Using NGINX in Grid and Cloud middleware

Laura Cappelli

With the contribution of the CNAF-SD team
Federica Agostini, Francesco Giacomini, Roberta Miccoli, Enrico Vianello

28 April 2023

Outline

- The backstory of NGINX
- NGINX behavior: architecture and processes
- Configuration files
  - Serving static content
  - Run NGINX with some examples
- Using NGINX as reverse proxy
- NGINX modules
  - The *ngx_http_voms_module*
- Scripting
- Other feature: caching, TLS termination, load balancing & health check, TCP/UDP stream
- A use-case: the *WLCG StoRM Tape REST API* reverse proxy

Backstory

- NGINX is a free and open-source software (FreeBSD License) used as:
  - HTTP server
  - HTTP, mail and generic TCP/UDP proxy server
- Written by Igor Sysoev and released in 2004
  - The codebase was written from scratch in C and uses its own libraries
  - It has been ported to many architectures and operating systems (Linux, FreeBSD, Solaris, Mac OS X, AIX and Microsoft Windows)
- The NGINX company was founded in 2011
  - It provides NGINX Plus paid software and support for the open-source version
  - In March 2019, the company was acquired by F5 for $670 million
- According to Netcraft, NGINX served or proxied 21.37% busiest sites in March 2023, overtook Apache for the first time

The C10k problem

- The C10k problem was coined in 1999 by software engineer Dan Kegel
  - Problem of optimizing network sockets to handle 10˙000 clients at the same time
- Example: a simple Apache-based web server which serves a 100 KB web page
  - A fraction of a second to generate or retrieve the page
  - 10 seconds to transmit the page to a client with 10 KB/s bandwidth before freeing the connection
  - 1˙000 simultaneous connections with 1 MB of extra memory each: about 1 GB of extra memory devoted to serving just 1000 clients 100 KB of content
- To provide high levels of performance and concurrency, a website should be based on:
  - Efficient hardware, network capacity, application and data storage architectures
  - The web server should be able to scale better than linearly the memory and the CPU usage with the growing number of simultaneous connections and requests per second
- NGINX was created to solve the C10k problem

The beginning of NGINX

- From the beginning, NGINX was focused on:
  - High performance & concurrency
  - Low memory usage
  - Load balancing
  - Caching
- Base principles of NGINX:
  - NGINX doesn’t spawn new processes or threads for each request because it is computational expensive (requires a new runtime environment and execution context, heap and stack memory allocation, …)
  - It is based on the event-driven approach with a modular, asynchronous, single-threaded and non-blocking architecture
  - Connections are processed in a highly efficient run-loop in a limited number of single-threaded processes called workers
    - Within each worker nginx can handle many thousands of concurrent connections and requests per second with typical hardware
    - Even as load increases, memory and CPU usage remain manageable

NGINX Architecture

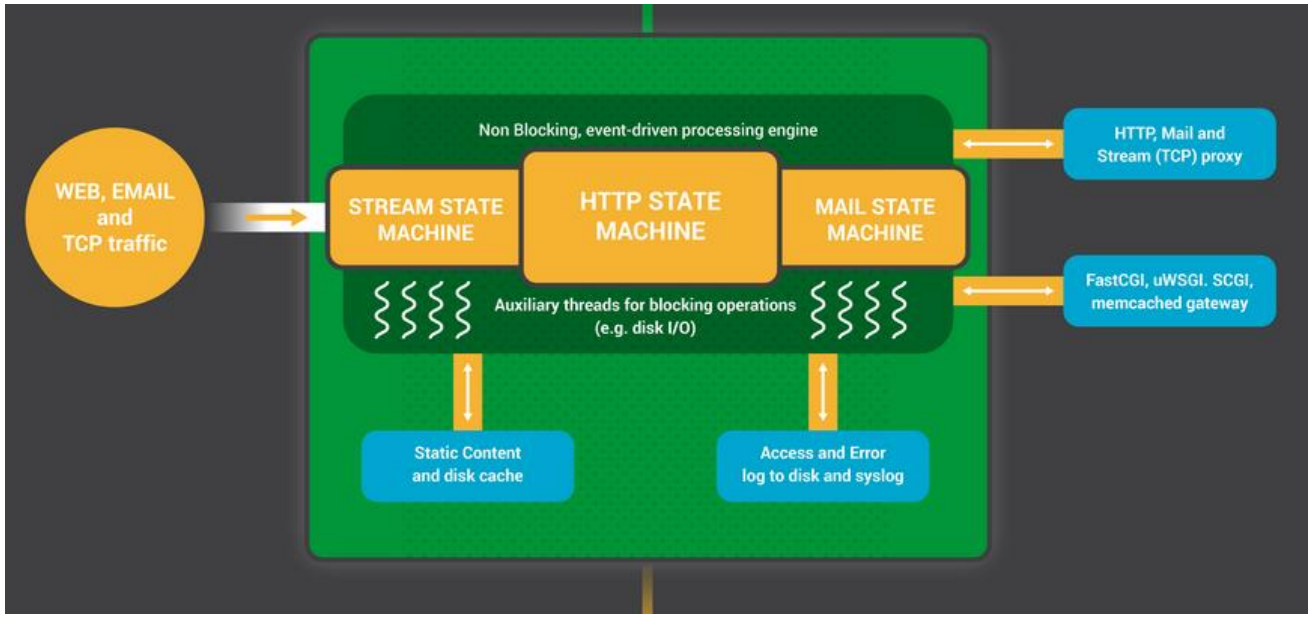
- NGINX has a limited number of single-threaded processes:
  - A master process
  - Many worker processes as the core number
  - The cache manager and the cache loader
- The processes can communicate using shared memory for:
  - Shared cache data
  - Session persistence data
  - Other shared resources

