Liam Crilly of F5 January 20, 2021

Deploying NGINX as an API Gateway, Part 1

API gateway

This is the first blog post in our series on deploying NGINX Open Source and NGINX Plus as an API gateway:

- *This post provides detailed configuration instructions for several use cases. Originally published in 2018, it has been updated to reflect current best practice for API configuration, using nested* `location` *blocks to route requests, instead of rewrite rules.*
- *Part 2 extends those use cases and looks at a range of safeguards that can be applied to protect and secure backend API services in production.*
- *Part 3 explains how to deploy NGINX Open Source and NGINX Plus as an API gateway for gRPC services.*

Note: Except as noted, all information in this post applies to both NGINX Open Source and NGINX Plus. For ease of reading, the rest of the blog refers simply to “NGINX”.

At the heart of modern application architectures is the HTTP API. HTTP enables applications to be built rapidly and maintained easily. The HTTP API provides a common interface, regardless of the scale of the application, from a single-purpose microservice to an all-encompassing monolith. By using HTTP, the advancements in web application delivery that support hyperscale Internet properties can also be used to provide reliable and high-performance API delivery.

For an excellent introduction to the importance of API gateways for microservices applications, see Building Microservices: Using an API Gateway on our blog.

As the leading high-performance, lightweight reverse proxy and load balancer, NGINX has the advanced HTTP processing capabilities needed for handling API traffic. This makes NGINX the ideal platform with which to build an API gateway. In this blog post we describe a number of common API gateway use cases and show how to configure NGINX to handle them in a way that is efficient, scalable, and easy to maintain. We describe a complete configuration, which can form the basis of a production deployment.

Introducing the Warehouse API

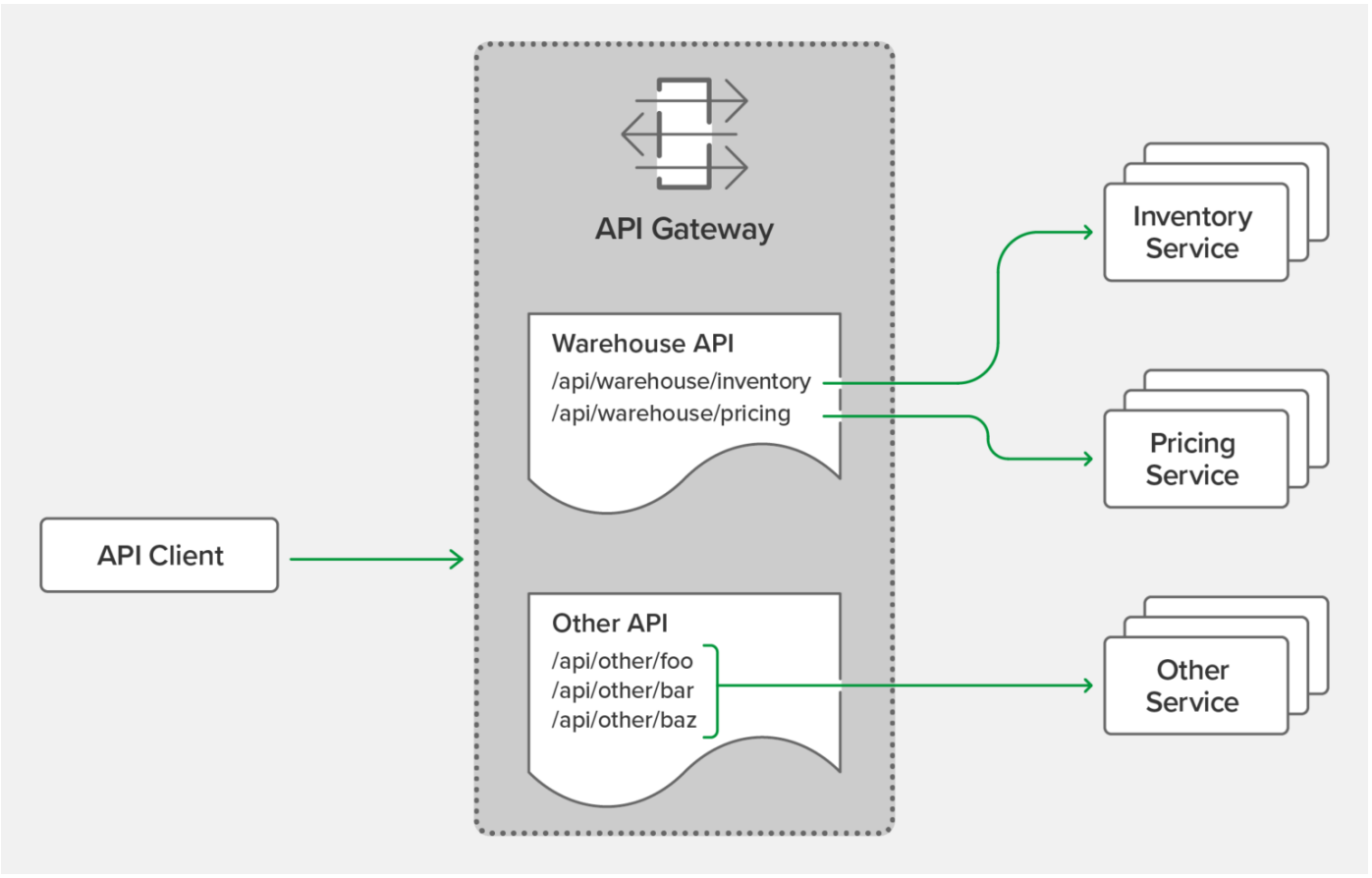
The primary function of the API gateway is to provide a single, consistent entry point for multiple APIs, regardless of how they are implemented or deployed at the backend. Not all APIs are microservices applications. Our API gateway needs to manage existing APIs, monoliths, and applications undergoing a partial transition to microservices.

