THE DEFINITIVE GUIDE TO **OPTIMIZING MAGENTO 2**

Magento™ 2

NEXCESS™
beyond hosting.

Nexcess – Beyond Hosting

sales@nexcess.net

NEXCESS™

beyond hosting.

OUR STORY

We believe in the promise of the Cloud: scalability, security, performance, and ease of use. Together with our team, clients, and partners, we've built something better.

Since 2000, we've been deploying application-specific hosting platforms that empower the way you do business. With over 10 years of extensive testing, support, and hosting experience – with tens of thousands of Magento Commerce and Open Source stores – we've created a Cloud platform designed around you.

Deploy fast, perform faster, and scale when you need it with Nexcess Cloud.

EXECUTIVE SUMMARY

Magento 2 is the second major release of the popular Magento eCommerce platform. Today, Magento is one of the most widely used open source eCommerce solutions in the world. This is largely due to its flexibility for merchants looking to design and customize their store. In part, this flexibility stems from the extensive Magento ecosystem of third party plug-ins. Moving beyond extensions, the Magento 2 architecture allows developers to further customize the store to the merchant's requirements.

With this versatile nature, Magento requires considerable resources to remain performant. A sluggish, unresponsive store will drive users away from even the best designed sites. All aspects of the software stack require proper tuning for both software itself and the hardware beneath. Improper tuning of even seemingly insignificant variables can cause bottlenecks and cascading issues that derail performance.

This guide provides configuration and best practices to keep your Magento 2 store at peak performance. This information derives from 10 years' experience with Magento installations, and includes everything from startup stores to enterprise clusters. The system and service configuration settings mirror the values we use on our systems, with noted exceptions for variables specific to store size and hardware configuration. Near the end of this paper, we provide a benchmark to demonstrate the effectiveness of these optimizations.

01

SOFTWARE CHANGES FROM MAGENTO 1 TO MAGENTO 2

CONTENTS

The original version of Magento 1 was released in 2008. When writing Magento 2, its developers noted its limitations and worked to improve performance and flexibility.

MODERN CODE BASE
FULL-PAGE CACHING

NATIVE VARNISH CACHING SUPPORT

PCI DSS-FRIENDLY FEATURES

IMPROVED CLI TOOLS

DATABASE IMPROVEMENTS

MODERN CODE BASE

The original version of Magento 1 was released in 2008. When writing Magento 2, its developers noted its limitations and worked to improve performance and flexibility. Magento 2 also supports PHP 7, which provides huge performance gains on identical hardware.

FULL-PAGE CACHING

Magento 2 now supports full-page caching in both the free Open Source (formerly Community Edition) and Magento Commerce (formerly Enterprise Edition) versions. This feature was previously exclusive to Open Source, leaving Magento Commerce users with third party extensions as their only option. The full-page cache (FPC) has been proven to provide significant performance increases in Magento 1, and its availability in the free Open Source version of Magento 2 is a welcome change.

NATIVE VARNISH CACHING SUPPORT

Varnish is a reverse-proxy cache that allows a Magento store to provide cached assets to visitors, rather than generate all assets dynamically upon every page load. Varnish can increase store visitor concurrency and lower load times significantly without scaling out hardware.

However, implementing Varnish with Magento 1 was problematic at best. Varnish configuration files, edge-side includes, and managing non-cacheable content like shopping carts and checkout pages were headaches even for skilled developers. To improve Varnish–Magento 1 compatibility, we developed the Turpentine extension.

With Magento 2, Varnish support is native. The entire Varnish configuration can be managed by Magento itself, boosting performance without adding the complexity of third party extensions like Turpentine.

PCI DSS-FRIENDLY FEATURES

PCI DSS compliance is a requirement for any Magento store that accepts payment by credit card. During the Magento 1 lifecycle, certain revisions to PCI requirements made achieving compliance difficult for many Magento users. Audit trails, admin login histories, secure authentication, and password policies are some examples. Magento 2 improved these areas, giving merchants stronger inherent security and making it easier to achieve PCI DSS compliance.

IMPROVED CLI TOOLS

The CLI tools available with Magento 2 are extensive, with many operations previously only executable through the admin interface now available. This makes the platform more developer-friendly, accelerating tasks such as enabling, disabling, and flushing the cache during deployments.

DATABASE IMPROVEMENTS

While exclusive to the Enterprise Edition of Magento 2, it is worth noting that databases can be distributed across multiple instances. With this capability, the tables storing orders run on hardware separate from the primary database running the site. This accelerates performance during the checkout process – a pain point for Magento 1 during high traffic. While configuring this distribution is beyond the scope of this white paper, all high traffic Enterprise-based stores should consider using this feature.

02

CONTENTS

TECHNOLOGICAL LANDSCAPE: THEN VERSUS NOW

Many things have changed since the publication of our original Magento performance guide in 2013.

CENTOS 7

PHP 7

SOLID STATE DRIVES

VIRTUALIZED ENVIRONMENTS

TECHNOLOGICAL LANDSCAPE: THEN VERSUS NOW

Many things have changed since the publication of our original Magento performance white paper in 2013:

CentOS 6 was the most recent CentOS release.

CentOS 7 saw release in 2015, with many improvements.

In June 2013, PHP 5.5 was the most current version of PHP, which has since reached end of life.

PHP 5.6 is the only remaining supported version of PHP 5, with all current development effort focused on PHP 7.1 and later.

CPUs, memory, and solid state drives are markedly better.

Current hardware is now generations ahead of the hardware we used to benchmark Magento 1.

Virtualized environments are widely available,

which provide better scalability, versatility, and performance than bare metal. Our current Magento plans use OpenStack, which, among other things, greatly simplifies migrations and development site creation.

03

OPTIMIZATIONS

CONTENTS

Modern applications like Magento 2 are resource-hungry and dependent on proper system and service configurations to run efficiently.

Running an application like Magento 2 on a stock system with the base variables set for services like MySQL, PHP-FPM, and Apache almost guarantees poor performance. Modern applications like Magento 2 are resource-hungry and dependent on proper system and service configurations to run efficiently. Good information about how to make these adjustments is scarce, and much of it is outdated. Even our 2013 white paper, ***Magento Hosting – Best Practices for Optimum Performance***, has drifted into obsolescence.

The next section focuses on various system services and the optimum configuration for each. Each entry explains the reasoning behind the settings and includes any other considerations worth taking into account, and we applied these variables when benchmarking Magento 2 for this white paper. As noted above, all system and service configuration settings mirror the values on Nexcess systems.