NGINX processes

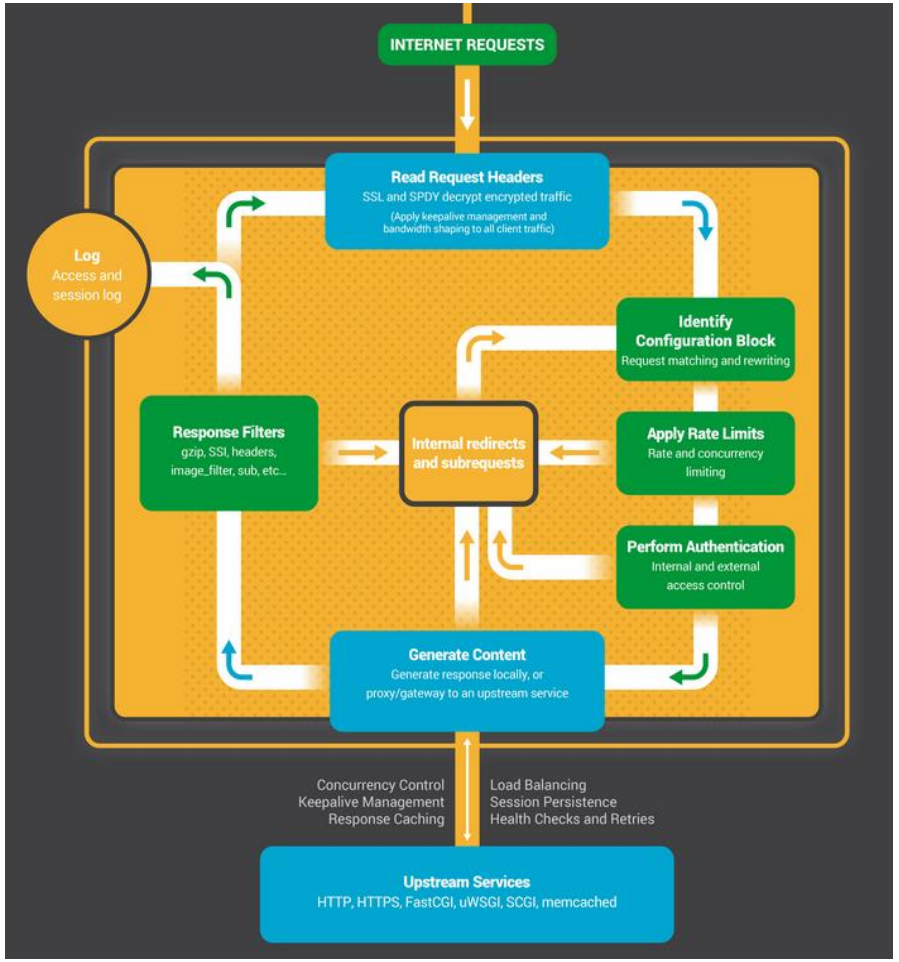
- The master process run as root user, and it is responsible for:
  - Reading and validating configuration
  - Creating, binding and closing sockets
  - Starting, terminating and maintaining the other processes
  - Perform online reconfigurations and upgrades
  - Compiling embedded scripts
- The cache loader process runs at startup:
  - Loads the disk-based cache into in-memory database with cache metadata
  - Updates the relevant entries in shared memory
  - Exits
- The cache manager process runs periodically and prunes entries from the disk caches depending on expiration and invalidation

