

NGINX CORE

Flawless Application Delivery

Trainer Intro

James Tacker

Technology Consultant &
Content Developer

Previous Training Work:

- Sauce Labs
- New Relic
- Salesforce
- Atlassian

james.tacker@servicerocket.com



Prerequisites/Expectations

- Sysadmin, DevOps, Solution Architect
- Some familiarity with Web Servers
- Some familiarity with Linux
- Text Editor: Vim, Vi, Emacs etc.
- Some knowledge of Networking

The Training Environment

- AWS EC2 Instances
- Ubuntu
- NGINX Plus

Log Into AWS

If you haven't done so already, please take the time to SSH into your EC2 Instances (Windows users use **PuTTY**).

Check your email for the login credentials, check your spam folder!

```
ssh student<number>@<ec2-server-hostname>
```

Course Administration

- Course Duration: 4 hours
- Takes place over 2 days
- Ask questions at any time!

Agenda: Day One



- Overview
- Serving Static Content
- Proxying Connections
- Logging
- Security
- Variables
- Routing Connections

What is NGINX?



HTTP traffic



Reverse Proxy

Caching, Load Balancing...



Webserver

Serve content from disk



Application Gateway

FastCGI, uWSGI, Passenger...

NGINX OVERVIEW

Module Objectives

This module enables you to:

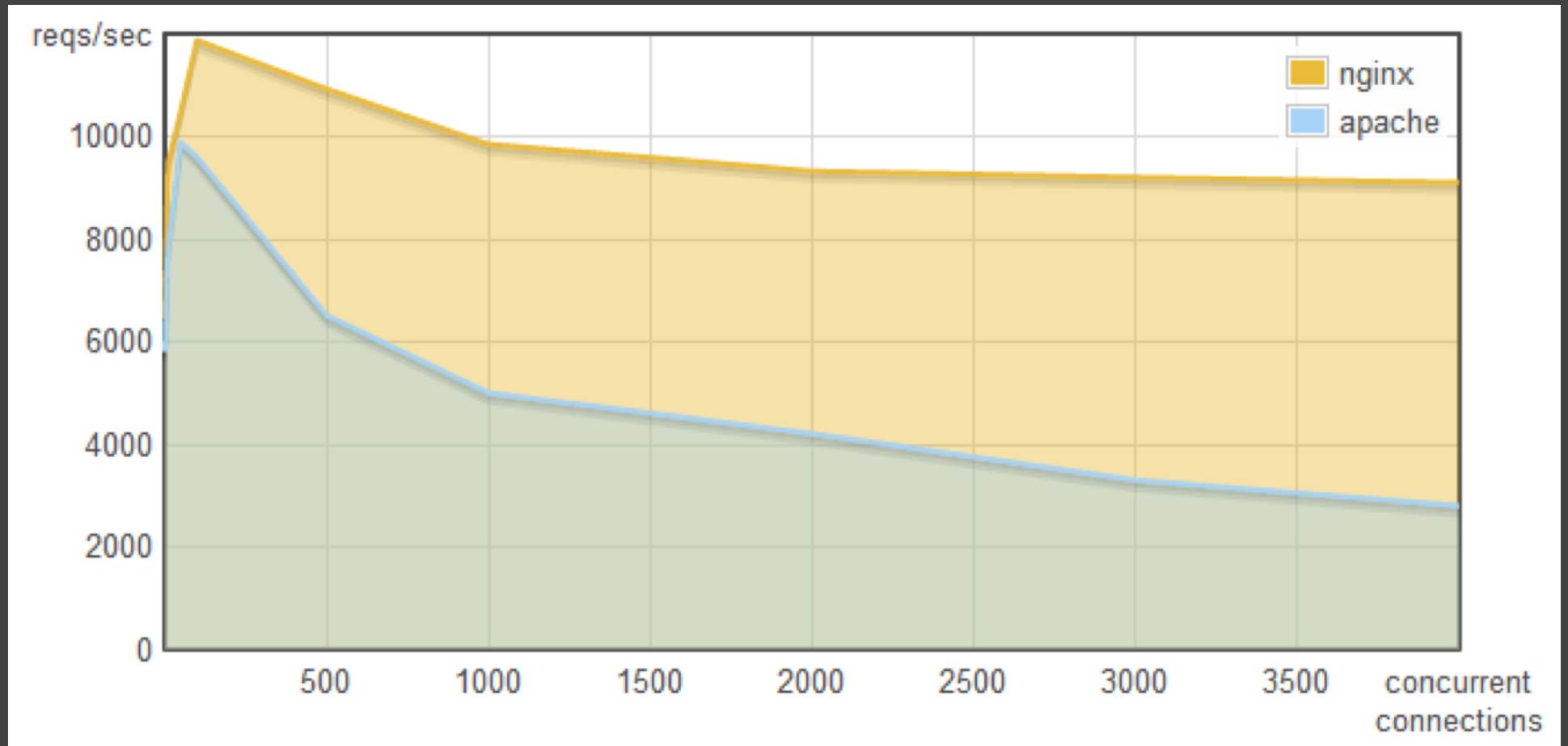
- Gain a basic understanding of NGINX's features
- Learn about the history of NGINX
- Understand the various use cases that NGINX supports

Origins of NGINX

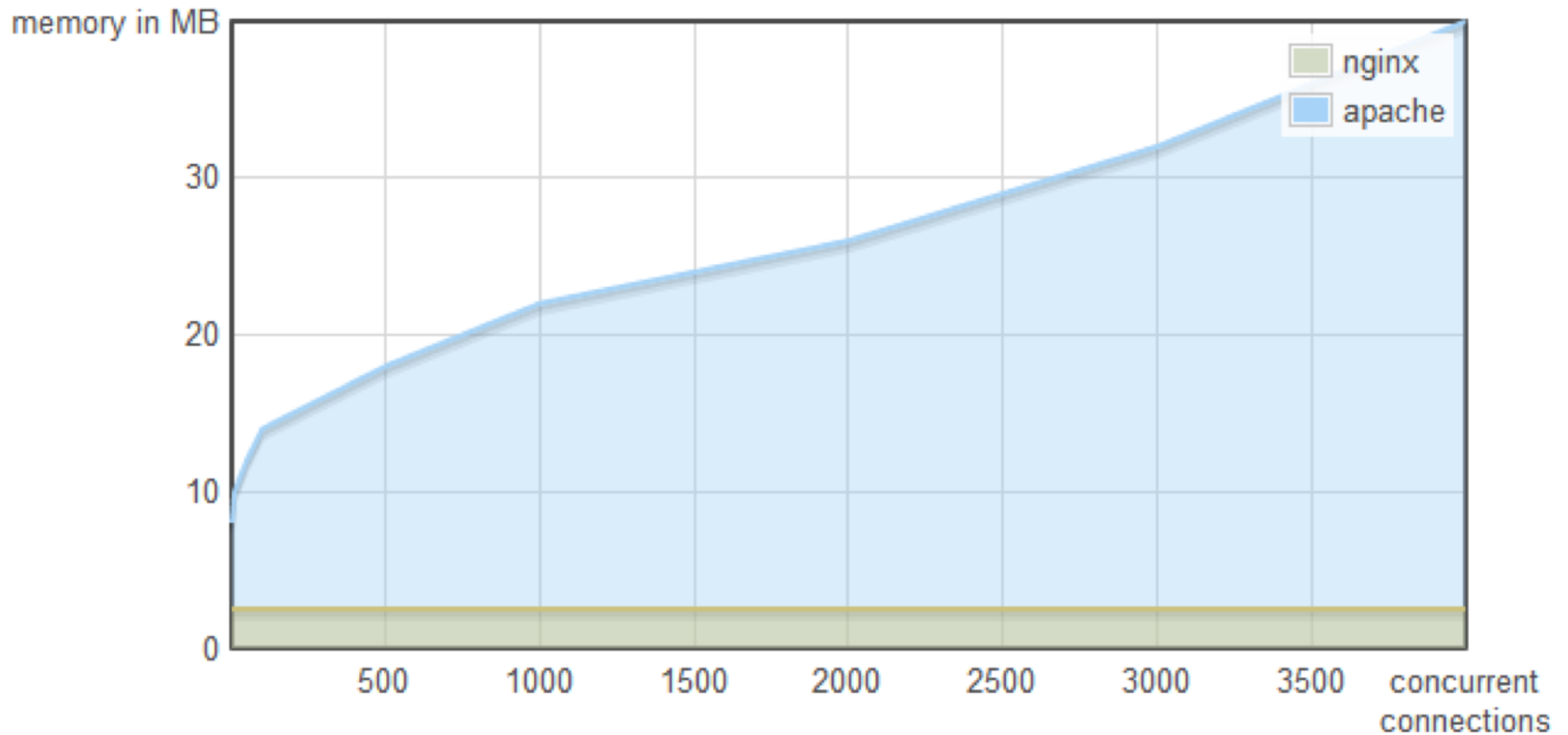


- 2002: Igor Sysov working for rambler.ru
- 2004: First OSS release
- 2011: Company founded
- 2016-Present: 500+ customers and 80+ employees

High Concurrency



Low Memory



NGINX Request Processing

1. Translate Request
2. Evaluate Configuration
3. Select Virtual Server
4. Process Request
5. Serve Response
6. Log Connection

NGINX Architecture

Master Process: Evaluates Config

Child Processes: Adjust module behavior

Shared Memory: Counters, rate limits, etc.

