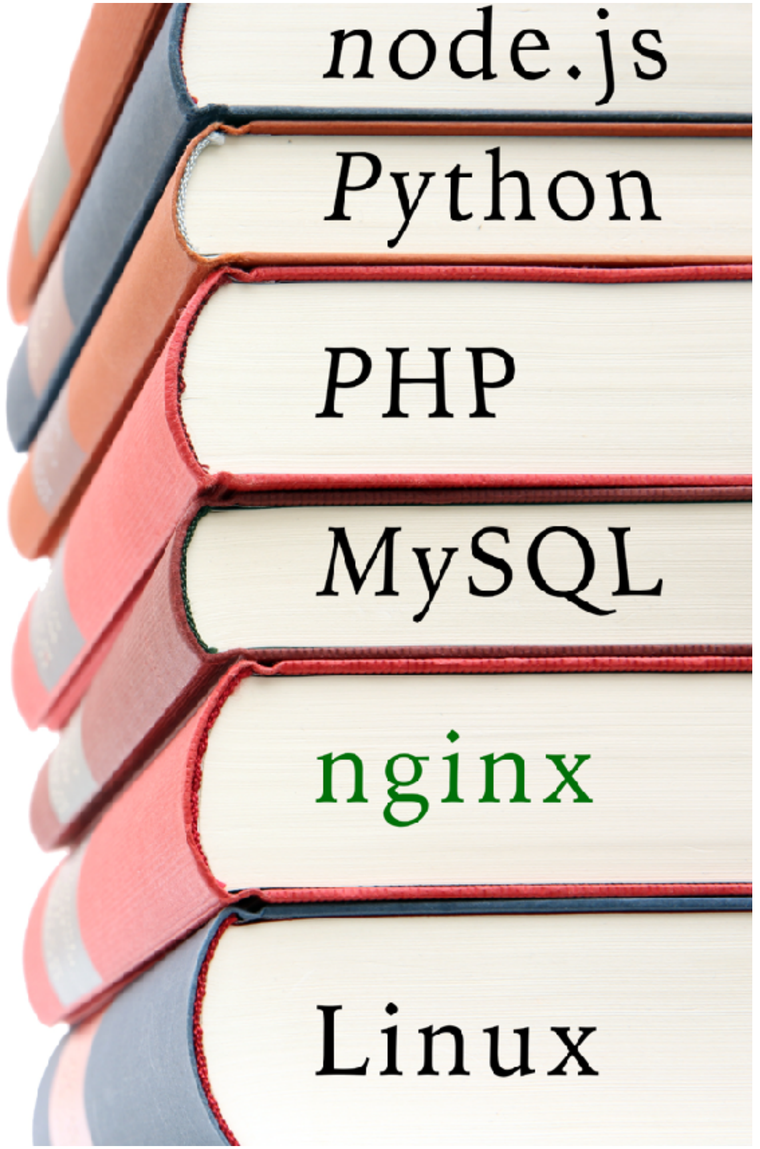

# From zero to hero

# The LEMP stack

nginx as a reverse proxy



# Static vs. dynamic websites

- Static websites are as old as the Internet itself. When the first ever web server was created, it served text files containing hyperlinks to other text files. Those files were static because:

- They were physically saved somewhere on the web server and that physical location is translated to the web address. For example, `/var/www/html/mywebsite/hr/staff.html` is `http://www.mywebsite.com/hr/staff.html`

- They did not (and could not be) changed at runtime. A `staff.html` containing a list of employee information will not give the HR manager more information about each employee (like their salaries) nor will it prevent non-HR visitors from viewing sensitive information that such a list may contain (like e-mail addresses).

- On the other hand, a dynamic webpage:

- Does not necessarily need to be physically saved on the web server with the same name/path that appears on the web address. For example, the staff.html page can be accessed as `http://mywebsite/hr/staff` while the actual file that serves this content might be `/var/www/html/mywebsite/index.php`, with the requested data passed at runtime in the form of a query string (like index.php?page=staff).

- The same page can have different content depending on whatever condition the developer requires. For example, e-mail and salaries information can be shown only to authorized staff.

- Nowadays modern websites use both types of content to serve visitors, with the dynamic content being for the main service, and the static content for assets like images, JS files, CSS files and so on.