OPERATING SYSTEM

Our plans are designed around CentOS 7, an open source upstream-compatible version of RedHat Enterprise Linux 7. Major versions of CentOS have support for 10-year intervals. For CentOS 7, updates will continue until June 30 2024, providing long term support that will likely outlast its hardware.

CentOS ships with fairly conservative default variables, but providing an exhaustive list of tunable variables is beyond the scope of this paper. This exercise instead targets four general settings that provide the greatest return when trying to get the most out of Magento 2.

OPERATING SYSTEM

03

FILE SYSTEM AND MOUNT OPTIONS

Most modern operating systems such as CentOS 7 have switched from EXT4 to XFS as the default file system. XFS is a modern journaling file system that has numerous advantages including greater scalability, quick recovery from file system issues, and faster raw I/O performance. The use of XFS is highly recommended for any CentOS 7-based system, and EXT4 has no place in any modern configuration.

Many older distributions of linux use EXT4 and by default, most EXT4 Linux file systems log the time of access `(atime)`. This means that every time a file is read, the system updates the file's metadata to include the most recent atime. When files number in the tens of thousands, as is the case with Magento 2 and similar applications, ths creates significant overhead. Therefore, disabling `atime` tends to increase performance. However, XFS uses `relatime` instead of `atime`, and so is unaffected by this change.

To disable in EXT4, add the noatime option to the end of your mount options list in `/etc/fstab` for the partition containing your Magento installation. The change will take effect the next time the partition is mounted.

CHANGE DEFAULT DISK SCHEDULER

The Linux kernel provides a method to adjust the disk scheduler to tune I/O performance to match the given hardware. There are currently three scheduler options available in CentOS: `noop`, `deadline`, and `cfq` (completely fair queuing). To restrain scope, it is enough to know they exist to help optimize the I/O workload for the given hardware.

A server running Magento 2 benefits most from SSDs, or fast SAS drives and a RAID controller. For this type of hardware, allowing the kernel to re-order I/O writes wastes resources because the RAID controller's scheduler already performs this function.

To prevent the kernel from duplicating this function, change noop by issuing the command below, where sdX is the hardware device or devices used by the system.

Attention: Issue the command in `rc.local` or equivalent, or it will not persist over boot.

```
echo noop > /sys/block/sdX/queue/scheduler
```

DISC SCHEDULER QUEUE SIZE

The Linux disk I/O schedulers uses a queue to process I/O requests. This queue has a default size of 128 operations. Increasing this queue puts more I/O operations into memory and allows more data in the queue, all of which is properly ordered by the scheduler.

If you are running HDDs and a scheduler other than noop, increasing the queue to 256 will improve write efficiency.

Attention: Issue the command in `rc.local` or equivalent, or it will not persist over boot.

```
echo 256 > /sys/block/sda/queue/nr_requests
```

If you are instead using SSDs and the noop scheduler, increasing this setting will have little to no effect.

PRECISE PRIORITIES

In Linux distributions, most processes are set to run at the same priority level. This creates potential scheduling contention between processes that need to run in at higher priorities (PHP, MySQL) versus processes that do not (cron jobs). We recommend strictly prioritizing all system processes in categories from low to real-time by setting values in `limits.conf`, as well as setting specific application process priorities and Nice levels, as appropriate. For more information, see the process-priority schedule in GitHub at https://github.com/nexcess/magento-whitepaper-april-2018.

PHP

Both Magento 1 and Magento 2 are written in PHP. Generally speaking, stores that run the most current and supported version of PHP tend to be faster and more responsive than ones that do not.

Current Nexcess Cloud-based plans include support for PHP 5.6 through 7.2. We strongly recommend running Magento 2 with PHP 7.0 and later for optimal performance. As of Magento 2.2.0, PHP 7.0 is now a minimum requirement, and PHP 7.1 the most current and supported.

PHP and Magento: The Last 5 Years

Magento 1 was developed on the PHP 5 major release, supporting up to PHP 5.6, which was released in 2014. PHP 7.0 saw release In December 2015, delivering fundamental changes and and improvement to the PHP interpreter. PHP reported a 50 percent increase in speed over the previous 5.6 release. Other improvements include significantly reduced memory usage, improved exception handling, consistent 64-bit support for modern operating systems, and the removal of many old and unsupported SAPIs and extensions.

Seeking to capitalize on these improvements and faster load times, Magento chose to support PHP 7 over PHP 5 when developing Magento 2. The claim of "50 percent increase in speed" was validated in our original benchmarking of Magento 2 2015, and again in our Magento 2 white paper, ***Magento 2: Premium Performance with PHP 7 and Varnish***.

As of April 2018, Magento 1 does not natively support PHP 7, although some third party extensions attempt to breach this gap.

PHP 7.1 saw delivery in December 2016 with small improvements. PHP 7.1 is likewise supported by Magento and will remain in active development until December 1, 2018. At time of publication, PHP 7.2 is now available, but not yet actively supported by Magento 2. This is primarily due to the removal of the mcrypt libraries, which are an older cryptographic library now considered insecure by the community. Newer cryptographic libraries, such as libsodium and others, are part of the PHP 7.2 release. At time of publication, there is no timeline for official PHP 7.2 support.

PHP 03

PHP RUNTIME VARIABLES

Of the many variables within PHP's primary configuration file, `php.ini`, only some affect performance. These variables are discussed in detail below.

max_execution_time

`max_execution_time` sets the maximum time in seconds a PHP script is allowed to run before being killed. This variable prevents runaway scripts from consuming system resources. As some Magento admin functions take considerable time to run, we recommend `max_execution_time` of 600 seconds.

```
max_execution_time = 600
```

max_input_time

`max_input_time` sets the maximum time in seconds a PHP script is allowed to parse POST and GET input data. The default is set to -1, which will inherit the value from `max_execution_time`.

```
max_input_time = -1
```

memory_limit

`memory_limit` restricts how much memory in bytes a script consumes. Any script reaching this limit is killed, which limits the possibility of a runaway script consuming excessive amounts of server memory. Magento 2 recommends this be set to 768MB, as do we.

```
memory_limit = 768M
```

upload_max_filesize

`upload_max_filesize` sets the maximum size in bytes allowed for each uploaded file. The default size is relatively small at 2MB. Since Magento file imports are often of significant size, we recommend setting to 512 MB.

```
upload_max_filesize = 512M
```

post_max_size

`post_max_size` sets the largest allowed amount of POST data and is related to the `upload_max_filesize` variable. As such, we recommend setting this to 512MB.

```
post_max_size = 512M
```

realpath_cache_size

`realpath_cache_size` designates the size of the cache within PHP that stores the paths to PHP files. When properly configured, it can significantly improve performance for large PHP applications that contain thousands of PHP files.

