NGINX
Part of F5

EBOOK

Deploying NGINX as an API Gateway

By Liam Crilly, F5, Inc.

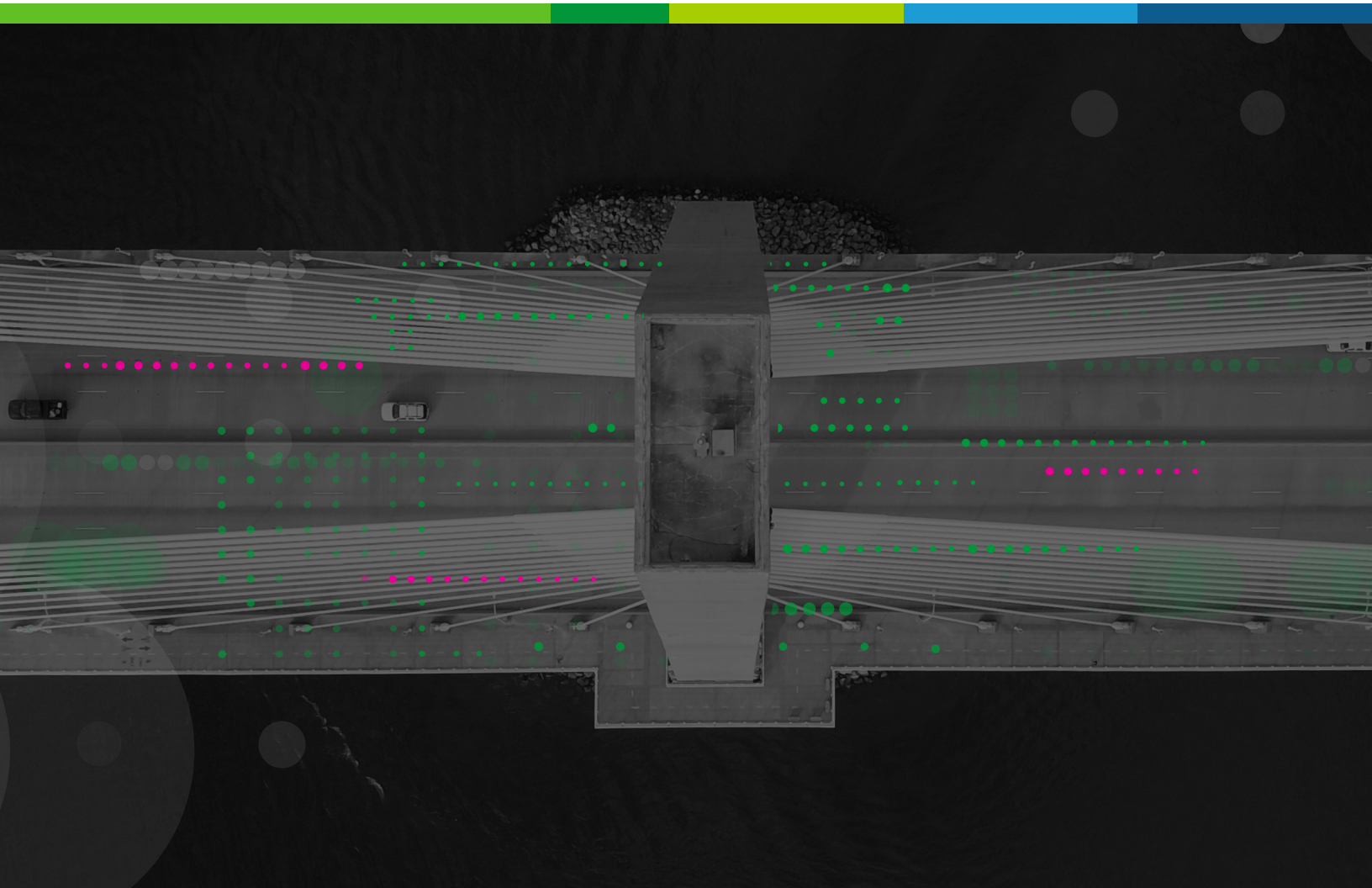

Table of Contents

Foreword

Application development and delivery is mission-critical for most organizations today, especially larger existing businesses and innovative startups in all kinds of fields.

As various challenges have arisen in application development and delivery, different kinds of solutions have come to prominence, engineered to fit a specific problem. Examples of these point solutions are hardware load balancers – also called application delivery controllers, or ADCs (such as F5 BIG-IP and Citrix ADC); content distribution networks (CDNs), which include Akamai; and API management tools such as Apigee.

A LIGHTER-WEIGHT AND MORE FLEXIBLE APPROACH HAS DEVELOPED, WITH NGINX AT THE FOREFRONT

Alongside this profusion of point solutions, a lighter-weight and more flexible approach has developed, with NGINX at the forefront. As inexpensive, generic server hardware grows in capability, organizations use simple, flexible software to manage tasks easily, from a single point of contact.

So today, many organizations use NGINX load-balancing features to complement existing hardware ADCs, or even to replace them completely. They use NGINX for multiple levels of caching, or even develop their own CDN to complement or replace commercial CDN usage. (Most commercial CDNs have NGINX at their core.)

The API gateway use case is now beginning to yield to the inexorable combination of increasing generic hardware power and NGINX's steadily growing feature set. As with CDNs, many existing API management tools are built on NGINX.

