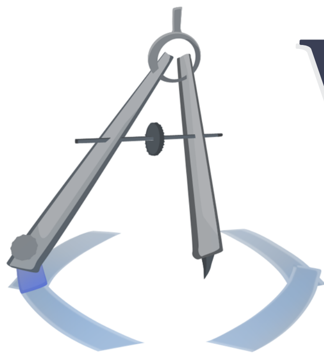


Website Architecture

Lezione 8

Apache HTTP Server

A getting-started guide for Apache and a survey of the expressive power of its most commom features.

Michael Serritella

Summer 2010

© Michael Serritella, 2010 • CIS4930: Website Architecture • The Florida State University

Intro to Apache

Apache is an HTTP server. That means that it is a program which understands the language of HTTP - it can listen for requests and write proper responses. It is the most common HTTP server in the world. It allows you to configure programmatic rules which govern the way requests are handled. This allows you much more expressive power than you would otherwise have, as hinted in the prior lesson on request processing. It is a solid program, but its configuration-file language sports a disappointingly hack design, as is typical of free-software projects. Fortunately, only a little knowledge is required to make a large impact on your site.

The Apache program itself is named httpd, which you may see from time to time. The "d" suffix is a common Unix naming scheme for programs which persist in the background, since they are known as daemons ("demons"). Since that is a fairly dumb name, you will increasingly see them called services, which is what they are called on Windows.

Motivational examples

Apache can do very many complex things, but it can also do some simple tasks that you may have always wanted. We'll dive into a few motivational examples, showing a preview of Apache's configuration-file language. We will see the code necessary for each example, taken out of context, as we will see the context later.

Directory listings on or off

If you go to the URI of a directory name, if Apache cannot find an "index.*", then it may show a listing of the files in the directory. However, you may not want this. If you want to toggle the directory listing on or off, it's just one line of code.

Apache

```
Options +Indexes
```

And, of course, there's the opposite:

Apache

```
Options -Indexes
```

You can also get Apache to build customized directory listings - including extra features into the page, omitting some files from the listing, etc. - using the ORLY module.

Custom error pages

We saw this in the last lesson when we saw HTTP response codes. You can give a custom error page to handle errors such as 404/not found errors, or else the server or the browser can make their own. Here is a way to give your own page in place of 404 errors:

Apache
```
# This is an absolute path, so to speak, relative to the
# root of your site within the actual filesystem.
ErrorDocument 404 /errorpages/myown404.html
```

Compressing output

We saw before that the you may save a lot of time & space resources by compressing content, such as HTML, CSS, and JavaScript, before sending it to the user. Apache allows configuration options for this; here is a simplified example:

Apache

```
# The "DEFLATE" is a postprocessor function (an "output filter")
# which compresses the output using gzip, so long as it has one
# of the given MIME types.
AddOutputFilterByType DEFLATE text/html text/css text/javascript
```

Not all browsers support gzipped content, so in reality, you would want to add some conditionals to make sure that the right browser gets the right content. See **an official documentation page** for a more responsible recommendation.

Configuration files

Now, let us zoom out and give some context to these examples. What is this Apache code? Where do you write it?

The code we've seen should occur in configuration files. Here, we will explore the different types of configuration files and the general structure of their contents. In the process, we will learn the basic design of Apache.

Directives

Each of the code lines that we've seen have demonstrated **directives**. These are the declarative statements which tell the Apache server what to do. We have not yet seen the statements which group them together or designate their context.

For an example of a directive, see the documentation snippet for the **ErrorDocument** directive. For a more ambitious one, which still has some small examples, see **Header**.

Configuration file hierarchy

Some configurations apply to Apache as a whole. Some apply to each site that Apache manages (one instance of the program may manage multiple sites/domains). And some apply on a per-directory basis.

So, we have two different types of configuration files, and within them, we can also use some language features to designate the context of directives. Generally, we have global configuration files and directory-specific configuration files.

Global configuration files

Configuration files may be placed in a "global" area of the filesystem, away from the actual files of the site. For instance, the most important such configuration file is called "httpd.conf" and exists in "/etc/apache2" within the filesystem. This is read when Apache starts, unless you specify another configuration file on the command line. Within this file, you may include other configuration files, similar to an include directive in C++.

Within the contents of these files, you may give directives which apply to anywhere within your sites - any folder or file - and you may give directives which apply to your server or site as a whole.

Directory-specific configuration files

Within your main configuration files, you can specify whether to allow directory-specific configuration files. If you do, small configuration files may be placed within each directory, and they may override the global configurations. They do so in a cascading manner, walking through each directory which is in the path of a request. For instance, if the user requests "images/photos/bob.jpg", the server will check for directory-specific files within the root directory, then "images/", and then "images/photos/". More local configuration files may override less local ones.

