with the URI itself instead of the filesystem. There are also regular expression variants which can be used to apply configuration more flexibly throughout the filesystem.

While Apache has the ability to operate on both the underlying filesystem and the webspace, it leans heavily towards filesystem methods. This can be seen in some of the design decisions, including the use of `.htaccess` files for per-directory configuration. The Apache docs themselves warn against using URI-based blocks to restrict access when the request mirrors the underlying filesystem.

Nginx

Nginx was created to be both a web server and a proxy server. Due to the architecture required for these two roles, it works primarily with URIs, translating to the filesystem when necessary.

This can be seen in some of the ways that Nginx configuration files are constructed and interpreted.Nginx does not provide a mechanism for specifying configuration for a filesystem directory and instead parses the URI itself.

For instance, the primary configuration blocks for Nginx are `server` and `location` blocks. The `server` block interprets the host being requested, while the `location` blocks are responsible for matching portions of the URI that comes after the host and port. At this point, the request is being interpreted as a URI, not as a location on the filesystem.

For static files, all requests eventually have to be mapped to a location on the filesystem. First, Nginx selects the server and location blocks that will handle the request and then combines the document root with the URI, adapting anything necessary according to the configuration specified.

This may seem similar, but parsing requests primarily as URIs instead of filesystem locations allows Nginx to more easily function in both web, mail, and proxy server roles. Nginx is configured simply by laying out how to respond to different request patterns. Nginx does not check the filesystem until it is ready to serve the request, which explains why it does not implement a form of `.htaccess` files.

Modules

Both Nginx and Apache are extensible through module systems, but the way that they work differ significantly.

Apache

Apache's module system allows you to dynamically load or unload modules to satisfy your needs during the course of running the server. The Apache core is always present, while modules can be turned on or off, adding or removing additional functionality and hooking into the main server.

Apache uses this functionality for a large variety tasks. Due to the maturity of the platform, there is an extensive library of modules available. These can be used to alter some of the core functionality of the server, such as `mod_php`, which embeds a PHP interpreter into each running worker.

Modules are not limited to processing dynamic content, however. Among other functions, they can be used used for rewriting URLs, authenticating clients, hardening the server, logging, caching, compression, proxying, rate limiting, and encrypting. Dynamic modules can extend the core functionality considerably without much additional work.

Nginx

Nginx also implements a module system, but it is quite different from the Apache system. In Nginx, modules are not dynamically loadable, so they must be selected and compiled into the core software.

For many users, this will make Nginx much less flexible. This is especially true for users who are not comfortable maintaining their own compiled software outside of their distribution's conventional packaging system. While distributions' packages tend to include the most commonly used modules, if you require a non-standard module, you will have to build the server from source yourself.

Nginx modules are still very useful though, and they allow you to dictate what you want out of your server by only including the functionality you intend to use. Some users also may consider this more secure, as arbitrary components cannot be hooked into the server. However, if your server is ever put in a position where this is possible, it is likely compromised already.

Nginx modules allow many of the same capabilities as Apache modules. For instance, Nginx modules can provide proxying support, compression, rate limiting, logging, rewriting, geolocation, authentication, encryption, streaming, and mail functionality.

Support, Compatibility, Ecosystem, and Documentation

A major point to consider is what the actual process of getting up and running will be given the landscape of available help and support among other software.

Apache

Because Apache has been popular for so long, support for the server is fairly ubiquitous. There is a large library of first- and third-party documentation available for the core server and for task-based scenarios involving hooking Apache up with other software.

Along with documentation, many tools and web projects include tools to bootstrap themselves within an Apache environment. This may be included in the projects themselves, or in the packages maintained by your distribution's packaging team.

Apache, in general, will have more support from third-party projects simply because of its market share and the length of time it has been available. Administrators are also somewhat more likely to have experience working with Apache not only due to its prevalence, but also because many people start off in shared-hosting scenarios which almost exclusively rely on Apache due to the `.htaccess` distributed management capabilities.

Nginx

Nginx is experiencing increased support as more users adopt it for its performance profile, but it still has some catching up to do in some key areas.

In the past, it was difficult to find comprehensive English-language documentation regarding Nginx due to the fact that most of the early development and documentation were in Russian. As interest in the project grew, the documentation has been filled out and there are now plenty of administration resources on the Nginx site and through third parties.

In regards to third-party applications, support and documentation is becoming more readily available, and package maintainers are beginning, in some cases, to give choices between auto-configuring for Apache and Nginx. Even without support, configuring Nginx to work with alternative software is usually straight-forward so long as the project itself documents its requirements (permissions, headers, etc).

Using Apache and Nginx Together

After going over the benefits and limitations of both Apache and Nginx, you may have a better idea of which server is more suited to your needs. However, many users find that it is possible to leverage each server's strengths by using them together.

The conventional configuration for this partnership is to place Nginx in front of Apache as a reverse proxy. This will allow Nginx to to handle all requests from clients. This takes advantage of Nginx's fast processing speed and ability to handle large numbers of connections concurrently.

For static content, which Nginx excels at, the files will be served quickly and directly to the client. For dynamic content, for instance PHP files, Nginx will proxy the request to Apache, which can then process the results and return the rendered page. Nginx can then pass the content back to the client.

This setup works well for many people because it allows Nginx to function as a sorting machine. It will handle all requests it can and pass on the ones that it has no native ability to serve. By cutting down on the requests the Apache server is asked to handle, we can alleviate some of the blocking that occurs when an Apache process or thread is occupied.

This configuration also allows you to scale out by adding additional backend servers as necessary. Nginx can be configured to pass to a pool of servers easily, increasing this configuration's resilience to failure and performance.

Conclusion

As you can see, both Apache and Nginx are powerful, flexible, and capable. Deciding which server is best for you is largely a function of evaluating your specific requirements and testing with the patterns that you expect to see.

There are differences between these projects that have a very real impact on the raw performance, capabilities, and the implementation time necessary to get each solution up and running. However, these usually are the result of a series of trade offs that should not be casually dismissed. In the end, there is no one-size-fits-all web server, so use the solution that best aligns with your objectives.

Justin Ellingwood

♡ Upvote (243) Subscribe Share