Load Balancing MODX

Garry Nutting, Senior Developer at MODX, LLC

This presentation, while remaining fairly high-level, will demonstrate how to load balance MODX. It is a setup we have rolled out to a few of our clients and I will provide a brief summary of one of them to demonstrate the level of scaling this environment can accommodate.

Of course, there are many, many ways to configure a load balancing environment so please remember that this is not a de-facto standard, just one possible way to do this.

> “Distributing processing and communications activity evenly across a computer network so that no single device is overwhelmed.”
>
> – webopedia.com

This is the simplest description of load balancing I could find and succinctly captures the purpose of load balancing. There are many more - some are great bedtime reading if you’re struggling to sleep. However, there are some deeper aspects that we can derive from this explanation.

In more detail …

- Optimizing resource usage
- Maximizing throughput
- Minimizing response times
- Avoiding overload of any one of the resources
- Improving reliability through redundancy

There are five main facets to load balancing and it's important to realize that not all of these items have to be tackled at once. Typically, your first foray into load balancing will be because you have a need to handle the overloading of resources and dealing with increasing amounts of visitor traffic.

It is sometimes better to treat load balancing as a series of layers, adding features over time (for example, transitioning from a basic load balancing environment to one that has redundancy)

But ...

Load balancing a poorly performing website or codebase is not necessarily solving your problem

There is always a but! Load balancing a poor performing website or codebase is really not solving any of the base performance issues. It is in fact a quick way to spend your hard-earned cash as you will generally need to add more server resources to account for the performance issues.

So …

- Look for problematic snippets and plugins
  - Are snippets uncached when they could be cached? For example, getResources and Wayfinder
  - PHPThumb?
  - Find and fix slow running scripts and queries
    - MySQL slow query log
    - PHP-FPM slow log
    - Cachegrind (beware, “grind” is what it will do)
- Look for errors and 404s in your logs
- Cache!
  - Custom snippets
  - getCache
  - microcache

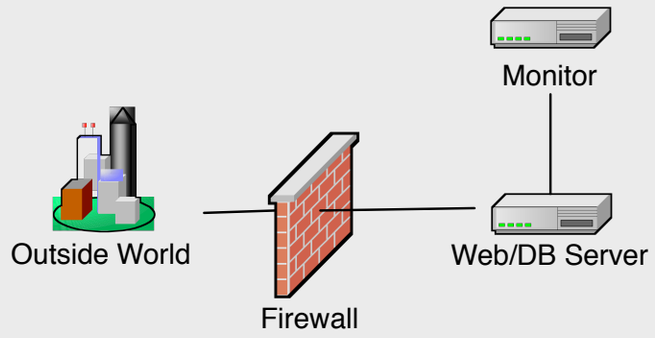
The items outlined on this slide should really be part of an iterative site maintenance process but sometimes once a site goes live, other priorities take over and these items slip. In addition, performance issues can sometimes only become evident once visitor traffic starts to increase.

One of the main things we see in MODX support is people coming to us with problems about page loading times - in most cases, we can correlate the page loading time to uncached snippets that should actually be cached. Wayfinder is one example, it only needs to refresh when a resource is changed in the backend and should always be cached on the frontend.

PHPThumb can be a server killer. A signature of PHPThumb is when a client says, “I keep seeing my site periodically grinding to a halt and after a period of time it returns to normal”. The problem is, PHPThumb has made us lazy and uploading large images may not seem a problem because we are offloading to PHPThumb to reduce the size for us on the frontend. However, image processing in PHP can be CPU intensive.

Using the analogy of vehicles on a road, the CPU usage can be represented by how many vehicles are on the road at any one time. But CPU usage is only part of analysing server performance, another main measure is the load average. In our analogy, the load average can be illustrated by vehicles that are trying to get onto the road. In the case of PHPThumb (for example, if generating multiple thumbnails on a gallery page), PHPThumb can provide a blockage for vehicles trying to get onto the road. As we know, what can start off as a few vehicles waiting to get onto the road can quickly escalate into a

How we started ...