# The LEMP stack

- LEMP stands for Linux, eNginx, MySQL, and PHP. This is how web developers often refer to the PHP development stack when combined with Linux and MySQL served on an nginx web server.

- The most important thing to notice first is that nginx does not have modules for serving dynamic content. In other words, there is no PHP module, for example, that can be compiled and enabled with nginx so that it can dynamically process and serve *.php files.

- Instead, nginx does what it does best: it proxies the request to whichever backend components that should handle dynamic content, then it routes the result from those components to the end client.

- This way, you benefit from serving non-static content to your visitors while also gaining the enhanced performance and scalability offered by nginx.

# LAB: install MySQL/MariaDB

- We already have the L, and the E. Let's install the M (MySQL).
- A quick note: on Red Hat 7 based systems (like Centos 7), you may want to install MariaDB instead of MySQL.
- MariaDB is a fork of MySQL created by some of the developers who developed the original MySQL database engine. The fork was created as Oracle acquired MySQL and there were fears that the product will not be maintain the open-source license that it originally had. It is a *drop-in* replacement for MySQL and can by used the same way.
- The following commands will install MariaDB components on Centos.

```
yum -y install mariadb
```

- Now let's set a new password for the root user:

```
mysql -u root
use mysql;
update user set password=PASSWORD("admin") where User='root';
flush privileges;
```

# PHP integration

- PHP stands for PHP Hypertext Preprocessor. It is a very well-known web programming language used in a lot of dynamic web applications. When installed, PHP can run in one of the following modes:

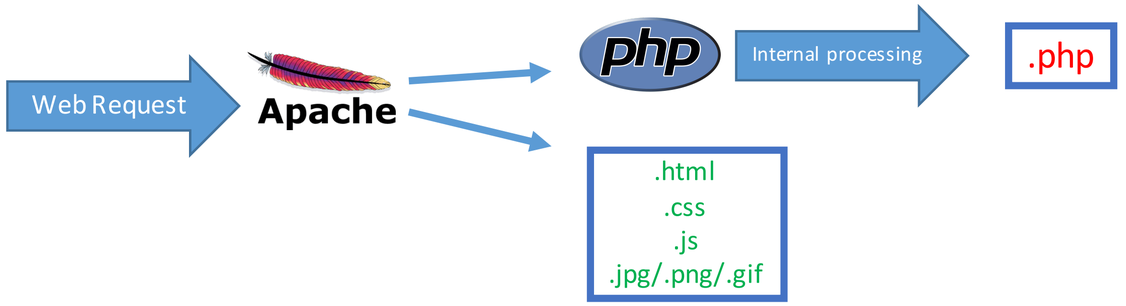
  - **mod_php:** this can only be used with Apache. When enabled, each process spawned by Apache has the PHP interpreter embedded inside. This enhances performance on busy web applications because no external process needs to be called. The downside is that Apache consumes more resources even when serving static (non-PHP) files. Any changes done to php.ini means that Apache must be restarted for the changes to be reflected.

  - **FastCGI:** this is the successor of the legacy CGI (Common Gateway Interface). It allows the PHP interpreter and the web server to be split. The web server can handle static content and only route requests that need to be processed by the PHP interpreter to the backend service. Only one process is spawned that can serve multiple requests. This method offers less resource consumption and is not bound to one specific web server.

