Understanding and Implementing FastCGI Proxying in Nginx

Introduction

Nginx has become one of the most flexible and powerful web server solutions available. However, in terms of design, it is first and foremost a proxy server. This focus means that Nginx is very performant when working to handle requests with other servers.

Nginx can proxy requests using http, FastCGI, uwsgi, SCGI, or memcached. In this guide, we will discuss FastCGI proxying, which is one of the most common proxying protocols.

Why Use FastCGI Proxying?

FastCGI proxying within Nginx is generally used to translate client requests for an application server that does not or should not handle client requests directly. FastCGI is a protocol based on the earlier CGI, or common gateway interface, protocol meant to improve performance by not running each request as a separate process. It is used to efficiently interface with a server that processes requests for dynamic content.

One of the main use-cases of FastCGI proxying within Nginx is for PHP processing. Unlike Apache, which can handle PHP processing directly with the use of the `mod_php` module, Nginx must rely on a separate PHP processor to handle PHP requests. Most often, this processing is handled with `php-fpm`, a PHP processor that has been extensively tested to work with Nginx.

Nginx with FastCGI can be used with applications using other languages so long as there

is an accessible component configured to respond to FastCGI requests.

FastCGI Proxying Basics

In general, proxying requests involves the proxy server, in this case Nginx, forwarding requests from clients to a backend server. The directive that Nginx uses to define the actual server to proxy to using the FastCGI protocol is `fastcgi_pass`.

For example, to forward any matching requests for PHP to a backend devoted to handling PHP processing using the FastCGI protocol, a basic location block may look something like this:

```
# server context

location ~ \.php$ {
    fastcgi_pass 127.0.0.1:9000;
}

. . .
```

The above snippet won't actually work out of the box, because it gives too little information. Any time that a proxy connection is made, the original request must be translated to ensure that the proxied request makes sense to the backend server. Since we are changing protocols with a FastCGI pass, this involves some additional work.

While http-to-http proxying mainly involves augmenting http headers to ensure that the backend has the information it needs to respond to the proxy server on behalf of the client, FastCGI is a separate protocol that cannot read http headers. Due to this consideration, any pertinent information must be passed to the backend through other means.

The primary method of passing extra information when using the FastCGI protocol is with parameters. The background server should be configured to read and process these, modifying its behavior depending on what it finds. Nginx can set FastCGI parameters using the `fastcgi_param` directive.

The bare minimum configuration that will actually work in a FastCGI proxying scenario for

PHP is something like this:

```
# server context

location ~ \.php$ {
    fastcgi_param REQUEST_METHOD $request_method;
    fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    fastcgi_pass 127.0.0.1:9000;
}

. . .
```

In the above configuration, we set two FastCGI parameters, called `REQUEST_METHOD` and `SCRIPT_FILENAME`. These are both required in order for the backend server to understand the nature of the request. The former tells it what type of operation it should be performing, while the latter tell the upstream which file to execute.

In the example, we used some Nginx variables to set the values of these parameters. The `$request_method` variable will always contain the http method requested by the client.

The `SCRIPT_FILENAME` parameter is set to a combination of the `$document_root` variable and the `$fastcgi_script_name` variable. The `$document_root` will contain the path to the base directory, as set by the `root` directive. The `$fastcgi_script_name` variable will be set to the request URI. If the request URI ends with a slash (/), the value of the `fastcgi_index` directive will be appended onto the end. This type of self-referential location definitions are possible because we are running the FastCGI processor on the same machine as our Nginx instance.

Let's look at another example:

```
# server context
root /var/www/html;

location /scripts {
    fastcgi_param REQUEST_METHOD $request_method;
    fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    fastcgi_index index.php;
    fastcgi_pass unix:/var/run/php5-fpm.sock;
```

```
    }

    . . .
```

If the location above is selected to handle a request for `/scripts/test/`, the value of the `SCRIPT_FILENAME` will be a combination of the values of the `root` directive, the request URI, and the `fastcgi_index` directive. In this example, the parameter will be set to `/var/www/html/scripts/test/index.php`.