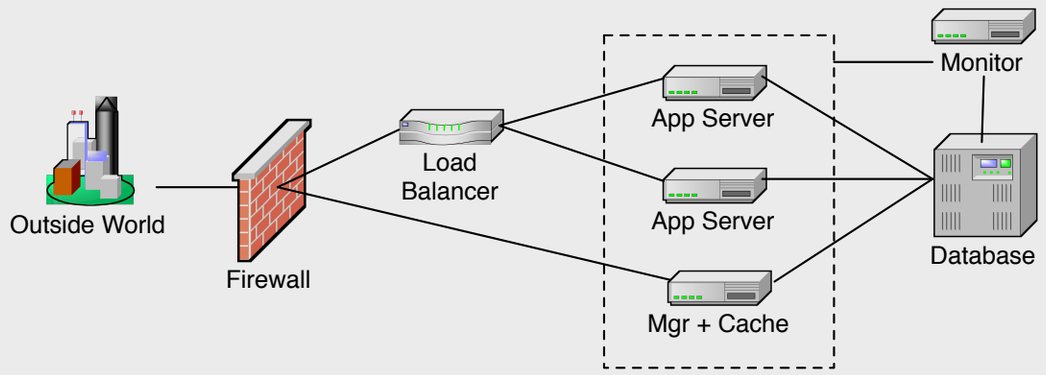


- Host managed firewall
- Combined Web/DB Server - 8 CPU/24GB
- Availability and DB Monitoring Server

So, this was a client's environment three months ago. We couldn't scale the number of CPUs on this one web server because the host caps their VPSes at 8 CPUs. The RAM on the server is 24Gb because of the database (PHP-FPM and NGINX use a relatively small amount of RAM in comparison).

In this setup, we have a dedicated monitoring server - you may not have this but it is highly recommended if you are going to a multi-server environment. There are numerous third-party tools that can also provide this monitoring and may be a cheaper option than a dedicated server.

… and what we wanted



This is the environment we wanted to move to, you'll see that we have three distinct layers to the environment:

1. The database layer.
2. The application layer.
3. The load balancing layer.

The main configuration of this environment actually happens in the application layer - you'll see we have a master instance that also acts as a global cache for MODX. The app servers are what will handle the visitor traffic, the master instance is where site admins and content editors will login to the manager and where any file changes (e.g. uploads) will be conducted.

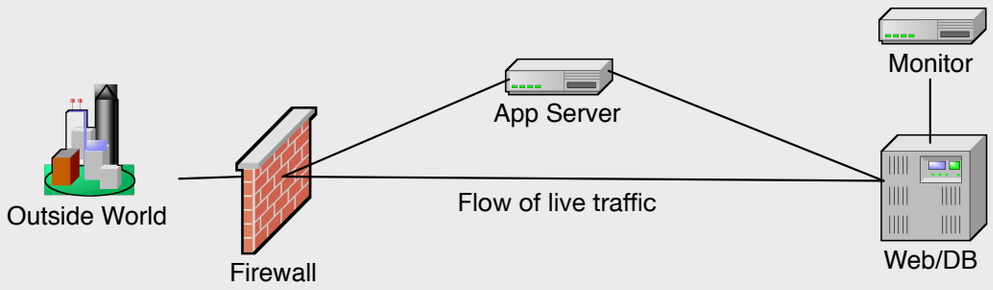
Step 1: Separate the DB

- Commission the first app server: 8CPU/4GB
- Install NGINX and PHP-FPM and copy over configuration files from existing web server
- If changing the number of CPUs from the existing web server, adjust worker_processes value in nginx.conf
- Install PHP-FPM Memcached extension
- Copy over site files keeping file locations the same

The first step is to separate the database. In our case, we decided to actually move the web files onto a newly-procured app server than move the database. The main reason for this is the extensive monitoring in place for the database and the fact that we implemented this environment around the live database.

The app servers can be scaled down (we need CPUs but only a small amount of RAM). In most cases, the configuration files for NGINX and PHP-FPM can be copied over from the existing server with only minor changes required.

