NGINX

From zero to hero

The directives

Talking to Nginx

- Of course we do. But to talk, you need a language that is understood by whom you are talking to. In our case, that's Nginx.
- Nginx's language is made up of directives. Think of them as instructions to the server to behave in one way or another.
- Nginx supports two types of directives: simple and block.
- A simple directive is made up of one statement ending with a semi colon (;).
- A number of simple directives can be grouped together in a block enclosed in curly braces ({}).
- Both types of directives exist in the configuration file, typically located in `/etc/nginx/nginx.conf`.
- Blocks can also be called "contexts". They can be nested inside each other.

Simple directive examples

- As mentioned, simple directives are made up of single lines terminated by semicolons. The most common examples of single directives include:
  - `user`: This specifies the user and group account under which nginx operates. A user and group must be created on the server before nginx can use them.
    `user nginx nginx`
    Where the first nginx refers to the user, the second one refers to the group.
  - `Worker`_process: This defines the number of processes that the daemon will create (spawn). Typically, this should reflect the number of cores installed on the machine. The default value is 1 but it can be also set to auto, where nginx will try to determine the number of cores.
  - `Error_log`: This instructs nginx where to log encountered errors and "how verbose" it should do that. The level of verbosity can be specified by setting the minimum level of logging errors. Error levels in ascending order are:
    - `Info` – logs information-level events in addition to all higher error levels (the most verbose).
    - `Notice` – logs notice-level events and all high error levels
    - `Warn` – logs warnings plus higher error levels
    - `Error` – logs error-level errors as higher error levels

    - `Crit` – logs critical- as well as all higher error levels
    - `Alert` – logs alert- and emergency-level errors
    - `Emerg` – logs only emergency-level errors (the least verbose).

- To see a live demonstration of this, change the error log level to info, and monitor the log file by issuing `tail –f /var/log/nginx/error.log`. Then open another terminal and build the following while loop:
  ```
  while true; do wget -p 127.0.0.1 -O /dev/null; done
  ```
  what this is doing is that it is continuously hammering the server with a huge number of get requests. Have a look at the log file and examine the different messages that get logged. Change the log level to less verbose one, issue `service nginx reload` or `/usr/sbin/nginx –s` reload to reflect the changes and see the difference.

- `pid`: this directive contains the path to the file that stores the process id of the nginx master process (parent process). This is useful whenever you want to quickly determine process id of nginx for scripting reasons. The alternative way would be to issue a command like `ps -ef | grep nginx | grep master | awk '{print $2}'` instead of simply `cat /var/run/nginx.pid`

The Events context

- There can only be one events context in the whole configuration file, and it can only be placed in the main (outermost) context.
- It controls various aspects of nginx behavior. It has the following simple directives:
  - `worker_connections`: this controls the maximum amount of concurrent worker connection that nginx would handle (multiplied by the number of worker processes or CPU cores). That is, how many people can be served simultaneously by nginx. The default value of 1024 is fine for most cases.
  - `use`: instructs nginx to use a specific connection processing method. This is set automatically and you'll seldom need to change it. For your reference, the less efficient methods are select and poll. They are used on platforms that lack more efficient methods (like Microsoft Windows). More efficient methods include kqueue, /dev/poll, eventport, and epoll (used on Linux kernels of version 2.6 or higher).
  - `multi_accept`: specifies whether or not nginx worker process can accept more than one connection at the same time. This is disabled by default (off|on).
  - `accept_mutex`: when enabled (which is the default), it makes nginx workers accept new connections one process after another. If disabled, all the processes will be notified when new connections are received. The problem here is that if the volume of connections is low, the processes will just be wasting system resources without actually serving requests.
  - `accept_mutex_delay`: this works only if accept_mutex is enabled. Because accept_mutex makes only one worker process accept new connections, other processes would wait for the time specified by this directive (in milliseconds) before they attempt to "check" whether or not the worker process has finished to start taking their turn. This is 500 ms by default.