In this blog post we refer to a hypothetical API for inventory management, the “Warehouse API”. We use sample configuration code to illustrate different use cases. The Warehouse API is a RESTful API that consumes JSON requests and produces JSON responses. The use of JSON is not, however, a limitation or requirement of NGINX when deployed as an API gateway; NGINX is agnostic to the architectural style and data formats used by the APIs themselves.

to different backends. So the API's path structure is:

```
api
└── warehouse
    ├── inventory
    └── pricing
```

As an example, to query the current warehouse inventory, a client application makes an HTTP `GET` request to **/api/warehouse/inventory**.



API gateway architecture for multiple applications

Organizing the NGINX Configuration

One advantage of using NGINX as an API gateway is that it can perform that role while simultaneously acting as a reverse proxy, load balancer, and web server for existing HTTP traffic. If NGINX is already part of your application delivery stack then it is generally unnecessary to deploy a separate API gateway. However, some of the default behavior expected of an API gateway differs from that expected for browser-based traffic. For that reason we separate the API gateway configuration from any existing (or future) configuration for browser-based traffic.

To achieve this separation, we create a configuration layout that supports a multi-purpose NGINX instance, and provides a convenient structure for automating configuration deployment through CI/CD pipelines. The resulting directory structure under **/etc/nginx** looks like this.

```
    ├── api_conf.d/ ..................................... Subdirectory for per-API configuration
    │   └── warehouse_api.conf ..... Definition and policy of the Warehouse API
    ├── api_backends.conf ..................... The backend services (upstreams)
    ├── api_gateway.conf ...................... Top-level configuration for the API gateway
server
    ├── api_json_errors.conf ............ HTTP error responses in JSON format
    ├── conf.d/
    │   ├── ...
    │   └── existing_apps.conf
    └── nginx.conf
```

The directories and filenames for all API gateway configuration are prefixed with **api_**. Each of these files and directories enables a different feature or capability of the API gateway as explained in detail below. The **warehouse_api.conf** file is a generic stand-in for the configuration files discussed below that define the Warehouse API in different ways.