As we used memcached for the global cache, it is important to ensure that the memcached PHP extension is installed. Keeping site files in the same location as the live site is also advisable (we don’t want to introduce possible pain points into the implementation).

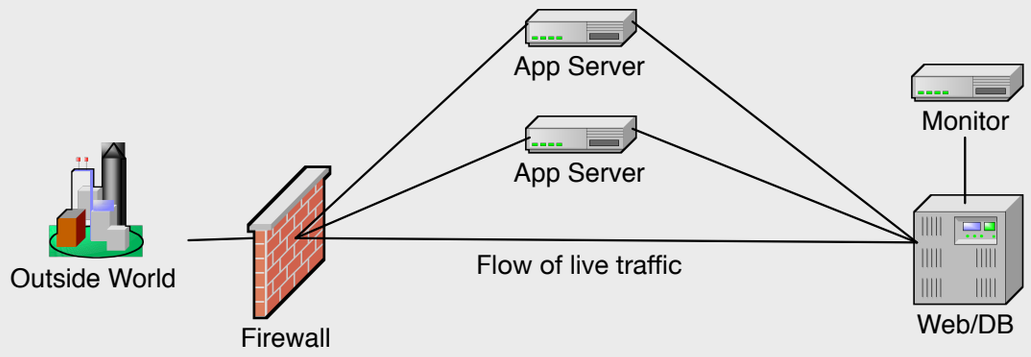


This is the environment once the first step has been implemented. It’s important to note that, during the implementation, we kept the live traffic going to the existing website on what will become the database server.

Step 2: Create second app server

- Repeat step 1 to create the second app server
- Ensure configurations remain identical
- Use host provider's ability to create VPS images if available

This is a repeat of step 1, just remember we want the app servers to be identical. Your host provider may provide tools to make this step easier by providing the ability to take an image of the first app server.



This is the environment once the second step has been implemented.

Step 3: The manager instance

- Smaller size as will not be receiving main visitor traffic: 2CPU/2GB
- Complete setup of PHP-FPM and NGINX as per the app server setup
- Install memcached and rsync
- Configure memcached partitions
- Configure rsync to run between this server and the two app servers

We are now ready to implement the master server and this step will cover implementing the global cache and rsync which is the way that we keep files synchronised between the master server and the two app servers.

memcached

- Caches to partitions stored in RAM
- A single cache entry is limited to 1Mb in size
- You can only delete individual entries or flush the entire partition
- Restarting memcached will flush all the partitions
- No dynamic resizing of partitions
- Memcached will use up to the memory limit specified for each partition. Once a partition is full, old entries will start to be removed to accommodate new cache entries
- If caching is critical to the application, consider having a script to prime the cache if a restart is required

These are some pertinent points about memcached.

The main thing to remember is that memcached is only resident in RAM so any restarts will clear the cache (a priming script may be an option but you will have to weigh up the consequences of this based on your specific application, priming a cache could introduce higher loads during the time it is running)

To determine the size of your memcached partitions, you can look at the core/cache folder and the folder sizes inside there. The default partition size for memcached is 64Mb but it's important to remember that this is not pre-allocated so aiming high and setting partition sizes slightly higher than you may need is fine in most cases.

Cache Partition	MODX System Setting	Configuration Location	Size	Port(s)
Default	memcached_server	/etc/memcached_default.conf	64M	11211
Resource	resource_memcached_server	/etc/memcached_resource.conf	64M	11212
Scripts & Includes	scripts_memcached_server includes_memcached_server	/etc/memcached_scripts.conf	64M	11213
System Settings	system_settings_memcached_server	/etc/memcached_system_settings.conf	32M	11214
Context Settings	context_settings_memcached_server	/etc/memcached_context_settings.conf	32M	11215
Custom	custom_memcached_server	/etc/memcached_custom.conf	384M	11216 - 11221

This slide illustrates a sample memcached setup, each partition has a related MODX system setting.

The partition size and ports are specified in the memcached configuration files. Just remember to check that these ports are open between your master instance and the app servers.