The configuration files are named **.htaccess** (note the leading period) by default. You may change that with the **AccessFileName** directive. They are enabled by default, which you

may disable with **AllowOverride**.

Syntax: configuration sections

Now, within each file, you may surround directives with some syntax to say that they apply to certain directories, files, URIs, or sites. They bascially look similar to HTML tags, and they are called **configuration sections**. Here are some examples:

Apache

```
<Location /images>
# A directive for the given URI..
</Location>

<Directory /images/photos>
# A directive for the given filesystem directory..
# e.g. ErrorDocument, Options, etc.

  <Files contactus.html>
  # A directive that applies to the given file..
  </Files>
</Directory>
```

We will see each type of configuration section. For an overview, you can read **this page** at the official documentation.

Virtual Hosts

Apache may service more than one site at a time. More relevantly: it may service more than one subdomain at a time. Apache has to be configured to listen for requests for all relevant domains, and *you* need a way to give configuration options specific to each domain. Enter the **VirtualHost** section.

If you want to serve more than one (sub)domain, you have to set up virtual hosts. Within a global configuration file, you must do two things to get virtual hosts working: You have to declare that you are listening for those requests, and you have to specify where the requests go.

The declaration of interest looks like this:

Apache

```
# Listen for any IP address (*) on port 80.
# You could just do an asterisk with no specific port.
NameVirtualHost *:80
```

Note: This declaration alone does not ensure that people will be able to find your site on the Internet at that address. People still have to find your server machine first. Then, people will assail it with requests for whatever domain. You are telling

Apache that it should listen and respond to those requests for the given domain.

The specification of locations for the domain looks like this:

```apache
# For the main domain.
<VirtualHost *:80>
    DocumentRoot /var/www/
    ServerName www.cis4930.com

    # Nested directives..
</VirtualHost>

# For the subdomain.
<VirtualHost *:80>
    DocumentRoot /var/www/somethingDirectory
    ServerName something.cis4930.com

    # Nested directives..
</VirtualHost>
```

Locations and Directories

More commonly useful are location and directory sections. But wait - what's the difference?

We saw before that requests need not correspond to files in the filesystem. In fact, requests can be completely abstract - the

requested file need not exist, or it may exist somewhere else. So, when the client gives a request, the request is an abstract.. Location. And when it finally maps to a directory in the filesystem, it is a.. Directory.

You may give specific URIs or paths for these sections, or you may give regular expressions, so that any matching URI or path will receive the effect of the configuration. Additionally, all child URIs and paths will be affected by the configuration, so applying a directive for "Location /" will apply it to the whole site.

See the documentation for **Location** and **Directory**.

Revisiting per-directory configuration

Writing directives in an .htaccess file is functionally identical to writing them in a directory section within a main configuration file. The .htaccess files do offer some unique flexibility, but they should generally be avoided unless you cannot write to the main configuration file. This is usually the case in shared Web hosting, however, so you may have to use .htaccess files.

Revisiting directives

In the documentation, each directive has a standard listing of its powers and abilities. In it, one of its most important distinctions is its **Context**, which is the list of configuration sections in which it may appear. See an example with **ForceType**, and read the **documentation** on the types of attributes used to describe directives.

Modules

Apache is extensible, and it has a module system by which modules can provide new directives. There is a "core" module which contains all of Apache's default directives (see **documentation**), and there are several dozen modules that come standard with Apache. You could conceivably write your own in C/C++. To see a listing of officially-recognized modules with links to their documentation, see **here**.

Guideposts to common features

Apache's features are broken up into fuzzily-divided modules, and the directives have a completely nonstandard naming scheme. Even if you know Apache's capabilities - which is the most you should care about for this course - you still may not know where to find them or how to execute them.

Here is a list of some useful modules and their summaries. Each of the entries has a link to the documentation page for the module, where you can find information about directives.

mod_core (Docs) A hodgepodge of directives; look here first.

mod_actions (Docs) Execute scripts when certain MIME types are requested, instead of serving the file.

mod_alias (Docs) Give aliases for files, possibly using regular expressions; give redirection responses; etc. Sometimes this is all you need.

mod_deflate (Docs) Serve compressed content.

mod_dir (Docs) Control the default page (e.g. index.html) of a directory, and control trailing-slash redirects.

mod_expires (Docs) Give the client caching information in the response headers.

mod_headers (Docs) Write arbitrary headers in the response.

mod_include (Docs) Allow for HTML files to include other HTML files, just like in programming languages (warning: taxes CPU and overall speed).

mod_mime (Docs) Control the sending of the `Content-Type` reponse header.

mod_rewrite (Docs) Rewrite request URIs using regular expressions.

mod_speling (Docs) Allow for spelling or capitalization mistakes in requests.