Defining the Top-Level API Gateway

All NGINX configuration starts with the main configuration file, **nginx.conf**. To read in the API gateway configuration, we add an `include` directive in the `http` block in **nginx.conf** that references the file containing the gateway configuration, **api_gateway.conf** (line 28 just below). Note that the default **nginx.conf** file uses an `include` directive to pull in browser-based HTTP configuration from the **conf.d** subdirectory (line 29). This blog post makes extensive use of the `include` directive to aid readability and to enable automation of some parts of the configuration.

```
28      include /etc/nginx/api_gateway.conf; # All API gateway configuration
29      include /etc/nginx/conf.d/*.conf;    # Regular web traffic
```

nginx.conf hosted with ♡ by GitHub view raw

The **api_gateway.conf** file defines the virtual server that exposes NGINX as an API gateway to clients. This configuration exposes all of the APIs published by the API gateway at a single entry point, **https://api.example.com/** (line 9), protected by TLS as configured on lines 12 through 17. Notice that this configuration is purely HTTPS – there is no plaintext HTTP listener. We expect API clients to know the correct entry point and to make HTTPS connections by default.

This configuration is intended to be static – the details of individual APIs and their backend services are specified in the files referenced by the `include` directive on line 20. Lines 23 through 26 deal with error handling, and are discussed in Responding to Errors below.

```
1   include api_backends.conf;
2   include api_keys.conf;
3
4   server {
5       access_log /var/log/nginx/api_access.log main; # Each API may also log to a
6                                                      # separate file
7
8       listen 443 ssl;
9       server_name api.example.com;
10
11      # TLS config
12      ssl_certificate      /etc/ssl/certs/api.example.com.crt;
13      ssl_certificate_key  /etc/ssl/private/api.example.com.key;
14      ssl_session_cache    shared:SSL:10m;
15      ssl_session_timeout  5m;
16      ssl_ciphers          HIGH:!aNULL:!MD5;
17      ssl_protocols        TLSv1.2 TLSv1.3;
18
```

```
22      # Error responses
23      error_page 404 = @400;         # Invalid paths are treated as bad requests
24      proxy_intercept_errors on;     # Do not send backend errors to the client
25      include api_json_errors.conf;  # API client friendly JSON error responses
26      default_type application/json; # If no content-type then assume JSON
```

api_gateway.conf hosted with ♡ by GitHub view raw

Single-Service vs. Microservice API Backends

Some APIs may be implemented at a single backend, although we normally expect there to be more than one, for resilience or load balancing reasons. With microservices APIs, we define individual backends for each service; together they function as the complete API. Here, our Warehouse API is deployed as two separate services, each with multiple backends.

```
1   upstream warehouse_inventory {
2       zone inventory_service 64k;
3       server 10.0.0.1:80;
4       server 10.0.0.2:80;
5       server 10.0.0.3:80;
6   }
7
8   upstream warehouse_pricing {
9       zone pricing_service 64k;
10      server 10.0.0.7:80;
11      server 10.0.0.8:80;
12      server 10.0.0.9:80;
13  }
```

api_backends.conf hosted with ♡ by GitHub view raw

All of the backend API services, for all of the APIs published by the API gateway, are defined in **api_backends.conf**. Here we use multiple IP address-port pairs in each `upstream` block to indicate where the API code is deployed, but hostnames can also be used. NGINX Plus subscribers can also take advantage of dynamic DNS load balancing to have new backends added to the runtime configuration automatically.

Defining the Warehouse API

The Warehouse API is defined by a number of `location` blocks in a nested configuration, as illustrated by the following example. The outer `location` block (**/api/warehouse**) identifies the base path, under which nested locations specify the valid URIs that get routed to the backend API services. Using an outer block enables us to define common policies that apply to the entire API (in this example, the logging configuration on line 6).

```
1   # Warehouse API
2   #
3   location /api/warehouse/ {
4       # Policy configuration here (authentication, rate limiting, logging...)
5       #
6       access_log /var/log/nginx/warehouse_api.log main;
7
8       # URI routing
9       #
10      location /api/warehouse/inventory {
11          proxy_pass http://warehouse_inventory;
12      }
13
14      location /api/warehouse/pricing {
```

```
18        return 404; # Catch-all
19    }
```

warehouse_api_simple.conf hosted with ♡ by GitHub view raw

NGINX has a highly efficient and flexible system for matching the request URI to a section of the configuration. The order of the `location` directives is not important – the most specific match is chosen. Here, the nested locations on lines 10 and 14 define two URIs that are more specific than the outer `location` block; the `proxy_pass` directive in each nested block routes requests to the appropriate upstream group. Policy configuration is inherited from the outer location unless there is a need to provide a more specific policy for certain URIs.