If you don’t specify a separate partition for a MODX-generated cache partition then MODX will cache that information in the default partition. For example, the action_map partition really doesn’t solicit it’s own partition due to it’s small size.

- The MODX system settings values are added to the MODX configuration file
- Ensures the cache settings are loaded before the database connection is established
- Each partition setting value takes the IP address and port of the master server
- Ensure that these ports are open between the master server and the app servers.

```
$config_options = array (
  'cache_handler' => 'cache.xPDOMemCached',
  'memcached_server' => 'localhost:11211',
  'resources_memcached_server' => 'localhost:11212',
  'scripts_memcached_server' => 'localhost:11213',
  'includes_memcached_server' => 'localhost:11213',
  'system_settings_memcached_server' => 'localhost:11214',
  'context_settings_memcached_server' => 'localhost:11215',
  'custom_memcached_server' => ‘localhost:11216,localhost:11217,localhost:11218,localhost:
11219,localhost:11220,localhost:11221'
);
```

The memcached system settings are actually added to the MODX config.inc.php because we want the cache settings to take effect before a database connection is made. The settings are added to the $config_options array.

With those configuration options in place, you will now have MODX caching to memcached. That’s all there is to it!

rsync

- rsync runs over SSH at a specified interval - for this environment, it's every 1min
- If you need a shorter sync interval, consider lsyncd which is a low-level wrapper for rsync
- Configure password-less SSH login between the master server and app servers
  - Use ssh-keygen to create a private/public key on the master server
  - Use ssh-copy-id to copy over the public key over to each of the app servers
  - Test the SSH login
- Create bash script to run rsync and setup as a cron job to run every minute

We decided that rsync was fine for the client's environment running via a cron job at 1 minute intervals (the site does not have a lot of file updates or uploads).

Faster (almost instantaneous) sync times can be achieved by using lsyncd. This is a wrapper for rsync that listens to low-level directory events in Linux and so can synchronise a file across the servers as soon as it changes - the delay is in the order of a few milliseconds.

In order for rsync to work, you will need to create a password-less SSH login between the master instance and the two app servers. This is not difficult to do but I would recommend you test the SSH login prior to setting up rsync in your crontab.

We decided to use a simple bash script to handle the rsync command, it just provides a simple way to add in additional app servers should we need them in the future.

```
#!/bin/bash
APPSERVERIPS=(
   xxx.xxx.xxx.xxx
   xxx.xxx.xxx.xxx
)

for IP in ${APPSERVERIPS[*]}
do
   rsync \
       --verbose \
       --progress \
       --include-from ‘/root/rsyncInclude.txt’ \
       --exclude-from ‘/root/rsyncExclude.txt’ \
       --stats \
       --compress \
       --rsh=/usr/bin/ssh \
       --recursive \
       --times \
       --perms \
       --owner \
       --group \
       --links \
       --delete \
       -e "ssh -i /root/.ssh/id_rsa"  /my/path/to/webroot/  $IP: /my/path/to/webroot/
done


* * * * * /bin/bash /root/rsync_to_app_servers.sh > /dev/null 2>&1
```

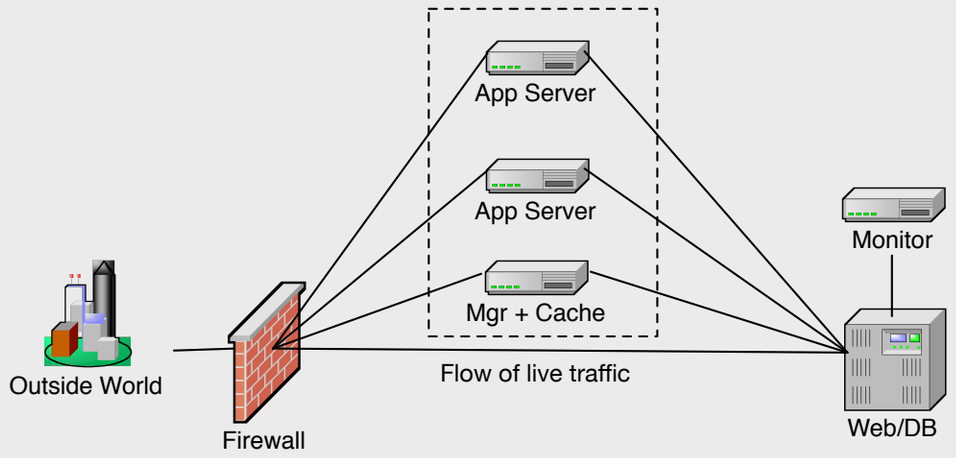
As you can see, the bash script is essentially a simple loop, iterating over each app server IP address and running rsync.