Client authentication

Apache can be used as a basic means of authenticating the client with a username & password. The method is typically called "HTTP authentication", since it is a simple protocol which uses HTTP headers to pass the client's credentials.

You've surely seen this kind of authentication before. You go to a URI for a file or folder, and the browser usually gives you a small dialog box (popup message) prompting you for a name and password. You press OK or Cancel, and that's it. This is not the same kind of authentication you use in most modern sites, with "Login"/"Sign Out" buttons that are written into the page; that is significantly more complicated. This is a cheap solution that works decently well and which is independent of the site content (i.e. you don't have to rewrite your pages with new buttons and new semantics).

Basic concept

This is our first foray into authentication and security, so perhaps we should introduce a few small concepts. What's a good

way to check if a user has the right password?

When giving a username & password, rather than send the password literally, it is more secure to send a "digest" of the password, also known as a "hash" of it. This digest/hash is the return value of a function (a "hash function") which accepts the password; it should look like gibberish, and it should more or less directly correspond to the password. It should be hard for someone to work in the inverse direction and figure out the password if given the digest.

So, similarly, rather than *store* the password, it's more secure to store the digest. Then, the client sends you a digest, you check it against your copy of the digest, and you can be virtually certain that they know the password[1].

Protocol

There are two flavors of the protocol, each of which builds upon HTTP by using some new headers. There is **basic authentication** and **digest authentication**.

[1]Modern security protocols have evolved from this thinking, but this is a good basis and is still used.

In either case, the client makes a request for a resource which may or may not be protected; the client does not know. Then, the server returns a response with code 401/unauthorized. The user sends back some credentials in a new request, which is otherwise a copy of the old request. If the server is satisfied, the original resource is served as normal.

In basic authentication, the client sends the username and password plainly, without any encryption. The username and password are encoded in Base64, but this does not offer any protection; there is no secret key and it is reversible, so anyone who sees it can decode it. Base64 merely serves to eliminate any problematic characters from the string, like colons, spaces, etc.

In digest authentication, a more complex protocol is used, where the digest is sent, as described above. Mixed into the digest is some other challenge-and-response-based data, which makes the protocol more robust against offline eavesdroppers.

Apache configuration

To achieve this in Apache, you basically have to make a file which contains the acceptable credentials, perhaps using a tool to do so, and then tell Apache to require credentials from the client and to match them against your file. There is a thorough walkthrough **here**, in the official documentation.

Advantages and disadvantages

HTTP authentication is simple, and it will stop most people from snooping around. However, it is vulnerable to man-in-the-middle attacks, where the network connection between the client and server is insecure. In practice, it shouldn't be used by clients who are in a LAN with untrusted people, since the request may pass through them (and come entirely within their control) and since the response is entirely unencrypted. HTTP authentication also lacks a way for you as the site author to manage the session of the client, so you can terminate the session when you wish, like after five minutes of inactivity. You have to wait for the user to close the window/tab and/or clear the history/cache (browser-dependent, of course).

CGI programming

As we mentioned before, the server may accept a request and then, instead of returning the contents of an existing file, run a program and return its output. The type of program which runs is called a CGI program.

Intro to CGI

CGI stands for Common Gateway Interface. The CGI standard describes a way for a server to pass Web-request information to any program which it can instantiate. This is done by populating some environment variables with standard names. To glance at an overview, see **Wikipedia**.

Theoretically, any text-based C or C++ program can already produce a Web page, since it can write text which has HTML tags. Now, it can receive the right inputs. If you're not familiar with them, environment variables are like mega-global variables that exist between programs; they can be passed to a program when it starts, like **argc** and **argv**, and they are passed by the program which instantiates the recipient program.

PHP as a server-side scripting language

Virtually any language will do, but PHP makes a particularly good choice. It has built-in functions for handling inputs from requests, writing responses and their headers, and manipulating strings in ways that are typically needed on a Web server. PHP is a language with a small learning curve which can build complex programs. Its Apache module, **mod_php**, is the most popular Apache module, and it is the most or second-most popular Web language (depending on whether you count Java as a Web language, since it has its JSP flavor).

Serve it plain or with some hot sauce?