Any URIs that do not match one of the nested locations are handled by the outer location, which includes a catch-all directive (line 18) that returns the response `404 (Not Found)` for all invalid URIs.

Choosing Broad vs. Precise Definition for APIs

There are two approaches to API definition – broad and precise. The most suitable approach for each API depends on the API's security requirements and whether it is desirable for the backend services to handle invalid URIs.

In **warehouse_api_simple.conf** above, we use the broad approach for the Warehouse API, defining URI prefixes on lines 10 and 14 such that a URI that begins with one of the prefixes is proxied to the appropriate backend service. With this broad, prefix-based location matching, API requests to the following URIs are all valid:

/api/warehouse/inventory
/api/warehouse/inventory/
/api/warehouse/inventory/foo
/api/warehouse/inventoryfoo
/api/warehouse/inventoryfoo/bar/

If the only consideration is proxying each request to the correct backend service, the broad approach provides the fastest processing and most compact configuration. On the other hand, a more precise approach enables the API gateway to understand the API's full URI space by explicitly defining the URI path for each available API resource. Taking the precise approach, the following configuration for URI routing in the Warehouse API uses a combination of exact matching (**=**) and regular expressions (**~**) to define each and every valid URI.

```
8     # URI routing
9     #
10    location = /api/warehouse/inventory { # Complete inventory
11        proxy_pass http://warehouse_inventory;
12    }
13
14    location ~ ^/api/warehouse/inventory/shelf/[^/]+$ { # Shelf inventory
15        proxy_pass http://warehouse_inventory;
16    }
17
18    location ~ ^/api/warehouse/inventory/shelf/[^/]+/box/[^/]+$ { # Box on shelf
19        proxy_pass http://warehouse_inventory;
20    }
21
22    location ~ ^/api/warehouse/pricing/[^/]+$ { # Price for specific item
23        proxy_pass http://warehouse_pricing;
24    }
```

warehouse_api_precise.conf hosted with ♡ by GitHub view raw

requests, at the cost of some small additional overhead for regular expression matching. With this configuration in place, NGINX accepts some URIs and rejects others as invalid:

Valid URIs	Invalid URIs
/api/warehouse/inventory	/api/warehouse/inventory/
/api/warehouse/inventory/shelf/foo	/api/warehouse/inventoryfoo
/api/warehouse/inventory/shelf/foo/box/bar	/api/warehouse/inventory/shelf
/api/warehouse/inventory/shelf/-/box/-	/api/warehouse/inventory/shelf/foo/bar
/api/warehouse/pricing/baz	/api/warehouse/pricing
	/api/warehouse/pricing/baz/pub

Using a precise API definition enables existing API documentation formats to drive the configuration of the API gateway. It is possible to automate the NGINX API definitions from the OpenAPI Specification (formerly called Swagger). A sample script for this purpose is provided among the Gists for this blog post.

Rewriting Client Requests to Handle Breaking Changes

As APIs evolve, it's sometimes necessary to make changes that break strict backward compatibility and require clients to be updated. One such example is when an API resource is renamed or moved. Unlike a web browser, an API gateway cannot send its clients a redirect (code `301 (Moved Permanently)`) naming the new location. Fortunately, when it's impractical to modify API clients, we can rewrite client requests on the fly.

In the following example, we use the same broad approach as in **warehouse_api_simple.conf** above, but in this case the configuration is replacing a previous version of the Warehouse API where the pricing service was implemented as part of the inventory service. The `rewrite` directive on line 3 converts requests to the old pricing resource into requests to the new pricing service.

```
1   # Rewrite rules
2   #
3   rewrite ^/api/warehouse/inventory/item/price/(.*)  /api/warehouse/pricing/$1;
4
5   # Warehouse API
6   #
7   location /api/warehouse/ {
8       # Policy configuration here (authentication, rate limiting, logging...)
9       #
10      access_log /var/log/nginx/warehouse_api.log main;
11
12      # URI routing
13      #
14      location /api/warehouse/inventory {
15          proxy_pass http://warehouse_inventory;
16      }
17
```

```
21
22      return 404; # Catch-all
23  }
```

warehouse_api_rewrites.conf hosted with ♡ by GitHub view raw

Responding to Errors

One of the key differences between HTTP APIs and browser-based traffic is how errors are communicated to the client. When NGINX is deployed as an API gateway, we configure it to return errors in a way that best suits the API clients.