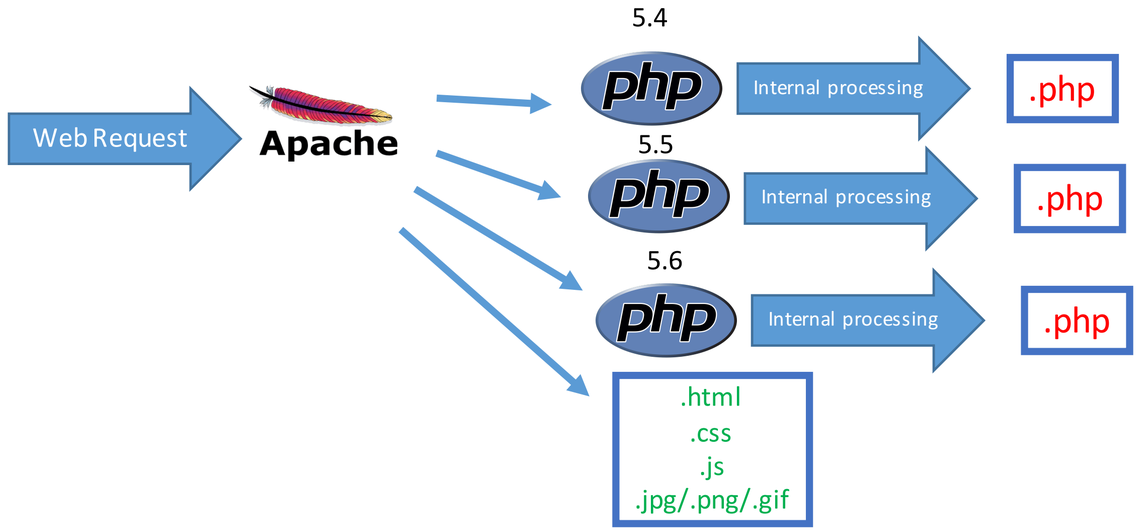
  - **PHP-FPM:** it is the PHP implementation of FastCGI. It offers more features than FastCGI like, for example, you can run multiple PHP processes, each with a different php.ini file or even with a totally different PHP version.

# mod_php


# FastCGI




# PHP-FPM



# LAB: Install PHP and enable it on nginx

- The following command will install PHP and php-fpm service: `yum -y install php php-mysql php-fpm`
- Let's create a new user that will be used for managing PHP files (nginx user will be for the static files):
  `useradd webuser;`
  `passwd nginx --stdin <<< 'P@ssw0rd'`
- Once done you need to make some adjustments to the php-fpm configuration. Edit `/etc/php-fpm.d/www.conf` and ensure that user and group are set to `webuser`. Restart php-fpm service by running `service php-fpm restart` (or `sysctl restart php-fpm`).
- Edit /etc/nginx/conf.d/servers.conf to be as follows:

```
server {
        listen 80;
        server_name localhost 127.0.0.1 10.240.3.240;
        root /usr/share/nginx/html;
        index index.php index.html;
        location / {
                try_files $uri $uri/ =404;
        }
        error_page 404 /404.html;
        error_page 500 502 503 504 /50x.html;
        location ~ \.php$ {
                try_files $uri =404;
                fastcgi_pass 127.0.0.1:9000;
                fastcgi_index index.php;
                fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
                include fastcgi_params;
        }
}
```

- As you might guess, FastCGI (through php-fpm) is invoked by the last location block. Let's have a quick look at what each directive in this block does:
  - First, it instructs nginx that this location block will be applied only on URLs that end in .php (case sensitive). The case-sensitivity is required here so as you don't get penalized by search engines for serving duplicate content.
  - Then it avoids routing any bad requests to the php-fpm servers by using try_files directive and serving 404 response as a fallback.
  - The actual routing happens when `fastcgi_pass` directive is used to pass requests to the FastCGI server. Notice here how you can replace 127.0.0.1 with a totally different physical server that would be dedicated just for serving PHP files.
  - The fast_cgi index defines the default page to be served if none was supplied at the end of the URL. For example, if the request comes like `example.com/users/ FastCGI` should infer that it `example.com/users/index.php` is the requested file.
  - Now this the next directive is the heard of FastCGI. It is what translates the URL arriving at nginx to an address that is understandable by FastCGI. The `SCRIPT_FILENAME` is a FastCGI parameter. Nginx is telling FastCGI that whenever a file is requested, for example /login.php, it should interpret it as `/usr/share/nginx/html/` (defined as `$document_root`) followed by `login.php` (`$fastcgi_script_name`), unless there is not file passed to the URL, then FastCGI should assume it is index.php (defined as `$fastcgi_index`).
  - The last directive is to include `fastcgi_params` file. This file contains the rest of FastCGI-related directives. It ships by default with nginx installations.

# LAB: install WordPress

- First we need to create a database to host our cms:

```
mysql –u root –p
#Enter the configured password
create database wordpress;
GRANT ALL ON wordpress.* to 'wordpress_user'@'localhost' IDENTIFIED
BY 'wordpress_password';
exit;
```

- The download the latest WordPress installation source file:

```
cd /usr/share/nginx/html
rm –rf wordpress # Delete the old wordpress directory if it exists
wget https://wordpress.org/latest.tar.gz
tar –xvf wordpress-4.7.3.tar.gz
chown –R webuser:webuser wordpress
```

- Now navigate to the web server's URL either through localhost, 127.0.0.1, or the public IP address and follow the installation wizard.