In this eBook, we tell you how to take an existing NGINX Open Source or F5 NGINX Plus configuration and extend it to manage API traffic as well. When you use NGINX for API management, you tap into the high performance, reliability, robust community support, and expert professional support (for NGINX Plus customers) that NGINX is famous for.

With that in mind, we recommend using the techniques described here to access NGINX capabilities wherever possible. Then use complementary solutions for any capabilities that they uniquely offer.

In this foreword, we take a look at how NGINX fits as a potential complement or replacement for existing API gateway and API management approaches. We then introduce the rest of the eBook, which shows you how to implement many important API gateway capabilities with NGINX.

HOW IS NGINX CURRENTLY USED AS AN API GATEWAY?

Today, NGINX is deployed as an API gateway in three different manners:

- **Native NGINX functionality** – Organizations frequently use the capabilities of NGINX to manage API traffic directly. They recognize that API traffic is HTTP or gRPC, and they translate their API management requirements into NGINX configuration to receive, route, rate-limit, and secure API requests.
- **Extending NGINX using Lua** – The NGINX-based OpenResty software adds a Lua interpreter to the NGINX core, enabling users to build rich capabilities on top of NGINX. The Lua modules can also be compiled and loaded into the vanilla NGINX Open Source and NGINX Plus builds. There are dozens of open source NGINX-based API gateway implementations on GitHub, many using Lua and OpenResty.
- **Third-party stand-alone API gateways** – Stand-alone API gateways are single-purpose products that focus on API traffic alone. The majority of stand-alone API gateway products use NGINX and Lua at their core, whether they are dedicated open source solutions or commercial products.

THE MAJORITY OF STAND-ALONE API GATEWAY PRODUCTS USE NGINX AND LUA AT THEIR CORE

Given these options, we have two recommendations:

- **Be mindful of performance and request latency** – Lua is a good way to extend NGINX, but it can compromise NGINX's performance. Our own tests for simple Lua scripts show decreases in performance ranging from 50% to 90%. If your chosen solution relies heavily on Lua, make sure to verify that it meets your peak performance requirements without adding latency to requests.
- **Take a converged approach** – NGINX can manage API traffic right alongside regular web traffic. An API gateway is a subset of the functionality of NGINX. (In some cases, API gateway offerings include Lua code that duplicates functionality found in NGINX, but with potentially worse reliability and performance.) Selecting a stand-alone, single-purpose API gateway limits your architectural flexibility, because using separate products to manage web and API traffic increases the complexity you experience in DevOps, CI/CD, monitoring, security, and other application development and delivery functions.

A converged approach is particularly important in the face of modern, distributed applications. Remember that NGINX does not just operate as a reverse proxy for web and API traffic; it also operates as a cache, an authentication gateway, a web application firewall, and an application gateway.