With Apache configuration, for a given request, you can choose to invoke a CGI program or not. Here is an example directive by which you can force all files in a directory to trigger PHP execution (assume this is in a directory configuration section):

Apache
```
ForceType application/x-httpd-php
```

And what if you have a directory of PHP scripts that you want

to serve as plain text, like if you are giving PHP code examples?

```
Apache
ForceType text/plain

# Or perhaps this.
AddType text/plain .php
```

Using such configurations, for example, you can actually redirect all image requests to PHP scripts that build images on the fly or conditionally reject the user's request for the image.

PHP installation details

PHP is often bundled within Apache's executable, so that Apache does not have to launch a new program in order to run PHP. It may also be run as a separate CGI program, as it may be good for security and reliability to decouple the programs. One can crash and the other will not be as affeced, for instance. This is slightly slower and loses a little bit of language power on PHP's part, since, as a module, PHP could ask Apache to do things (for the future: see **Apache Functions** on PHP's documentation site). Still, some hosting companies choose to keep PHP separate as a CGI program.

Request paths and security

You may be wondering a few things about how Apache fields requests for sensitive files, like script files (PHP) or configuration files (.htaccess). Can you be sure that the server will not serve your PHP files as plain text, instead of executing them and returning their contents? What if there's a glitch? Believe it or not, once per aeon, this happens. Let's learn about this and learn how to minimize the risk of anything too terrible happening as a result.

Background knowledge: Race conditions

A race condition is an undesirable relationship among a set of events - like a set of computing tasks - where the result depends on a race between the operations, not on a well-enforced policy. Many race conditions exist in the wild, but perhaps most commonly, this may happen when multiple programs in an operating system are running concurrently and they need to access a file - like if one reads and another writes. If you are very careful, you can almost certainly avoid this problem.

Background knowledge: Human nature

Most people are not super careful all the time.

Server files and race conditions

Although Apache is very commonly used, conceptually, there exists a risk of a race condition between Apache's reading of files and your writing them. Configuration files and principal site files may be incorrectly read or incorrectly regarded as blank.

Race conditions may not be the culprit behind scripts served as text; it's difficult to tell when the occurrence is so extremely rare. It is likely that they are at fault, but at the very least, they are a good enough reason to be careful.

Parsing of configuration files and type sniffing

In light of these problems, let us look simply at the way Apache decides how to serve a requested file. First, it checks for configuration files. Perhaps a configuration file is momentarily blank or incomplete. The configuration file may have told Apache

how to handle that particular request, and now we have a problem.

Alternatively, the configuration file is inconsequential. After all, most of the time, you don't have to specify to serve JPEG files as image/jpeg. Apache figures it out, either by checking the filename's extension (".jpg") or by "sniffing" the file - checking its first few bytes. Some files are not properly named, or maybe they have no extension. If the file is incorrectly read as a result of a race condition and it needs to be sniffed, there may be a problem.

Perhaps the most important lessons here are that many race conditions may exist, and they correspond to different areas of code within Apache. They are effectively independent, and maybe you can try to depend on the parts of code within Apache that are most simple, which are least likely to have careless errors.

Apache seems least likely to serve a configuration file if it is requested; that part of code is likely uncomplicated. The file may not be named ".htaccess", which does raise a complication,

so perhaps you could avoid giving the configuration files custom names. Even still, there isn't much you could do about this problem. You can more or less trust that it will work. Let's turn our attention to the problem of serving script files as plain text.

Background knowledge: The document root

Each site managed by the server has a "document root", which is a directory in the filesystem that serves as the root directory of the site. Now, if a user makes a request with "../" in the URI, Apache will (perhaps unfortunately) resolve the URI; for example, requesting "www.example.com/asdf/subdir/../hello.html" will go to "www.example.com/asdf/hello.html". *However*, requests may not go higher than the document root. Apache will stop there.

The code which performs this check is likely very simple, and it is extremely rare that it will result in a race condition. Technically, it does depend on a configuration file to define the document root, but that may be in a main configuration file that almost never changes.

Read more about the document root at the **official documentation**.

Finally: Hiding script implementations

That's great, but how can we take advantage of this document-root rule? Well, the script files that people can request may very rarely become exposed. At the very least, it is conceivable. And there isn't too much you can do about that.

However, within those script files, you may include other script files, just like in C++ or any programming language, and those inclusions are not limited by Apache's document-root rules, since they have nothing to do with HTTP requests. The PHP interpreter may simply access the filesystem in any normal way. *So,* the files under the document root may include files which are *outside* the document root, such as within a directory that is the sibling of the document root. The bulk of a script's implementation can be in those files, and the (potentially) exposed script need only include the a file and call one function. The hidden implementation can take care of the rest.