The realpath cache is used for PHP functions such as `include()` or `require()`, among others. When one of these PHP function calls is invoked, PHP first checks its realpath cache to see if the path is already cached, which would allow PHP to call the file directly instead of performing many `stat()` or `lstat()` calls to the actual file on disk. For Magento 2, which uses deep file paths, this can save tens of thousands of calls per page load.

As noted above, the `realpath_cache_size` specifies the size of this cache in bytes. PHP 7.2 recently increased this value from 16kB to 4MB. This cache is not shared across PHP children, so each process will have its own independent cache.

If open_basedir is enabled in PHP, it disables the realpath cache. Safe mode would also disable the cache, but safe mode was removed in PHP 5.4 and is no longer a concern in modern releases.

```
realpath_cache_size = 4M
```

realpath_cache_ttl

`realpath_cache_ttl` sets the frequency in seconds at which realpath cache entries expire. We recommend increasing this value from the default of 120 seconds to 300 seconds. The goal is to keep the cache populated as long as possible. However, setting it longer than 300 seconds may conflict with the PHP-FPM child lifetime variable.

```
realpath_cache_ttl = 300
```

PHP-FPM

PHP-FPM, or FastCGI process manager, has three available process managers that handle PHP-FPM processing: static, ondemand, and dynamic.

We recommend the dynamic PM for Magento 2 environments. Though it grants excellent performance and memory management, improper tuning can lead to memory exhaustion or poor performance.

Static is a simple process manager (PM) that designates the amount of children available to handle incoming PHP requests. While this can be performed well, careless use has the potential to exhaust system memory resources. As the name suggests, static uses the configured number of PHP processes at all times, even when the system is idle.

Ondemand only creates children as new requests come in, then destroys them when additional requests exceed the `idle_timeout` or `max_request` variables. Although this keeps memory leaks from consuming excessive system resources, it involves some overhead and is not the best solution in terms of performance.

Dynamic preforks a configured amount of the PHP processes in the same manner as static, while also allowing scalability. In times of high traffic, dynamic creates additional PHP processes to meet demand. When no longer needed, these additional processes are destroyed after their request counters are exceeded to free memory.

TUNING THE DYNAMIC PM

03

PHP-FPM defaults are far from ideal. We recommend the following settings:

Process Manager

As discussed earlier, set the process manager to dynamic.

```
pm = dynamic
```

Max Children

This sets the maximum creatable number of child processes when using the static or dynamic process managers. Similar to the Apache MaxClients directive, this sets a hard limit on how many parallel PHP-FPM requests can be processed.

It is important to consider available system memory and the amount used by each PHP process when setting this variable. Setting it too high can result in out-of-memory conditions. Benchmarking various settings will also allow for proper tuning. Even in systems with a hefty surplus of memory, there are finite CPU cores available to process each request.

```
pm.max_children = 96
```

Start Servers

Start servers sets the number of child processes to be listening upon starting the PHP-FPM process. WIth the dynamic PM, this normally does not require a very high setting because new child processes will be created as demand increases. Setting it with a moderate value helps curb memory usage, and offers a solid compromise between the static and ondemand PMs.

```
pm.start_servers = 8
```

Minimum and Maximum Spare Servers

Since the dynamic PM creates additional processes based on demand, it will create new child processes up to the value set by `pm.max_children`. The number of child processes fluctuates and is determined by `pm.min_spare_servers` and `pm.max_spare_servers`. We typically set these to one-half and double the `pm.start_servers` variable, respectively.

```
pm.min_spare_servers = 4
pm.max_spare_servers = 16
```

Maximum Number of Requests

The maximum number of requests-per-process can be specified with `pm.max_requests`. This prevents potential memory leaks by limiting the memory used by a single process.

We recommend setting this variable relatively high to limit the unnecessary destruction and recreation of processes. Depending on site traffic, a process may sit idle for minutes to days. If it appears an application consumes excessive memory due to such a leak, this value can be decreased.

```
pm.max_requests = 16384
```

PHP OPCACHE

Opcache is a PHP-caching extension that accelerates performance by optimizing and storing precompiled script bytecode in shared memory. With Opcache, frequently used static code can be read directly from memory, skipping the intensive compilation process.

Opcache has been bundled every version of PHP since PHP 5.5, including the most current PHP 7.2. Opcache has replaced the use of older caching utilities such as eAccelerator and APC. As Opcache can drastically increase site performance, it is worth spending time tuning its variables to match the environment and application.

Enabling Opcache within PHP will allow Opcache to function, but for large applications like Magento 2, the default variables are not ideal and will limit the potential performance gains when using Opcache. However, when configuring Opcache for a specific application, it is critical to provide adequate memory for Opcache. Otherwise, the entire cache runs the risk of becoming invalidated, essentially disabling it.

The recommended variable changes for Magento 2 are listed below. These are applied to our own systems as well to the benchmark later in this paper. The following variables are sufficient for most Magento 2 installations; however, it is best practice to monitor the Opcache status variables to make sure your cache is operating effectively.

PHP OPCACHE

03

OPCACHE.MEMORY_CONSUMPTION

`opcache.memory_consumption` sets the size of the shared memory storage used by Opcache. It defaults to 64MB for caching all compiled scripts, which is inadequate for a Magento 2 installation.

Calculating how much memory is necessary for a given application requires some trial and error. Opcache provides some tools for monitoring internal variables with the `opcache_get_status()` function. The "free memory" value lists how much memory is available to Opcache and can be used to determine the sizing of `opcache.memory_consumption`. For another metric, if `opcache_get_status()` reports `oom_restarts` as greater than zero, this indicates an out-of-memory event.

Attention: Opcache will only restart if wasted memory inside Opcache exceeds the `max_wasted_percentage`. If this is not met and the cache is full, Opcache will not restart and any cached items will not be used, effectively disabling your cache.

Based on our research and experience, a setting of 512MB is adequate for most Magento 2 installations. With this setting, out-of-memory events are rare, but still sometimes occur on larger sites with many extensions or a surplus of code. Servers running multiple sites call for additional caution. Because Opcache is bound to the master PHP process, all sites on the server use the same Opcache configuration, even if those sites are in different PHP-FPM pools. For these reasons, the Opcache status should be monitored for out-of-memory conditions.

OPCACHE.MAX_ACCELERATED_FILES

`opcache.max_accelerated_files` is the maximum number of keys or files in the Opcache hash table. This defaults to 2000 cacheable files, but internally this is increased to a higher prime number to ensure consistent hashing.