We made one other significant change in the configuration above in that we specified the FastCGI backend using a Unix socket instead of a network socket. Nginx can use either type of interface to connect to the FastCGI upstream. If the FastCGI processor lives on the same host, typically a Unix socket is recommended for security.

Breaking Out FastCGI Configuration

A key rule for maintainable code is to try to follow the DRY ("Don't Repeat Yourself") principle. This helps reduce errors, increase reusability, and allows for better organization. Considering that one of the core recommendations for administering Nginx is to always set directives at their broadest applicable scope, these fundamental goals also apply to Nginx configuration.

When dealing with FastCGI proxy configurations, most instances of use will share a large majority of the configuration. Because of this and because of the way that the Nginx inheritance model works, it is almost always advantageous to declare parameters in a general scope.

Declaring FastCGI Configuration Details in Parent Contexts

higher, parent
fied at higher
ccurs. This means
that multiple locations can use the same config.

For instance, we could modify the last configuration snippet from the above section to make it useful in more than one location:

```
# server context
root /var/www/html;

fastcgi_param REQUEST_METHOD $request_method;
fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
fastcgi_index index.php;

location /scripts {
    fastcgi_pass unix:/var/run/php5-fpm.sock;
}

location ~ \.php$ {
    fastcgi_pass 127.0.0.1:9000;
}

. . .
```

In the above example, both of the `fastcgi_param` declarations and the `fastcgi_index` directive are available in both of the location blocks that come after. This is one way to remove repetitive declarations.

However, the configuration above has one serious disadvantage. If any `fastcgi_param` is declared in the lower context, *none* of the `fastcgi_param` values from the parent context will be inherited. You either use *only* the inherited values, or you use none of them.

The `fastcgi_param` directive is an *array* directive in Nginx parlance. From a users perspective, an array directive is basically any directive that can be used more than once in a single context. Each subsequent declaration will append the new information to what Nginx knows from the pr[illegible] The `fastcgi_param` directive was designed as an array directive in order to allow users to set multiple parameters.

Array directives inherit to child contexts in a different way than some other directives. The information from array directives will inherit to child contexts *only if they are not present at any place in the child context*. This means that if you use `fastcgi_param` within your location, it will effectively clear out the values inherited from the parent context completely.

For example, we could modify the above configuration slightly:

```
# server context
root /var/www/html;

fastcgi_param REQUEST_METHOD $request_method;
fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
fastcgi_index index.php;

location /scripts {
    fastcgi_pass unix:/var/run/php5-fpm.sock;
}

location ~ \.php$ {
    fastcgi_param QUERY_STRING $query_string;
    fastcgi_pass 127.0.0.1:9000;
}

. . .
```

At first glance, you may think that the `REQUEST_METHOD` and `SCRIPT_FILENAME` parameters will be inherited into the second location block, with the `QUERY_STRING` parameter being additionally available for that specific context.

What actually happens is that *all* of the parent `fastcgi_param` values are wiped out in the second context, and only the `QUERY_STRING` parameter is set. The `REQUEST_METHOD` and `SCRIPT_FILENAME` parameters will remain unset.

A Note About Multiple Values for Parameters in the Same Context

One thing that is definitely worth mentioning at this point is the implications of setting multiple values for the same parameters within a single context. Let's take the following example as a discussion point:

```
# server context

location ~ \.php$ {
    fastcgi_param REQUEST_METHOD $request_method;
    fastcgi_param SCRIPT_FILENAME $request_uri;

    fastcgi_param DOCUMENT_ROOT initial;
    fastcgi_param DOCUMENT_ROOT override;
```

```
        fastcgi_param TEST one;
        fastcgi_param TEST two;
        fastcgi_param TEST three;

        fastcgi_pass 127.0.0.1:9000;
    }

    . . .
```

In the above example, we have set the `TEST` and `DOCUMENT_ROOT` parameters multiple times within a single context. Since `fastcgi_param` is an array directive, each subsequent declaration is added to Nginx's internal records. The `TEST` parameter will have declarations in the array setting it to `one`, `two`, and `three`.