# Solving the permalinks problem

- WordPress is an amazing platform that powers millions of websites worldwide. One of the great features that it offers is the ability to serve pretty URLs, so `http://mywordpress.local/index.php?hello-world` can be offered to visitors (and search engines) as `http://mywordpress.local/hello-world`.

- On Apache, this works out of the box. Even if rewrite rules need to be added, WordPress writes the necessary rules in `.htaccess` file for you.

- On nginx, this has to be done manually. Fortunately it only involves changing one line in the `servers.conf` file:

```
location / {
     #try_files $uri $uri/ =404;
     try_files $uri $uri/ /index.php?$args;
}
```

- Then reload your nginx server: `service nginx reload`.

# Working with Python Flask

- Flask is a micro server that is built using Python. It is used for creating small applications or APIs (Application Programming Interfaces).
- It can be used when you need a “service” rather than a web server. For example, it can be used to connect to a database, do some processing and send the output to the client (which is usually another server in the stack rather than the end client).
- When used in that way, it is called a RESTful service. It uses the HTTP protocol and data is sent back and forth using JSON rather than traditional HTML.
- But you can definitely use it as an application server that generates HTML. But Flask alone cannot be used in a production environment as it does not scale well with increasing number of requests. Besides, it can serve only one request at a time by default. Seems like a perfect candidate for nginx which can fill this gap.
- To make it play nice with nginx, we need an intermediary component: a WSGI (Web Server Gateway Interface) server. A WSGI is used the same way FastCGI and php-fpm is used: it translates request from Flask to nginx and vice versa.
- In the following lab, we’ll deploy a sample Flask application and use nginx as the front end web server.

# LAB: install Flask and deploy a sample app

- Python is usually shipped with Linux. But if not, you can install it by running `yum -y install python` or `apt-get install python` depending on your platform.

- We also need pip installed. This is a Python package manager that will be used to deploy Flask easily:
```
curl "https://bootstrap.pypa.io/get-pip.py" -o "get-pip.py"
python get-pip.py
```

- Install Flask by running `pip install flask`

- Instead of building a new application from scratch we'll use an already made one: flaskr. This is an example Flask app that can be downloaded from https://github.com/pallets/flask

- Now let's deploy the application using the instructions found on its help page:
```
cd flask/examples/flaskr/flaskr
pip install --user --editable .
export FLASK_APP=flaskr
flask initdb
flask run --host=0.0.0.0
```

- If you have a firewall, issue the following command (using root) to enable traffic on port 5000. On Centos 7: `firewall-cmd --zone=public --add-port=5000/tcp --permanent && firewall-cmd --reload` and on Ubuntu: `iptables -I INPUT -p tcp --dport 5000 -j ACCEPT && service iptables save`.

- Verify that the application has been successfully deployed by navigating to http://yourip:5000. You should see a minimal microblog. Login with the following credentials:
  user: admin
  password: default

# LAB: installing uWSGI

- Python is compatible with a number of application servers that can be used with nginx. In this lab we are going to use uWSGI (an implementation of the WSGI protocol). To install it, make sure that `python-devel` package is installed or run `yum -y install python-devel` .Then run the following command: `pip install uwsgi`

- Create a file wsgi.py (you can call it any name you want) and place it somewhere on your filesystem. Add the following content to it:

```
from flaskr import app
if __name__ == "__main__":
    app.run()
```

- With `uswgi` installed, we need to run the application. Issue the following command:

```
uwsgi --http :5000 --chdir /home/webuser/flask/examples/flaskr/flaskr
--wsgi-file /home/webuser/flask/examples/flaskr/flaskr/wsgi.py --
callable app
```

- Navigate again to http://yourip:5000 and ensure that the application is running as expected.

# LAB: link uWSGI to nginx

- Now it's time to invoke nginx. To do this, we have to first run uWSGI in a way that allows nginx to act as its front end server. Run uWSGI (as `webuser`) using the following command:
  nohup `uwsgi --socket :5000 --module wsgi --chdir /home/webuser/flask/examples/flaskr/flaskr --wsgi-file /home/webuser/flask/examples/flaskr/flaskr/wsgi.py --callable app` &

- As the root user, edit the servers.conf to look as follows:

```
server {
listen 80;
server_name localhost 127.0.0.1 192.168.1.111;
  location  / {
      include uwsgi_params;
      uwsgi_pass 127.0.0.1:5000;
  }
}
```

- Issue `service nginx reload` and navigate to http://yourip. Ensure that the application is working as expected.