The list of primes is hard-coded into the Opcache code and as follows: 5, 11, 19, 53, 107, 223, 463, 983, 1979, 3907, 7963, 16229, 32531, 65407, 130987, 262237, 524521, 1048793, 2097397, 4194103, 8388857, 16777447, 33554201, 67108961, 134217487, 268435697, 536870683, 1073741621, 2147483399

Note that when setting this variable, the value will be rounded up. In cases of the larger variables, this may result in a much larger variable than anticipated. For example, 140000 would round up to 262237.

The maximum value is 100000 in PHP 5.5.6 and earlier, and 1000000 in all later versions.

A default Magento 2 installation contains roughly 35,000 PHP files. We recommend increasing the default to 65407 which roughly doubles the number of Magento PHP files . This prevents most out-of-memory conditions, and it gives headroom for custom development, plug-ins, and other customizations.

OPCACHE.VALIDATE_TIMESTAMPS

`opcache.validate_timestamps` enables the option to check each file for changes at a specified interval. This allows Opcache to flush any cached files that may have changed on disk. This validation is unnecessary in production, where files change only during planned code deployments. Setting this value to zero in production systems removes the overhead of Opcache checking for file modifications at the interval specified by `opcache.revalidate_freq`, which defaults to 2 seconds:

```
opcache.validate_timestamps=0
```

With `opcache.validate_timestamps=0`, you must manually restart your PHP processes after any code changes to force a cache flush. On development systems or production systems implementing frequent small code changes, we recommend setting this variable to 1 to prevent the need to restart PHP after each change.

```
opcache.validate_timestamps=1
```

OPCACHE.REVALIDATE_FREQ

The revalidation frequency is how often Opcache checks for file changes when `opcache.validate_timestamps` is enabled. The default of 2 seconds is somewhat aggressive; we recommend doubling this value to 4 seconds.

```
opcache.revalidate_freq=4
```

OPCACHE.INTERNED_STRINGS_BUFFER

String-interning is a space-saving method of storing duplicate strings in memory by only storing one copy of each string, and referencing the copies with pointers. This variable sets the amount of memory in megabytes to allocate string-interning. By default, Opcache sets this variable to only 4MB; we recommend increasing this value to 48MB.

```
opcache.interned_strings_buffer=48
```

OPCACHE.MAX_WASTED_PERCENTAGE

`opcache.max_wasted_percentage` is the percentage of wasted space in Opcache necessary to trigger a restart. This variable defaults to 5 percent, which is generally regarded as safe. This value can be increased to prevent unnecessary restarts that would invalidate the cache when the cache is not full. Generally, this is not a concern if the `opcache_memory_consumption` is set high enough.

It is critical to provide adequate memory for Opcache. Otherwise, the entire cache runs the risk of becoming invalidated, essentially disabling it.

APACHE

One of the most major changes from CentOS 6 to CentOS 7 was the switch to the Apache 2.4 webserver. CentOS 6 included Apache 2.2 within its base packages, which now lack some modern features, namely HTTP/2.

HTTP/2 is a revision of the original HTTP 1.1 protocol released in 1999. It focuses on improving performance, perceived end-user latency, and use of a multiplex connection between a webserver and browser. HTTP/2 is currently supported by all major browsers and is supported by Apache 2.4.17 and Nginx 1.9.5, and later. If your web server supports it, we strongly recommend using HTTP/2.

Apache 2.4 adds a third type of multi-processing module (MPM) to the existing prefork and worker MPMs: event. The event MPM brings similar benefits found in Nginx to Apache. Each has pros and cons, though the event MPM earns our recommendation.

APACHE 03

EVENT MPM: KEY BENEFITS

The chief advantages of event MPM are the ability to process more requests than worker MPM while using the same memory footprint. Event MPM also allows longer keepalives without the downside of holding up threads.

Because of these advantages, we recommend event MPM over prefork and worker. The event configuration itself is highly dependent on the system hardware responsible for running Apache. Available memory, processing power, and consideration for any other services running on the machine will guide configuration.

In our benchmark, our virtual machine has 20 cores and 24GB of memory, and used the Apache configuration below:

```
<IfModule event.c>
    StartServers             4
    ServerLimit              32
    ThreadLimit              64
    ThreadsPerChild          64
    MaxRequestWorkers        2048
    MaxConnectionsPerChild   8192
</IfModule>

KeepAlive            On
MaxKeepAliveRequests 100
KeepAliveTimeout     5
```

PREFORK MPM

Prefork MPM is the simplest of the three multi-processing modules. Prefork is non-threaded, meaning that individual full Apache child processes are spawned when Apache launches, and each process can only handle a single connection.

Because of this, each Apache process contains all Apache configuration, modules, and so on, which is not memory efficient. Since there are no threads to help handle connections, a full Apache process must handle each connection individually. No other connections can be handled by a child process until the existing connection is closed. Prefork is best used for non-thread-safe application and handler configurations; for example, where `mod_php` is used over the more efficient PHP-FPM.

WORKER MPM

Worker MPM solves the memory-overhead issue of prefork by supporting threading. With worker, it is possible to start a fewer number of child Apache processes from the parent process. Each of these child processes will then process incoming connections in threaded processes, which uses far less memory. Threaded processes share the memory with the child process that spawned it, with each thread handling its own connection and including all requests for the connection.

EVENT MPM

Event MPM is similar to the worker MPM and improves performance in several key areas. With worker, each thread is bound to a single connection, and no other connections can be processed by a thread until its connection is closed. Event, however, uses threads to handle requests, not connections. Each established connection ties up a thread with handling requests, but only until the requests are complete. Once complete, even if the connection is still established, event MPM makes these threads available to process additional requests. The connections are managed by the parent process, leaving these threads available to handle any additional requests.

MYSQL

Our CentOS 6 platform used Percona Server instead of the standard version of MySQL. As detailed in our white paper, *Percona Server Versus Percona XtraDB Cluster: A Magento Case Study*, we chose Percona because it 1) is compatible with MySQL, 2) is open source, and 3) performs better than MySQL. We have historically used Percona server for all Magento 1 and Magento 2 plans with great success.

Moving to CentOS 7, we switched from Percona to MariaDB, which is another high performance, open source replacement for MySQL. MariaDB is now included with base CentOS 7, translating to a platform with less external dependencies and a simplified deployment process.

MariaDB has an extensive amount of tunable variables. For the entire production `my.cnf` file usd in this benchmark, refer to the GitHub repo at https://github.com/nexcess/magento-whitepaper-april-2018.