What is important to realize at this point is that all of these will be passed to the FastCGI backend without any further processing from Nginx. This means that it is completely up to the chosen FastCGI processor to decide how to handle these values. Unfortunately, different FastCGI processors handle the passed values completely differently.

For instance, if the above parameters were received by PHP-FPM, the *final* value would be interpreted to override any of the previous values. So in this case, the `TEST` parameter would be set to `three`. Similarly, the `DOCUMENT_ROOT` parameter would be set to `override`.

However, if the above value is passed to something like FsgiWrap, the values are interpreted very differently. First, it makes an initial pass to decide which values to use to run the script. It will use the `DOCUMENT_ROOT` value of `initial` to look for the script. However, when it passes the actual parameters to the script, it will pass the final values, just like PHP-FPM.

This inconsistency and unpredictability means that you cannot and should not rely on the backend to correctly interpret your intentions when setting the same parameter more than one time. The only safe solution is to only declare each parameter once. This also means that there is no such thing as safely overriding a default value with the `fastcgi_param` directive.

Using Include to Source FastCGI Configuration from a Separate File

There is another way to separate out your common configuration items. We can use the `include` directive to read in the contents of a separate file to the location of the directive declaration.

This means that we can keep all of our common configuration items in a single file and include it anywhere in our configuration where we need it. Since Nginx will place the actual file contents where the `include` is called, we will not be inheriting downward from a parent context to a child. This will prevent the `fastcgi_param` values from being wiped out, allowing us to set additional parameters as necessary.

First, we can set our common FastCGI configuration values in a separate file in our configuration directory. We will call this file `fastcgi_common`:

```
fastcgi_param REQUEST_METHOD $request_method;
fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
```

Now, we can read in this file wherever we wish to use those configuration values:

```
# server context
root /var/www/html;

location /scripts {
    include fastcgi_common;

    fastcgi_index index.php;
    fastcgi_pass unix:/var/run/php5-fpm.sock;
}

location ~ \.php$ {
    include fastcgi_common;
    fastcgi_param QUERY_STRING $query_string;
    fastcgi_param CONTENT_TYPE $content_type;
    fastcgi_param CONTENT_LENGTH $content_length;

    fastcgi_index index.php;
    fastcgi_pass 127.0.0.1:9000;
}

. . .
```

Here, we have moved some common `fastcgi_param` values to a file called `fastcgi_common` in our default Nginx configuration directory. We then source that file when we want to insert the values declared within.

There are a few things to note about this configuration.

The first thing is that we did not place any values that we may wish to customize on a per-location basis in the file we plan to source. Because of the problem with interpretation we mentioned above that occurs when setting multiple values for the same parameter, and because non-array directives can only be set once per context, only place items the common file that you will not want to change. Every directive (or parameter key) that we may wish to customize on a per-context basis should be left out of the common file.

The other thing that you may have noticed is that we set some additional FastCGI parameters in the second location block. This is the ability we were hoping to achieve. We were able to set additional `fastcgi_param` parameters as needed, without wiping out the common values.

Using the fastcgi_params File or the fastcgi.conf File

With the above strategy in mind, the Nginx developers and many distribution packaging teams have worked towards providing a sane set of common parameters that you can include in your FastCGI pass locations. These are called `fastcgi_params` or `fastcgi.conf`.

These two files are largely the same, with the only difference actually being a consequence of the issue we discussed earlier about passing multiple values for a single parameter. The `fastcgi_params` file does not contain a declaration for the `SCRIPT_FILENAME` parameter, while the `fastcgi.conf` file does.

The `fastcgi_params` file has been available for a much longer period of time. In order avoid breaking configurations that relied on `fastcgi_params`, when the decision was made to provide a default value for `SCRIPT_FILENAME`, a new file needed to be created. Not doing so may have resulted in that parameter being set in both the common file and FastCGI pass location. This is described in great detail in Martin Fjordvald's excellent post on the history of these two files.