The top-level API gateway configuration includes a section that defines how to handle error responses.

```
22      # Error responses
23      error_page 404 = @400;         # Invalid paths are treated as bad requests
24      proxy_intercept_errors on;     # Do not send backend errors to the client
25      include api_json_errors.conf;  # API client friendly JSON error responses
26      default_type application/json; # If no content-type then assume JSON
```

api_gateway.conf hosted with ♡ by GitHub view raw

The `error_page` directive on line 23 specifies that when a request does not match any of the API definitions, NGINX returns the `400` (`Bad Request`) error instead of the default `404` (`Not Found`) error. This (optional) behavior requires that API clients make requests only to the valid URIs included in the API documentation, and prevents unauthorized clients from discovering the URI structure of the APIs published through the API gateway.

Line 24 refers to errors generated by the backend services themselves. Unhandled exceptions may contain stack traces or other sensitive data that we don't want to be sent to the client. This configuration adds a further level of protection by sending a standardized error response to the client.

The complete list of standardized error responses is defined in a separate configuration file referenced by the `include` directive on line 25, the first few lines of which are shown below. This file can be modified if an error format other than JSON is preferred, with the `default_type` value on line 26 of **api_gateway.conf** changed to match. You can also have a separate `include` directive in each API's policy section to reference a different file of error responses which override the global responses.

```
1   error_page 400 = @400;
2   location @400 { return 400 '{"status":400,"message":"Bad request"}\n'; }
3
4   error_page 401 = @401;
5   location @401 { return 401 '{"status":401,"message":"Unauthorized"}\n'; }
6
7   error_page 403 = @403;
8   location @403 { return 403 '{"status":403,"message":"Forbidden"}\n'; }
9
10  error_page 404 = @404;
11  location @404 { return 404 '{"status":404,"message":"Resource not found"}\n'; }
```

api_json_errors.conf hosted with ♡ by GitHub view raw

With this configuration in place, a client request for an invalid URI receives the following response.

```
HTTP/1.1 400 Bad Request
Server: nginx/1.19.5
Content-Type: application/json
Content-Length: 39
Connection: keep-alive

{"status":400,"message":"Bad request"}
```

Implementing Authentication

It is unusual to publish APIs without some form of authentication to protect them. NGINX offers several approaches for protecting APIs and authenticating API clients. For information about approaches that also apply to regular HTTP requests, see the documentation for IP address-based access control lists (ACLs), digital certificate authentication, and HTTP Basic authentication. Here, we focus on API-specific authentication methods.

API Key Authentication

API keys are a shared secret known by the client and the API gateway. An API key is essentially a long and complex password issued to the API client as a long-term credential. Creating API keys is simple – just encode a random number as in this example.

```
$ openssl rand -base64 18
7B5zIqmRGXmrJTFmKa99vcit
```

On line 2 of the top-level API gateway configuration file, **api_gateway.conf**, we include a file called **api_keys.conf**, which contains an API key for each API client, identified by the client's name or other description. Here are the contents of that file:

```
1  map $http_apikey $api_client_name {
2      default "";
3
4      "7B5zIqmRGXmrJTFmKa99vcit" "client_one";
5      "QzVV6y1EmQFbbxOfRCwyJs35" "client_two";
6      "mGcjH8Fv6U9y3BVF9H3Ypb9T" "client_six";
7  }
```

api_keys.conf hosted with ♡ by GitHub view raw

The API keys are defined within a `map` block. The `map` directive takes two parameters. The first defines where to find the API key, in this case in the `apikey` HTTP header of the client request as captured in the `$http_apikey` variable. The second parameter creates a new variable (`$api_client_name`) and sets it to the value of the second parameter on the line where the first parameter matches the key.