We set up a few configuration parameters for the rsync command, some are self-explanatory but these are some to bear in mind:

1. The include-from and exclude-from parameters can include additional files or directories (e.g. you have your core folder outside the web root) or exclude files or directories (e.g. the core/cache/* folders). Wildcards are accepted in the directory or file paths.
2. The perms, owner and group parameters retain the permissions, you’ll need these to avoid possible access issues to synchronised files.
3. The final line includes the location of our SSH private key, the source folder and the destination folder.

Remember, after creating your bash script file, you need to make it executable: chmod +x /root/rsync_to_app_servers.sh

The line in red on this slide is what you would add to your crontab to have this script run at 1min intervals.



So now, we are close to having a load balancing environment to test. The load balancer is the last piece of the load balancing stack to get in place.

Step 4: The load balancer

- Many different load balancers: zeus load appliance, NGINX
- Several types of load balancing algorithms
- Session persistance, i.e. “sticky sessions”
- How to handle IP forwarding
- Some load balancers have SSL termination

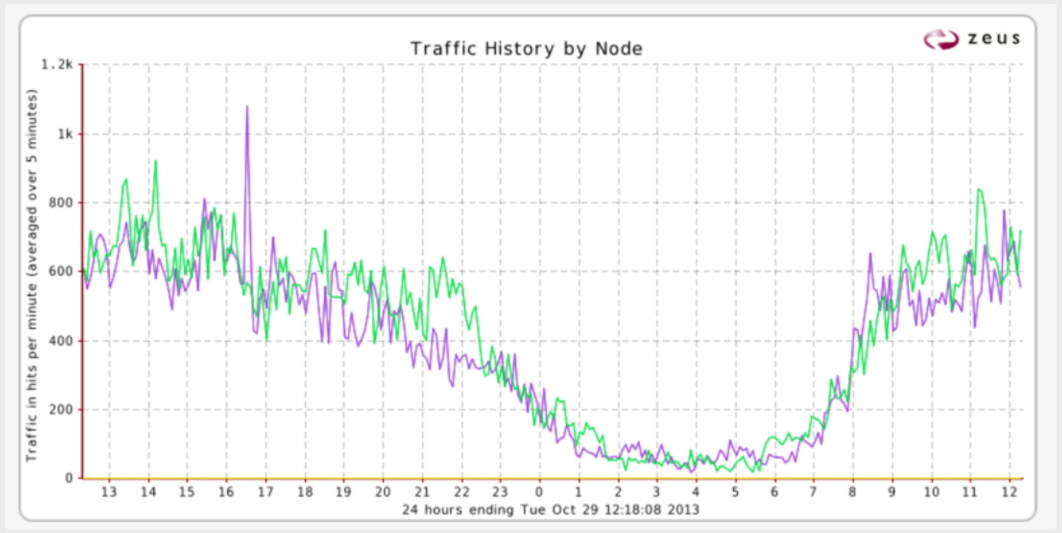
I will discuss two load balancing options to give an idea of the differences between load balancing appliances. The Zeus load balancer has a full web UI whereas NGINX is configuration-file based, also Zeus does have more advanced options for load balancing available than the basic NGINX load balancing implementation.

The next few slides will not only show the differences in those two load balancing options but will outline considerations for each of the items listed on this slide.