Many package maintainers for popular distributions have elected to include only one of

these files or to mirror their content exactly. If you only have one of these available, use the one that you have. Feel free to modify it to suit your needs as well.

If you have both of these files available to you, for most FastCGI pass locations, it is probably better to include the `fastcgi.conf` file, as it includes a declaration for the `SCRIPT_FILENAME` parameter. This is usually desirable, but there are some instances where you may wish to customize this value.

These can be included by referencing their location relative to the root Nginx configuration directory. The root Nginx configuration directory is usually something like `/etc/nginx` when Nginx has been installed with a package manager.

You can include the files like this:

```
# server context

location ~ \.php$ {
    include fastcgi_params;
    # You would use "fastcgi_param SCRIPT_FILENAME . . ." here afterwards

    . . .

}
```

Or like this:

```
# server context

location ~ \.php$ {
    include fastcgi.conf;

    . . .

}
```

Important FastCGI Directives, Parameters, and Variables

In the above sections, we've set a fair number of parameters, often to Nginx variables, as a means of demonstrating other concepts. We have also introduced some FastCGI

directives without too much explanation. In this section, we'll discuss some of the common directives to set, parameters that you might need to modify, and some variables that might contain the information you need.

Common FastCGI Directives

The following represent some of the most useful directives for working with FastCGI passes:

- `fastcgi_pass`: The actual directive that passes requests in the current context to the backend. This defines the location where the FastCGI processor can be reached.
- `fastcgi_param`: The array directive that can be used to set parameters to values. Most often, this is used in conjunction with Nginx variables to set FastCGI parameters to values specific to the request.
- `try_files`: Not a FastCGI-specific directive, but a common directive used within FastCGI pass locations. This is often used as part of a request sanitation routine to make sure that the requested file exists before passing it to the FastCGI processor.
- `include`: Again, not a FastCGI-specific directive, but one that gets heavy usage in FastCGI pass contexts. Most often, this is used to include common, shared configuration details in multiple locations.
- `fastcgi_split_path_info`: This directive defines a regular expression with two captured groups. The first captured group is used as the value for the `$fastcgi_script_name` variable. The second captured group is used as the value for the `$fastcgi_path_info` variable. Both of these are often used to correctly parse the request so that the processor knows which pieces of the request are the files to run and which portions are additional information to pass to the script.
- `fastcgi_index`: This defines the index file that should be appended to `$fastcgi_script_name` values that end with a slash (`/`). This is often useful if the `SCRIPT_FILENAME` parameter is set to `$document_root$fastcgi_script_name` and the location block is configured to accept requests with info after the file.
- `fastcgi_intercept_errors`: This directive defines whether errors received from the FastCGI server should be handled by Nginx or passed directly to the client.

The above directives represent most of what you will be using when designing a typical FastCGI pass. You may not use all of these all of the time, but we can begin to see that they interact quite intimately with the FastCGI parameters and variables that we will talk about next.

Common Variables Used with FastCGI

Before we can talk about the parameters that you are likely to use with FastCGI passes, we should talk a bit about some common Nginx variables that we will take advantage of in setting those parameters. Some of these are defined by Nginx's FastCGI module, but most are from the Core module.

- `$query_string` **or** `$args`: The arguments given in the original client request.
- `$is_args`: Will equal "?" if there are arguments in the request and will be set to an empty string otherwise. This is useful when constructing parameters that may or may not have arguments.
- `$request_method`: This indicates the original client request method. This can be useful in determining whether an operation should be permitted within the current context.
- `$content_type`: This is set to the `Content-Type` request header. This information is needed by the proxy if the user's request is a POST in order to correctly handle the content that follows.
- `$content_length`: This is set to the value of the `Content-Length` header from the client. This information is required for any client POST requests.
- `$fastcgi_script_name`: This will contain the script file to be run. If the request ends in a slash (/), the value of the `fastcgi_index` directive will be appended to the end. In the event that the `fastcgi_split_path_info` directive is used, this variable will be set to the first captured group defined by that directive. The value of this variable should indicate the actual script to be run.
- `$request_filename`: This variable will contain the file path for the requested file. It gets this value by taking the value of the current document root, taking into account both the `root` and `alias` directives, and the value of `$fastcgi_script_name`. This is a very flexible way of assigning the `SCRIPT_FILENAME` parameter.