The worker processes

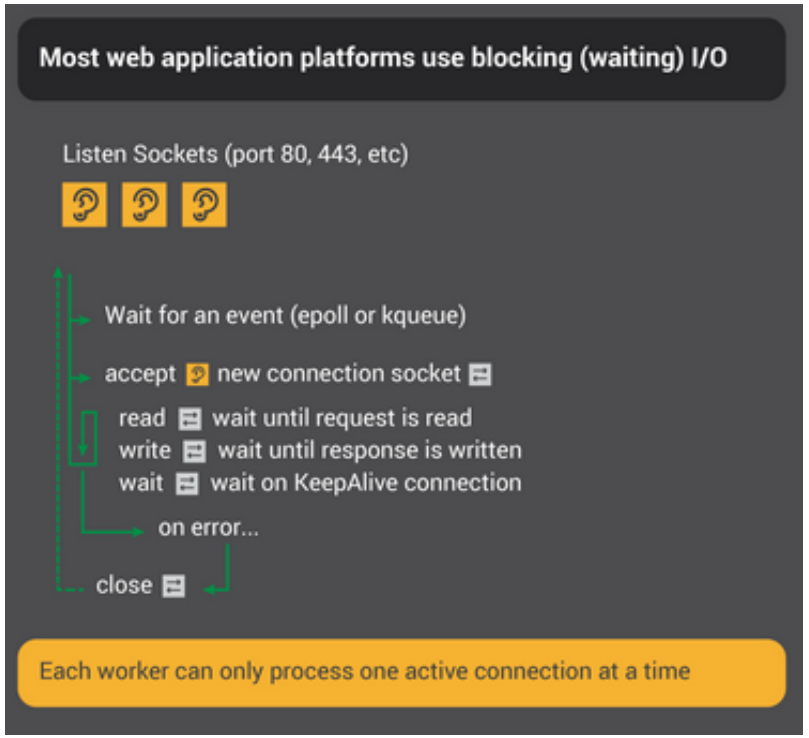
- The worker processes do the work independently from all the other processes
  - Handle multiple network connections
  - Read and write content to disk
  - Communicate with upstream servers
- Key principle: to be as non-blocking as possible
  - Uses heavily asynchronous tasks
  - A run-loop is the core of the worker process
    - Waiting for events on the listen sockets
    - Events are initiated by new incoming connections that are assigned to a state machine (eg: the HTTP state machine)

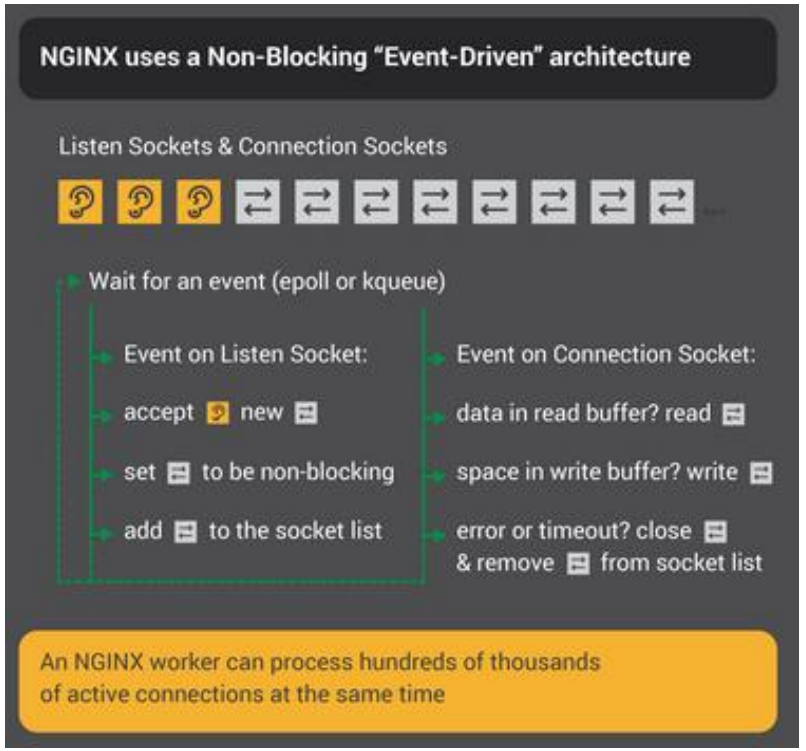


The HTTP state machine



- The state machine is the set of instructions that tell how to process a request
  - Most web servers use a similar state machine, the difference lies in the implementation





NGINX configuration

- The configuration is kept in a few text files
  - Typically in `/usr/local/etc/nginx` or `/etc/nginx` folders
  - The main configuration file is called `nginx.conf`
  - Parts of the configuration can be put in separate files (typically in the `/etc/nginx/conf.d` folder) which can be included in the main one
- When NGINX is started, the configuration files are read and verified by the master process
  - A compiled form of the configuration is passed to the worker processes as they are created
  - Configuration structures are automatically shared by the usual virtual memory management mechanisms
- The configuration is composed by:
  - Simple directives: name and parameters separated by spaces and ends with a semicolon
  - Complex directives or context: set of directives inside braces ({})