Cache Manager and Loader: Cache settings

Worker Processes: Handle requests and responses

Basic NGINX Commands

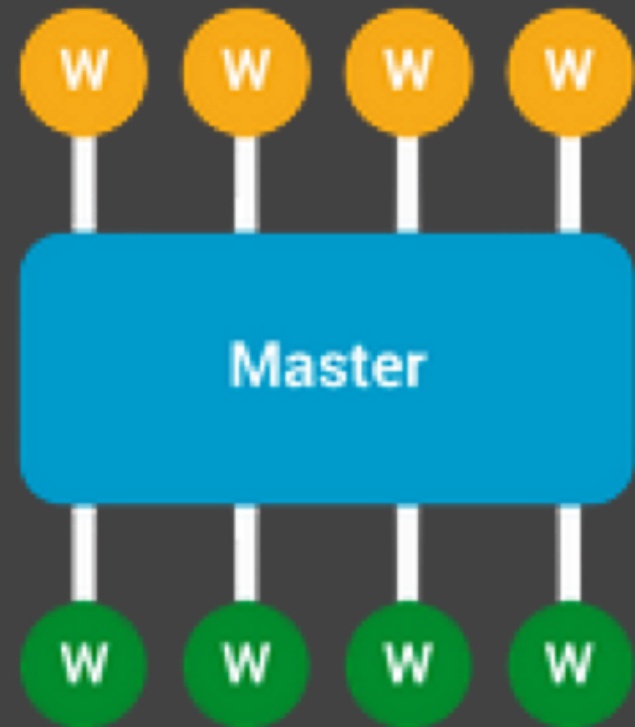
Below commands can reload, gracefully stop, terminate, or check running config

```
$ nginx -s reload  
$ nginx -s quit  
$ nginx -s stop  
$ nginx -t  
$ nginx -T
```


Reloading the Configuration

1. SIGHUP signal
2. Master Process evaluates new config
3. Checks for errors
4. Forks new workers while old workers shut down

Documentation: [Controlling the Configuration](#)



SERVING STATIC CONTENT

Module Objectives

This module enables you to:

- Understand the Configuration File
- Configure a Basic Setup
- Explore server selection methods

NGINX Processes

To check running processes, run the following command:

```
$ ps aux | grep nginx
```

Configuration File

Global Configuration Path

```
$ cat /etc/nginx/nginx.conf
```

Additional Configuration(s) Path

```
$ cat /etc/nginx/conf.d/*.conf
```

Documentation: [nginx.conf Example](#)

Include Directive

The following line in `nginx.conf` allows NGINX to search for additional configurations

```
include /etc/nginx/conf.d/*.conf;
```

Configuration File Structure

- Directives
- Blocks
- Contexts

Documentation: [How .conf Files Work](#)

Directives

Configuration statement that controls **NGINX Modules**

```
listen 80;  
root /usr/share/nginx/html;  
index index.html index.htm index.php;
```


Blocks

Contains mixture of directives and data—begins and ends with curly brackets.

```
server {  
    listen 80;  
    root /usr/share/nginx/html;  
    index index.html index.htm index.php;  
}
```

Contexts

Nested Blocks implying a hierarchy. Colloquially, 'Block' and 'Context' are interchangeable.

```
http {  
    include      /etc/nginx/mime.types;  
    default_type application/octet-stream;  
    gzip on;  
    log_format  
  
    server {  
        listen 80;  
        root /usr/share/nginx/html;  
        index index.html index.htm index.php;  
  
        location {  
            proxy_pass http://backend;  
        }  
    }  
}
```

Documentation: [Further Explanation](#)

Serving Content

Requirements:

- **HTTP Block**—Handles high level processing (logging, compression, caching etc.)
- **Server Block**—Virtual Server (Host) that handles the request
- **Location Block**—Further processing based on the URI of the request

Server Block Example

```
server {  
    listen 80;  
    server_name www.example.com;  
    root /etc/student1/public_html;  
}
```

Location Block Example

```
location /application1 {  
}
```

Two most common types of locations:

- **Prefix**
- **Regex**

Prefix Location

Always checked first, and NGINX selects and remembers longest match

```
location /application1 {  
  
}  
  
location /application1/images {  
    alias /media/data;  
}
```

Second location is selected and matched if given request:

<http://somedomain.com/application1/images/?img2>

Location Modifiers

- Exact String Match =
- Case Insensitive Regex ~*
- Case Sensitive Regex ~
- Stop Request Processing ^~
- Named Location Routing @

Regex Location

Matched sequentially and only after prefix locations.

```
location /application1 {  
  
}  
  
location ~* ^\.(gif|jpg|jpeg|png)$ {  
    alias /media/data;  
}
```


Location Order

Configuration Example

```
server {  
    listen 80 default_server;  
    root /usr/share/nginx/html;  
  
    location = / {  
    }  
  
    location ~* ^\.(png|jpg)$ {  
    }  
  
    location ^~ /app1 {  
    }  
}
```

Selection Order:

- `location = / {`
- `location ^~ /app1 {`
- `location ~* ^\.(png|jpg)$ {`

Defining Server Names

```
server {  
    server_name mycompany.com *.mycompany.com;  
}
```

```
server {  
    server_name mycompany.net *.mycompany.net;  
}
```

Default Server

```
server {  
    listen 80 default_server;  
    server_name example.net www.example.net;  
}
```

IP vs. Server_Name

```
server {  
    listen 192.168.1.1:80;  
    server_name example.org www.example.org;  
}
```

```
server {  
    listen 192.168.1.1:80;  
    server_name example.net www.example.net;  
}
```

```
server {  
    listen 192.168.1.2:80;  
    server_name example.com www.example.com;  
}
```

Bad Requests

Use empty string for server_name directive to prevent bad requests

```
server {  
    listen 80;  
    server_name "";  
    return 444;  
}
```

Lab 1: Serve Pages and Images

1. Open `/etc/nginx/conf.d` and backup `default.conf`.
2. Create a new configuration called `server1.conf`.
3. Create one `server` context with a `listen` and `root` directive pointing to `"/home/student1/public_html."`
4. Add two `location` prefixes for:
 - `/application1`
 - `/application2`
5. Close and save the configuration. Then reload NGINX.
6. Open your browser and test your server URL. Then test the `"/application1"` and `"/application2"` URI. What do you observe?

PROXYING CONNECTIONS

Module Objectives

This module enables you to:

- Configure Proxy Server
- Understand how Proxy Buffering works
- Demonstrate use of `proxy_pass` directive

Reverse Proxy Servers

Receives requests, passes them to backend servers

NGINX supports proxy for HTTP, HTTPS, TCP, UDP, and other protocols.

```
server {  
    listen 80;  
    server_name mydomain.com;  
  
    location / {  
        proxy_pass http://backend;  
    }  
}
```

`proxy_pass` Directive

Sets the address, and protocol, of the proxied server(s)

Syntax:

`proxy_pass url`

Example:

```
location / {  
    proxy_pass http://backend:8080/application  
}
```

Proxy With/Without a Path

WITH A PATH

```
location /application1 {  
    proxy_pass http://localhost:8080/otherapp;  
}
```

WITHOUT A PATH

```
location /application2 {  
    proxy_pass http://localhost:8080  
}
```

Proxy Scenario

```
server {  
    listen 80;  
    root /home/student1/public_html;  
  
    location / {  
        proxy_pass http://localhost:8080;  
    }  
  
    location /application1 {  
        proxy_pass http://localhost:8080/sampleApp;  
    }  
  
    location /images {  
        root /data;  
    }  
}  
  
server }
```

Lab 2: Setup a Reverse Proxy

1. Create a new configuration file called `server2.conf` inside `/etc/nginx/conf.d`
2. Define a server context that listens on port `8080` and has a `root` directive pointing to `/data/server2`
3. Open `server1.conf` from the previous exercise and modify the `/application1` context by adding a `proxy_pass` with the URI:
`http://localhost:8080/sampleApp/`
4. Save and Reload NGINX
5. Open your browser and test your server url/application1. What do you see?

Proxy Buffers

- `proxy_buffering on / off` sets proxy buffering
- `proxy_buffers number size` sets amount and size used for reading entire response for one connection
- `proxy_buffer_size` sets size for reading first part of a response (usually response headers) received from proxied server

proxy_busy_buffers

This directive sets max size of “client-ready” buffers.

The “client-ready” buffers are then placed in a queue.