- `$request_uri`: The entire request as received from the client. This includes the script, any additional path info, plus any query strings.
- `$fastcgi_path_info`: This variable contains additional path info that may be available after the script name in the request. This value sometimes contains another location that the script to execute should know about. This variable gets its value from the second captured regex group when using the `fastcgi_split_path_info` directive.
- `$document_root`: This variable contains the current document root value. This will be set according to the `root` or `alias` directives.
- `$uri`: This variable contains the current URI with normalization applied. Since certain directives that rewrite or internally redirect can have an impact on the URI, this variable will express those changes.

As you can see, there are quite a few variables available to you when deciding how to set the FastCGI parameters. Many of these are similar, but have some subtle differences that will impact the execution of your scripts.

Common FastCGI Parameters

FastCGI parameters represent key-value information that we wish to make available to the FastCGI processor we are sending the request to. Not every application will need the same parameters, so you will often need to consult the app's documentation.

Some of these parameters are necessary for the processor to correctly identify the script to run. Others are made available to the script, possibly modifying its behavior if it is configured to rely on the set parameters.

- `QUERY_STRING`: This parameter should be set to any query string supplied by the client. This will typically be key-value pairs supplied after a "?" in the URI. Typically, this parameter is set to either the `$query_string` or `$args` variables, both of which should contain the same data.
- `REQUEST_METHOD`: This parameter indicates to the FastCGI processor which type of action was requested by the client. This is one of the few parameters required to be set in order for the pass to function correctly.

- `CONTENT_TYPE`: If the request method set above is "POST", this parameter must be set. It indicates the type of content that the FastCGI processor should expect. This is almost always just set to the `$content_type` variable, which is set according to info in the original request.
- `CONTENT_LENGTH`: If the request method is "POST", this parameter must be set. This indicates the content length. This is almost always just set to `$content_length`, a variable that gets its value from information in the original client request.
- `SCRIPT_NAME`: This parameter is used to indicate the name of the main script that will be run. This is an extremely important parameter that can be set in a variety of ways according to your needs. Often, this is set to `$fastcgi_script_name`, which should be the request URI, the request URI with the `fastcgi_index` appended if it ends with a slash, or the first captured group if using `fastcgi_fix_path_info`.
- `SCRIPT_FILENAME`: This parameter specifies the actual location on disk of the script to run. Because of its relation to the `SCRIPT_NAME` parameter, some guides suggest that you use `$document_root$fastcgi_script_name`. Another alternative that has many advantages is to use `$request_filename`.
- `REQUEST_URI`: This should contain the full, unmodified request URI, complete with the script to run, additional path info, and any arguments. Some applications prefer to parse this info themselves. This parameter gives them the information necessary to do that.
- `PATH_INFO`: If `cgi.fix_pathinfo` is set to "1" in the PHP configuration file, this will contain any additional path information added after the script name. This is often used to define a file argument that the script should act upon. Setting `cgi.fix_pathinfo` to "1" can have security implications if the script requests are not sanitized through other means (we will discuss this later). Sometimes this is set to the `$fastcgi_path_info` variable, which contains the second captured group from the `fastcgi_split_path_info` directive. Other times, a temporary variable will need to be used as that value is sometimes clobbered by other processing.
- `PATH_TRANSLATED`: This parameter maps the path information contained within `PATH_INFO` into an actual filesystem path. Usually, this will be set to something like `$document_root$fastcgi_path_info`, but sometimes the later variable must be replaced by the temporary saved variable as indicated above.