Rewriting requests (!)

Now for the big show. This is doubtlessly one of Apache's most powerful features, and it can streamline your entire site architecture, especially including its layout on the filesystem. We have mentioned that request URIs do not necessarily have a direct correspondence with filesystem paths. Now, we're going to blow that wide open. Requests may be *entirely rewritten* by Apache and then handled internally as sub-requests that the user does not see.

Simple aliasing

Using **mod_alias**, you can essentially make shortcuts from one part of the filesystem to another. More precisely, you can make "shortcuts" from an URI to any part of the filesystem. You can make shortcuts for whole directories, specific files, or even patterns (regular expressions) of URIs. The destination path can even include some parts of the source URI.

This is fine for simple tasks and even some fairly advanced ones. See the documentation for more explanation and examples.

The documentation recommends to use mod_alias to help hide script implementations. You can do this by aliasing an URI to a place in your filesystem that is outside of the document root and cannot be directly requested. This is a fine-enough way of doing the job, but it has implications on your filesystem structure that may seem unnatural. For instance, most sites have a directory, such as "/products", with some scripts, and "/products/images" with some images. Or at least it is common to have some kind of subdirectory of a directory which contains scripts. You would either have to put all of your content in a place which is not in the document root, or you would have to make a combination of aliasing rules to move the "...images" requests back to their natural place. You might want to consider just keeping it simple and hiding your script implementations as described above.

Traditional redirection

Still using mod_alias, you can tell the user that the requested resource has moved - i.e. you can issue a redirect. You may give temporary redirects or permanent redirects, which, as you

may imagine, has consequences on the client side, such as with search engines or browser caches.

Rewriting using mod_rewrite

Finally, there is **mod_rewrite**. This is a much more powerful alternative to mod_alias. It can consider the state of the environment, including the existence of files, the current time, environment variables, and more; it can include randomization and input from user-generated (administrator-generated) programs to assist in its redirection choice; it can chain many rules together; set cookies; and more.

Official documentation

A pretty good walkthrough of mod_rewrite is available on its documentation page. Additionally, Apache provides an **URL Rewriting Guide** which helps introduce related technology (e.g. regular expressions) and provide useful examples.

Turning it on first

Before you start rewriting URLs, you need to actually set a directive to enable mod_rewrite. Though it is likely included in your Apache installation, it does not execute by default.

In a configuration file and in the appropriate configuration section (directory, etc.), do this:

```
Apache
RewriteEngine on
```

Though most Apache directives are declarative, this appears to be an imperative command; how cute.

If you create rewriting rules within a .htaccess file, you need to do two things. You need to inform Apache of the URI directory which corresponds to that directory, so it knows which part of the URI to rewrite. See the documentation on **RewriteBase**. Second, you need to make sure the "FollowSymLinks" option is enabled for the directory in question, as described in the documentation for **RewriteRule**.

Specifying conditions upon which you rewrite

You may test for a wide range of conditions to see if a rewriting rule should be applied to a given URI. The most interesting thing is that the conditions may have little to do with the URI itself. But first, let's look a those more simple cases.

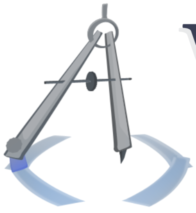
First example

Let's say you want to rewrite URIs that begin with "rewriteMe", followed by three capital letters and an underscore, such as "rewriteMeXYZ_index.html". You can specify this condition in its own directive, which appears before the directive which actually specifies the rewriting expression. Your directive would look like this:

Apache

```
RewriteCond %{REQUEST_URI} ^.*/rewriteMe[A-Z]{3}_[^/]*$
```

Let's walk through these regular expressions. The first parameter tells Apache which string it should test, and we are testing the request URI, of course. Next is the pattern; if this matches, the entire RewriteCond condition is satisfied. The



"^" and "$" frame the regular expression, since they mean "the beginning of the string" and "the end of the string." In some patterns, you may not care where the pattern matches, as long as it matches somewhere within the string, but in this case, we want to make sure it appears in the last part of the URI. The "." means any character, and the "*" means the previous thing (the dot) 0 or more times. Then we have a slash, as it would literally appear in the URI. Then we have a literal string, and then we have "any character in between A and Z, inclusive", and after that, "the previous thing 3 times". Then a literal underscore, and then "any character that is not a slash", "zero or more times" (i.e. slashes cannot come after this point).

Explaining regular expressions in general is way out of the scope of this course, so you are not required to be skillful for this course. Basically, in real life, if you can imagine the pattern, there is probably a regular expression for it, and it may take you a half hour or an hour to suss it out. You generally only have to do this when setting up your site, so you shouldn't mind having this occasional, time-intensive cost.