The standard version of `nginx.conf`

Main context
Global context

```
user  nginx;
worker_processes  auto;

error_log  /var/log/nginx/error.log notice;
pid        /var/run/nginx.pid;
```

Event context
Workers configuration

```
events {
    worker_connections  1024;
}
```

HTTP context
Manage HTTP/HTTPS traffic

```
http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';
    access_log  /var/log/nginx/access.log  main;

    sendfile        on;
    keepalive_timeout  65;

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

Serving static content

The `default.conf` file in the `conf.d` folder contains the following code:

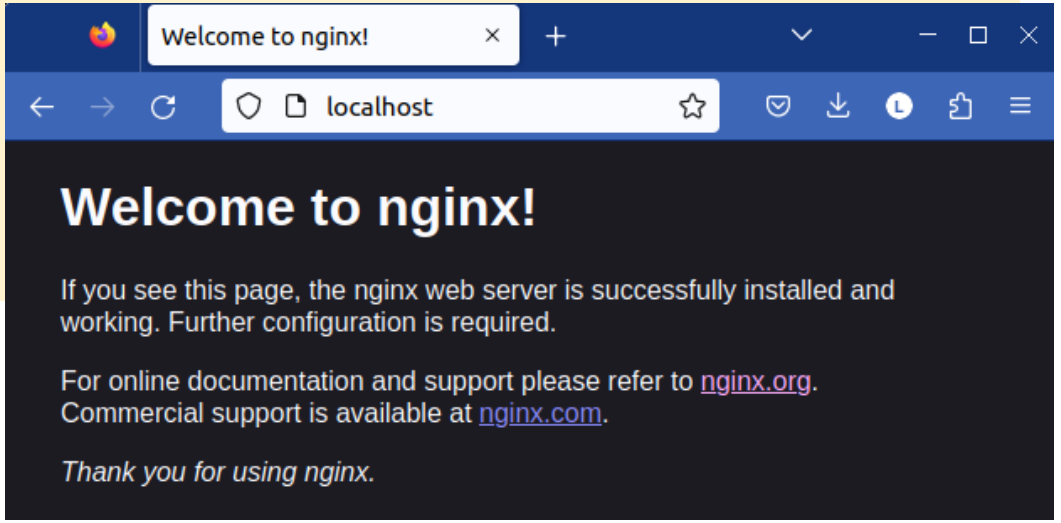
```
http {
    ...
    server {
        listen       80;
        server_name  localhost;

        location / {
            root   /usr/share/nginx/html;
            index  index.html index.htm;
        }

        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   /usr/share/nginx/html;
        }
    }
    ...
}
```

Server context

Server configuration



Run NGINX

- The NGINX execution command is: `nginx`
  - The possible options are:

```
# nginx -help
nginx version: nginx/1.24.0
Usage: nginx [-?hvVtTq] [-s signal] [-p prefix]
             [-e filename] [-c filename] [-g directives]

Options:
  -?,-h         : this help
  -v            : show version and exit
  -V            : show version and configure options then exit
  -t            : test configuration and exit
  -T            : test configuration, dump it and exit
  -q            : suppress non-error messages during configuration testing
  -s signal     : send signal to a master process: stop, quit, reopen, reload
  -p prefix     : set prefix path (default: /etc/nginx/)
  -e filename   : set error log file (default: /var/log/nginx/error.log)
  -c filename   : set configuration file (default: /etc/nginx/nginx.conf)
  -g directives : set global directives out of configuration file
```

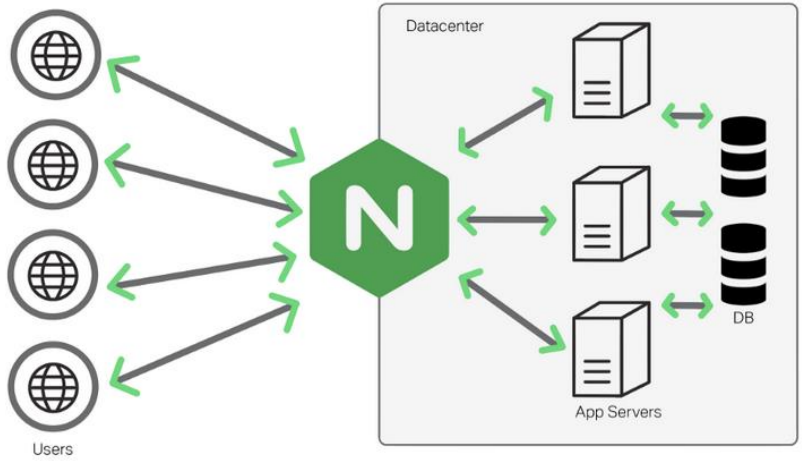
First simple exercises

- Prerequisite: install NGINX in an appropriate environment:
  - You can find the documentation on the official website https://nginx.org/en/
  - At this link there is a repo with a ready-to-use Dockerfile
- Start NGINX with the standard configuration and show its welcome page
- Modify the configuration to print "Hello <your name>, welcome to nginx!" and reload NGINX
- Send your name to NGINX as URL parameter and return the string "Hello <your name>!"
  - Hint: use the NGINX variable `$arg_<parameter-name>` and the `return` directive
  - You can add the new location `/hello` and query NGINX from command line:

    ```
    curl http://localhost/hello?person=laura
    ```

Reverse proxy

- A reverse proxy is an application that sits in front of back-end applications and forwards client requests to those applications
- Proxying is typically used to:
  - Distribute the load among servers
  - Hide the existence and the characteristics of origin servers
  - Provide a single public IP address for multiple web-servers listen on different ports in the same or on different machines
  - Cache content for reducing the load
  - Add access authentication and TLS encryption



- NGINX can be configured as reverse proxy for HTTP and other protocols
  - E.g. TCP/UDP, FastCGI, uwsgi, SCGI, and memcached

Reverse proxy with NGINX

- To pass a request to an HTTP proxied server, the `proxy_pass` directive is specified inside a `location` context
- Example: serving static content provided by a virtual server that listen on a different port (that could be not directly reachable from the client side)