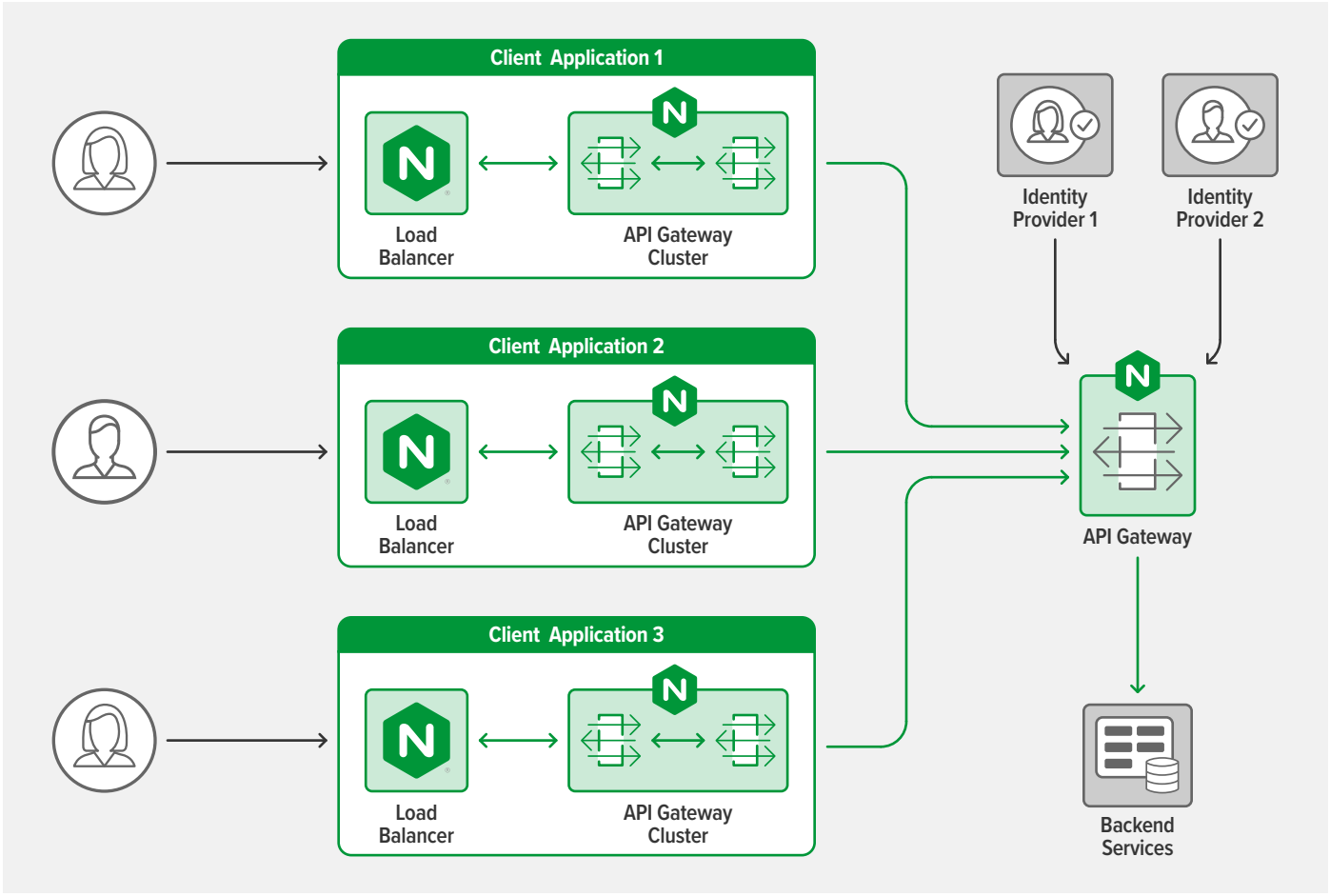

Figure 1: Sample Application Architecture using NGINX as an API Gateway

WHY IS NGINX IDEALLY SUITED TO BE THE ENABLER OF AN API GATEWAY?

The following table examines API gateway use cases for managing API requests from external sources and routing them to internal services.

	API GATEWAY USE CASE	NGINX REVERSE PROXY
Core Protocols	REST (HTTPS), gRPC	HTTP, HTTPS, HTTP/2, gRPC
Additional Protocols	TCP-borne message queue	WebSocket, TCP, UDP
Routing Requests	Requests are routed based on service (host header), API method (HTTP URL), and parameters	Very flexible request routing based on host header, URLs, and request headers
Managing Lifecycle of APIs	Rewriting legacy API requests, rejecting calls to deprecated APIs	Comprehensive request rewriting and rich decision engine to route or respond directly to requests
Protecting Vulnerable Applications	Rate limiting by APIs and methods	Rate limiting by multiple criteria, including source address and request parameters; connection limiting to backend services
Offloading Authentication	Interrogating authentication tokens in incoming requests	Multiple authentication methods, including JWT, API keys, OpenID Connect, and other external auth services
Managing Changing Application Topology	Various APIs to accept configuration changes and support blue-green workflows	APIs and service discovery to locate endpoints (NGINX Plus); orchestration of APIs for blue-green deployment and other use cases

Being a converged solution, NGINX can also manage web traffic with ease, translating between protocols (HTTP/2 and HTTP, FastCGI, uwsgi) and providing consistent configuration and monitoring interfaces. NGINX is sufficiently lightweight to be deployed in container environments or as a sidecar, with minimal resource footprint.

FURTHER ENHANCING NGINX SOLUTIONS

NGINX was originally developed as a gateway for web (HTTP) traffic, and the primitives by which it is configured are expressed in terms of HTTP requests. These are similar, but not identical, to the way you expect to configure an API gateway, so a DevOps engineer needs to understand how to map API definitions to HTTP requests.

For simple APIs, this is straightforward to do. For more complex situations, the three chapters in this book describe how to map a complex API to a service in NGINX, and then how to perform common tasks, such as:

- Rewriting API requests
- Correctly responding to errors
- Managing API keys for access control
- Interrogating JWT tokens for user authentication
- Rate limiting
- Enforcing specific request methods
- Applying fine-grained access control
- Controlling request sizes
- Validating request bodies

F5 NGINX CONTROLLER AND API GATEWAY FUNCTIONALITY

F5 NGINX Controller is the control plane for the NGINX Application Platform, which builds on the multi-functional nature of NGINX and NGINX Plus to deliver powerful application delivery capabilities in a simple and consistent way.

NGINX Controller has the capability to define API management policies in an API-friendly way, pushing them out to multi-function NGINX gateways deployed across your application platform. For more information on NGINX Controller and to get a free trial, visit our website.

1. Getting Started with NGINX and Your API Gateway

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.

NGINX HAS THE ADVANCED HTTP PROCESSING CAPABILITIES NEEDED FOR HANDLING API TRAFFIC

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 chapter 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.

Note: Except as noted, all information in this chapter applies to both NGINX Open Source and F5 NGINX Plus.

INTRODUCING THE WAREHOUSE API

The primary function of an 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 chapter we refer to a hypothetical API for inventory management, the "Warehouse API". We use sample NGINX configurations 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.

The Warehouse API is implemented as a collection of discrete microservices and published as a single API. The inventory and pricing resources are implemented as separate services and deployed 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**.

