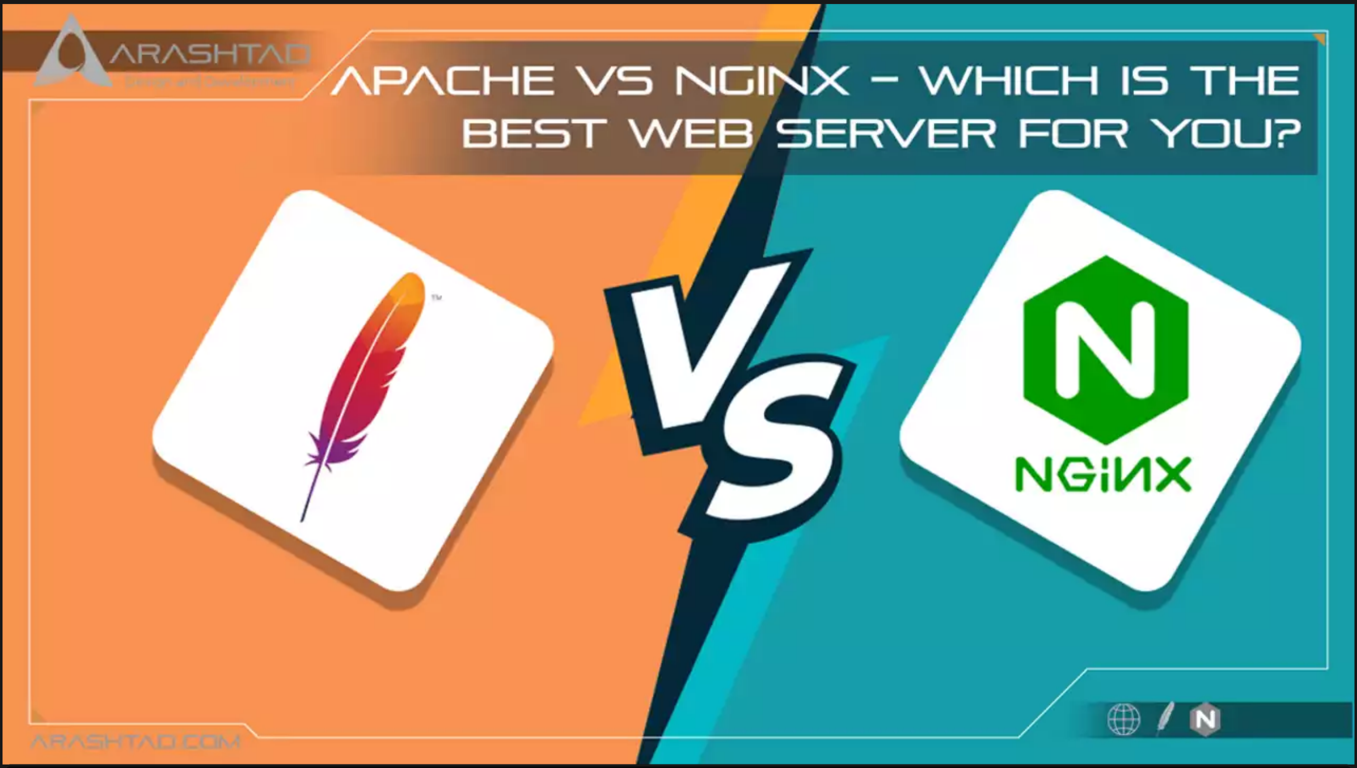
Apache Vs NGINX – Which Is The Best Web Server for You?

No comments



If you are looking for open-source web servers, Apache and Nginx are the two most popular ones. 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 this article will cover the strengths and weaknesses of each.In this article, before we dive into the differences between Apache and Nginx, we will take a quick look at the background of these two projects and their general characteristics.

Apache Background

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 was the most popular server on the internet from at least 1996 through 2016. 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 near-universal support. It is extensible through a dynamically loadable module system and can directly serve many scripting languages, such as PHP, without requiring additional software.

Nginx Background

In 2002, Igor Sysoev began work on Nginx as an answer to the C10K problem, which was an outstanding challenge for web servers to be able to handle ten thousand concurrent connections. Nginx was publicly released in 2004, and met this goal by relying on an asynchronous, events-driven architecture.
Nginx has since surpassed Apache in popularity due to its lightweight footprint and its ability to scale easily on minimal hardware. Nginx excels at serving static content quickly, has its own robust module system, and can proxy dynamic requests off to other software as needed.
Nginx is often selected by administrators for its resource efficiency and responsiveness under load, as well as its straightforward configuration syntax.