```
server {
    listen    80;
    location / {
        proxy_pass http://localhost:8080;
    }
}

server {
    listen    8080;
    root /tmp/simple-reverse-proxy;
    location / {
        index  proxy-index.html;
    }
}
```

Passing Request Headers

- By default, in proxied requests NGINX eliminates the empty header fields and redefines the following header fields:
  - `Host` is set to the `$proxy_host` variable
  - `Connection` is set to `close`
- To change the header field in proxied request, use the `proxy_set_header` directive

```
location /some/path/ {
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_pass http://localhost:8000;
}
```

- To prevent a header field from being passed to the proxied server, set it to an empty string

```
location /some/path/ {
    proxy_set_header Accept-Encoding "";
    proxy_pass http://localhost:8000;
}
```

Reverse proxy – a more complex example

- Three servers running in three different containers behind a NGINX reverse proxy
  - The servers are listening on the port 8080 and they are reachable from http://localhost/one, http://localhost/two, http://localhost/three
- Some key points of the solution
  - We use the `upstream` context to define each server

```
upstream service-one {
    server service-one:8080; # this will point to the Docker Container DNS
}
```

  - In the `server` context we define the locations with the `proxy_pass` directives, for example:

```
location /one {
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_set_header X-Forwarded-Proto $scheme;
    proxy_pass http://service-one;
}
```

  - The solution is available here

NGINX Modules

- NGINX is a collection of modules
  - About one hundred are part of the core (`http`, `stream`, `mail`, `ngx_http_proxy_module`, ...)
  - There are thousands of 3rd party modules listed here (e.g. HTTP Healthcheck, HTTP echo, LDAP Auth)
- The modules can be:
  - Static: the module is compiled into the NGINX server binary at compile time

    ```
    ./configure --prefix=/opt/nginx --add-module=/path/to/my-module
    make install
    ```

  - Dynamic: the module can be loaded or unloaded into NGIN at runtime based on configuration files

    ```
    ./configure --add-dynamic-module=/opt/source/ngx_my_module
    make modules && make install
    ```

    - To enable dynamic modules compatibility, compile the modules with the `--with-compat` option
    - To load the module into the *.conf* files use the `load_module` directive

      ```
      load_module modules/ngx_my_module.so;
      ```

- In both the cases, the NGINX source file is needed

The ngx_http_voms_module

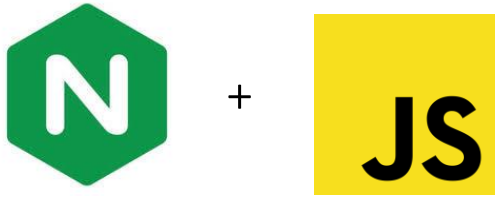
- It is possible developing a customized module
  - A good understanding of the NGINX internal architecture is required
  - It must be event-based and non-blocking
- We develops and maintains a module to integrate VOMS in NGINX (VOMS termination)
  - It enables client-side authentication based on X.509 proxy certificates augmented with the VOMS AC obtained from a VOMS server
  - The module defines a set of *embedded* variables, whose values are extracted from the first Attribute Certificate found in the certificate chain
  - The repo is on baltig: https://baltig.infn.it/storm2/ngx_http_voms_module
  - A docker image with NGINX 1.24.0, the VOMS module and the independent HTTPG patch is available on DockerHub at this link

Scripting

- If your desired behavior is not possible to handle with the configuration file, the next stop would be implementing it with scripting
  - Scripting allows you to use existing and widely known languages to extend the functionality of NGINX
- NGINX supports 3 scripting methods:
  1. Perl modules with the experimental `ngx_http_perl_module` (the complete one is only for NGINX Plus) used for less complex use-cases
  2. Lua code with OpenResty, a web platform that integrates NGINX, LuaJIT, Lua libraries and 3rd-party NGINX modules
     - It's a 3rd-party software with a modified version of the NGINX core and many dependencies
     - You can write Lua code inside the .conf files

Scripting with njs

3. Using JavaScript with the *nginscript* module, or *njs*, developed and maintained by NGINX
   - It is a subset of the JavaScript language with a compiler that produce an *executable* when the NGINX process starts
   - It is theoretically faster than the others scripting methods
   - All the JS code must be collocated in a sort of library files that you can import and use in the NGINX configuration



- Some considerations based on our experience:
  - JavaScript is a better-known language than Lua
  - The njs module is a small implementation of JS and it is still under development
  - Theoretically, you can use JS modules and TypeScript to extend njs, but we are experimenting several issues on their use
  - There are several useful example on https://github.com/nginx/njs-examples

Caching

- NGINX can cache all the content requested from the clients to the origin servers it serves
  - If a client requests a cached content, NGINX returns the content directly
  - Only two directives are needed to enable basic caching:
    - `proxy_cache_path` – sets the cache path and configuration
    - `proxy_cache` – activates the cache configuration for a specific location
  - Basic example