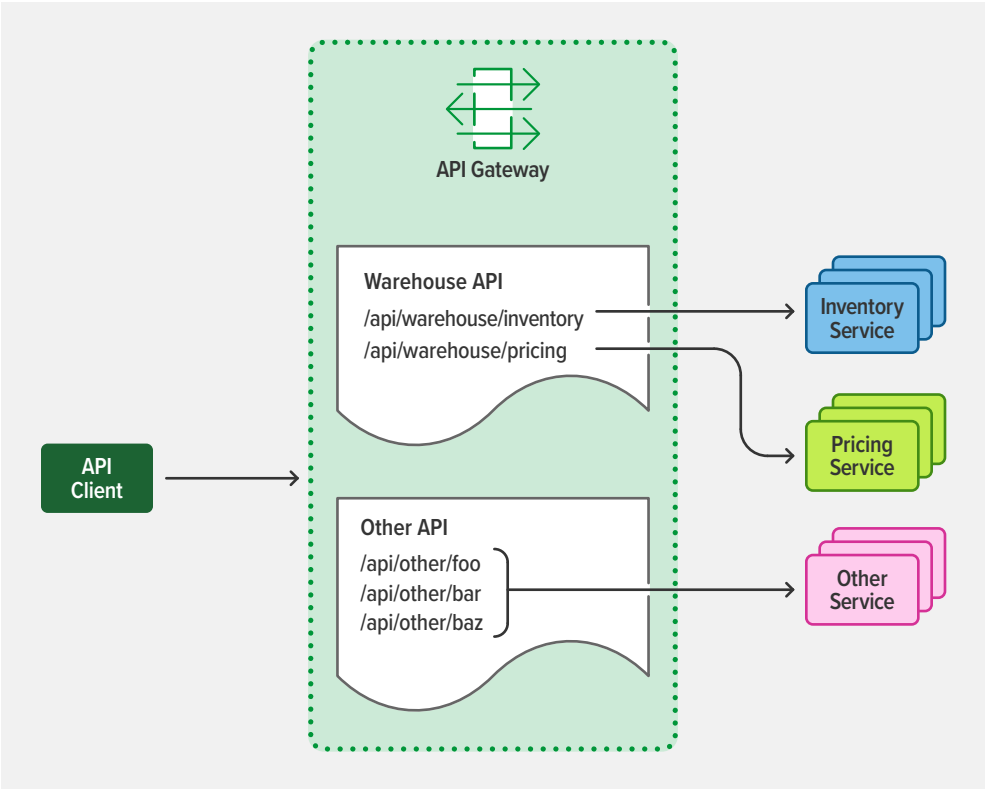

Figure 2: API Gateway Architecture for Multiple Applications

ORGANIZING THE NGINX CONFIGURATION

IF NGINX IS ALREADY PART OF YOUR APPLICATION DELIVERY STACK THEN IT IS GENERALLY UNNECESSARY TO DEPLOY A SEPARATE API GATEWAY

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.

```
etc/
└── nginx/
    ├── api_conf.d/ ..... Subdirectory for per-API configuration
    │   └── warehouse_api.conf ..... Definition and policy of the Warehouse AP
    ├── 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 various configuration files we describe below for different ways to define the Warehouse API.

DEFINING THE TOP-LEVEL API GATEWAY

All NGINX configuration starts with the main configuration file, **nginx.conf**. To read in the API gateway configuration, in the `http` block in **nginx.conf** we add an `include` directive that references the file containing the gateway configuration, **api_gateway.conf** (line 28 in the configuration snippet 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 chapter 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
```

View raw on GitHub

The following file, **api_gateway.conf**, 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 8), protected by TLS as configured on lines 12 through 16. 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
6                                                        # to a 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
19     # API definitions, one per file
20     include api_conf.d/*.conf;
21
22     # Error responses
23     error_page 404 = @400;         # Treat invalid paths as bad requests
24     proxy_intercept_errors on;     # Do not send backend errors to client
25     include api_json_errors.conf;  # API client-friendly JSON errors
26     default_type application/json; # If no content-type, assume JSON
27 }
```

View raw on GitHub

SINGLE-SERVICE VS. MICROSERVICE API BACKENDS

Some APIs are implemented at a single backend, although we normally expect there to be more than one backend, 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.

```
upstream warehouse_inventory {
    zone inventory_service 64k;
    server 10.0.0.1:80;
    server 10.0.0.2:80;
    server 10.0.0.3:80;
}

upstream warehouse_pricing {
    zone pricing_service 64k;
    server 10.0.0.7:80;
    server 10.0.0.8:80;
    server 10.0.0.9:80;
}
```

View raw on GitHub

NGINX PLUS SUBSCRIBERS CAN ALSO TAKE ADVANTAGE OF DYNAMIC DNS LOAD BALANCING

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 {
15         proxy_pass http://warehouse_pricing;
16     }
17
18     return 404; # Catch-all
19 }
```

View raw on GitHub

NGINX HAS A HIGHLY EFFICIENT AND FLEXIBLE SYSTEM FOR MATCHING THE REQUEST URI

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 above 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 depends on each API's security requirements and whether it is desirable for the backend services to handle invalid URIs.

In the configuration in Defining the Warehouse API 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 }
```

View raw on GitHub

This configuration is more verbose, but more accurately describes the resources implemented by the backend services. This has the advantage of protecting the backend services from malformed client 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 NGINX API definitions from the OpenAPI Specification (formerly called Swagger). A sample script for this purpose is provided among the Gists for this chapter.

Rewriting Client Requests to Handle Breaking Changes

IT'S SOMETIMES NECESSARY TO MAKE CHANGES THAT BREAK STRICT BACKWARD COMPATIBILITY

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 Defining the Warehouse API 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
18     location /api/warehouse/pricing {
19         proxy_pass http://warehouse_pricing;
20     }
21
22     return 404; # Catch-all
23 }
```