MYSQL 03

INNODB_BUFFER_POOL_SIZE

`innodb_buffer_pool_size` contains the working data set for all tables using the InnoDB storage engine. The optimal size is one that can accommodate the entire working data set within the buffer pool while also providing room for future growth. Maintaining a sufficiently large buffer pool provides space for new entries and removes the need to flush infrequently used pages to disk to conserve that space.

Extremely undersized buffer pools can lead to excessive disk writes and performance degradation. If necessary, systems dedicated to MariaDB allow for settings as high as 75 - 80 percent of available memory.

On instances where the database runs on the same server as the webserver, we set this value to 50 percent of available memory.

INNODB_BUFFER_POOL_INSTANCES

This variable sets the number of regions into which the total `innodb_buffer_pool_size` will be split. For example, a 20GB `innodb_buffer_pool_size` configured with four `innodb_buffer_pool_instances` results in four buffer pool regions of 5GB. Each buffer pool instance contains its own buffer pool-specific structures such as pool mutex, free and LRU lists, and so on. For a larger number of buffer pool instances, the end result is a higher potential InnoDB throughput.

We set the number of pool instances to 4.

INNODB_LOG_BUFFER_SIZE

This, along with the InnoDB log file, constitute part of the redo log system. The log buffer holds redo log records before they are written to the log file. Poor or undersized log buffers canresult
in frequent writes to the log files in order to free pages for the recording of incoming changes. For transactions that change large amounts of rows, maintaining a larger log buffer size prevents excessive disk I/O. The best way to tune this is by testing performance with the anticipated traffic
in production.

On our servers, we set this value to one-sixth the size of the `innodb_log_file_size.`

INNODB_LOG_FILE_SIZE

The log files (often with filenames of ib_`logfile[0-9]*)` contain encoded change requests to InnoDB tables. These recorded changes can be replayed by MariaDB during crash recovery to correct data from incomplete transactions. The log file size should be large enough to accommodate the logging from routine data changes; overly small log files can stall write-inquiries and degrade performance whenever InnoDB attempts to free space within the log files.

Historically, it was unwise to use exceedingly large log files because they delayed server startup. Due to a more efficient recovery process in newer versions of MariaDB, this is no longer a cause for concern. The best way to tune this is by testing performance with transaction traffic that resembles anticipated traffic in production.

On our servers, this defaults to 10 percent of the size of the `innodb_buffer_pool_size` value.

QUERY_CACHE_SIZE

This variable designates the size of the cache for storing evaluated query results. If the same queries are frequently submitted to MariaDB, enabling this cache will accelerate response times.

There is a performance penalty when searching for query results not stored in the query cache. Furthermore, for high concurrency traffic coupled with a high queries-per-second (QPS) rate, performance may suffer when attempting to acquire a mutex on the query cache. Due to this, increasing the query cache requires some consideration. For workloads like Magento 2, setting the query cache too high will hurt performance.

We recommend setting `query_cache_size` to 2 percent of available memory, with an upper limit of 128MB.

TMP_TABLE_SIZE AND MAX_HEAP_TABLE_SIZE

The lower of these two values determines the maximum size of an in-memory temporary table before it is converted into an on-disk MyISAM temporary table. These values should be high enough to fit the majority of temporary data sets into memory. Note the entire amount of memory is not pre-allocated at the time of thread creation.

MYSQL SETTINGS AT A GLANCE

`innodb_buffer_pool_size`
On instances where the database runs on the same server as the webserver, we set this value to 50 percent of available memory.

`innodb_buffer_pool_instances`
We set the number of pool instances to 4.

`innodb_log_buffer_size`
On our servers, we set this value to one-sixth the size of the innodb log file size.

`innodb_log_file_size`
On our servers, this defaults to 10 percent of the size of the `innodb_buffer_pool_size` value.

`query_cache_size`
We recommend setting `query_cache_size` to 2 percent of available memory, with an upper limit of 128MB.

`tmp_table_size` and `max_heap_table_size`
These values should be high enough to fit the majority of temporary data sets into memory.

MAGENTO 2 CACHING

Like its predecessor, Magento 2 provides several layers of caching to improve store performance. The size and complexity of Magento 2 means enabling and properly configuring the built-in caching is crucial.

SESSION CACHING

03

Magento 2 provides three options for user session storage: files, database, or a memory-based backend like Memcached or Redis.

FILES

Files serve as the default storage location for all user session data. While this setting works for staging, development, and low volume production environments, it does not scale to high traffic sites due to the latency of file I/O operations. With solid state drives it is less of a concern, although memory-backed solutions still offer advantages. In most load-balanced, clustered environments, where a user may be spread across multiple web nodes, memory-backed solutions are ideal.

DATABASE

Using the database for sessions storage is an option, but adding queries to a busy database already handling thousands of queries per second is sub-optimal.

Due to Redis persistence, the Magento team prefers Redis over Memcached for session storage, and so do we.

MEMORY-BACKED (REDIS OVER MEMCACHED)

Due to the above reasons, we recommend memory-backed caching for high performance session storage. Magento 2 natively supports two memory-based solutions, Memcached and Redis.

Due to Redis persistence, the Magento team prefers Redis over Memcached for session storage, and so do we. Implemented in Redis version 2 and later, persistence makes it possible to back up cached data like cart contents. In the event of an unexpected restart, Redis retains this data; Memcached does not. After a Memcached flush, store visitors returning to your site will find their carts empty.

To prevent unwelcome surprises during deployment to the production environment, we also recommend Redis for staging and development environments. Several options are available for configuring Redis persistence, but this is a topic best served by a separate publication.

Previously, we have recommended Memcached for sessions due to Redis causing session-locking bugs with Magento 2. These issues were reported resolved in Magento 2.0.6, but persisted until Magento 2.2.2, as confirmed by the community and our own testing. We recommend checking your version to verify compatibility.

Memcached Configuration for Session

If you must use Memcached instead of Redis, we recommend configuring Memcached to listen on a local Unix socket. Local sockets reduce networking overhead and provide a slight performance increase.

If your Memcached deployment is an external instance or a cluster, a standard TCP or UDP connection will be required. Two other variables within the Memcached config will need to be considered.

Maxconn

Defines the maximum amount of simultaneous connections to Memcached. Any additional connections beyond this number will be queued. The default for this limit is 1024; however, we recommend increasing it to 4096, which is sufficient for most deployments.

If you are a very high traffic site, monitor the Memcached `listen_disabled_num` statistic for blocked connections.

Cachesize

is the upper limit of the cache in megabytes. We set this value to 256MB for our deployments. Memory usage can be monitored by comparing bytes to `limit_maxbytes` in the stats.