- Note that the above example could be greatly enhanced to boot performance using the techniques that you already know by now. For example, you could add another location directive for serving static files (like css, js, html, and images) and instruct nginx to serve them instead of passing all the requests to the backend Flask server.

- You could also split this architecture so that Flask runs on a  separate server than nginx, which would give you more control and better performance.

# Working with node.js

- Ok I know that node.js does not fit in the LEMP stack; it does not start with P, in addition, it has its own MEAN stack (Mongodb, Express, AngularJS, Node.js). However, it provides an excellent example of how nginx can be used as a reverse proxy for performance enhancement.

- So what is node.js? It is an application server built using the V8 JavaScript engine (used in Google Chrome). It provides a robust performance and non-blocking I/O (just like nginx).

- It is often used to build high-performance network applications , especially realtime ones like chatting apps. But it can also be used as a traditional web server serving HTML. The express.js module greatly facilitates this task.

- Now why should you use nginx with node? For the same reasons a reverse proxy is used: an application server should be only concerned with running the application it should not:

  - Serve static files

  - Perform redirects

  - Generate HTTP error codes

  - Handle SSL traffic

- Those should be performed by the reverse proxy (nginx in our case) to achieve the best performance. In the rest of this section we are going to install Node.js and Express.js, build a sample app, and connect it with nginx as a reverse proxy.

# LAB: install node.js and express.js

- There is more than one way to install node.js. I will go for compiling it from source. This ensures that I am getting the latest version out there and also provides a procedure that works equally on all supported platforms.
- Download the source code gzipped file by issuing `wget https://nodejs.org/dist/v6.10.0/node-v6.10.0.tar.gz`
- Uncompress the file: `tar -xvf node-v6.10.0.tar.gz`
- Build the application by issuing: `cd node-v6.10.0 && ./configure && make && make install`
- Once done, install Express.js by running (as root) `npm install express -g`. Then we'll install Express generator, which will create a number of scaffolding files that will provide a skeleton application out of the box: `npm install express-generator -g`.
- Switch back to webuser, and ssue the following command: express nodeapp. This will create the application is a directory called nodeapp.
- Now issue `npm install` followed by `npm start` to start the service in the foreground. Express by default runs your applications on port 3000 so you must enable this port on the firewall. As root issue the following command: `firewall-cmd --zone=public --add-port=3000/tcp --permanent && firewall --reload` or on Ubuntu: `iptables -I INPUT -p tcp --dport 3000 -j ACCEPT`
- Navigate to http://yourip:3000 and you should see the default page of Express.js scaffolded application.
- Of course running the application in the foreground is far from efficient. We could usfore `nohup` and `&` as we did with Flask, but fortunately Node.js has already a number of process managers that will make sure the service runs in the background while providing a lot more features. I will choose Forever for this lab but you can visit https://expressjs.com/en/advanced/pm.html and pick what you favor.
- To install Forever, drop back to root (press CTRL-D) and issue the following command: `npm install forever -g`
- Once done, switch back to `webuser` and inside nodeapp directory issue the following command: `forever start ./bin/www`. This will start the server in the background. Issuing `forever list` will give you more information about the running servers. Now try to visit http://yourip:3000 and you should see the same result as before.

# LAB: plugging in nginx

- Let's introduce nginx to the architecture and let it handle HTTP requests from clients, route it to node.js and sends back the response it receives.

- Edit servers.conf so that the contents look as follows:

```
server {
        listen 80;
         server_name localhost 127.0.0.1 192.168.1.111;
         location  / {
                 proxy_pass http://127.0.0.1:3000;
         }
}
```

- Reload nginx by issuing service nginx reload. Now navigate to http://yourip, you should see exactly the same page you saw when visiting the same URL but at port 8000.

- Again, you can use the location directive techniques that we learned so serve static files without passing their requests to node.js, issue friendly error messages or even serve pretty URLs (although the way node.js and express.js work does not need URL rewriting).