View raw on GitHub

RESPONDING TO ERRORS

ONE OF THE KEY DIFFERENCES BETWEEN HTTP APIs AND BROWSER-BASED TRAFFIC IS HOW ERRORS ARE COMMUNICATED TO THE CLIENT

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;         # Treat invalid paths as bad requests
24     proxy_intercept_errors on      # Do not send backend errors to client
25     include api_json_errors.conf;  # API client-friendly JSON errors
26     default_type application/json   # If no content-type, assume JSON
```

View raw on GitHub

The `error_page` directive on line 23 above 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.

UNHANDLED EXCEPTIONS MAY CONTAIN STACK TRACES OR OTHER SENSITIVE DATA

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 above 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'; }
```

View raw on GitHub

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

```
$ curl -i https://api.example.com/foo
HTTP/1.1 400 Bad Request
Server: nginx/1.21.3
Content-Type: application/json
Content-Length: 39
Connection: keep-alive

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

IMPLEMENTING AUTHENTICATION

NGINX OFFERS SEVERAL APPROACHES FOR PROTECTING APIs AND AUTHENTICATING API CLIENTS

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

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** (shown in full in Defining the Top-Level API Gateway), we use an `include` directive to reference the following file, **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  }
```

View raw on GitHub

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.

Here we enable API key authentication by amending the "broad" configuration from Defining the Warehouse API to include an `auth_request` directive in the policy section that 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 }
```

View raw on GitHub

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     }
```

View raw on GitHub

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.

```
$ curl https://api.example.com/api/warehouse/pricing/item001
{"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 Chapter 2.

SUMMARY

This chapter details a complete solution for deploying NGINX as an API gateway. You can review and download the complete set of files discussed in this chapter from our GitHub Gist repo.

2. Protecting Backend Services

This chapter extends the basic use cases presented in Chapter 1 and looks at a range of safeguards that can be applied to protect and secure backend API services in production:

- Rate limiting
- Enforcing specific request methods
- Applying fine-grained access control
- Controlling request sizes
- Validating request bodies

Note: Except as noted, all information in this chapter applies to both NGINX Open Source and NGINX Plus.

RATE LIMITING

INDIVIDUAL API CLIENTS ARE ABLE TO PLACE HUGE LOADS ON YOUR APIs

Unlike browser-based clients, individual API clients are able to place huge loads on your APIs, even to the extent of consuming such a high proportion of system resources that other API clients are effectively locked out. Not only malicious clients pose this threat: a misbehaving or buggy API client might enter a loop that overwhelms the backend. To protect against this, we apply a rate limit to ensure fair use by each client and to protect the resources of the backend services.

NGINX can apply rate limits based on any attribute of the request. The client IP address is typically used, but when authentication is enabled for the API, the authenticated client ID is a more reliable and accurate attribute.

Rate limits themselves are defined in the top-level API gateway configuration file and can then be applied globally, on a per-API basis, or even per URI.

```
1  include api_backends.conf;
2  include api_keys.conf;
3
4  limit_req_zone $binary_remote_addr zone=client_ip_10rs:1m rate=1r/s;
5  limit_req_zone $http_apikey        zone=apikey_200rs:1m   rate=200r/s;
6
7  server {
8      access_log /var/log/nginx/api_access.log main;  # Each API may also
9                                                      # log to a separate file
10
11     listen 443 ssl;
12     server_name api.example.com;
13
14     # TLS config
15     ssl_certificate        /etc/ssl/certs/api.example.com.crt;
16     ssl_certificate_key    /etc/ssl/private/api.example.com.key;
17     ssl_session_cache      shared:SSL:10m;
18     ssl_session_timeout    5m;
19     ssl_ciphers            HIGH:!aNULL:!MD5;
20     ssl_protocols          TLSv1.2 TLSv1.3;
21
22     # API definitions, one per file
23     include api_conf.d/*.conf;
24
25     # Error responses
26     error_page 404 = @400;         # Treat invalid paths as bad requests
27     proxy_intercept_errors on;     # Do not send backend errors to client
28     include api_json_errors.conf;  # API client-friendly JSON errors
29     default_type application/json; # If no content-type, assume JSON
49 }
```

View raw on GitHub

In this example, the `limit_req_zone` directive on line 4 defines a rate limit of 10 requests per second for each client IP address (`$binary_remote_addr`), and the one on line 5 defines a limit of 200 requests per second for each authenticated client ID (`$http_apikey`). This illustrates how we can define multiple rate limits independently of where they are applied. An API may apply multiple rate limits at the same time, or apply different rate limits for different resources.

Then in the following configuration snippet we use the `limit_req` directive to apply the first rate limit in the policy section of the "Warehouse API" described in Chapter 1. By default, NGINX sends the `503 (Service Unavailable)` response when the rate limit has been exceeded. However, it is helpful for API clients to know explicitly that they have exceeded their rate limit, so that they can modify their behavior. To this end we use the `limit_req_status` directive to send the `429 (Too Many Requests)` response instead.