The HTTP context

- This is the most important part of nginx configuration if you are going to use it as a web server (you can disable the HTTP module by compiling nginx with --without-http)

- The HTTP context contains several child contexts and simple directives that define the behavior of nginx when used as a web server:

  - `include`: this allows you to point to the location of another file, of which contents will be placed in the same spot. This allows for organizing the configuration file thus increasing readability. For example, include /etc/nginx/mime.types; will grab the contents of /etc/nginx/mime.types as if they were part of the nginx.conf. The mime.types file contains the types context. This defines the different MIME types that nginx should serve based on the file extension. For example, application/javascript for *.js, text/plain for *.txt, and image/jpeg for *.jpg and *.jpeg. You can try and add your own extension to this file and test it. For example: application/json json conf; will make nginx serve *.conf to the browser as JSON-formatted files so that the browser can render it accordingly instead of downloading it.

  - `default_type`: it has the default value of application/octet-stream. This is the fallback MIME type that nginx would use if the file extension does not match any of the specified in mime.types. That's why your browser will attempt to download the file for which no MIME type is known for nginx rather than display it; because it is served to it as application/octet-stream (downloadable file).

  - `log_format`: specifies how nginx should write a line of data to the log file. It contains two parameters: the name of the format (main by default) and a number of placeholders that would be filled when actual data is written.

- `access_log`: the name and path of the access log file. This file records each and every request received by nginx. The directive also accepts the name of the format by which log lines will get written to this file.

- `sendfile`: it is recommended to keep its value to on, which is the default setting. This directive ensures that nginx disk operations will not block disk I/O. That is, when a large file is read from the disk, nginx will retrieve its contents in chunks. This greatly improves performance especially with large files.

- `tcp_nopush`: this is commented by default. It instructs nginx to serve the response headers and the beginning of the file in one packet. Then sending the file contents in full packets.

- `keepalive_timeout`: This defaults to 65 seconds. It defines the amount of time nginx would keep the connection open with the client. This is recommended so that the browser needn't open a separately new connection for each and every component on the web page (think of images, CSS files, JS files…etc.). If you are having a lot of hits, you may want to decrease this setting a little so that connections are not wasted on clients that have already finished loading the page in ten seconds or less.

- `gzip`: this enables data compression to enhance performance. It is enabled by default and recommended to be left like that.

The server context

- The next directive is an instruction to include the `*.conf` under `/etc/nginx/conf.d/`. This basically means `/etc/nginx/conf.d/default.conf`, but you can add other configuration files like an SSL configuration file if you plan to server HTTPS content. Any file(s) you place in this directory will be nested inside the http context.

- The first context in the `default.conf` file is the server context. This can be nested in more than one context not just http. You can also have more than one server context in the same `default.conf` file. This allows you to host more than one virtual server on the same nginx engine. This is quiet similar to Virtual Servers in Apache.

- In the following lab we are going to create a new configuration file, `servers.conf`. Inside this file we'll create two server contexts that define two separate applications (two directories each having an index.html file).

LAB 01: create two virtual servers on nginx

- Comment out this line in /etc/nginx/nginx.conf by adding a # at the start to be as follows:
```
    #include /etc/nginx/conf.d/*.conf;
```

- Create a two new directories under /usr/share/nginx/html: wordpress and joomla:
```
mkdir /usr/share/nginx/html/joomla
mkdir /usr/share/nginx/html/wordpress
```

- Add a file index.html in each directory containing the following text respectively:
```
<h1>Welcome to Wordpress</h1>
<h2>Welcome to Nginx</h1>
```

- Create a new file `/etc/nginx/conf.d/servers.conf` and add the following content:
```
server {
        listen 80;
        server_name www.wordpress.example.com  worpdress.example.com;
        location / {
                root /usr/share/nginx/html/wordpress;
        }
}
server {
        listen 80;
        server_name www.joomla.example.com joomla.example.com;
        location / {
                root /usr/share/nginx/html/joomla;
        }
}
```