Proxy Busy Buffers Example

```
server {  
    proxy_buffering on;  
    proxy_buffer_size 1k;  
    proxy_buffers 24 4k;  
    proxy_busy_buffers_size 8k;  
    proxy_max_temp_file_size 2048m;  
    proxy_temp_file_write_size 32k;  
  
    location / {  
        proxy_pass http://example.com;  
    }  
}
```


LOGGING

Module Objectives

This module enables you to:

- Setup Logging to audit connections to NGINX
- Demonstrate use cases for log levels
- Differentiate between access and error log

error_log Directive

- Configures the logging settings for error messages
- Syntax: **error_log** *file log_level*
- "Log Level" specifies the detail of the output

Log Levels

debug

Detailed Trace

info

General Info

notice

Something Normal

warn

Something Strange

error

Unsuccessful

crit

Important Issue(s)

alert

Fix Now!

emerg

Unusable

access_log Directive

- Records all attempts to access the server
- Default log type is *combined*

#Example

```
access_log /var/log/nginx/server3.access.log combined;
```

Logging Best Practices

Keep a separate log files for each server

```
server {  
    server_name server1.com;  
    root /data/server1.com;  
    error_log logs/server1.error.log    info;  
}  
  
server {  
    server_name server2.com;  
    root /data/server2.com;  
    error_log logs/server2.error.log    info;  
}
```

Rotating Logs

Run this shell script in a *cron* job

```
#Get Yesterday's date as YYYY-MM-DD
YESTERDAY=$(date -d 'yesterday' '+%Y-%m-%d')
PID_FILE=/run/nginx.pid
LOG_FILE=/var/log/error.log
OLD_FILE=/var/log/error-$YESTERDAY.log

#Rotate yesterday's log.
mv $LOG_FILE $OLD_FILE

#Tell nginx to open the log file
kill -USR1 $(cat $PID_FILE)
```

Documentation: [Log Rotation](#)

syslog Protocol

Use this protocol when sending messages to syslog servers such as: *splunk*, or *syslog-ng*

```
error_log syslog:server=192.168.1.1 debug;
```

```
access_log syslog:server=unix:/var/log/nginx.sock,nohostname;
```

```
access_log syslog:server=[2001:db8::1]:12345,facility=local7,tag=nginx,se
```

Documentation: [Syslog](#)

Lab 3: Setup Logging

1. Open `server1.conf` at `/etc/nginx/conf.d` and add an `error_log` and `access_log` directive
2. Set the `error_log` to `info`, and the `access_log` to `combined`
3. Repeat the same steps for `server2.conf`
4. Save and Reload NGINX
5. Open your browser and test your `http://server:8080/`
6. Back in your terminal, run a `tail -f` command. What do you see?

SECURITY

Module Objectives

This module enables you to:

- Configure a HTTPS server
- Understand how NGINX handles SSL
- Set limit rates

http_ssl_module

- Provides directives for configuring and managing HTTPS servers
- Requires `openssl` Library

ssl vs. proxy_ssl



SSL Termination

SSL terminates at NGINX level, NGINX handles handshake overhead to take load off the backends

```
http {  
    ssl_session_cache    shared:SSL:10m;  
    ssl_session_timeout  10m;  
  
    server {  
        listen            443 ssl;  
        server_name       www.example.com;  
        keepalive_timeout 70;  
  
        ssl_certificate    www.example.com.crt;  
        ssl_certificate_key www.example.com.key;  
        ssl_protocols     TLSv1 TLSv1.1 TLSv1.2;  
        ssl_ciphers       HIGH:!aNULL:!MD5;  
        ...  
    }  
}
```

Documentation: [SSL Termination](#)

Configuring an HTTPS Server

- Define `ssl` on listen port
- Generate trusted certificate and private key
- Configure protocols and ciphers

Certs and Keys

`ssl_certificate` = the public certificate

`ssl_certificate_key` = the private key

```
server {  
    listen 443 ssl;  
    root /data;  
  
    error_log /var/log/nginx/secure_server.error.log  
  
    ssl_certificate /etc/nginx/nginx.crt;  
    ssl_certificate_key /etc/nginx/nginx.key;  
}
```


Protocols and Ciphers

ssl_protocols

```
ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
```

ssl_ciphers

```
ssl_ciphers ECDH+AESGCM:ECDH+AES256:ECDH+AES128:DH+3DES:!ADH:!AECDH:!MD5
```

Documentation: [SSL Ciphers](#)

Perferred Ciphers

Configure NGINX to tell client there is a preferred order of available cipher suites

```
ssl_prefer_server_ciphers on;
```

Dual Stack "RSA" and "ECC"

```
server {  
    listen 443 ssl;  
    server_name example.com;  
  
    #RSA cert  
    ssl_certificate example.com.rsa.crt;  
    ssl_certificate_key example.com.rsa.key;  
  
    #ECC cert  
    ssl_certificate example.com.ecdsa.crt;  
    ssl_certificate_key example.com.ecdsa.key;  
}
```

Popular Use Cases

Combine HTTP & HTTPS

```
server {  
    listen 443;  
    listen 80;  
    server_name example.com;  
    root /home/student1/public_html;  
}
```

Popular Use Cases

Force incoming traffic to HTTPS

```
#server 1
server {
    listen 80;
    return 301 https://$host$request_uri;
}

#server 2
server {
    listen 443 ssl;
    ssl_certificate /etc/nginx/ssl/server.crt;
    ssl_certificate_key /etc/nginx/ssl/server.key;
}
```

Lab 4: Configure HTTPS

1. Use openssl to generate a cert and key in `/etc/nginx:`

```
sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout nginx.key
```

2. In `/etc/nginx/conf.d` open `example_ssl.conf`
3. Uncomment the example HTTPS server block
4. Replace your cert and key with the one you generated
5. Save and reload NGINX
6. Test `http://server/` index page using https
7. Click-through the warning message, what do you see?

Restricting Access

- `allow` specifies access to a server or location block
- `deny` directive prevents access

Basic HTTP Authentication

Requires users to specify a login and password via the browser. Syntax is:

```
auth_basic realm | off
```


Basic Auth Example

```
server {  
    listen 8080;  
    root /home/johnny/public_html;  
  
    auth_basic "restricted server";  
    auth_basic_user_file /etc/nginx/htpasswd;  
  
    location /public_area {  
        auth_basic off;  
    }  
}
```

Password File

- `crypt()`
- `openssl passwd`

```
name1:password1  
name2:password2  
#comments
```

auth_request Module

```
server {  
    listen 443 ssl;  
    root /usr/share/nginx/html;  
    ...  
    location /private {  
        auth_request /auth;  
        auth_request_set $user $upstream_http_x_user;  
        proxy_set_header x-user $user;  
        proxy_pass http://backend_server;  
    }  
}
```

Lab 5: Setup Basic Authentication

1. Open the password file called “htpasswd” inside
`/etc/nginx`
2. Delete the password for the admin entry
3. Close and save the file. Back in the terminal, run
`openssl passwd`
4. Specify any user and password. Copy the encrypted output to your clipboard and paste into htpasswd
5. Open server1.conf and enable authentication on the server level
6. Save and reload NGINX. Open a browser and hit `http://server`
7. Login with your username and password you specified

Setup Limit Rates

Limitations based on:

- Number of Connections
- Number of Requests
- Download Speeds

Limiting Connections

- Define a Shared Memory Zone
- `limit_conn_zone` defines: zone, size, name, and key
- Syntax is:

```
limit_conn_zone $variable zone=name:size;
```

limit_conn

Specify shared memory zone in desired context

Syntax:

```
limit_conn zone number_of_connections;
```

```
http {  
    limit_conn_zone $binary_remote_addr zone=ip:10m;  
  
    server {  
        ...  
        location /downloads {  
            limit_conn ip 5;  
        }  
    }  
}
```

Limiting Request Rate

Set request rate, name, key, and duration in Shared Memory Zone

Example:

```
limit_req_zone $server_name zone=one:10m rate=1r/s
```


limit_req

Syntax:

```
limit_req zone='name' [burst=n.o | nodelay];
```

```
http {  
    limit_req_zone $server_name zone=ten:10m rate=10r/s  
    ...  
    server {  
        ...  
        location /data {  
            limit_req zone=ten burst=15;  
        }  
    }  
}
```

Limiting Download Rate

Syntax:

limit_rate speed

```
location /media {  
    limit_conn ipzone 5;  
    limit_rate 50k;  
}
```

limit_rate_after

Throttle download speeds after client builds a buffer

```
location /videos {  
    limit_rate_after 500k;  
    limit_rate 50k;  
}
```

VARIABLES

Module Objectives

This module enables you to:

- Gain knowledge of NGINX's predefined variables
- Understand Variable Scope with regards to NGINX
- Define your own custom variables

Core Module Variables

Variable	Value
<code>\$host</code>	Host name from request line
<code>\$request_uri</code>	The full URI, including arguments
<code>\$uri</code>	Normalized URI (no arguments)
<code>\$scheme</code>	Request scheme (HTTP or HTTPS)
<code>\$request_method</code>	GET, POST, PUT etc.
<code>\$request_filename</code>	Absolute file path for current request
<code>\$remote_addr</code>	IP address of client

Variable Example

Given URL: `http://localhost:8080/test?arg=1`

- `$host` = `localhost`
- `$request_uri` = `/test?arg=1`
- `$uri` = `/test`
- `$scheme` = `http://`
- `$args` = `?arg=1`

Usage Example:

```
server {  
    listen 80;  
    return 301 https://$host$request_uri;  
}
```

set Directive

Syntax:

`set $variable_name value;`

```
set $foo hello;  
set $bar "hello world";  
set $combo $foo
```


Declaration and Scope

```
server {  
    listen 8080;  
  
    location /test1 {  
        return 200 = "foo $foo \name example = $example \n";  
    }  
  
    location /test2 {  
        set $foo hello;  
        return 200 = "foo = $foo \n";  
    }  
}  
  
server {  
    listen 8081;  
  
    set $example 42;
```

map Directive

Creates a mapping relationship between two variables

Syntax:

```
map $var1 $var2 {value value}
```

```
map $uri      $path {  
    /test1    /path1  
    /test2    /path2  
    /test3    /path3  
}  
  
server {  
    listen 80;  
  
    location /test1 {  
        return 200 "$path \n";  
    }  
}
```

Default Value

```
map $arg $value {  
  default 1;  
  test1 2;  
}
```

Regex in Maps

```
map $uri $path {  
    ~*/test/.*\.php$ /path4;  
    /example /examplePath;  
}
```