```
# Warehouse API
#
location /api/warehouse/ {
    # Policy configuration here (authentication, rate limiting, logging...)
    #
    access_log /var/log/nginx/warehouse_api.log main;
    limit_req zone=client_ip_10rs;
    limit_req_status 429;

    # URI routing
    #
    location /api/warehouse/inventory {
        limit_except GET {
            deny all;
        }
        error_page 403 = @405;   # Convert deny response from '403 (Forbidden)'
                                 # to '405 (Method Not Allowed)'
        proxy_pass http://warehouse_inventory;
    }

    location /api/warehouse/pricing {
        limit_except GET PATCH {
            deny all;
        }
        error_page 403 = @405;
        proxy_pass http://warehouse_pricing;
    }

    return 404;     # Catch-all
}
```

View raw on GitHub

You can use additional parameters to the `limit_req` directive to fine-tune how NGINX enforces rate limits. For example, it is possible to queue requests instead of rejecting them outright when the limit is exceeded, allowing time for the rate of requests to fall under the defined limit. For more information about fine-tuning rate limits, see Rate Limiting with NGINX and NGINX Plus on our blog.

ENFORCING SPECIFIC REQUEST METHODS

WITH RESTFUL APIs, THE HTTP METHOD IS AN IMPORTANT PART OF EACH API CALL

With RESTful APIs, the HTTP method (or verb) is an important part of each API call and very significant to the API definition. Take the pricing service of our Warehouse API as an example:

- `GET /api/warehouse/pricing/item001` returns the price of item001
- `PATCH /api/warehouse/pricing/item001` changes the price of item001

We can update the URI-routing definitions in the Warehouse API to accept only these two HTTP methods in requests to the pricing service (and only the GET method in requests to the inventory service).

```
10      # URI routing
11      #
12      location /api/warehouse/inventory {
13          limit_except GET {
14              deny all;
15          }
16          error_page 403 = @405;  # Convert deny response from '403 (Forbidden)'
17                                  # to '405 (Method Not Allowed)'
18          proxy_pass http://warehouse_inventory;
19      }
20
21      location /api/warehouse/pricing {
22          limit_except GET PATCH {
23              deny all;
24          }
28          error_page 403 = @405;  # Convert deny response from '403 (Forbidden)'
29                                  # to '405 (Method Not Allowed)'
30          proxy_pass http://warehouse_pricing;
31      }
```

View raw on GitHub

With this configuration in place, requests to the pricing service using methods other than those listed on line 22 (and to the inventory service other than the one on line 13) are rejected and are not passed to the backend services. NGINX sends the `405 (Method Not Allowed)` response to inform the API client of the precise nature of the error, as shown in the following console trace. Where a minimum-disclosure security policy is required, the `error_page` directive can be used to convert this response into a less informative error instead, for example `400 (Bad Request)`.

```
$ curl https://api.example.com/api/warehouse/pricing/item001
{"sku":"item001","price":179.99}
$ curl -X DELETE https://api.example.com/api/warehouse/pricing/item001
{"status":405,"message":"Method not allowed"}
```

APPLYING FINE-GRAINED ACCESS CONTROL

Chapter 1 describes how to protect APIs from unauthorized access by enabling authentication options such as API keys and JSON Web Tokens (JWTs). We can use the authenticated ID, or attributes of the authenticated ID, to perform fine-grained access control.

Here we show two such examples.

- The first example controls access to a specific API resource, extending a configuration presented in Chapter 1 and using API key authentication to verify that a given API client is on the allowlist.
- The second example controls which HTTP methods a client is allowed to use. It implements the JWT authentication method mentioned in Chapter 1, using a custom claim to identify qualified API clients. (Note that JWT support is exclusive to NGINX Plus.)

Of course, other authentication methods are applicable to these sample use cases, such as HTTP Basic authentication and OAuth 2.0 token introspection.

Controlling Access to Specific Resources

Let's say we want to allow only "infrastructure clients" to access the **audit** resource of the Warehouse API inventory service. With API key authentication enabled, we use a `map` block to create an allowlist of infrastructure client names so that the variable `$is_infrastructure` evaluates to 1 when a corresponding API key is used.

```
11  map $api_client_name $is_infrastructure {
12     default         0;
13
14     "client_one"    1;
15     "client_six"    1;
16  }
```

View raw on GitHub

In the definition of the Warehouse API, we add a `location` block for the inventory **audit** resource (lines 15–20). The `if` block ensures that only infrastructure clients can access the resource.

```
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/inventory/audit {
16         if ($is_infrastructure = 0) {
17             return 403;     # Forbidden (not infrastructure)
18         }
19         proxy_pass http://warehouse_inventory;
20     }
21
22     location /api/warehouse/pricing {
23         proxy_pass http://warehouse_pricing;
24     }
25
26     return 404;             # Catch-all
27 }
```

View raw on GitHub

Note that the location directive on line 15 uses the = (equals sign) modifier to make an exact match on the **audit** resource. Exact matches take precedence over the default path-prefix definitions used for the other resources. The following trace shows how with this configuration in place a client that isn't on the allowlist is unable to access the inventory **audit** resource. The API key shown belongs to **client_two** (as defined in Chapter 1).

```
$ curl -H "apikey: QzVV6y1EmQFbbxOfRCwyJs35"
https://api.example.com/api/warehouse/inventory/audit
{"status":403,"message":"Forbidden"}
```

Controlling Access to Specific Methods

WITH JWT AUTHENTICATION, PERMISSIONS FOR EACH CLIENT ARE ENCODED AS CUSTOM CLAIMS

As defined above, the pricing service accepts the GET and PATCH methods, which respectively enable clients to obtain and modify the price of a specific item. (We could also choose to allow the POST and DELETE methods, to provide full lifecycle management of pricing data.) In this section, we expand that use case to control which methods specific users can use. With JWT authentication enabled for the Warehouse API, the permissions for each client are encoded as custom claims. The JWTs issued to administrators who are authorized to make changes to pricing data include the claim `"admin":true`. We now extend our access-control logic so that only administrators can make changes.