If bytes is approaching `limit_maxbytes`, increase your cache size.

03 DEFAULT AND FULL-PAGE CACHES

Magento 2 supports various options for the default and full-page caches (FPCs), including the file system, database, or Redis. For production stores, we recommend enabling both the default and full-page caches, with the exception of those running Magento 2 with Varnish. This exception is discussed in greater detail on page 47.

As with the session cache, file or database-backed storage are recommended only for development environments and rarely, if ever, for production. Recommend by the Magento team, Redis has proven to be the preferred method of cache storage in terms of performance and scalability. Desirable features include replication and backup support, as well as tag support to allow for flushing only portions of the cache.

As discussed in the Session Caching section on page 46-47, backups allow data to persist through restarts and replication, and allow for high availability configurations when it becomes necessary to scale beyond one Redis instance.

Two options exist for configuring the Redis instances for the default and full-page caches. In the first, a single Redis instance uses two separate databases, one for each cache. The second option uses two separate Redis instances and dedicates one to each cache. The Magento 2 configuration in env.php supports both setups.

Our benchmark uses a single instance of Redis with three separate databases, one each for the default, session, and full-page caches. Larger deployments would benefit having separate Redis instances for each cache type, but for the limited scale of a single virtual machine, a single instance is adequate and creates no observable connection contention.

When using a Redis database for sessions within another instance of Redis, it is critical to note that Redis eviction policies are global, not database-specific. If Redis runs out of memory, there is the possibility that client session data could be evicted. This serves as another example of why it is crucial to make sure Redis does not run out of memory and begin evicting data.

MAXMEMORY

`maxmemory` sets the maximum amount of memory to be used by Redis to cache data. We normally start at a value of 512MB for both the default and full-page caches, then monitor Redis as the cached data grows to verify evictions from Redis reaching the memory limit. Failure to set this value places no memory limit, potentially allowing Redis to consume all system memory.

```
maxmemory 512MB
```

MAXMEMORY POLICY

`maxmemory-policy` sets the eviction policy for Redis. For Magento 2, we recommend using the least recently used algorithm, which will remove low-demand keys and conserve memory.

```
maxmemory-policy allkeys-lru
```

To verify adequate memory, confirm the `used_memory` statistic is less than maxmemory, and monitor the `evicted_keys` variable to confirm the absence of forced evictions.

VARNISH

Varnish is an HTTP reverse proxy that can dramatically increase site performance by caching content in memory in front of the web server. Processes normally handled by Apache and PHP are now handled by Varnish as it directly delivers assets from memory to users' browsers.

The downside to Varnish is its complexity. Controlling which content is cached, which is not, and maintaining proper invalidation of cached data are difficult tasks. For Magento 1, our Turpentine extension simplified the configuration of Magento with Varnish. With Magento 2, Varnish support is built-in, and it directly replaces the full-page cache within the platform.

03 USING VARNISH WITH HTTPS PROTOCOL

Varnish does not natively support TLS connections.

Varnish does not natively support TLS connections. Only unsecured traffic can pass through it, and it cannot handle any type of HTTPS traffic. With modern security standards requiring all traffic to be encrypted, this is impractical for Magento 2 stores.

The solution to this is to use a TLS terminator in front of Varnish. This offloads TLS processing to a separate service, which then passes all traffic as HTTP traffic to Varnish.

There are many candidates for TLS terminator services, including Pound, Nginx, Stunnel, Hitch, Stud, and Haproxy, among others. Though these are all high performance solutions, we prefer Haproxy and Nginx.

HAPROXY AS A REVERSE PROXY

Haproxy, a TCP/HTTP load balancer, can function as a TLS terminator while also providing scaling and redundancy. When configured in front of Varnish, Haproxy can bypass it if becomes unavailable and route traffic directly to Apache. Haproxy can be also used to easily scale horizontally because all TLS termination occurs in front of the web servers. As of release 1.8, Haproxy supports HTTP/2.

The benchmarks using Varnish within this guide used Haproxy as a TLS terminator.

NGINX AS A REVERSE PROXY

Nginx also qualifies, but it much more than just a TLS terminator. As a full-featured web server itself, it can be configured as a reverse proxy while it provides additional benefits.

One of these benefits is microcaching. The concept behind microcaching is to cache all static and dynamic content in a site for time periods as short as 1 second. For high-traffic sites with heavy dynamic content, this prevents resource-hungry content from being generated more than once per second, regardless of how many visitors on your site.

In most cases, allowing this content to be delayed for seconds is acceptable. However, in Magento 2, it is not possible to cache dynamic content because every page load is unique to a user session. This explains why integrations with reverse proxies like Varnish tend to be complex; pages are typically separated into blocks of cacheable static data and dynamic data.

Due to this limitation, Nginx microcaching cannot easily be used for all Magento 2 content. Typical Magento 2 page-load waterfalls often involve well over 100 requests for JavaScript and CSS files, which places heavy load onto the web server. Using the Nginx microcache to cache these static files removes load from the web server, freeing it to focus on dynamic requests. This method makes it possible to set the cache-lifetime for 30 seconds or more.

As demonstrated in the benchmark, the above configuration tends to greatly increase performance in Magento 2.

04

MAGENTO 2 BENCHMARK

CONTENTS

To test various Magento 2 configurations and system settings, we used the LoadImpact load testing tool.

Each test ran three times, then we averaged the results, which are made more reliable by the randomized virtual user behavior.

METHOD

04

To test various Magento 2 configurations and system settings, we used the LoadImpact load testing tool. Compared to many traditional load testing tools that provide results in terms like requests-per-second or transactions-per-second, LoadImpact tests better simulate real user traffic.

LoadImpact test results attempt to reveal how many users can be on a site before it breaks down, at which point users experience increasing page-load times or pages failing to load altogether. This is a more useful metric to Magento 2 merchants, as its results are inline with the information provided by Google Analytics.

Our testing consisted of two user scenarios running in parallel from Ashburn, VA and Palo Alto, CA. This simulates traffic from the east and west U.S. coasts to the tested system in our data center in Southfield, MI. The scenarios consisted of two types running in parallel. In one, a virtual user added four items to cart as described above, and in the other, a virtual user hit only the main page of the site.

Unless specified otherwise, all tests ran for 10 minutes and started with one virtual user ramping up to 250 users. Each test ran three times, then we averaged the results, which are made more reliable by the randomized virtual user behavior. The final data is then put through a moving average to smooth the graphs for easier test comparison.