- Round Robin
  - Simplest form of load balancing
  - Equal distribution of traffic across available app server nodes
  - Does not take into account how busy an app server node may be
- Least Connections
  - Directs traffic to the app server node with fewest connections
  - Session persistance can work against this algorithm
- Fastest Response Time
  - Directs traffic to the app server node with the fastest response time
  - Does not take into account available server resources
- Perceptive
  - Predict the most appropriate node using a combination of historical and current data
  - Takes time to build historical data so load balancing effectiveness is reduced after restarts

These are the different types of load balancing. We decided to use least connections (with session persistence) for the client.

Despite session persistence (the ability to route a specific visitor to a specific app server) being used, the next slide shows a graph showing that the load balancing is actually quite well-balanced.



Just an example of the least connection load balancing over a 24hr period. The spike around 16:30 is actually fine, it's explainable by events in the application at the time and not a problem with the load balancing.

- Session persistance ensures traffic from a specific visitor is always directed to the same app server
- Should not be an issue with database sessions but could be if using file-based sessions
- Numerous types of session persistance algorithms are available:
  - IP-based: Send all requests from the same source address to the same node.
  - Transparent session affinity: Insert a cookie into the response to track sessions.
  - Application Cookie: Monitor a specified application cookie to identify sessions.
- Define what happens if the app server node is not available, e.g. choose new node or close connection

Session persistence could be important to consider if you are using file-based sessions, it shouldn't be a problem if you are using database sessions.

There are a couple of ways that session persistence can be handled, the simplest being that the load balancer will log the IP address of the visitor and use that to route them to a specific app server. This is the only option for session persistence that NGINX supports without requiring a commercial subscription.

Transparent session affinity, although sounding complex, is really just inserting a cookie for tracking purposes. An application generated cookie can also be used on the Zeus load balancer if available.

When implementing session persistence, you will need to define what happens if the app server a visitor is using becomes unavailable. In most cases, redirecting to a different app server is okay although logged in visitors may need to re-authenticate.

- By default, the source IP address of a request hitting the app server will be the load balancer IP
- Use the X_FORWARDED_FOR header to pass the visitor's IP in the request
- Ensure your NGINX access logs are configured to log the X_FORWARDED_FOR IP address
- In nginx.conf, under the http section:

```
log_format  main '$remote_addr - $remote_user [$time_local] "$request" '
                 '$status $body_bytes_sent "$http_referer" '
                 '"$http_user_agent" "$http_x_forwarded_for"';
```

IP forwarding is needed because the remote address variable passed in the request will be the load balancer IP address by the time the request hits one of the app servers. In most cases, you will want to be capturing the visitor IP address for your server access logs amongst other reasons.

This can be achieved by using the X_FORWARDED_FOR header. You can tell the load balancer to populate the visitor's IP address into this header before the request is forwarded to the app server.

In NGINX, you will need to make a small change to add the header assignment into the access log format which is shown at the bottom of the slide.

Configuring the Zeus LB

- Traffic Manager: This represents the physical LB appliance.
- Traffic IP Group: This allows a Traffic Manager to be assigned to a network interface(s), i.e. an IP address
- Virtual Servers: The virtual server will accept traffic on the specified port for processing:
  - HTTP: Listens on port 80. Adds the HTTP_X_FORWARDED_PROTO header and sets it to HTTP. Sets the X_FORWARDED_FOR to the requester's IP. The request is then forwarded onto the HTTP pool.
  - HTTPS: Listens on port 443. Decrypt SSL traffic for communication with backend app servers on port 80. Adds the HTTP_X_FORWARDED_PROTO header and sets it to HTTPS. Sets the X_FORWARDED_FOR to the requester's IP. The request is then forwarded onto the HTTP pool.
- Pools: One pool for HTTP. The app server IPs can easily be changed/added/removed from here.
- Catalogs: Catalogs contain a variety of items for the LB configuration. The only one of importance is the SSL Catalog, this is where the SSL certificate is uploaded and assigned to the HTTPS virtual server.

This slide explains some of the terminology used by the Zeus load balancer. The process of setting up the Zeus load balancer in their web UI also follows this slide from top to bottom. Although each different make of load balancer will be configured in different ways, a lot of the terminology described on this slide will have direct parallels on other systems.