Nginx and Apache differences

There are a number of differences between Nginx and Apache that we should know in order to choose them more wisely for our needs. The differences are in the following aspects:

1. Differences in Architecture
2. Differences in Static and Dynamic Content
3. Differences in the System Configuration
4. Differences in Interpretation

Differences in Configuration

Apache and Nginx differ significantly in their approach to allowing overrides on a per-directory basis.

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. Apache was originally developed at a time when it was advantageous to run many heterogeneous web deployments side-by-side on a single server, and delegating permissions made sense. Nginx was developed at a time when individual deployments were more likely to be containerized and to ship with their own network configurations, minimizing this need. This may be less flexible in some circumstances 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. Keep in mind that it is possible to turn off .htaccess interpretation in Apache if these concerns resonate with you.

Differences in 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 or blocks, while it utilizes 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. Essentially, 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 blocks is a method of working with the URI itself instead of the filesystem. There are also regular expression variants that can be used to apply configuration more flexibly throughout the filesystem.
While Apache has the ability to operate on both the underlying filesystem and other web URIs, 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 is evident in the way that Nginx configuration files are constructed and interpreted. Nginx does not provide a mechanism for specifying the 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 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.

Differences in Connection Handling Architecture

One difference between Apache and Nginx is the specific way that they handle connections and network traffic. This is perhaps the most significant difference in the way that they respond under load.

Apache

Apache provides a variety of multi-processing modules (Apache calls these MPMs) that dictate how client requests are handled. This allows administrators to configure its connection handling architecture. These are:
1. mpm_prefork: This processing module spawns processes with a single thread each to handle requests. 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 always thread-safe, so this MPM has been recommended as a safe way of working with mod_php, the Apache module for processing these files.
2. 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.
3. 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.
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 sites face at scale. As a result, 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. Within the loop, events are processed asynchronously, allowing work to be handled in a non-blocking manner. When a connection closes, it is removed from the loop.
This style of connection processing allows Nginx to scale 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.

Differences in Static and Dynamic Contents

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 was a direct contributor to the popularity of LAMP (Linux-Apache-MySQL-PHP) architectures, as PHP code can be executed natively by the web server itself.

Nginx

Nginx does not have any ability to process dynamic content natively. To handle PHP and other requests for dynamic content, Nginx has to hand off a request to an external library for execution and wait for output to be returned. The results can then be relayed to the client.
These requests must be exchanged by Nginx and the external library using one of the protocols that Nginx knows how to speak (http, FastCGI, SCGI, uWSGI, memcache). In practice, PHP-FPM, a FastCGI implementation, is usually a drop-in

solution, but Nginx is not as closely coupled with any particular language in practice.
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.

Using Both Servers Together:

After reviewing the benefits and limitations of both Apache and Nginx, you may have a better idea of which server is more suited to your needs. In some cases, 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 handle all client requests. 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, files or other directives 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 facilitates horizontal scaling by adding additional backend servers as necessary. Nginx can be configured to pass requests to multiple servers, increasing this configuration's performance.

Conclusion

In this article, you got familiar with both Nginx and Apache in terms of their background, architecture, configuration (centralized or distributed), interpretation, and static and dynamic contents. You also learned about the details of using the two kinds of web servers together.
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 implementation time necessary to use either solution in production. Use the solution that best aligns with your objectives.

Join Arashtad Community

Follow Arashtad on Social Media

We provide variety of content, products, services, tools, tutorials, etc. Each social profile according to its features and purpose can cover only one or few parts of our updates. We can not upload our videos on SoundCloud or provide our eBooks on Youtube. So, for not missing any high quality original content that we provide on various social networks, make sure you follow us on as many social networks as you're active in. You can find out Arashtad's profiles on different social media services.

Get Even Closer!

Did you know that only one universal Arashtad account makes you able to log into all Arashtad network at once? Creating an Arashtad account is free. Why not to try it? Also, we have regular updates on our newsletter and feed entries. Use all these benefitial free features to get more involved with the community and enjoy the many products, services, tools, tutorials, etc. that we provide frequently.

SIGN UP NEWSLETTER RSS FEED