For example, when a client presents the API key `7B5zIqmRGXmrJTFmKa99vcit`, the `$api_client_name` variable is set to `client_one`. This variable can be used to check for authenticated clients and included in log entries for more detailed auditing. The format of the `map` block is simple and easy to integrate into automation workflows that generate the **api_keys.conf** file from an existing credential store.

delegates the authentication decision to a specified location.

```
1   # Warehouse API
2   #
3   location /api/warehouse/ {
4       # Policy configuration here (authentication, rate limiting, logging...)
5       #
6       access_log /var/log/nginx/warehouse_api.log main;
7       auth_request /_validate_apikey;
8
9       # URI routing
10      #
11      location /api/warehouse/inventory {
12          proxy_pass http://warehouse_inventory;
13      }
14
15      location /api/warehouse/pricing {
16          proxy_pass http://warehouse_pricing;
17      }
18
19      return 404; # Catch-all
20  }
```

warehouse_api_apikeys.conf hosted with ♡ by GitHub view raw

With the `auth_request` directive (line 7) we can, for example, have authentication handled by an external authentication server such as OAuth 2.0 token introspection. In this example we instead add the logic for validating API keys to the top-level API gateway configuration file, in the form of the following `location` block called **/_validate_apikey**.

```
28      # API key validation
29      location = /_validate_apikey {
30          internal;
31
32          if ($http_apikey = "") {
33              return 401; # Unauthorized
34          }
35          if ($api_client_name = "") {
36              return 403; # Forbidden
37          }
38
39          return 204; # OK (no content)
40      }
```

api_gateway_apikey.conf hosted with ♡ by GitHub view raw

The `internal` directive on line 30 means that this location cannot be accessed directly by external clients (only by `auth_request`). Clients are expected to present their API key in the `apikey` HTTP header. If this header is missing or empty (line 32), we send a `401 (Unauthorized)` response to tell the client that authentication is required. Line 35 handles the case where the API key does not match any of the keys in the `map` block – in which case the `default` parameter on line 2 of **api_keys.conf** sets `$api_client_name` to an empty string – and we send a `403 (Forbidden)` response to tell the client that authentication failed. If neither of those conditions match, the API key is valid and the location returns a `204 (No Content)` response.

With this configuration in place, the Warehouse API now implements API key authentication.

```
{"status":401,"message":"Unauthorized"}
$ curl -H "apikey: thisIsInvalid"
https://api.example.com/api/warehouse/pricing/item001
{"status":403,"message":"Forbidden"}
$ curl -H "apikey: 7B5zIqmRGXmrJTFmKa99vcit"
https://api.example.com/api/warehouse/pricing/item001
{"sku":"item001","price":179.99}
```

JWT Authentication

JSON Web Tokens (JWTs) are increasingly used for API authentication. Native JWT support is exclusive to NGINX Plus, enabling validation of JWTs as described in Authenticating API Clients with JWT and NGINX Plus on our blog. For a sample implementation, see Controlling Access to Specific Methods in Part 2.

Summary

This first blog in a series details a complete solution for deploying NGINX Open Source and NGINX Plus as an API gateway. The complete set of files discussed in this blog can be reviewed and downloaded from our GitHub Gist repo.

Check out the other posts in this series:

- Part 2 explores more advanced use cases for protecting backend services from malicious or badly behaved clients.
- Part 3 explains how to deploy NGINX as an API gateway for gRPC services.

To try NGINX Plus, start your free 30-day trial today or contact us to discuss your use cases.

smanne • 3 years ago

Is there any article which tells how to compose multiple micro services into 1 api. Something like this product page example https://www.nginx.com/blog/...

4 ^ | v • Reply • Share ›

Adam Patterson ➚ smanne • 3 years ago

Can you use a proxy_pass under a location?

https://gist.github.com/ada...

^ | v • Reply • Share ›

Faisal Memon ➚ Adam Patterson • 3 years ago

yes, proxy_pass is supported inside location blocks.

4 ^ | v • Reply • Share ›

vrno ➚ Faisal Memon • 3 years ago

How would you configure proxying a socket that is created by a backend websocket server after the client binds to an API like ws://ahostname:3000/bind, which is proxied by nginx?

^ | v • Reply • Share ›

Faisal Memon ➚ vrno • 3 years ago

See this blog for how to use NGINX to proxy web socket: https://www.nginx.com/blog/...