The main thing here is HTTP_X_FORWARDED_PROTO header being added in the virtual servers. This is another custom header and identifies whether the original request was HTTPS or not. We need this as we're using SSL termination (SSL termination is discussed in detail later in the presentation)

Configuring the NGINX LB

- Not as feature packed as a dedicated load balancing appliance
- Many of the advanced features being introduced are for commercial subscriptions only
- Will need to have the NGINX Upstream module installed
- Will need to commission a server with NGINX installed: 2CPU/2GB

NGINX can be used as a simple load balancer however it currently doesn't have a full feature set compared to dedicated load balancing appliances. The more advanced features appearing in NGINX also require a commercial subscription however the approach discussed in this presentation can be achieved with a free NGINX install (remember to check the Upstream module is compiled into NGINX).

If the upstream module isn't compiled, a lot of Linux distros supply an nginx-extra package which contains all of the available modules.

You will need a server to place NGINX on, I've estimated a 2CPU/2Gb would be fine (it's what the Zeus load balancer runs on). As this server is essentially just routing traffic and not doing the processing, we shouldn't need a high specification server.

```
upstream backend {
    ip_hash;
    least_conn;

    server xxx.xxx.xxx.xxx max_fails=3 fail_timeout=15s weight=1;
    server xxx.xxx.xxx.xxx max_fails=3 fail_timeout=15s weight=2;
    server xxx.xxx.xxx.xxx max_fails=3 fail_timeout=15s backup;
}

server {
    listen 80;

    location / {
        proxy_pass http://backend;
        proxy_set_header X-Forwarded-Proto http;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}

server {
    listen 443;
    ssl on;
    ssl_certificate /etc/nginx/ssl/server.crt;
    ssl_certificate_key /etc/nginx/ssl/server.key;

    location / {
        proxy_pass http://backend;
        proxy_set_header X-Forwarded-Proto https;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

This is a boiler-plate NGINX site configuration that is equivalent to the items outlined on the Zeus load balancer slide.

The upstream block is the direct equivalent of the Zeus's pool definition. Here, the ip_hash command tells NGINX to use IP-based session persistence and the least_conn command tells NGINX to use the least connections method of load balancing (without this, round robin would be used).

Below these two commands, the server statements provide the app server IP addresses followed by some additional configuration for each app server:

1. max_fails and fail_timeout: Maximum number of connection failures allowed within the fail_timeout. In the case of this maximum being reached, NGINX will not send any more requests to the app server for the time period, again specified by fail_timeout.
2. weight: The weight can be used to send more or less traffic to a specific app server. You may want to use this if your app servers have differing amounts of CPUs.
3. backup: The backup command signifies that that app server will only be used if one of the other app servers becomes unavailable. Not shown on the slide is the 'down' statement which will mark an app server as being unavailable (e.g. if you need to take an app server down for maintenance), remember to reload NGINX if you add that to any of the app servers.

App Server SSL Termination

In your nginx.conf, inside http{} definition:

```
map $http_x_forwarded_proto $fastcgi_https {
   default off;
   https on;
}
```

In your NGINX site configuration, inside location{} definition:

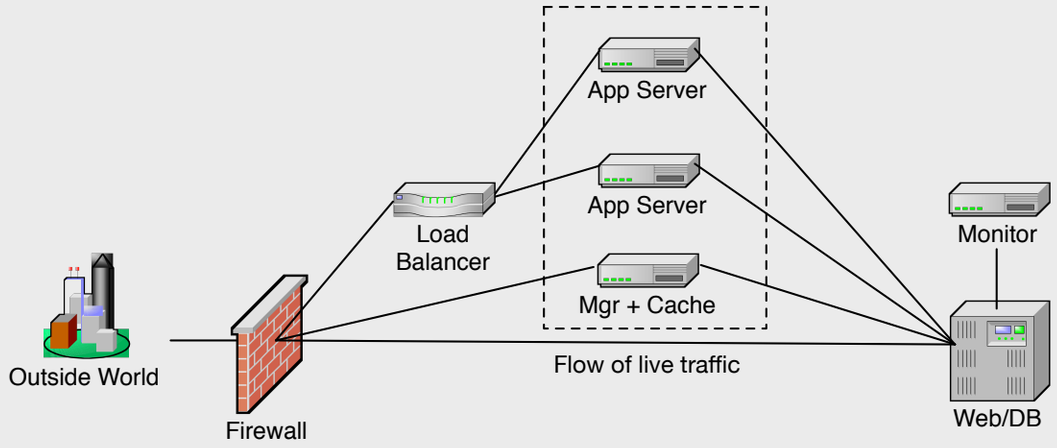
```
fastcgi_param HTTPS $fastcgi_https;
```

This way MODX will know if the original request was HTTPS.

MODX needs to know if the original request was sent over SSL or not (one example is so it can use the correct scheme when formatting link tags).

As we are sending the X_FORWARDED_PROTO header in the request to the app server, we can use this in conjunction with the NGINX map function to correctly assign our HTTPS header (which will then be handed over to PHP-FPM and subsequently MODX).

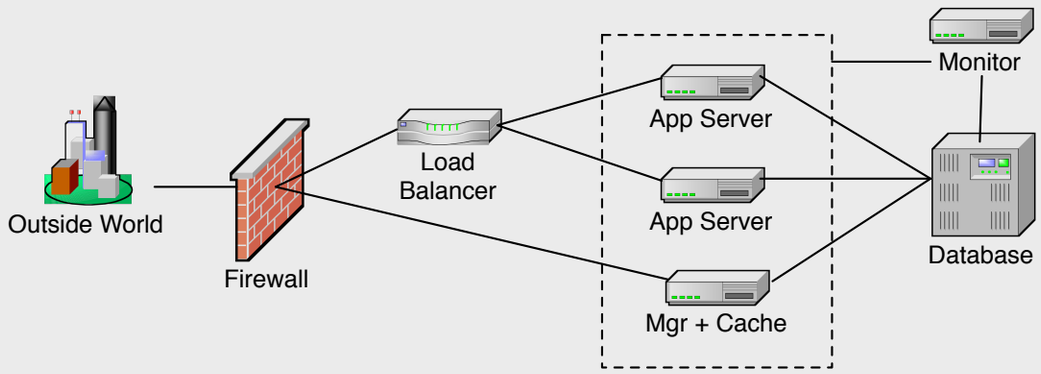
In the location block (where you're sending data over to PHP-FPM, we need to ensure we are passing our HTTPS header as a parameter, this may or may not already exist in your NGINX fastcgi_params configuration file.



So, this is the environment prior to go-live, at this point you can test your environment.

Immediately prior to go-live, remember to synchronise your web files from the existing live site (which will be decommissioned after go-live) to the master instance. This is also a good test to see if the master instance synchronises the files to the app servers.

And we're there!



After testing, you can go-live. This is essentially a two-step process:

1. Change your site's DNS entry to point to the load balancer IP address.
2. Remove/shutdown the website on the database server.

And that's it!

A couple of the questions were surrounding the application layer.

There was a discussion about using Redis instead of memcached for the global cache as Redis is persistent and therefore will retain the cache data on restarts. This is an interesting proposition and while we decided that memcached was okay to use for this environment, this is definitely worth further investigation to determine what would be required to implement Redis for MODX caching.

Another question focussed on rsync and if there could be a problem if an rsync process took longer than a minute to synchronise (i.e. possible overlapping of file syncing) - it is to my knowledge that lsyncd would avoid this.

There was a question on how to redirect the manager traffic to the manager instance. This can be handled on the load balancer, both Zeus and NGINX (and therefore I assume other load balancer appliances will too) support the ability to do a forced redirect based on the URL.

Final question was on whether (to avoid the rsync) NFS could be used instead. NFS is an option for centralised file storage (mounting the storage on each of the app servers and having the web root on the NFS). This may be an option for new sites but may be difficult to implement on existing sites, MODX has had issues with file locking on NFS mounts previously too so we need to bear that in mind. Finally, it does add another component to the overall load