```
proxy_cache_path /path/to/cache levels=1:2 keys_zone=my_cache:10m max_size=10g inactive=60m use_temp_path=off;
server {
    # ...
    location / {
        proxy_cache my_cache;
        proxy_pass http://my_upstream;
    }
}
```

  - NGINX has other optional settings for fine-tuning the cache and its performance (e.g. set different timing options, specify the cache key, splitting the cache across multiple hard drives, …)

Configuring HTTPS servers - TLS termination

- To configure an HTTPS server
  - The `ssl` parameter must be enabled in the server block
  - The locations of the server certificate (sent to every client that connects to the server) and the private key files should be specified

```
server {
    listen 443 ssl;
    ssl on;
    server_name          www.example.com;
    ssl_certificate      certs/example.com.pem;
    ssl_certificate_key  certs/example.com.key;
    ssl_trusted_certificate file; | ssl_client_certificate  file;
    ssl_protocols        TLSv1 TLSv1.1 TLSv1.2 TLSv1.3;
    …
    location / {
        proxy_pass http://127.0.0.1:8000;
    }
}
```

- There are several directives to optimize the performance of the SSL operations, such as:
  - Enable keepalive connections to send several requests via one connection
  - reuse SSL session parameters to avoid SSL handshakes for parallel and subsequent connections.

Load balancing & health check

- NGINX can be used as load balancer to distribute traffic to several application servers and to improve performance, scalability and reliability of web applications
- Supported load balancing mechanism:
  - Round-robin (default) – requests distributed in a round-robin fashion
  - Least-connected – next request is assigned to the server with the least number of active connections
  - Hash methods – a hash-function is used to determine what server should be selected for the next request; the IP-hash is used when there is the need to tie a client to a particular application server
- It is also possible to influence nginx load balancing algorithms by using server weights

```
http {
    upstream myapp1 {
        # least_conn; ip_hash;
        server srv1.example.com;
        server srv2.example.com;
        server srv3.example.com;
        # server srv4.example.com weight=3;
    }

    server {
        listen 80;
        location / {
            proxy_pass http://myapp1;
        }
    }
}
```

- Reverse proxy includes server health checks: if the response from a particular server fails with an error, nginx will mark this server as failed, and will try to avoid selecting this server for subsequent inbound requests for a while

Stream context

- NGINX can proxy and load balance not only HTTP or HTTPS protocols, but also TCP and UDP traffic
  - Instead of using `http` context, you can use the `stream` block with one or more `server` context
  - The `listen` directive in a stream-server context uses TCP as default protocol, otherwise you can specify `udp` as parameter

TCP server

UDP server