How LoadImpact Simulates Traffic

To simulate real world traffic, LoadImpact provides a browser tool to record user scenarios. For our tests, we used this tool to simulate the activity of a user visiting a typical site. Our test scenarios followed a simple model of a user landing on the main home page of the site, browsing to various categories, adding four products to their cart, then proceeding to checkout. To simulate real users reading content on each page, we included randomized delays of 2 to 10 seconds between each user interaction. This process is then repeated, continually ramping up the number of users to a set maximum amount. The output from this testing shows complete load time for the scenario. By comparing these load times to the number of virtual users, it is possible to deduce where the site breaks down under a given load.

Most typical sites have low conversion ratios, so our test is a best case scenario of every visitor on the site adding items to cart and proceeding to checkout. We chose to run our tests this way as it is more demanding on the system and application by creating more frequent writes of session data.

To more accurately simulate latency, LoadImpact also runs tests from remote locations across the world, and allows testers to pick those locations during setup. LoadImpact also permits for the selection of browser types, test criteria and scenarios, number of virtual users, and test duration.

HARDWARE AND VIRTUAL MACHINE CONFIGURATION

- Dell PowerEdge R430
- 2x Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz
- 96GB DDR4-2133 ECC RAM
- 8x 400GB Intel SC SSD Hard drives in RAID-10 in writethrough
- Dell Perc H730 with 1GB cache & BBU

The virtual machine for testing itself was configured with 24GB of RAM, 20 CPU cores, and 400GB of disk space.

SOFTWARE CONFIGURATION

- Magento Open Source version 2.2.2, including the Luma default demo theme and store
- All system and service variables were set as described in the preceding sections
- LetsEncrypt SSL certificate, generated and installed; all testing traffic took place over HTTPS
- Magento 2 was set to the production mode
- Single tenant compiler was run
- Magento 2 sessions cache: Redis
- Magento 2 default cache: Redis
- For full-page cache testing, the FPC was enabled with Redis
- For Varnish testing, Varnish was configured and enabled along with HaProxy for TLS termination
- No external CDN (to allow benchmarking of all assets being delivered from the server)

RESULTS

04

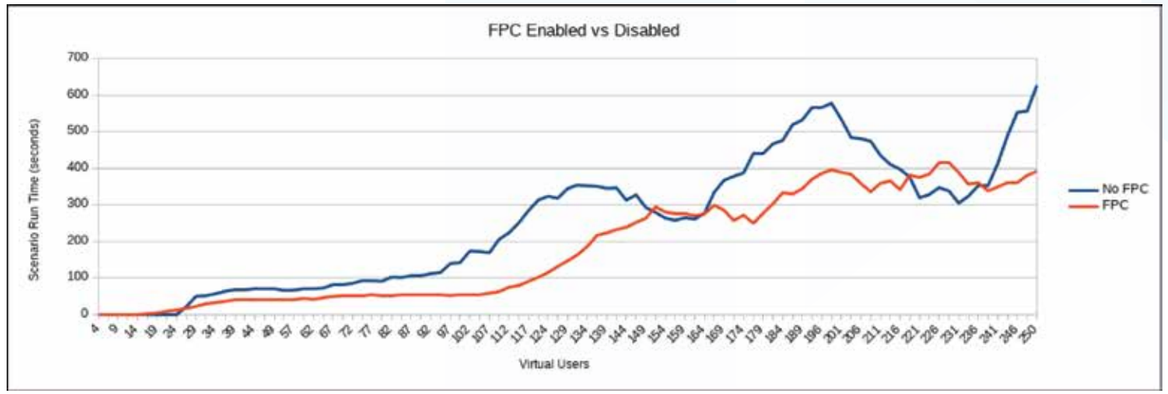
Test 1:
FPC with Redis Versus Disabled FPC

Test 2:
FPC Versus Varnish

Test 3:
Nginx Reverse Proxy Versus FPC Versus Varnish

TEST 1:
FPC WITH REDIS VERSUS DISABLED FPC

The first test reflects a baseline for a store configured on the specified hardware. Graph 1 compares the performance differences from the use of the full-page cache enabled with Redis, versus the full-page cache disabled.

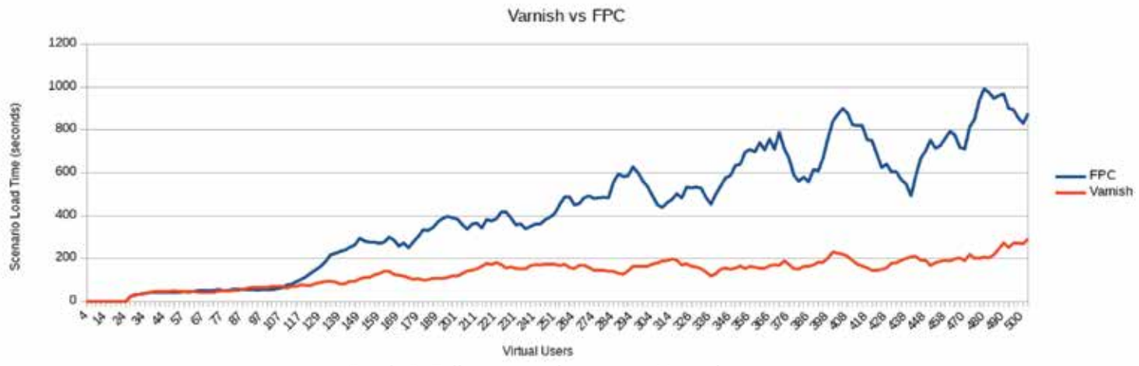


Graph 1: Full-page cache with Redis versus disabled full-page cache.

The results show the advantages of full-page cache. When enabled, run-times show consistency until the breaking point of about 110 virtual users. At this point, performance degrades, and the site will be noticeably less responsive for visitors. Comparing this to running with the FPC disabled, scenario run-times are longer and begin to degrade at roughly 100 virtual users. At this point, scenario run times are double those when using the full-page cache.

TEST 2: FPC VERSUS VARNISH

As shown in Graph 2, FPC can provide good results, but switching to Varnish takes those improvements further. For this test, Magento 2 was configured with Varnish instead of the Redis-based FPC. We also increased the number of virtual users from 250 to 500 in order to stress the configuration with additional user load. HaProxy was used as a TLS terminator, handing http traffic off to Varnish.