Lab 6: Map Example

1. Create a map for a variable called `$is_redirect` that depends on the predefined `$uri` variable.
2. Set variable relationships as follows:
 - `default 0;`
 - `test1 1;`
 - `test2 2;`
 - `test3 3;`
3. Save the file and reload Nginx. Run a `curl` command against `http://server/test1`. What do you observe? Try the other URIs

ROUTING CONNECTIONS

Module Objectives

This module enables you to:

- Use specific directives in NGINX to reroute traffic
- Define URL Rewrites
- Understand Rewrite Request processing

alias Directive

Allows you to specify a replacement path

Syntax:

alias *path*;

```
server {  
    root /home/public_html;  
  
    location /test {  
        alias /data/app1;  
    }  
}
```


Lab 7: Alias Replacement

1. Change directory to `/etc/nginx/conf.d` and open `server1.conf`.
2. In `/application2` prefix, specify a replacement path of `/data/test` using the `alias` directive.
3. Save the file and reload Nginx. Test against `http://server/application2/logo.png` What do you observe? Try the other URIs

alias with Regex

```
location ~ ^/pictures/(.+\.(:gif|jpe?g|png))$ {  
    alias /data/images/$1;  
}
```

Lab 7.5: Regex Alias

1. Open `server1.conf` and edit the `/images` prefix to be a case insensitive regex, with a URI match of `/pictures.`
2. In the regular expression, ensure that the capture group includes 1 or more occurrences of any character followed by a "." and ending in either `gif`, `jpeg`, `jpg`, or, `png`
3. Inside the location block, define the replacement path as: `/data/images/$1`
4. Save the file and reload Nginx. Test against `http://server/pictures/logo.png` What do you observe?

return Directive

Return a HTTP response code and URL to the client

Syntax:

return code url; Or *return url;*

```
server {
    listen 8080;
    root /home/public_html;

    location /test {
        return http://localhost:8081$uri;
    }

server {
    listen 8081;
    root /data/app1;
    autoindex on;
}
}
```

rewrite Directive

Regex pattern matched against URI, replacement string rewrites the URI.

Syntax:

rewrite regex replacement [flag]

```
server {  
    listen 8080;  
    root /home/public_html;  
    rewrite ^/shop/products/(\d+) /myshop/products/product$1.html;  
}
```

Lab 8.1: Setting Up Rewrite Data

1. Inside your `public.html` folder, ensure that there are the following directories and files:
`/shop/products/product1.html product2.html product3.`
2. Inside `product1.html` edit the paragraph tag by removing `8` and replacing `<server>` with your ec2 url

```
<p>  
  
</p>
```

Lab 8.2: Rewrite URLs

1. Open `server2.conf` in `/etc/nginx/conf.d`
2. Inside the `server` context, define a `rewrite` regex:

```
^/shop/greatproducts/(\d+)$
```

and a replacement string:

```
/shop/products/product$1.html
```

3. Save your file and reload nginx
4. Open your browser and test against:
 - `<server>/shop/greatproducts/2`
 - `<server>/shop/greatproducts/3`
 - `<server>/shop/greatproducts/1`

Lab 8.3: Rewrite URLs (Continued)

1. Now try `<server>/shop/greatproducts/1`
2. What do you notice about the image?
3. Add another `rewrite` at the `server` level with the following regex:

```
^/media/pics/(.+\. (gif|jpe?g|png))$
```

and replacement string:

```
/images/$1
```

4. Save and reload NGINX
5. Re-test `/shop/greatproducts/1`

Rewrite Process Cycle

Highlights:

- Executed sequentially
- Executed upon server selection
- All rewrites in higher context are executed first
- Flags will stop further processing

rewrite flags

Prevents further rewrite processing

- last
- break
- redirect
- permanent

Lab 9.1: Rewrite Flags

1. Open `server1.conf` and define a new location block with the prefix `/shop`
2. Cut and paste the `rewrite` regarding `/greatproducts`, and paste it into the new `location`
3. Add a new `rewrite` with regex:

```
^/shop/./+/(\\d+)$
```

and replacement string:

```
/shop/services/service$1.html
```

4. Add a `return` directive with code `403`
5. Save and reload NGINX. Re-test
`/shop/greatproducts/1`

Lab 9.2: Rewrite Flags (Continued)

1. Re-open `server1.conf` and place `break` flags after every `rewrite` in the `location` context
2. Save and reload NGINX
3. Re-test `/shop/greatproducts/1`

END OF DAY ONE:

THANK YOU!

Agenda: Day Two



- Load Balancing
- Live Activity Monitoring
- Caching
- Compression
- Dynamic Configuration
- Installation

LOAD BALANCING

Module Objectives

This module enables you to:

- Setup basic load balancer
- Identify load balancing methods
- Enable session persistence

ngx_http_upstream_module

Key Take-Aways:

- Defines a group of servers
- Leverages `proxy_pass` directive
- Server definition can be:
 - Unix socket
 - DNS
 - IP:Port

Load Balancing

`proxy_pass` forwards request to `upstream` link

```
upstream myServers {  
    server training.example.com;  
    server training.example1.com:8080;  
    server 192.168.245.27;  
}
```

Specifying Server Priorities

`weight` indicates `server` priority

`max_fails` indicates server-level failures

`fail_timeout` indicates timeout and duration of downtime

```
upstream myServers {  
    server backend.server1 weight=5 max_fails=10 fail_timeout=90s;  
    server backend.server2 weight=3 max_fails=4 fail_timeout=60s;  
    server backend.server3 weight=4 max_fails=2 fail_timeout=30s;  
}
```

Lab 10.1: Setup "Backends"

1. In `/etc/conf.d/` backup `server2.conf` to `server2.conf.bak`
2. In the `conf.d` directory, create a new `.conf` called `backends.conf` and define a `server` context with the `root` directive pointing to `/data/backend1`. The server should listen on `8081`
3. Repeat steps 1 and 2 for the remaining servers. Servers should listen on `8082` and `8083` respectively, and the `root` directories are `/data/backend2` and `/data/backend3`

Lab 10.2: Configure Load Balancing

1. Create a file called `myServers.conf` and define an `upstream` with three backend servers using `127.0.0.1:<port>`
2. Create a `server` context that listens on `8080`
3. Set the `root` directive to your `public_html` directory.
4. Define an `error_log` with a level of `info` and an `access_log` with a level of `combined`
5. Create a `location` context that matches all requests
6. Add a `proxy_pass` to forward all request to the `upstream`

ngx_stream_core_module

Key Take-Aways:

- Used for TCP/UDP Load Balancing
- Also leverages `proxy_pass` directive
- Similar syntax with `ngx_http_upstream` module
- Version compatibility:
 - TCP: r5 or greater
 - UDP: r9 or greater
- Exists in `main context`

stream Use Cases

- **TCP:** mSQL, LDAP, RTMP
- **UDP:** DNS, Syslog, RADIUS

mySQL Example

Load Balancings across three SQL servers

```
stream {  
    server {  
        listen 3306;  
        proxy_pass db;  
    }  
  
    upstream db {  
        server db1:3306;  
        server db2:3306;  
        server db3:3306;  
    }  
}
```


DNS Example

Load balances UDP traffic across two DNS servers

```
stream {  
    upstream dns_upstreams {  
        server 192.168.136.130:53;  
        server 192.168.136.131:53;  
    }  
  
    server {  
        listen 53 udp;  
        proxy_pass dns_upstreams;  
        proxy_timeout 1s;  
        proxy_responses 1;  
        error_log logs/dns.log;  
    }  
}
```

Load Balancing Methods

- `least_conn`
- `least_time`
- `hash`
- `ip_hash`

least_conn Directive

```
upstream backendServers {  
    least_conn;  
  
    server backend1.com;  
    server backend2.com;  
    server backend3.com;  
}
```

least_time Directive

```
upstream myServers {  
    least_time header;  
  
    server backend1.com;  
    server backend2.com;  
    server backend3.com;  
}
```

hash Directive

```
upstream myServers {  
    hash $request_uri;  
  
    server backend1.com;  
    server backend2.com;  
    server backend3.com;  
}
```

ip_hash Directive

Uses first three octets for IPv4, or entire IPv6 address

```
upstream myapp1 {  
    ip_hash;  
    server srv1.example.com;  
    server srv2.example.com;  
    server srv3.example.com;  
}
```

fail_timeout Parameter

```
upstream myServers{  
    server localhost:8080 max_fails=3 fail_timeout=30s;  
    ...  
}
```

max_conns Parameter

```
upstream backend {  
    server backend1.example.com max_conns=3;  
    server backend2.example.com;  
}
```


queue Directive

```
upstream backend {  
    server backend1.example.com max_conns=3;  
    server backend2.example.com;  
        queue 100 timeout=70;  
}
```

Session Affinity

For applications that require state data on backend servers

NGINX supports the following methods:

- `sticky cookie`
- `sticky learn`
- `sticky route`

sticky cookie

Syntax:

`sticky cookie name`

```
upstream backendServers {  
    server my.example1.com;  
    server my.example2.com;  
  
    sticky cookie my_srv expires=1h domain=example.com path=/cart;  
}
```

sticky learn

Syntax:

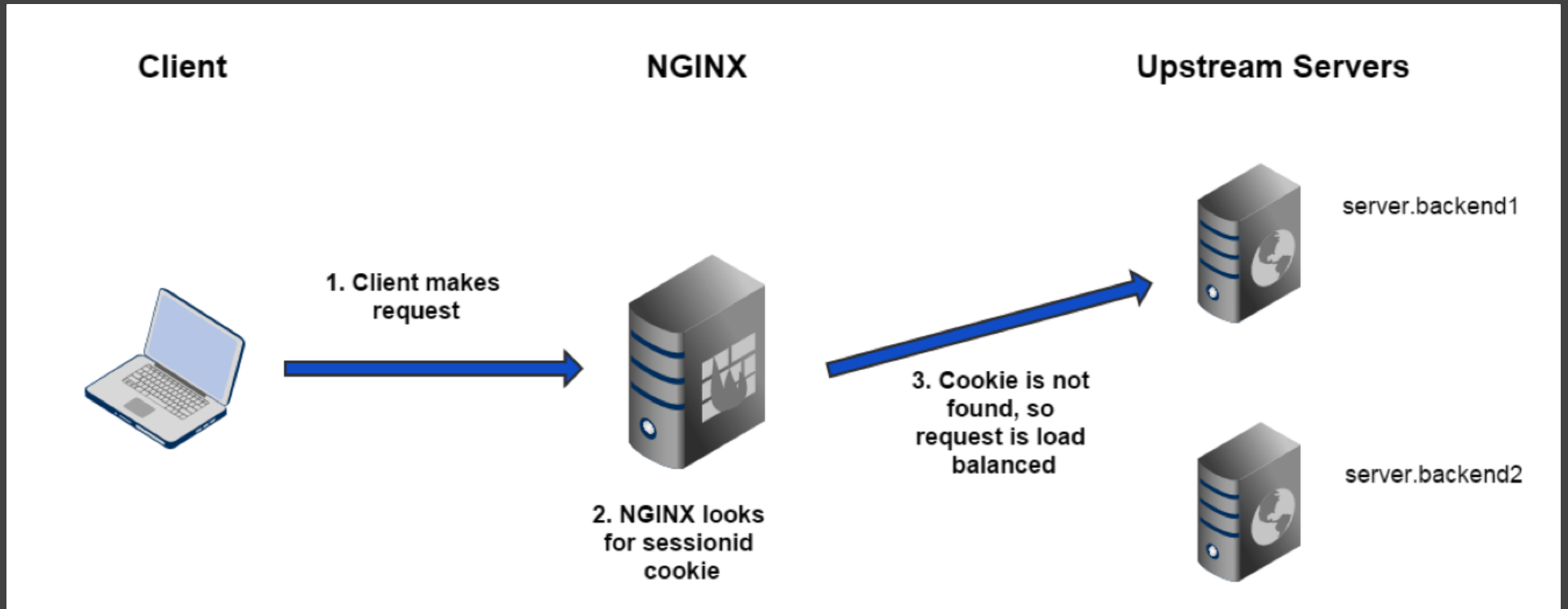
`sticky cookie name`

```
upstream backendServers {
    server my.example1.com:8081;
    server my.example2.com:8082;

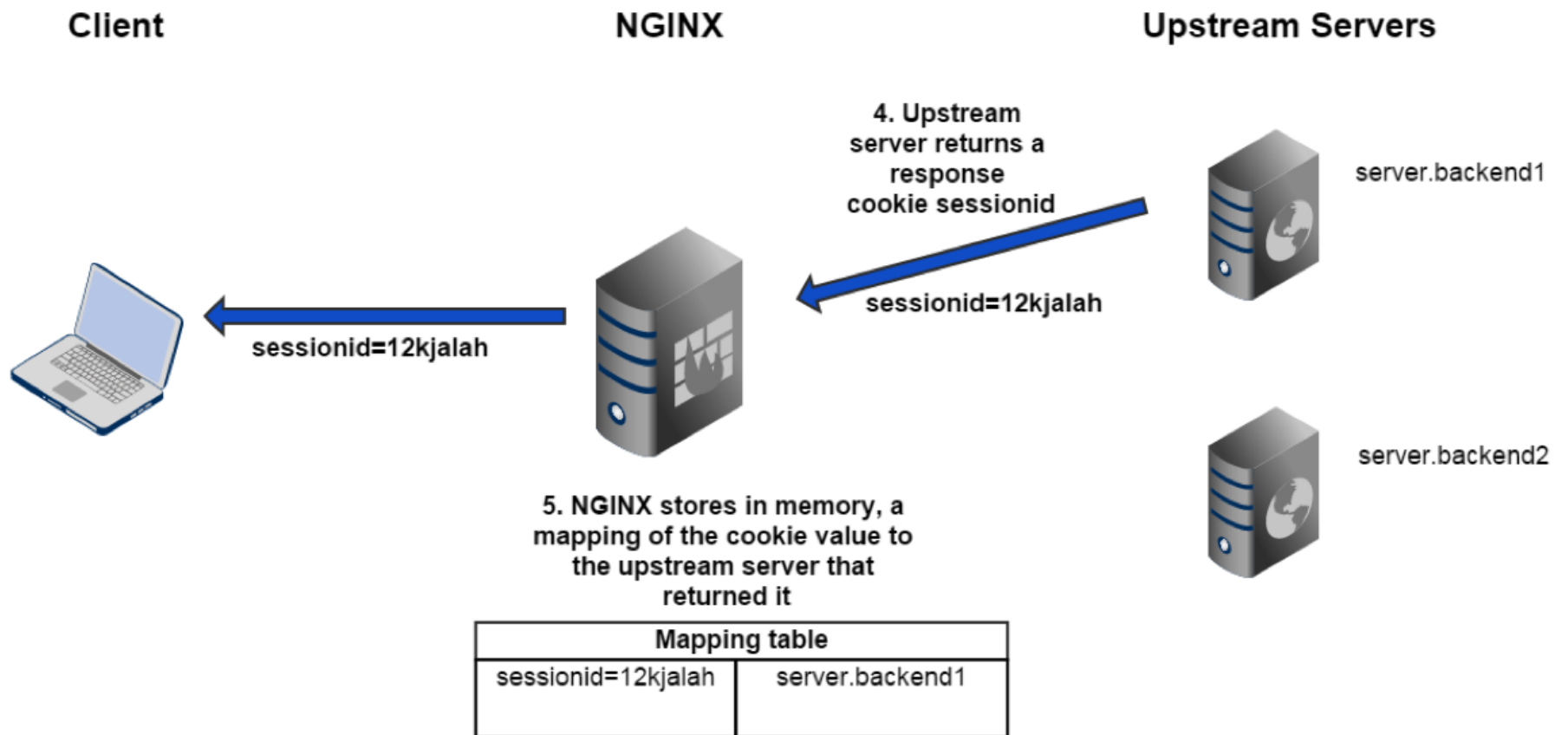
    sticky learn create=$upstream_cookie_sessionid lookup=$cookie_sessionid
}

server {
    location / {
        proxy_pass http://backendServers;
    }
}
```