```
stream {
    server {
        listen 12345;
        # ...
    }

    server {
        listen 53 udp;
        # ...
    }
    # ...
}
```

Stream context – examples

- VOMS AA use the HTTPG protocol, so NGINX uses a stream block with a TCP server to communicate with it
- A useful module in this context could be `ngx_stream_ssl_preread_module`
  - It allows extracting information from the ClientHello message without terminating SSL/TLS
  - E.g: selecting an upstream based on server name requested through Server Name Indication (SNI)
    - TLS does not provide a mechanism for a client to tell a server the name of the server it is contacting
    - It may be desirable for clients to provide this information to facilitate secure connections to servers that host multiple 'virtual' servers at a single underlying network address
    - To provide any of the server names, clients MAY include an extension of type "server_name" in the ClientHello message

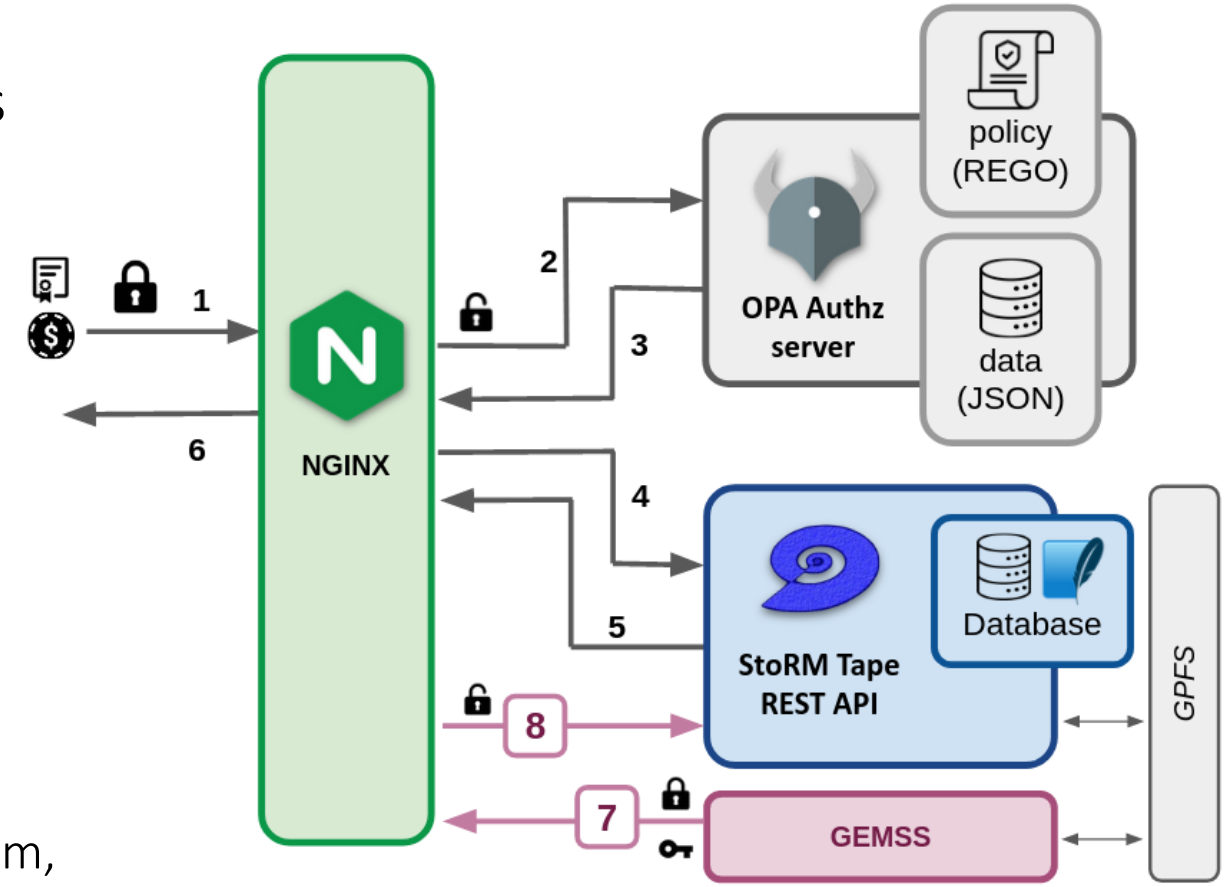
```
map $ssl_preread_server_name $name {
    backend.example.com backend;
    default backend2;
}
```

```
upstream backend {
    server 192.168.0.1:12345;
    server 192.168.0.2:12345;
}
upstream backend2 {
    server 192.168.0.3:12345;
    server 192.168.0.4:12345;
}
```