```
60  # Access to write operations is evaluated by JWT claim 'admin'
61  map $request_method $admin_permitted_method {
62      "GET"       true;
63      "HEAD"      true;
64      "OPTIONS"   true;
65      default     $jwt_claim_admin;
66  }
```

View raw on GitHub

This `map` block, added to the bottom of the configuration file discussed in Rate Limiting, takes the request method (`$request_method`) as input and produces a new variable, `$admin_permitted_method`. Read-only methods are always permitted (lines 62–64) but access to write operations depends on the value of the **admin** claim in the JWT (line 65). We now extend our Warehouse API configuration to ensure that only administrators can make pricing changes.

```
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_jwt "Warehouse API";
8      auth_jwt_key_file /etc/nginx/idp_jwks.json;
9
10     # URI routing
11     #
12     location /api/warehouse/inventory {
13         limit_except GET {
14             deny all;
15         }
16         error_page 403 = @405; # Convert deny response from '403 (Forbidden)'
17                                # to '405 (Method Not Allowed)'
18         proxy_pass http://warehouse_inventory;
19     }
20
21     location /api/warehouse/pricing {
22         limit_except GET PATCH {
23             deny all;
24         }
25         if ($admin_permitted_method != "true") {
26             return 403;
27         }
28         error_page 403 = @405; # Convert deny response from '403 (Forbidden)'
29                                # to '405 (Method Not Allowed)'
30         proxy_pass http://warehouse_pricing;
31     }
32
33     return 404; # Catch-all
34 }
```

View raw on GitHub

The Warehouse API requires all clients to present a valid JWT (line 7). We also check that write operations are permitted by evaluating the `$admin_permitted_method` variable (line 25). Note again that JWT authentication is exclusive to NGINX Plus.

CONTROLLING REQUEST SIZES

THE REQUEST BODY CAN POSE AN ATTACK VECTOR TO BACKEND API SERVICES

HTTP APIs commonly use the request body to contain instructions and data for the backend API service to process. This is true of XML/SOAP APIs as well as JSON/REST APIs. Consequently, the request body can pose an attack vector to the backend API services, which may be vulnerable to buffer overflow attacks when processing very large request bodies.

By default, NGINX rejects requests with bodies larger than 1 MB. This can be increased for APIs that specifically deal with large payloads such as image processing, but for most APIs we set a lower value.

```
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      client_max_body_size 16k;
```

View raw on GitHub

The `client_max_body_size` directive on line 7 limits the size of the request body. With this configuration in place, we can compare the behavior of the API gateway upon receiving two different PATCH requests to the pricing service. The first `curl` command sends a small piece of JSON data, whereas the second command attempts to send the contents of a large file (/**etc/services**).

```
$ curl -iX PATCH -d '{"price":199.99}'
https://api.example.com/api/warehouse/pricing/item001
HTTP/1.1 204 No Content
Server: nginx/1.21.3
Connection: keep-alive

$ curl -iX PATCH -d@/etc/services
https://api.example.com/api/warehouse/pricing/item001
HTTP/1.1 413 Request Entity Too Large
Server: nginx/1.21.3
Content-Type: application/json
Content-Length: 45
Connection: close

{"status":413,"message":"Payload too large"}
```

VALIDATING REQUEST BODIES

BACKEND API SERVICES CAN BE SUSCEPTIBLE TO BODIES THAT CONTAIN INVALID OR UNEXPECTED DATA

In addition to being vulnerable to buffer overflow attacks with large request bodies, backend API services can be susceptible to bodies that contain invalid or unexpected data. For applications that require correctly formatted JSON in the request body, we can use the NGINX JavaScript module to verify that JSON data is parsed without error before proxying it to the backend API service.

With the JavaScript module installed, we use the `js_import` directive to reference the file containing the JavaScript code for the function that validates JSON data.

```
51 js_import json_validation.js;
52 js_set $json_validated json_validation.parseRequestBody;
```

View raw on GitHub

The `js_set` directive defines a new variable, `$json_validated`, which is evaluated by calling the `parseRequestBody` function.

```
1  export default { parseRequestBody };
2
3  function parseRequestBody(r) {
4      try {
5          if (r.variables.request_body) {
6              JSON.parse(r.variables.request_body);
7          }
8          return r.variables.upstream;
9      } catch (e) {
10         r.error('JSON.parse exception');
11         return '127.0.0.1:10415'; // Address for error response
12     }
13 }
```

View raw on GitHub

The `parseRequestBody` function attempts to parse the request body using the `JSON.parse` method (line 6). If parsing succeeds, the name of the intended upstream group for this request is returned (line 8). If the request body cannot be parsed (causing an exception), a local server address is returned (line 11). The `return` directive populates the `$json_validated` variable so that we can use it to determine where to send the request

```
8       # URI routing
9       #
10      location /api/warehouse/inventory {
11          proxy_pass http://warehouse_inventory;
12      }
13
14      location /api/warehouse/pricing {
15          set $upstream warehouse_pricing;
16          mirror /_get_request_body;          # Force early read
17          client_body_in_single_buffer on;    # Minimize memory copy operations
18                                              # on request body
19          client_body_buffer_size      16k;   # Largest body to keep in memory
20                                              # (before writing to file)
21          client_max_body_size         16k;
22          proxy_pass http://$json_validated$request_uri;
23      }
```

View raw on GitHub

In the URI routing section of the Warehouse API, we modify the `proxy_pass` directive on line 22. It passes the request to the backend API service as in the Warehouse API configurations discussed in previous sections (see, for example, lines 18 and 30 in the **warehouse_api_methods.conf** file in Controlling Access to Specific Methods), but now uses the `$json_validated` variable as the destination address. If the client body was successfully parsed as JSON then we proxy to the upstream group defined on line 15. If, however, there was an exception, we use the returned value of `127.0.0.1:10415` to send an error response to the client.