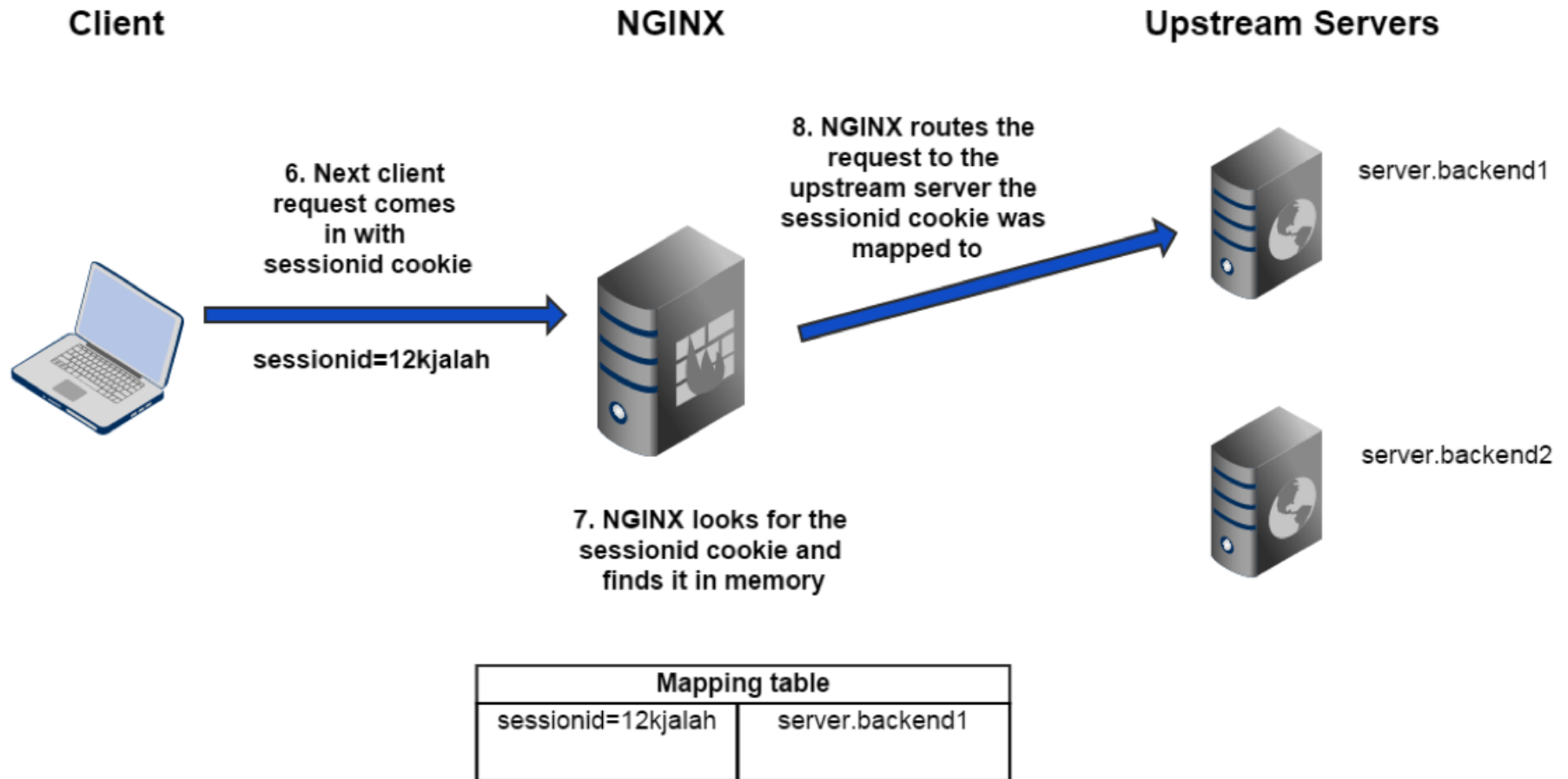
sticky learn Part 1



sticky learn Part 2



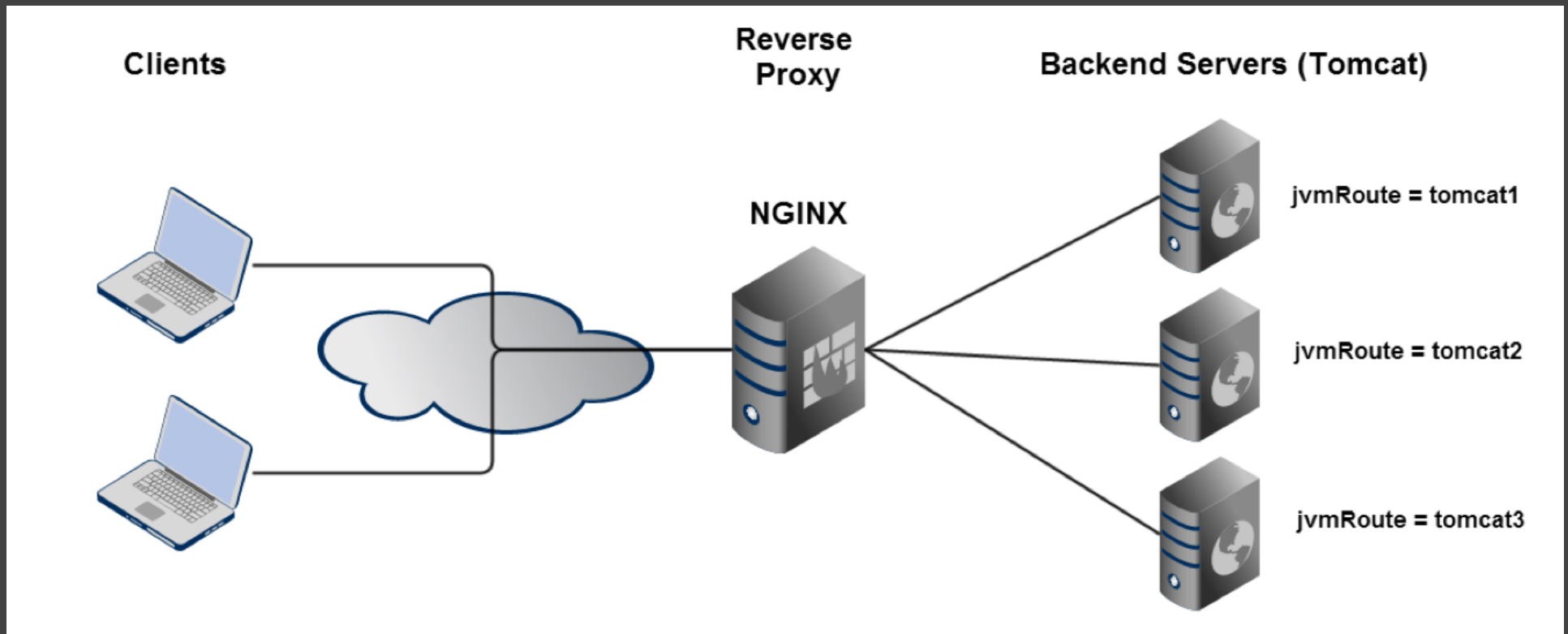
sticky learn Part 3



sticky route

```
upstream backend {  
    zone backend 64k;  
    server backend1.com route=tomcat1;  
    server backend2.com route=tomcat2;  
    server backend3.com route=tomcat3;  
  
    sticky route $route_cookie $route_uri;  
}
```


Tomcat Example



Routing Variables

```
map $cookie_JSESSIONID $route_cookie {  
    ~.+\\. (?P<route>\\w+)$ $route;  
}  
  
map $request_uri $route_uri {  
    ~JSESSIONID=\\. (?P<route>\\w+)$ $route;  
}
```

Lab 11.1: Tomcat Route

1. Open `myServers.conf` in `/etc/nginx/conf.d`
2. In the `http` context, create a `log_format` called `sticky`
3. Set the new log to:

```
"$request\tUpstream: $upstream_addr \t URL token: $arg_serverid\t Cookie:
```

4. Add the `sticky route` directive in your upstream group with variables: `$arg_serverid $cookie_serverid;`
5. Add the `route` parameter to each upstream like so:

```
upstream myServers {  
    zone backend 64k;  
    server <backend_url>:8081 route=backend1;  
    server <backend_url>:8082 route=backend2;  
    server <backend_url>:8083 route=backend3;  
}
```

ask your instructor about the backend urls!!!

Lab 11.2: Tomcat Route Test

1. In your shell, make the following `curl` requests:

```
curl http://<localhost>:8080
```

2. In a separate shell, run a `tail -f` command on your `upstream_access.log`
3. Do you notice the IP address changing?
4. Open a browser, and step through the app via the following URI:

```
<localhost>/examples/servlets/servlet/SessionExample
```

5. Execute the application, and refresh your browser several times. What can you observe in the log now? Which IP address is the request hitting?

LIVE ACTIVITY MONITORING

Module Objectives

This module enables you to:

- Use the `status` directive to get server metrics
- Configure `health_check` to monitor the availability of your upstream servers

status Directive

```
server {  
    listen 8080;  
  
    location = /status{  
        status;  
    }  
}
```

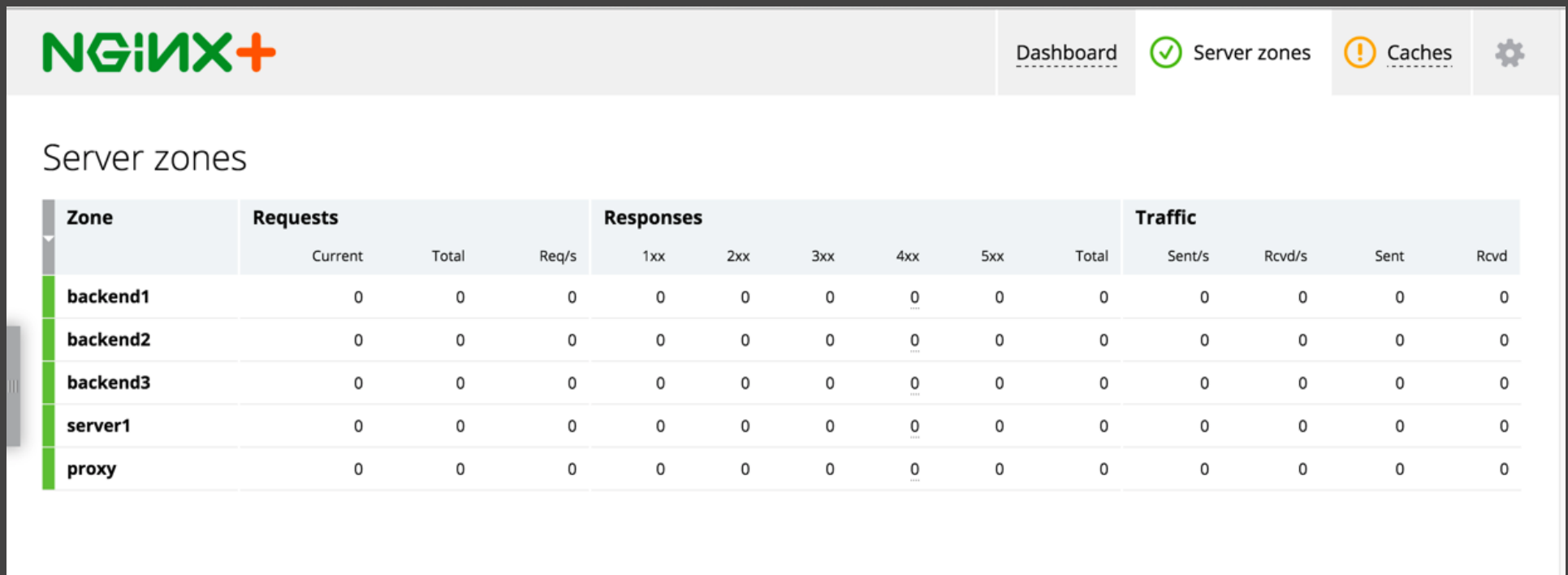
Securing the Status Request

```
location = /status{  
    allow 192.168.0.0/16;  
    deny all; # deny access from everywhere else  
  
    status;  
}
```


Default Status Page

NGINX Plus contains an html page that parses JSON

The page is located at **`/usr/share/nginx/html/status.html`**



The screenshot shows the NGINX Plus status page. At the top is the NGINX+ logo. To the right are navigation links: 'Dashboard' (underlined), 'Server zones' (with a green checkmark icon), 'Caches' (with a yellow warning icon), and a settings gear icon. Below the navigation bar, the title 'Server zones' is displayed. A table follows, showing metrics for five zones: backend1, backend2, backend3, server1, and proxy. The table has four main sections: Zone, Requests, Responses, and Traffic. Each section contains specific metrics as detailed in the table below.