^ | v • Reply • Share ›

vrno ➚ Faisal Memon • 3 years ago

Should nginx be forward or reverse proxy?
Also, I do not see anything in that example related to ws://aNginxServer:3000/bind
I can proxy this because it is client to nginx, and nginx to upstream. But when upstream creates a socket, how will that socket be proxied back from upstream to nginx to Client?

http {

map $http_upgrade $connection_upgrade {

default upgrade;

'' close;

}

upstream websocket {

see more

^ | v • Reply • Share ›

Faisal Memon ➚ vrno • 3 years ago

That is for nginx as a reverse proxy. Your server creates a web socket to the client? That is a little more challenging to implement unless the ip or hostname of the client is fixed.

• 3 years ago

Yes, the new socket is created on the upstream and I would like the communication over this socket to also go through nginx. Maybe SSE between nginx and clients and webhook between nginx and the upstream? Or a tunnel between nginx and the upstream. Clients are in the public zone and no fix addresses

Owen Garrett Mod → SuperDeterrent
• 3 years ago

Hi **@SuperDeterrent** - what you're seeking is generally not possible in a client-server architecture. Server endpoints (e.g. the NGINX load balancer) have publicly accessible IPs and open ports, whereas clients may be deployed behind NAT or firewalls and are unable to receive incoming connections.

Unless you have full control over every client, you should build a client-server architecture that is based on clients initiating connections to the server only. Once a client->server websocket connection is established, bidirectional communication is straightforward.

If you're building a peer-to-peer architecture, look at how protocols such as Skype or bittorrent make connections between peers, through NAT and firewalls.

SuperDeterrent → Owen Garrett
• 3 years ago

Hi Owen,
yes, I do have full control over behavior of each client. I guess then that 3-tier architecture will do in my case.

A question re http session: When nginx forwards the request to Upstream as described, is this forward going to take place under a brand new http session or the preexisting http user session will be used (e.g. a session that was created on nginx prior to this forwarding to Upstream)? Say, the browser was redirected to a web application by nginx in the prior step. And in the next step, the browser now calls an API on nginx server, but this time nginx forwards the request to Upstream. I hope all these forwards to different servers will happen under the same http user's session created for the user at the keyboard?

Owen Garrett Mod → SuperDeterrent
• 3 years ago

HTTP is stateless; there's no such thing as an HTTP session.

Applications that run on top of HTTP may be stateful, and require that HTTP requests in the same session (application session) are routed to the same application upstream. This process is known as "session persistence".

based on something that reliably identifies the client. The remoteip is commonly used to identify a client.
* NGINX Plus session persistence methods... NGINX Plus can learn the session identifier, automatically inject session identifiers (cookies), or use static routes for certain identifiers
* third-party session persistence modules; we've not looked at or tested any of these, but they are available

• Reply • Share ›

SuperDeterrent ➚ Owen Garrett
• 3 years ago

Maybe I didn't describe my use case in sufficient details. I have nginx and two unrelated Upstreams e.g. Left and Right. nginx reverse proxies both. The Browser's app session begins when he visits Left Upstream(s) (via nginx).

HTML from Left is being sent back to Browser, but first reaches nginx, at which point nginx enriches this HTML by adding <script> </script> tags at the end of HTML before returning enriched HTML to Browser (in that regard, I could not understand what /after_action meant in the add filter documentation; can you shed some light on that as well i.e. how to configure nginx add filter and /after_action to achieve this?).

see more

• Reply • Share ›

Owen Garrett Mod ➚ SuperDeterrent
• 3 years ago

Hi **@SuperDeterrent** - thanks for the explanation, it helps a lot.

With respect to enriching the HTML from the LEFT servers, you could look at the sub_filter module to do this. Put a placeholder in the original HTML, and then search-for-and-replace that placeholder text. That may be a simpler alternative to the http_addition_module.

For your session persistence requirements, if I understand correctly, you only need SP for requests to the RIGHT servers. For requests to the LEFT servers, you don't care which server each request is routed to. That should be straightforward - in the configuration where you location-match requests for the RIGHT servers, you can use one of the session persistence methods I described above.