Directive behavior

In addition, you could prefix the pattern with "!", meaning that the string should not match. There are some other prefix symbols that invoke non-regular-exprssion behvaior, such as "<" and ">" (lexicographical string compare). If you provide multiple rewrite conditions before you give the rewrite rule, then the conditions should all be satisfied if the rule is to be executed.

The directive can also be customized with flags. Most notably, it can be made case-insensitive. The flags come after the pattern, with the syntax " [FlagOne,FlagTwo]". The not-case-sensitive flag is "NC". See all options in the documentation.

Testing the environment

Now, that was a sort of simple example, in a way. The regular expression is annoying-looking, but it is also typical. But it was simple in that it only considers the URI. If you want, you can consider many more things, as shown in the documentation. Some of these are environmental flags, like whether the request has already been rewritten or whether the connection

is over SSL. These appear as strings which you can test for a pattern; in another cute "design" choice, some of these are "true"/"false", and some are "on"/"off".

Let's look at an example of checking the system time. Suppose you write a stock-ticker page, and the stocks are only active from 9:30 AM - 4:00 PM. Outside that range, you want to display a simple page that says "Sorry" and/or fetches the stock price from a different database. Within that time, you need to get the live stock price, and this page gets loaded a million times a day. Since PHP is an interpreted language, fetching the server time (perhaps a string?), parsing it out, comparing it, etc., will take more time than it would optimally, like if you were using a C program. You'd like to not execute that if-statement a million times a day. So, you'd like to redirect the request for "stockTicker.php" to "stockTicker_open.php" or "stockTicker_closed.php", to reflect whether the market is open or closed. The rewrite condition is more interesting than the rewrite rule, in this case. Let's look at it:

Apache

```
RewriteCond %{TIME_HOUR}%{TIME_MIN} >0930
RewriteCond %{TIME_HOUR}%{TIME_MIN} <1600
# Rewrite the URI to the "open" script
# "Else" (though no "else" sytnax necessary), rewrite to "closed"
```

The code in this example is more or less jacked from the URI Rewriting Guide, so check that out.

Another special power of rewrite conditions is that they can check whether the URI points to a file vs. a directory, a non-empty file, an executable file, or a few other things, as visible in the directive's documentation. These tests are ostensibly modeled on the tests you can do in Bash scripting. We will see an example later.

Specifying what to rewrite

Now, here it is. How do you do the deed? Simple cases are simple enough. But, of course, it can get complicated. In more complicated examples, we will see how to use *parts* of the pattern to reconstruct the replacement.

First example

Trivial examples will look a lot like mod_alias. Let's say we want to redirect all inquires to a "random.png" to a PHP script which generates a random image. This is not even pattern-based, and it does not even require a rewrite condition. But let's see it, anyway:

Apache

```
RewriteEngine on
RewriteRule random\.png randomImage.php
```

That's it. We have to escape the "." within the pattern so it does not appear like our earlier ".", which means "any one character." However, we don't need to do that within the replacement, since the replacement is not (so much) a pattern, and "any one character" would be meaningless there. A dot is taken as a literal dot.

Backreferences

Let's go back to our mod_alias example of the "rewriteMe" files. Say that we wanted to rewrite those files such that "rewriteMeQWE_something.html" goes to

"something_QWE.html". We want to use that part of the pattern and put it elsewhere within the replacement string. First, we need to designate that part of the pattern which is to be used later; we need to put it in parenthesis. There may be up to nine such parenthetical sections. Then, in the replacement string, we use the form "$N", going from "$1" to "$9", to signify those parenthetical sections.

To solve our example problem:

```
Apache
RewriteRule rewriteMe([A-Z]{3})_([^\.]*)\.(.*) $2_$1.$3
```

Here, let's look at the parentheticals. The first is the QWE (or whatever) capital-letter pattern. The second is the filename after the "_" but before the extension (using the "not-a-dot" trick, like our "not-a-slash" before). The third is the filename's extension.

You may notice that we didn't have the "^" and "$" in this one to frame the beginning and end of the pattern. This is to show you that you could relax some of the pattern in the

rewrite rule as long as you know the string already satisfied the rewrite condition; it's a design choice, and it may not always be valid. If the same "rewriteMeXYZ_something.else" pattern were to appear anywhere else in the URI, it would also be replaced. But you can usually be pretty sure that that isn't the case - perhaps only in the query string could there be a problem.