Checking Requests Before Passing to FastCGI

One very essential topic that we have not covered yet is how to safely pass dynamic requests to your application server. Passing all requests to the backend application, regardless of their validity, is not only inefficient, but also dangerous. It is possible for attackers to craft malicious requests in an attempt to get your server to run arbitrary code.

In order to address this issue, we should make sure that we are only sending legitimate requests to our FastCGI processors. We can do this in a variety of ways depending on the needs of our particular set up and whether the FastCGI processor lives on the same system as our Nginx instance.

One basic rule that should inform how we design our configuration is that we should never allow any processing and interpretation of user files. It is relatively easy for malicious users to embed valid code within seemingly innocent files, such as images. Once a file like this is uploaded to our server, we must ensure that it never makes its way to our FastCGI processor.

The major issue we are trying to solve here is one that is actually specified in the CGI specification. The spec allows for you to specify a script file to run, followed by additional path information that can be used by the script. This model of execution allows users to request a URI that may look like a legitimate script, while the actual portion that will be executed will be earlier in the path.

Consider a request for `/test.jpg/index.php`. If your configuration simply passes every request ending in `.php` to your processor without testing its legitimacy, the processor, if following the spec, will check for that location and execute it if possible. If it *does not* find the file, it will then follow the spec and attempt to execute the `/test.jpg` file, marking `/index.php` as the additional path information for the script. As you can see, this could allow for some very undesirable consequences when combined with the idea of user uploads.

There are a number of different ways to resolve this issue. The easiest, if your application does not rely on this extra path info for processing, is to simply turn it off in your processor. For PHP-FPM, you can turn this off in your `php.ini` file. For an example, on Ubuntu systems, you could edit this file:

```
sudo nano /etc/php5/fpm/php.ini
```

Simply search for the `cgi.fix_pathinfo` option, uncomment it and set it to "0" to disable this "feature":

```
cgi.fix_pathinfo=0
```

Restart your PHP-FPM process to make the change:

```
sudo service php5-fpm restart
```

This will cause PHP to only ever attempt execution on the last component of a path. So in our example above, if the `/test.jpg/index.php` file did not exist, PHP would correctly error instead of trying to execute `/test.jpg`.

Another option, if our FastCGI processor is on the same machine as our Nginx instance, is to simply check the existence of the files on disk before passing them to the processor. If the `/test.jgp/index.php` file doesn't exist, error out. If it does, then send it to the backend for processing. This will, in practice, result in much of the same behavior as we have above:

```
location ~ \.php$ {
        try_files $uri =404;

        . . .

}
```

If your application *does* rely on the path info behavior for correct interpretation, you can still safely allow this behavior by doing checks before deciding whether to send the request to the backend.

For instance, we could specifically match the directories where we allow untrusted uploads and ensure that they are not passed to our processor. For instance, if our application's uploads directory is `/uploads/`, we could create a location block like this that

matches before any regular expressions are evaluated:

```
location ^~ /uploads {
}
```

Inside, we can disable any kind of processing for PHP files:

```
location ^~ /uploads {
    location ~* \.php$ { return 403; }
}
```

The parent location will match for any request starting with `/uploads` and any request dealing with PHP files will return a 403 error instead of sending it along to a backend.

You can also use the `fastcgi_split_path_info` directive to manually define the portion of the request that should be interpreted as the script and the portion that should be defined as the extra path info using regular expressions. This allows you to still rely on the path info functionality, but to define exactly what you consider the script and what you consider the path.

For instance, we can set up a location block that considers the first instance of a path component ending in `.php` as the script to run. The rest will be considered the extra path info. This will mean that in the instance of a request for `/test.jpg/index.php`, the entire path can be sent to the processor as the script name with no extra path info.