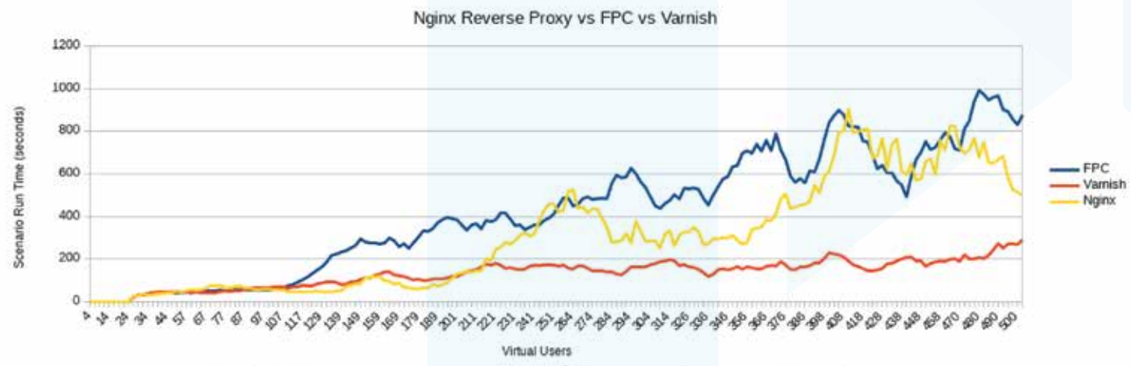


Graph 2: Full-page cache versus Varnish.

The results from this test are impressive. By using Varnish, overall scenario run-times remained relatively consistent. Run-times still increased with the number of virtual users, but at a much lower rate than when those with the Redis-backed FPC. While the use of Varnish does add some additional complexity to the system and Magento 2 configuration, if your hosting environment does support it, these results speak for themselves.

TEST 3: NGINX REVERSE PROXY VERSUS FPC VERSUS VARNISH

The final test was a test using Nginx as a reverse proxy in front of Apache with the microcache enabled. Because Varnish is not feasible for some instances and hosting plans, this test disabled Varnish to focus on the improvements provided by Nginx microcaching alone. Typically, Nginx is easier to implement than Varnish as a reverse proxy for static assets and is a standard component of all Nexcess Cloud plans.



Graph 3: Nginx reverse proxy versus full-page cache versus Varnish.

The results for Nginx microcaching are impressive. The number of virtual users on the site increased from 110 to 200, an 81% improvement over using the FPC alone. This gain on virtual users is made possible simply by enabling Nginx microcaching. Nginx will not help as much as Varnish or the FPC for very large user concurrencies, but Nginx is an way to nearly double store performance with no additional modifications.

BENCHMARKING CAVEATS

Benchmarks are inherently imperfect.

While we try to create tests that accurately simulate traffic, benchmarks are inherently imperfect. Ours is no exception. While our results provide valuable comparisons between different server and Magento 2 configurations, it is impossible to account for all possible variables within the stores themselves. It is prudent to note the following four caveats.

First, the default Magento 2 theme, Luma, has a limited number of categories and products. Because of this limitation, the results may differ from those in a store with a considerable number of categories and products.

Second, our tests declined the use of third party Magento 2 extensions. In real world stores, this is generally not the case. Third party extensions provide significant value and many additional features to a Magento 2 store. That said, not all extensions are created equal, and some can undermine performance. It is best practice to limit their use, and to benchmark the store both before and after enabling any extension.

Third, our tests simulate real visitor traffic, but not truly random visitor traffic. Generating a test that simulates thousands of visitors browsing a site and simulating a real world checkout process at determined conversion ratios is difficult with the LoadImpact tool.

Finally, our testing and resulting benchmarks do not complete the order process. This is a conventional omission due to practical limitations of testing, and we have adopted it here. Processing transactions from start to finish places more load on the server and database, though this load is much more significant when handling large numbers of orders per minute.

RECOMMENDATIONS

Run a modern version of PHP.

Run at least 7.0, and 7.1 is ideal until Magento 2 announces full support for 7.2. Moving from PHP 5 to PHP 7 alone can double your store's performance.

Tune your web stack.

Properly tuning Apache, Nginx, PHP-FPM, MySQL, and Opcache is crucial for high performance. The default options and directives are far from ideal for any Magento 2 store. Proper monitoring of these services is also critical, as some require adjustment as your store grows.

Enable the default and session caches with Redis.

Settings up the default cache and sessions caches to use Redis provides numerous benefits, including better performance and the ability to cluster at scale.

Enable the full-page cache.

Even if a Redis instance is not available for your hosting environment and you must use local files for caching, this is still a considerable improvement over not using the full-page cache at all. The full-page cache pulls load away from both the PHP interpreter and MySQL. Enabling Redis for the full-page cache will increase site performance even further.

Use Varnish for full-page cache.

If your environment supports it, enabling Varnish for the full-page cache over Redis significantly improves performance. In testing, Varnish outperformed Redis as a full-page cache by allowing several hundred more users on a site. The use of Varnish creates additional complexity, but the built-in support within Magento 2 makes it much easier than in the past. Using Varnish will also require a TLS terminator to be effective for both HTTP and HTTPS connections.

Use Nginx as a reverse proxy which provides TLS termination and static-content cache.

Placing Nginx in front of your site and configuring a microcache for static content can improve performance up to 81 percent. With the Nginx microcache enabled, static content such as JavaScript, CSS, and images are cached, providing instant responses to queries and keeping load off the primary webserver. This configuration can benefit any Magento 2 site configuration, with or without Varnish as the FPC. Nginx also handles TLS termination effectively for stores using Varnish.

CONCLUDING SUMMARY

The system and Magento 2 configurations in this guide are based on more than a decade of experience with the Magento and Magento 2 platform.

Stores applying these suggestions will significantly outperform stores that do not. Even so, a "one size fits all" solution is impractical, and many directives and variables specific to your environment must also be taken into consideration.

APPENDIX

ABOUT THE AUTHOR

Brad Boegler is Director of System Operations at Nexcess. With over a decade in systems administration, he oversees our internal systems and was the author of ***Magento Hosting – Best Practices for Optimum Performance***.

Few systems at Nexcess escape his insight. When not monitoring our hosting infrastructure or upholding PCI security standards, Brad celebrates life with his family and plays a mean game of Puerto Rico.

GITHUB RESOURCE

For details regarding all system configurations, visit our GitHub repo at **https://github.com/nexcess/magento-whitepaper-april-2018**.

NOTES

https://github.com/nexcess/magento-whitepaper-april-2018

Nexcess – Beyond Hosting

21700 Melrose Ave
Southfield, MI 48075

Phone: **+1.866.639.2377**
Fax: **+1.248.281.0473**

http://twittter.com/nexcess

NEXCESS
beyond hosting.

sales@nexcess.net