This behavior is the same as in other regular-expression-based programs; backreferences are a common feature. You can also add qualifiers to the parenthetical sections, like an extra "{2}" afterwards to signify that the contents of the parenthesis should appear twice, etc.

Whoops: also in mod_alias

Yeah, it turns out that backreferences are already a part of mod_alias. See the documentation for **AliasMatch**, which is quite powerful in and of itself. However, in mod_rewrite, there are two kinds of backreferences: There are backreferences from the pattern of the rewrite rule, and there are backreferences from the pattern of the (most recent) rewrite condition.

Flags

You can add flags to the rewrite rules, however, which is unique to mod_rewrite. These have the same syntax as the flags for the rewrite conditions, but the flags themselves are different, there are more of them, and some of them have surprisingly active behavior. For instance, you can set a cookie, send a forbidden or "gone" status code, set a MIME type for the response, and more.

You can see the documentation for RewriteRule, and you can see a **special part** of the URL Rewriting Guide the explains the flags.

Implications of rewriting

So, now we know that we can rewrite stuff. What kind of implications does this have on our site architecture? It turns out that rewriting - and knowledge of Apache in general - can greatly improve our site architecture, especially in the way that scripts and files are organized in the filesystem.

We will see a running example throughout this section. Imagine

that we have a site, www.store.com, and we want to have this URI: "http://www.store.com/Books/2007/APublisherName"[2]. This will work as a book search, searching for all books in 2007 with the given publisher name. We will see various ways of doing this with URI rewriting, closing in on a more and more efficient solution.

Intuitive URIs

So we want to do a book search. Most search pages have some URI like "search.php?bk=abc&hack=1&asdf...", but really, those are annoying. The user could never remember them or verbally communicate them to other people, and it has a large amount of information that the user shouldn't have to care about. Should they care that the solution is written in PHP? The variable names? No. Let's start to fix this. Our goal URI is pretty nice; it's also about as intuitive as it gets. You see the same kind of URIs for Wikipedia articles and a small but growing number of websites. They should be the wave of the future. So, how would you do it? Here's one way.

[2]We could have a trailing slash here in our example, though that changes things. See the documentation for mod_dir and the **"Trailing Slash Problem"** in the URL Rewriting Guide introduction for some background knowledge. Let's make some loose assumptions for now.

Say that there is actually a directory structure on your site like "/Books/2007/", and inside is a script called "APublisherName.php", another called "AnotherPublisherName.php", etc., and each of the scripts gives the proper search results (for 2007 books). How do we take the intuitively-written requests and transform them to the PHP scripts?

We check to see if the URI + ".php" is a valid regular file, and if so, we redirect the request there. Else, we let the 404 error be sent back as usual. Here is the final code:

Apache

```
RewriteEngine on
RewriteCond %{REQUEST_FILENAME}.php -f
RewriteRule ^(.*) $1.php
```

So that is a win. You may not want to do it for our current book-search example, but if you were making Wikipedia, you should look into it.

Virtualized filesystem structure

Now for the next step. We'd really like to not have all of these directories lying around. Long story short, a depthwise-complex filesystem structure will degrade performance linearly for each new level of depth - not good. And a breadthwise-complex one will degrade performance logarithmically for each new file in a directory. Any degradation is not great, but logarithmic beats linear. How can we "flatten" our directory structure in some way?

Let's shoot for this instead. In a folder "/bookSearches/", we have scripts like "search_2007_APublisherName.php". Let's use backreferencing to rewrite these requests. The solution is simple enough, actually.

```
Apache
RewriteEngine on
RewriteCond %{REQUEST_URI} /Books/([^/]+)/([^/]+)
RewriteRule ^.*$ /bookSearches/search_%1_%2.php
```

We use %-styled backreferences here to refer to parentheticals from the rewrite condition. The "+" means "one or more of the previous", as opposed to the "*", which allows zero or more.

Note that many types of filesystem reorganization are possible with URI rewriting; you don't simply have to flatten a directory structure.

Consolidation of program inputs

Now we come around to a most elegant solution. Let's not have all these script files, anyway. They are annoying to maintain, they incur a minor filesystem overhead which becomes noticeable when there are hundreds or thousands of files, and they even leave opportunities for error. Let's consolidate this down to *one* file. Let's have a mediaSearch.php, which takes GET parameters "media", "year", and "publisher".

The result may look like this:

mediaSearch.php?media=Books&year=2007&publisher=APublisherName

You can probably see where this is going. We can rewrite (or write) the query string!

```
Apache
RewriteEngine on
RewriteCond %{REQUEST_URI} /([^/]+)/([^/]+)/([^/]+)
RewriteRule ^.*$ /mediaSearch.php?media=%1&year=%2&publisher=%3
```

And yeah, this can also be done with mod_alias, since we don't use any environmental state, introduce nondeterministic behavior, etc.