- issue `nginx -t -c /etc/nginx/nginx.conf` to test the configuration file for any syntax errors.
- Issue `service nginx reload` (or `/usr/sbin/nginx –s` reload) to reflect the changes without killing the nginx process.
- Since both of them are listening on the same port 80, nginx can only distinguish between both of them through the server name that comes in the request, whether it was wordpress or joomla.
- But you can also change this behavior and let nginx serve the required application based on the port. Change joomla to be listening on port 8080 and now the user can see “Welcome to Joomla” regardless of the URL (as long as :8080 is appended to the server name/IP).
- Notice that if the user did not request any of the specified URLs, for example, visiting http://serverip. This makes nginx fallback to the default server, which is the first context in `servers.conf`.
- You can change this behavior by forcing nginx to serve a specific virtual server as a fallback by adding default_server to the listen directive. For example: listen 80 default_server;
- Notice that default_server can be configured multiple times but on different IP addresses and ports. There could not be more than one default_server directive per IP:port.
- Some security regulations require that the web server should not display any content unless visited by the specified URL. That is, visiting localhost or 127.0.0.1 or the server 192.16.1.111 should not bring up any content. To do this, add a third stanza the file as follows:

```
server {
        listen 80;
        server_name "" 127.0.0.1 localhost 192.168.1.111;
        return 404;
}
```

- The previous stanza states that:
  - Listening on port 80
  - If the server name was not supplied in the request or if it was localhost or 127.0.0.1 or the assigned server IP
  - Return 404 (an HTTP response that means page was not found).
  - Let's test nginx behavior on different IP addresses. If you are on VirtualBox. Create a new network card of any type other than NAT. Take note of the IP address. If you are working on a physical server with only one network adapter you can mimic this behavior by adding an IP alias using a command as follows:
    ifconfig eth0:0 192.168.1.112 up
  - Now that we have more than one IP address on the node, let's modify the joomla application stanza to listen on that IP address:
    ```
    server {
        listen 192.168.1.112:80;
        server_name joomla.example.com;
        location / {
        root /usr/share/nginx/html/joomla;
        }
    }
    ```
  - If you try to visit joomla.example.com it will serve the correct page. But if you modified your hosts file (or the DNS) to map joomla.example.com to 192.168.1.111 (the primary IP), nginx will serve wordpress (or the default_server stanza). This is because nginx checks the IP address and port before checking the host address. So when it didn't find joomla on 192.168.1.111, it served the fallback server.
  - You can even use wildcards (*) to specify the host address that should serve a specific application. However, nginx allows placing an asterisk only at the start or end of the hostname. For example, *.example.com is ok, joomla.example.* is also ok but not joomla.*.com.

The location context

- This is how nginx knows which files to serve based on the request it receives. For example, in the previous lab we used location / to indicate the root of the web address (the slash at the end of wordpress.example.com/ which is interpreted by the web server even if omitted from the address).

- Notice that index.html file was served by default even when it was not mentioned in the address. This can be changed by modifying the index directiventext. For example:

```
server {
    listen 10.240.3.243:8080;
    server_name "joomla";
    location / {
        root /usr/share/nginx/html/joomla;
        index primary.html primary.htm
    }
}
```

- This makes nginx serve primary.html or primary.htm files if no file was specified in the URL.

- You can also place more than one location context under server. This allows you to load different index files based on the requested URL. To demonstrate that, clear the contents of servers.conf and replace them with the following:

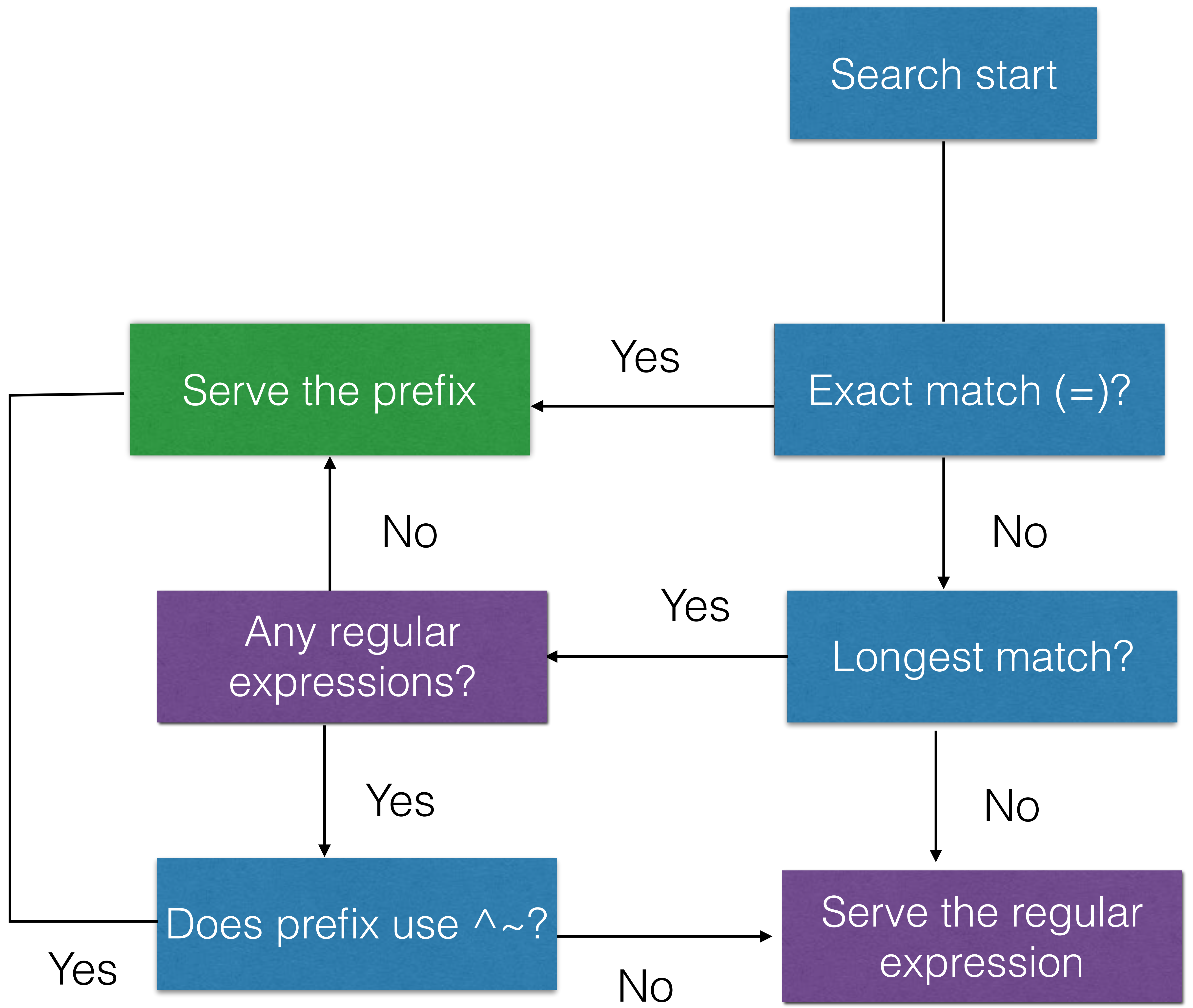
```
server {
   listen 80;
   server_name joomla localhost 127.0.0.1 192.168.1.111;
   location /joomla {
        root /usr/share/nginx/html/;
        index default.html;
   }
   location /wordpress {
        root /usr/share/nginx/html/;
        index index.html;
   }
}
```

- This could be better written by removing the redundant element root and placing it only once in the server context:

```
server {
   listen 80;
   root /usr/share/nginx/html/;
```

Location pattern-matching

- The location context also enables you to apply rules up to individual files. It is also capable of using Regular Expressions for even more control. The following are the symbols used for matching:
  - None: nginx regards the match as a prefix
  - ~* perform a case-insensitive match
  - ~ perform a case-sensitive match
  - ^~ stop regular expression checking if a match is found
  - = match the exact URI. No further search is performed.
  - To understand how those location modifiers work, we have to know first how nginx determines the location to serve:
    1. It searches for an exact match (=). If one is found it is immediately served. Search is stopped. If not, the second step is followed.
    2. It searches for the longest matching prefix (the most specific) and deals with it as follows:
       1. If the ^~ is used this location is served. Search is stopped.
       2. If not, nginx stores this longest matching prefix and the search continues to step 3.
    3. It examines regular expression locations that use case sensitive (~) and case insensitive (~*) modes. It does this sequentially and the first one found is served immediately and search is stopped. If not, step 4 is followed.
    4. Finally, the longest matching prefix location is found and served.
  - Now the question is: What if nginx is only left with the longest matching prefix and a regular expression match, which one would it choose? the answer is: it depends on whether or not the ^~ is used. ^~ instructs nginx to serve the prefix and ignore the regular expressions.