```
server {
    listen 12346;
    proxy_pass $name;
    ssl_preread on;
}
```

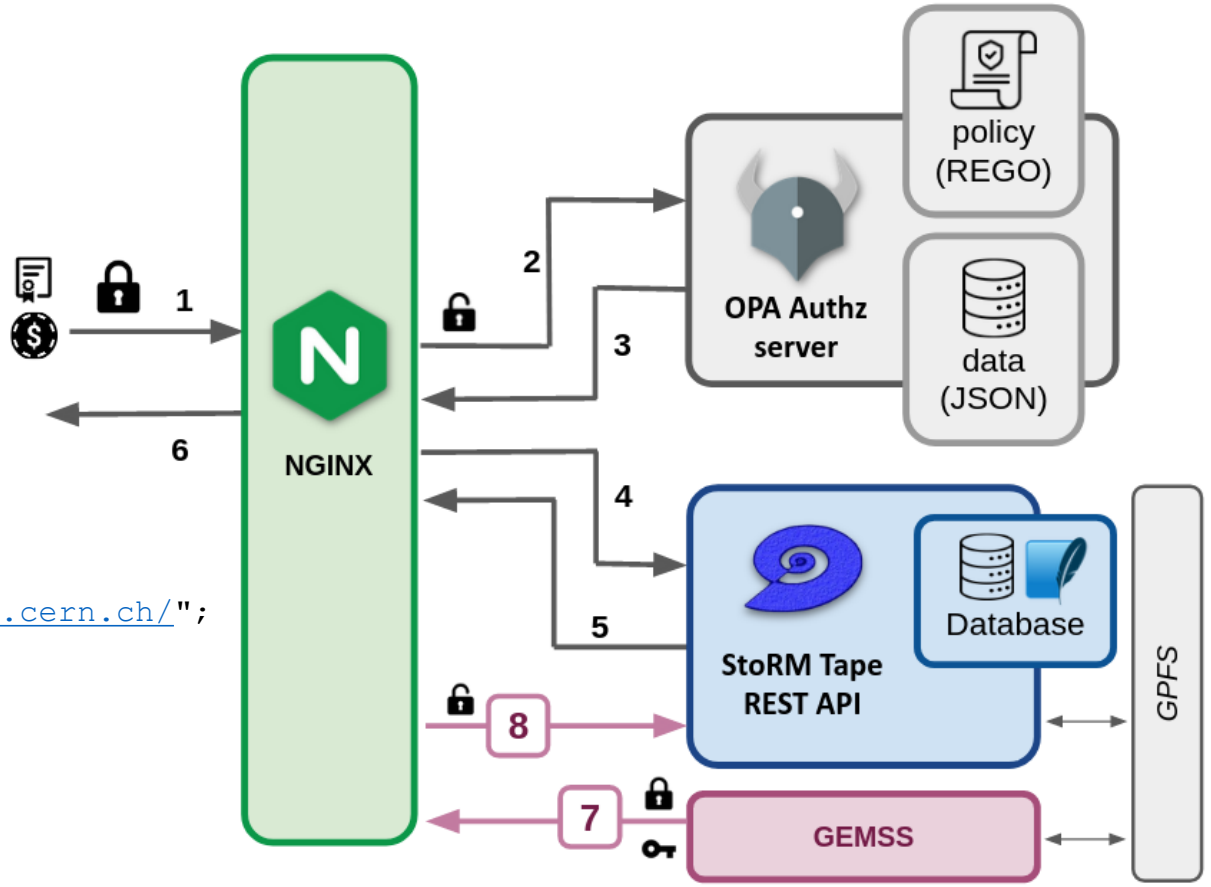
The WLCG StoRM Tape REST API

- The WLCG tape REST API allows clients to manage disk residency of tape stored files
- Software structure:
  - NGINX reverse proxy
  - OPA authorization server
  - StoRM Tape REST API
  - GEMSS
- Authentication is managed by NGINX and supports:
  - VOMS certificates with the `ngx_http_voms_module`
  - JWT (experimental) – authn written by SD team, but we hope to use some 3rd-party libraries



StoRM Tape REST API – NGINX config

```
load_module modules/ngx_http_voms_module.so;
load_module modules/ngx_http_js_module.so;
…
server{
  …
  location /api/v1 {
    auth_request /authz;
    proxy_set_header    X-SSL-Client-S-Dn $ssl_client_s_dn;
    proxy_set_header    x-voms_fqans $voms_fqans;
    …
    proxy_pass              http://storm-tape:8080;
  }
  location /authz {
    internal;
    js_var $trusted_issuers
     "https://wlcg.cloud.cnaf.infn.it/,https://cms-auth.web.cern.ch/";
    js_content auth_engine.authorize_operation;
  }
  location /_opa {
    internal;
    …
    proxy_pass http://opa:8181/;
  }
}
```



References

- *The Architecture of Open Source Applications (Vol 2)*, Chapter 14, A. Alexeev, Edited by Brown & Greg Wilson
- NGINX Documentation: http://nginx.org/en/
- NGINX Blog: https://www.nginx.com/blog/
- OpenResty: http://openresty.org/en/
- VOMS module: https://baltig.infn.it/storm2/ngx_http_voms_module
- StoRM Tape REST API: https://baltig.infn.it/cnafsd/storm-tape
- StoRM Tape REST API testsuite: https://baltig.infn.it/cnafsd/storm-tape-ts