Next step for the mighty: Can you make versatile rules so that users can go to approximately-correct URIs, like "/Books/APublisherName/2007" or just "/APublisherName/2007" or "/2007/APublisherName"? Imagine a concerted effort between well-designed PHP scripts and well-designed Apache rules, still using at most a handful of files.

Apache in the wild

If you start out on a shared Web host, you will almost certainly not have access to the httpd.conf file. You probably won't have any special influence over Apache except with .htaccess files. You should expect to work within this limitation.

This is usually not really a problem because you usually don't run a high-profile site on a "shared" server[3]. And if it's part of a somewhat-high-profile site, then it's probably just the CDN which is on a shared server. So you can't do any performance tuning, and you likely can't do some of the request-handling acrobatics that we've seen.

With the next step up - virtual private servers - you can control Apache completely, including stopping and restarting it.

[3]It still may be shared among multiple parties, but in the business, "shared" implies massively shared.

Performance issues

Now for a few performance issues. If you're running on a shared server, you can't care too much about these, because you may not be able to set the configuration files to do anything about these problems and, after all, your resources may be conceptually infinite. So you might not care. But keep these in mind in case you design a more pro-level site that has finite resources.

Minimizing disk access

Disk access is one of the most influential performance factors of probably any program which uses the disk an infinite number of times. Websites are no exception. In order to minimize disk access, you can virtualize your filesystem structure, as described in the section on URI rewriting.

Somewhat conversely, you can minimize the number of .htaccess files you use, as their use incurs a lot more disk checking than usual. Try to keep all directives in the main configuration files, and if you do use .htaccess files, don't make the filesystem too depthwise-complex, or else the directories at every level of

depth for every request will need to be checked.

Minimizing RAM and CPU usage

Probably, the most influential factors to RAM and CPU usage are your script's algorithms, so this is a little out of the scope of Apache configuration. However, it is important to know that you can only serve a certain number of people concurrently before you run into trouble, and using Apache, you can configure the number of connections that you serve simultaneously. More specifically, a Web server should never swap memory to disk, or it may incur a seemingly endless loop of users refreshing their browsers and issuing more requests.

To avoid this, you may take several routes, including distributing your site on a CDN such that your main site only serves the main document and no images, movies, etc. More simply, you can run some tests and determine a conservative maximum number of connections that your application can service. You can use Apache to enforce this limit using some of the "Max*" directives in the module **mpm_common**.

Compiling regular expressions, like in mod_rewrite and others, does take some CPU usage. It may be used in a tradeoff against extra disk access, in which case it's almost certainly worth it. But you should know that regular expressions are some of the more complex things that Apache can do computationally.

There is one feature that we have not seen, in which Apache can facilitate HTML files including other HTML files via an "#include" directive within HTML comments, which Apache parses before it sends out the page. You can see it at mod_include. This is useful for offering modularity to basic, static sites, but it isn't appropriate for large performance-conscious sites, as it incurs more RAM & CPU usage, since it must check every byte of the output instead of just transferring it in blocks.

The takeaway

Apache has some unnaturally-named commands and some hairy syntax. When designing your site, you don't really need to know the syntax; you mostly set it and forget it. More importantly, you should try to learn Apache's capabilities, so you can make appropriate designs. Here is a coarse summary of this lesson:

- Apache is a program that understands HTTP, receives requests, and sends responses.
- Apache's configuration comes in text files, which may exist "globally," with one file including another starting at httpd.conf, or within directories, in which case they apply in a cascading manner, in combination with the global configurations.
- Management of custom error pages, gzipped content, and custom directory listings are just some of the ways that Apache fits into a site architecture.
- Each (nontrivial) line of an Apache configuration file is a directive, and the first word of the directive is the directive name. Even if you see some kind of English-looking phrase, like "Allow from all", you should look for "Allow" as a directive name if you want to find it in the documentation.

- Apache may call any text-based program (commonly PHP) in order to generate text that it sends back as a response, instead of sending the contents of a file.
- Apache may implement the HTTP authentication method, which is a simple way of checking passwords for client authentication.
- Apache may rewrite requests and change them into entirely different beasts; this allows you to hide your scripts and present a clean interface to the client, among other things.
- Regular expressions may be used to enhance common configuration-file rules and the URI-rewriting engine.
- Each request in a Web server should not use too much RAM; Apache allows limitations on the number of concurrent users and other such performance parameters.