Zone	Requests			Responses						Traffic			
	Current	Total	Req/s	1xx	2xx	3xx	4xx	5xx	Total	Sent/s	Rcvd/s	Sent	Rcvd
backend1	0	0	0	0	0	0	0	0	0	0	0	0	0
backend2	0	0	0	0	0	0	0	0	0	0	0	0	0
backend3	0	0	0	0	0	0	0	0	0	0	0	0	0
server1	0	0	0	0	0	0	0	0	0	0	0	0	0
proxy	0	0	0	0	0	0	0	0	0	0	0	0	0

Server Zones

```
server {  
    listen 8081;  
    root /server/backend1;  
    status_zone apac;  
}
```

```
server {  
    listen 8082;  
    root /server/backend2;  
    status_zone apac;  
}
```

```
server {  
    listen 8083;  
    root /server/backend3;  
    status_zone us;  
}
```

Individual servers zones displayed in status.html:

Server zones

Zone	Requests			Responses			
	Current	Total	Req/s	1xx	2xx	3xx	4xx
apac	0	0	0	0	0	0	0
us	0	0	0	0	0	0	0

zone Directive

Shared memory zone makes upstream dynamically configurable

Allows worker processes to share counter information

```
upstream myServers {  
    zone backend 64k;  
    server backend1;  
    server backend2;  
    server backend3;  
}
```

Lab 12.1: Define a Status Page

1. Open `myServers.conf` located in `/etc/nginx/conf.d/`
2. In a new `server` context that listens on `9090` with a `root` location `/usr/share/nginx/html;`
3. Add a `location` prefix with an exact match `= /status`, and the `status` directive in this location block
4. Save and reload NGINX
5. Open a browser and access the following URI:

```
http://<server>:9090/status.html
```

6. Notice anything strange in the GUI?

Lab 12.2: Define Server Zones

1. In `backends.conf`, specify a server zone for each server by using the `status_zone` directive
2. Open `server1.conf` and define a `status_zone`
3. Open `server2.conf` and define a `status_zone`
4. Open `myServers.conf` and specify a shared memory zone in your `upstream` block with a size of `64k`
5. Save and reload NGINX
6. Refresh your browser listening on

```
http://<server>:9090/status.html
```

Server health_check

A request sent to upstream to check status based on conditions

```
upstream myUpstreams {
    zone backend 64k;

    server localhost:8081 weight=1;
    server localhost:8082 weight=4;
    server localhost:8083 weight=7;
}

server {
    listen 8080;
    error_log /var/log/nginx/upstream_error.log info;

    location / {
        proxy_pass http://myUpstreams;
        health_check;
    }
}
```

health_check Parameters

- interval
- fails
- passes
- uri
- match

match Block

Block directive that defines conditions for:

`health_check`

Conditions can be based on:

- Response Codes
- Header Values
- Text body of documents

match Directives

- status
 - status 200
 - status ! 403
 - status 200-399
- header
 - header Content-Type = text/html
 - header Cache-Control
- body
 - body ~ "Hellow World"
 - body !~ "hello world"

match Example

```
server {  
    location / {  
        proxy_pass http://myServers;  
        health_check match=conditions fails=2;  
    }  
}  
  
match conditions {  
    status 200;  
    header Content-Type = text/html;  
    body !~ "maintenance";  
}
```

Lab 13.1: Server Maintenance

1. Open the `myServers.conf`, file
2. Define a `match` block called `health_conditions` with the following directives:

```
match health_conditions {  
    status 200-399;  
    header Content-Type = text/html;  
    body !~ maintenance;  
}
```

Lab 13.2: Server Maintenance Continued

1. In the `location` prefix that matches all requests, add the following `match` parameters:

```
location / {  
    proxy_pass http://myServers;  
    health_check match=health_conditions  
    fails=2  
    uri=/health/test.html;  
}
```

2. Save and reload NGINX
3. Refresh your browser listening on

```
http://<server>:9090/status.html
```

CACHING

Module Objectives

This module enables you to:

- Define a reverse proxy cache for your upstream and other servers
- Purge old or stale content from the cache
- Identify other cache control techniques

Reverse Proxy and Caching

Common use case to have NGINX in front, caching static resources to improve performance

To compose a cache:

- Define a `cache_path`
- Configure the `proxy_pass`
- Reference the `cache_key`

Configuring the `proxy_cache`

- `proxy_cache_key`
- `proxy_cache`

```
location / {  
    proxy_pass http://application.com:8080;  
  
    proxy_cache_key "$scheme$host$request_uri";  
    proxy_cache my-cache;  
    proxy_cache_valid 1m;  
    proxy_cache_valid 404 1m;  
}
```


Validating the Cache

```
location / {  
    proxy_cache_valid any 1m;  
    proxy_cache_valid 404 1m;  
}
```

Passing Headers

```
proxy_set_header    Host      $host;  
proxy_set_header    X-Real-IP  $remote_addr;  
proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
```

add_header Directive

```
server {  
  [...]  
    add_header X-Proxy-Cache $upstream_cache_status;  
  
    location / {  
      proxy_cache myCache;  
      proxy_pass http://localhost:8081;  
    }  
}
```

Lab 14.1: Proxy Cache

1. Open `server1.conf`
2. Define a cache path in the `http` context:

```
proxy_cache_path /data/nginx/cache levels=1:2  
keys_zone=img_cache:20m inactive=5m;
```

3. In the `server` context, set the `proxy_cache_key` to the `$scheme`, `$host`, and `$request_uri`
4. Set the `proxy_set_header` to forward the client IP address

Lab 14.2: Proxy Cache Continued

1. Set the `proxy_cache` in the `application1` prefix, and set the validation of the cache for 10 minutes.
2. Change the `proxy_pass` port from `8080` to `90`
3. Change `server2.conf.bak` back to `server2.conf`
4. Open `server2.conf` and change the `listen` port to `90`
5. Save and reload NGINX
6. Reload NGINX and make a request to

```
http://<server>/application1
```

7. Hit refresh multiple times and then check your `status.html` in a separate tab. Notice the cache icon is now “warm” and the hit ratio is increasing

Lab 15.1: Proxy Upstream Cache

1. Open `Open /etc/nginx/conf.d/myServers.conf`
2. Create a `proxy_cache_path` with a `keys_zone` named `upstreamCache` that lasts for:
 - 10 minutes
 - Has a max size of 60mb
 - Timeouts after 60 minutes

Lab 15.2: Proxy Upstream Cache Continued

1. Set the `proxy_set_header` to forward the client host and IP
2. Use the `add_header` directive with the `X-Proxy-Cache` response header to return the upstream status
3. Add `proxy_cache` to the proxying location
4. Save and reload NGINX
5. Make a request to your upstream and notice the second cache in your status.html
6. Try making a `curl` request? What do you see with the `-I` parameter?

Caching Resources

Directives that control cached responses:

- `proxy_cache_min_uses`
- `proxy_cache_methods`

Caching limit rates:

- `proxy_cache_bypass`
- `proxy_no_cache`

Documentation: [Cache Admin Guide](#)

proxy_cache_min_uses

```
server {  
    proxy_cache myCache;  
    proxy_pass http://localhost:8081;  
    proxy_cache_min_uses 5;  
}
```

cache_methods and no_cache

Syntax:

`proxy_cache_methods <REQUEST METHOD>`

Syntax:

`proxy_no_cache argarg_comment`

```
map $request_uri $no_cache;  
    /default      0;  
    /test         1;  
  
server {  
    proxy_cache_methods GET HEAD POST;  
    proxy_no_cache $no_cache;  
}
```

Cache Manager and Loader

- `loader_threshold`
- `loader_files`
- `loader_sleeps`

```
proxy_cache_path /data/nginx/cache keys_zone=one:10m loader_threshold=300
```

proxy_cache_purge Directive

Allows you to remove full cache entries that match a configured value.

```
server {  
    proxy_cache myCache;  
    proxy_pass http://localhost:8081;  
    proxy_cache_purge $purge_method;  
}
```

Purge Methods

Partial Purge

- use `curl` command to send `PURGE HTTP` request, `map` evaluates request and enables the directive

Full Purge

- turn `purger` parameter on in the `proxy_cache_path`, all wildcard pages will also be purged

HTTP PURGE Example

Request:

```
$curl -X PURGE -D - "http://www.mysite.com"
```

```
# setting the default purge method will only delete matching URLs.
map $request_method $purge_method {
    PURGE 1;
    default 0;
}
server {
    listen 80;
    server_name www.mysite.com
    proxy_cache myCache;
    proxy_pass http://localhost:8081;
    proxy_cache_purge $purge_method;
}
```

purger Example

Request:

```
$curl -X PURGE -D - "http://www.mysite.com/*"
```

```
proxy_cache_path /data/nginx/cache levels=1:2 keys=myCache:10m purger=on
```

```
server {  
    listen 80;  
    server_name www.mysite.com;  
    location / {  
  
        proxy_cache_purge $purge_method;  
    }  
}
```

Lab 16.1: Configure Cache Purge

1. Open `myServers.conf` and use a `map` to create a custom variable called `purge_method` that depends on the predefined `request_method`
2. In the `location` where caching occurs, specify a condition for the cache purge request.
3. Save and reload NGINX
4. Send the purge command using the `curl` command, your machine url, and necessary port (`http://<server>:8081`).
5. A successful purge should return an HTTP 204 code (no content).