This location may look something like this:

```
location ~ [^/]\.php(/|$) {

    fastcgi_split_path_info ^(.+?\.php)(.*)$;
    set $orig_path $fastcgi_path_info;

    try_files $fastcgi_script_name =404;

    fastcgi_pass unix:/var/run/php5-fpm.sock;
    fastcgi_index index.php;
    include fastcgi_params;
```

```
        fastcgi_param SCRIPT_FILENAME $request_filename;
        fastcgi_param PATH_INFO $orig_path;
        fastcgi_param PATH_TRANSLATED $document_root$orig_path;
    }
```

The block above should work for PHP configurations where `cgi.fix_pathinfo` is set to "1" to allow extra path info. Here, our location block matches not only requests that end with `.php`, but also those with `.php` just before a slash (/) indicating an additional directory component follows.

Inside the block, the `fastcgi_split_path_info` directive defines two captured groups with regular expressions. The first group matches the portion of the URI from the beginning to the first instance of `.php` and places that in the `$fastcgi_script_name` variable. It then places any info from that point onward into a second captured group, which it stores in a variable called `$fastcgi_path_info`.

We use the `set` directive to store the value held in `$fastcgi_path_info` at this point into a variable called `$orig_path`. This is because the `$fastcgi_path_info` variable will be wiped out in a moment by our `try_files` directive.

We test for the script name that we captured above using `try_files`. This is a file operation that will ensure that the script that we are trying to run is on disk. However, this also has a side-effect of clearing the `$fastcgi_path_info` variable.

After doing the conventional FastCGI pass, we set the `SCRIPT_FILENAME` as usual. We also set the `PATH_INFO` to the value we offloaded into the `$orig_path` variable. Although our `$fastcgi_path_info` was cleared, its original value is retained in this variable. We also set the `PATH_TRANSLATED` parameter to map the extra path info to the location where it exists on disk. We do this by combining the `$document_root` variable with the `$orig_path` variable.

This allows us to construct requests like `/index.php/users/view` so that our `/index.php` file can process information about the `/users/view` directory, while avoiding situations where `/test.jpg/index.php` will be run. It will always set the script to the shortest component ending in `.php`, thus avoiding this issue.

We could even make this work with an alias directive if we need to change the location of

our script files. We would just have to account for this in both our location header and the `fastcgi_split_path_info` definition:

```
location ~ /test/.+[^/]\.php(/|$) {

    alias /var/www/html;

    fastcgi_split_path_info ^/test(.+?\.php)(.*)$;
    set $orig_path $fastcgi_path_info;

    try_files $fastcgi_script_name =404;

    fastcgi_pass unix:/var/run/php5-fpm.sock;
    fastcgi_index index.php;
    include fastcgi_params;

    fastcgi_param SCRIPT_FILENAME $request_filename;
    fastcgi_param PATH_INFO $orig_path;
    fastcgi_param PATH_TRANSLATED $document_root$orig_path;
}
```

These will allow you to run your applications that utilize the `PATH_INFO` parameter safely. Remember, you'll have to change the `cgi.fix_pathinfo` option in your `php.ini` file to "1" to make this work correctly. You may also have to turn off the `security.limit_extensions` in your `php-fpm.conf` file.

Conclusion

Hopefully, by now you have a better understanding of Nginx's FastCGI proxying capabilities. This ability allows Nginx to exercise its strengths in fast connection handling and serving static content, while offloading the responsibilities for dynamic content to better suited software. FastCGI allows Nginx to work with a great number of applications, in configurations that are performant and secure.

Tags: Nginx, Scaling, PHP

Heart

Subscribe Share

By: Justin Ellingwood

Related Tutorials

Automating the Deployment of a Scalable WordPress Site

How To Deploy a Rails App with Puma and Nginx on Ubuntu 14.04

How To Deploy a Rails App with Unicorn and Nginx on Ubuntu 14.04

How To Serve Flask Applications with Gunicorn and Nginx on CentOS 7

How To Serve Flask Applications with Gunicorn and Nginx on Ubuntu 14.04

0 Comments

Leave a comment...

Log In to Comment