2 • Reply • Share ›

SuperDeterrent ➚ Owen Garrett
• 3 years ago • edited

Hi Owen,

I do care about both LEFT servers and the RIGHT server. I, as a web user, can open 10 browser tabs, one tab for each of 10 different LEFT web servers. During each of these 10 browser tabs LEFT renderings, an interaction will also occur between the particular browser

browser tabs with the RIGHT server occur as one single application session (and not as 10 different sessions). I am not sure, but I think that nginx will have 10 sessions established for 10 LEFT servers as it makes all those web requests itself. RIGHT server needs to have a single queue for the user i.e. URL from which

see more

• Reply • Share ›

SuperDeterrent ➚ Owen Garrett
• 3 years ago

What I meant was web session (as a synonym for http session):
https://techterms.com/defin...

"A common type of client/server session is a Web or HTTP session. An HTTP session is initiated by a Web browser each time you visit a website.
While each page visit constitutes an individual session, the term is
often used to describe the entire time you spend on the website. For
example, when you purchase an item on an ecommerce site, the entire process may be described as a session, even though you navigated through several different pages."

• Reply • Share ›

Adam Raszkiewicz • a year ago

I was following all steps above but when trying to reach correct URL it throws 400 Bad Request. Similar issue was described here. Any clue? Thanks

1 • Reply • Share ›

Junior Alves • 2 years ago

This article saved my life.

• Reply • Share ›

Adam Raszkiewicz ➚ Junior Alves • a year ago

Hey Junior,

I'm trying to follow steps from that blog post but what I'm getting is 400 Bad Request for correct path and backend. Similar issue is described here:
https://stackoverflow.com/q... Any thought on that?

• Reply • Share ›

SuperDeterrent • 3 years ago

Does it make sense to use it only as a reverse proxy, web server, and load balancer for a single LOB service if all LOBs services are already protected by the main API Gateway?

• Reply • Share ›

Owen Garrett Mod ➚ SuperDeterrent • 3 years ago

Hi **@SuperDeterrent** - yes, it does make sense in that case.

Let's begin with the situation that there is already an API gateway in place, and assume that it's not appropriate to replace or update that gateway. In this situation, I see two common reasons why people would also deploy NGINX - one is functional, the other organizational.

Functional: NGINX forms the core of many commercial and open source API gateways; it also forms the core of many CDNs (for caching) and is the most common software detected in high-traffic websites (for application delivery). An API gateway alone, even one using NGINX, will not provide

translation and load balancing for both web and API traffic.

Organizational: In larger organizations, application teams

see more

• Reply • Share ›

SuperDeterrent ➚ Owen Garrett • 3 years ago

Thanks, this all makes sense. If a LOB has many subsystems e.g. each subsystem on a different port, then if everything must first go through nginx, CORS issues are resolved bona fide, which simplifies development etc.
Just to clarify a possible complexity and make sure it is feasible with nginx:

let's assume LOB has two subsystems, one at 5000 port and the other at 5001 port.

nginx is installed on the first subsystem/server i.e. fronting 5000 subsystem. Any Ajax call from the client/browser to the 5001 subsystem (with REST API endpoints) will be proxied through nginx.

My question is how would nginx handle HTTP websockets (e.g. socket.io sockets over http transport) if 5001 subsystem creates a socket for

see more

• Reply • Share ›

Owen Garrett Mod ➚ SuperDeterrent • 3 years ago

Hi **@SuperDeterrent** - yes, NGINX can handle websocket connections without difficulty [ref1, ref2]. The client upgrades their HTTP connection to a websocket; the server is not able to initiate the upgrade itself.

The client could also bypass NGINX by connecting directly to the upstream server. This is a little more complex to implement securely; the upstreams need publicly-routable addresses, and the client needs to discover the public address of the server it

NGINX Amplify

NGINX on Github

NGINX Open Source

NGINX Unit

NGINX Amplify

NGINX Kubernetes Ingress Controller

NGINX Microservices Reference Architecture

NGINX Crossplane

Connect With Us

STAY IN THE LOOP

FAQ

Learn

Glossary

Support

Professional Services

Training

Customer Portal Login

Certified Module Program

Company

About F5 NGINX

Careers

Leadership

Press

Events

F5

Shape Security

Volterra