COMPRESSION

HTTP **gzip** Module

Key Directives:

- gzip
- gzip_types
- gzip_proxied

gzip Example

```
http {  
    gzip on;  
    gzip_types text/plain text/css;  
    gzip_proxied any;  
}
```

gzip_min_length

Specifies the minimum length of the response to compress

```
gzip_min_length 1000;
```

gzip_proxied

NGINX doesn't compress proxied requests by default,

gzip_proxied instructs NGINX to check header fields

```
server {  
    gzip on;  
    gzip_types text/plain application/xml;  
    gzip_proxied no-cache no-store private expired auth;  
    gzip_min_length 1000;  
}
```

gzip_static

Syntax:

`gzip_static on | off`

```
server {  
  gzip on;  
  gzip_static on;  
  gzip_types text/plain application/xml;  
  gzip_proxied no-cache no-store private expired auth;  
  gzip_min_length 1000;  
}
```

HTTP **gunzip** Module

Decompresses client gzip responses, if gzip method isn't supported

Syntax:

gunzip_buffers *number size*

```
http {  
    gunzip on;  
    gunzip_buffers 32 4k;  
}
```

gzip_vary

Places the

“Vary: Accept-Encoding” response header if both `gzip_static` and `gunzip` are active.

```
server {  
    gzip on;  
    gzip_vary on;  
    gzip_types text/plain application/xml application/json;  
    gzip_proxied no-cache no-store private expired auth;  
    gzip_min_length 1000;  
}
```


DYNAMIC CONFIGURATION

Module Objectives

This module enables you to:

- Leverage NGINX API to dynamically configure server information
- Use the state directive to make changes persistent

Dynamic Configuration

Advantages:

- View, modify, and remove servers at runtime
- No need to reload NGINX to affect changes

Disadvantages

- Runtime config is not saved to conf file
- Changes revert back to conf settings after reload (unless using `state` directive)

Shared Memory Zone

Required to make server dynamically configurable

Distributes traffic more evenly

Necessary for

state changes (covered later)

upstream_conf Directive

```
upstream myServers {  
  
    server localhost:8081;  
    server localhost:8082;  
}  
  
server {  
    listen 8080;  
  
    location / {  
        proxy_pass http://myServers;  
    }  
  
    location /upstream_conf {  
        upstream_conf;  
        allow 127.0.0.1;  
        deny all;  
    }  
}
```

`upstream_conf` Parameters

Key Parameters:

- `add=`
- `remove=`
- `drain=`
- `upstream=name`
- `server=address`
- `id=number`

Sending Requests

View and modify server details

```
#View all primary servers in upstream group myServers  
curl http://<server>:8080/upstream_conf?upstream=myServers
```

```
#View individual server detail in upstream group  
curl http://<server>:8080/upstream_conf?upstream=myServers&id=<id number>
```

Add and Remove Servers

```
#Add a new server to the myServer group with address localhost:8083 and \
http://<server>:8080/upstream_conf?add=&upstream=myServers&server=localhost
```

```
#Remove server with id=0 from the myServers upstream group
http://<server>:8080/upstream_conf?remove=&upstream=myServers&id=0
```


More Examples


#Modify the server with id = 0 and set the weight to 5 and the max_fails
`http://<server>:8080/upstream_conf?upstream=myServers&id=0&weight=5&max_:`

#Modify the server with id = 0 and set the route parameter to tomcat1
`http://<server>:8080/upstream_conf?upstream=myServers&id=0&route=tomcat1`

#Modify the server with id = 0 and set the server address to newdomain.co
`http://<server>:8080/upstream_conf?upstream=myServers&id=0&server=newdoma`

Using the Status Dashboard

http_backends.test



Edit selected

Add server

▼	Server			Requests		Responses			
	<input type="checkbox"/>	Name	DT	W	Total	Req/s	...	4xx	5xx
	<input type="checkbox"/>	95.211.80.227:80	1d 1h	1	0	0		0	0
	<input type="checkbox"/>	95.211.80.227:80	1d 1h	1	0	0		0	0
	<input type="checkbox"/>	127.0.0.1:8080	1h 18m	1	0	0		0	0

Lab 17: Dynamic Config

1. Open `myServers.conf` and add an `upstream_conf` location prefix in the same server block where the `status` prefix is located.
2. Add the `upstream_conf` directive inside the `upstream_conf` prefix
3. Open your browser and hit `http://server:9090/upstream_conf?upstream=myServers`, note the id number of the first server
4. Use the API commands to `remove` the server, then `add` it back with a `weight` of 5
5. Try using the `status.html` page to change the server details

Persistent Changes

For SDP protocols that allow automatic detection of devices and/or services, changes must persist across reloads

- Documentation: [Microservices architecture](#)
- Demo: [Service Discovery with Consul](#)

state Directive

state name must match **zone** name

Syntax:

state *file/path.state*

```
upstream myServers {  
    zone backend 64k;  
    state /etc/nginx/conf.d/backend.state;  
}
```

Lab 18: Persistent Dynamic Config

1. Create a directory for `/var/lib/nginx/state` and change ownership of the path like so:

```
$ sudo mkdir -p /var/lib/nginx/state  
$ sudo chown nginx:nginx /var/lib/nginx/state
```

2. Comment out the existing servers in the `upstream`
3. Save NGINX and reload
4. Run in a new browser window enter the following urls:

```
<server>:9090/upstream_conf?add=&upstream=myServers&server=127.0.0.1:8081&v  
<server>:9090/upstream_conf?add=&upstream=myServers&server=127.0.0.1:8082&v  
<server>:9090/upstream_conf?add=&upstream=myServers&server=127.0.0.1:8083&v
```

5. Go to your status.html page and dynamically edit these server details

INSTALLATION

Module Objective

This module will enable you to:

- Install NGINX from a binary distribution
- Compile binary from source code

NGINX Signing Key

```
wget http://nginx.org/keys/nginx_signing.key  
sudo apt-key add nginx_signing.key
```

Distribution URL

Edit the sources.list to retrieve correct distribution components

```
#Open the sources.list file with vim  
sudo vim /etc/apt/sources.list
```

```
#For Debian, append the following distribution URLs  
deb http://nginx.org/packages/debian/ codename nginx  
deb-src http://nginx.org/packages/debian/ codename nginx
```

```
#For Ubuntu  
deb http://nginx.org/packages/ubuntu/ codename nginx  
deb-src http://nginx.org/packages/ubuntu/ codename nginx
```

Distribution Codename Reference

Debian:

- 7.x - wheezy
- 6.x - squeeze

Ubuntu:

- 16.04 - xenial
- 14.04 - trusty

```
deb-src http://security.ubuntu.com/ubuntu xenial-security main restricted
deb http://security.ubuntu.com/ubuntu xenial-security universe
deb-src http://security.ubuntu.com/ubuntu xenial-security universe
deb http://security.ubuntu.com/ubuntu xenial-security multiverse
deb-src http://security.ubuntu.com/ubuntu xenial-security multiverse

## Uncomment the following two lines to add software from Canonical's
## 'partner' repository.
## This software is not part of Ubuntu, but is offered by Canonical and the
## respective vendors as a service to Ubuntu users.
# deb http://archive.canonical.com/ubuntu xenial partner
# deb-src http://archive.canonical.com/ubuntu xenial partner
deb http://nginx.org/packages/ubuntu/ xenial nginx
deb-src http://nginx.org/packages/ubuntu/ xenial nginx
```

Install: apt-get

```
sudo apt-get update  
sudo apt-get install nginx
```

Check Installation

NGINX will run on port 80 by default



Location of Files

NGINX Executable

```
/usr/sbin/nginx
```

Configuration File

```
/etc/nginx
```

Log Files

```
/var/log/nginx
```

Building NGINX From Source

General Steps:

- Download `.tar` file
- Extract the archive
- Run the `.configure` tool
- Add modules with various parameters using:

```
./configure --<param>=<paramValue>
```

- Run `make && sudo make install`

Command Reference

```
sudo wget <nginx.org/en/download link address>  
tar -xvf <nginx_mainline_version>.tar.gz  
cd <nginx_mainline_verison>  
./configure --with-http_ssl_module --with-debug --with-<other_modules>  
make && sudo make install
```


Important Notes

Make sure to download requisite libraries PRIOR to compiling the binary

Specify file paths as needed:

- `--prefix=path`
- `--sbin-path=path`
- `--conf-path=path`
- `--error-log=path`
- `--http-log=path`

`#Example`

```
./configure --sbin-path=/usr/local/nginx/nginx --conf-path=/usr/local/nginx
```

ADDITIONAL RESOURCES

Further Information

- [NGINX Documentation](#)
- [NGINX Admin Guides](#)
- [NGINX Blog](#)

Q&A

- Survey!
- Sales: nginx-inquiries@nginx.com