Search start
Exact match (=)?
Yes
Serve the prefix
No
Longest match?
Yes
Any regular expressions?
No
Yes
No
Does prefix use ^~?
Yes
No
Serve the regular expression

LAB: play with location modifiers

- The only way to fully understand how nginx uses location modifiers to serve content is by experimenting with it. In this lab we are going to create a new website, brands. It contains a number of images of company brands. Image have different formats. Modify the servers.conf file to be as follows:

```
server {
     listen 80;
     server_name brands localhost 127.0.0.1 192.168.1.111;
     root /usr/share/nginx/html/brands;
     location  / {
          index index.html;
     }
     location  /hidden/ibm.jpg {
     }
     location /hidden {
          return 403;
     }
     location ~* \.jpg {
          return 406;
     }
}
```

- Now when you request / the index.html page is displayed. When you request /hidden a 403 header is displayed. However, when you request /hidden/ibm.jpg, and because this request is longer than /hidden, the second rule will be matched and you can see the image. Now if there is another image, oracle.jpg inside the hidden directory then:
  - With the current configuration, a 406 header will be displayed.
  - If you want the 403 header to be displayed instead, you have to modify the second rule to match /hidden as location ^~ /hidden

try_files directive

- The try_files directive can be used in the location context. It enables you to serve a file, a status code, or another location if the file the client is requesting does not exist.
- Let's say, for example, that you want to display a customized error message to the visitor when the requested brand image cannot be found:
    - First, create a new file in brands directory, call it 404.html and write whatever message in it.
    - In the location context, add the following line:
      try_files $uri /404.html;
- Now whenever a user navigates to an image/page that does not exist, the 404.html page will get displayed.
- Notice how the $uri variable is used to indicate the URI that the user has requested. The $uri always refers to files, if you want to use it to refer to a directory, append a slash to the end ($uri/)
- You have a couple of more options here:
    - You can add =*header* after the fallback page to serve that header while displaying the page (for example, 406)

The rewrite directive

- If you've worked with the Apache web server before, you've probably used the rewrite module at least once. A rewrite simply takes the requested URI from the client and translates it to another URI to be correctly interpreted by the web server. It is often used to produce "pretty URLs". It used Perl regular expressions to process the URI text.

- For example, you want to give your visitors (or search engines) the following URL: http://example.com/ibm but your application server is expecting the URL to be like this: http://example.com/index.html?id=ibm the rewrite directive can do this for you.

- In the location context, add the following line:

```
rewrite ^/(.*)$ /index.html?id=$1 break;
```

- The rewrite directive accepts an optional flag at the end of the line (break in the above example). The flag is one of the following:

  - last: instructs nginx to stop processing the current set of directives and starts searching for the new (changed) URI (perhaps to execute more rules on it)

  - break: stop the current set of rules. We used it in the example to avoid an endless loop.

  - redirect: issues a temporary redirect response 302.

  - permanent: issues a permanent redirect response 301. Both permanent and redirect flags are often used for search engine optimization reasons.

The error_page directive

- This allows you to customize the error pages that appears to your visitors.
- It can be used as globally as in the http context, it can also be more specific by being placed in server context to be per virtual server or even at the location context to serve different error pages for different locations on the same server.
- You can use it as follows:
  - `error_page 404 /friendly404.html`
  - `error_page 404 406 402 /friendly4x.html`
  - Or even `error_page 404 =200 /index.html` which will send a response with code 200 (ok) and presents the home page to the user.

Managing directory access

- By default, nginx block directory listing of directories that do not contain an index file (defined with the `index` directive). However, sometimes you may want to explicitly allow this. For example, a directory that contains files that are meant for download should have listing enabled for easier navigation.

- To do this to a directory called downloads, add the following line to the `location` context:
  `autoindex on`

- However, beware that you should either remove the index directive or set it to a file that does not exist. Otherwise nginx will attempt to serve that file to the client instead of listing the directory contents.

- You may also want to prevent visitors from navigating to a specific directory. Perhaps it contains protected information or configuration files. This can be done by adding the `deny all` directive to the appropriate location. For example:

```
location /protected {
        deny all;
}
```