```
54  server {
55      listen 127.0.0.1:10415;
56      return 415; # Unsupported media type
57      include api_json_errors.conf;
58  }
```

View raw on GitHub

When requests are proxied to this virtual server, NGINX sends the `415 (Unsupported Media Type)` response to the client.

With this complete configuration in place, NGINX proxies requests to the backend API service only if they have correctly formatted JSON bodies.

```
$ curl -iX POST -d '{"sku":"item002","price":85.00}'
https://api.example.com/api/warehouse/pricing
HTTP/1.1 201 Created
Server: nginx/1.21.3
Location: /api/warehouse/pricing/item002

$ curl -X POST -d 'item002=85.00' https://api.example.com/api/
warehouse/pricing
{"status":415,"message":"Unsupported media type"}
```

A Note about the $request_body Variable

The JavaScript function parseRequestBody uses the `$request_body` variable to perform JSON parsing. However, NGINX does not populate this variable by default, and simply streams the request body to the backend without making intermediate copies. By using the `mirror` directive inside the URI routing section (line 16) we create a copy of the client request, and consequently populate the `$request_body` variable.

```
14      location /api/warehouse/pricing {
15          set $upstream warehouse_pricing;
16          mirror /_get_request_body;          # Force early read
17          client_body_in_single_buffer on;    # Minimize memory copy operations
18                                              # on request body
19          client_body_buffer_size     16k;    # Largest body to keep in memory
20                                              # (before writing to file)
21          client_max_body_size        16k;
22          proxy_pass http://$json_validated$request_uri;
23      }
```

View raw on GitHub

The directives on lines 17 and 19 control how NGINX handles the request body internally. We set `client_body_buffer_size` to the same size as `client_max_body_size` so that the request body is not written to disk. This improves overall performance by minimizing disk I/O operations, but at the expense of additional memory utilization. For most API gateway use cases with small request bodies this is a good compromise.

As mentioned, the `mirror` directive creates a copy of the client request. Other than populating `$request_body`, we have no need for this copy so we send it to a "dead end" location (`@get_request_body`) that we define in the `server` block in the top-level API gateway configuration.

```
45      # Dummy location used to populate $request_body for JSON validation
46      location /_get_request_body {
47          return 204;
48      }
```

View raw on GitHub

This location does nothing more than send the `204 (No Content)` response. Because this response is related to a mirrored request, it is ignored and so adds negligible overhead to the processing of the original client request.

SUMMARY

In this chapter, we focused on the challenge of protecting backend API services in a production environment from malicious and misbehaving clients. NGINX uses the same technology for managing API traffic that is used to power and protect the busiest sites on the Internet today.

You can access the complete set of configuration files used in this chapter at our GitHub Gist repo.

3. Publishing gRPC Services

This chapter explains how to deploy NGINX as an API gateway for gRPC services.

Note: Except as noted, all information in this chapter applies to both NGINX Open Source and NGINX Plus.

The concepts and benefits of microservices application architectures have been well documented in recent years, and nowhere more so than on the NGINX blog. At the heart of microservices applications is the HTTP API, and Chapters 1 and 2 use a hypothetical REST API to illustrate how NGINX addresses this style of application.

Despite the popularity of REST APIs with JSON message formats for modern applications, it is not an ideal approach for every scenario, or every organization. The most common challenges are:

- **Documentation standards** – Without good developer discipline or mandated documentation requirements, it is all too easy to end up with a mishmash of REST APIs that lack an accurate definition. The Open API Specification has emerged as a generic interface description language for REST APIs, but its use is optional and requires strong governance within the development organization.
- **Events and long-lived connections** – REST APIs, and their use of HTTP as the transport, largely dictate a request-response pattern for all API calls. When the application requires server-generated events, using solutions such as HTTP long polling and WebSocket can help, but the use of such solutions ultimately requires building a separate, adjacent API.
- **Complex transactions** – REST APIs are built around the concept of unique resources, each represented by a URI. When an application event calls for multiple resources to be updated then either multiple API calls are required, which is inefficient, or a complex transaction must be implemented at the backend, which contradicts the core principle of REST.

IN RECENT YEARS, gRPC HAS EMERGED AS AN ALTERNATIVE APPROACH TO BUILDING DISTRIBUTED APPLICATIONS

In recent years, gRPC has emerged as an alternative approach to building distributed applications, and microservices applications in particular. Originally developed at Google, gRPC was open sourced in 2015, and is now a project of the Cloud Native Computing Foundation. Significantly, gRPC uses HTTP/2 as its transport mechanism, taking advantage of its binary data format and multiplexed streaming capabilities.

The primary benefits of gRPC are:

- Tightly coupled interface definition language (protocol buffers)
- Native support for streaming data (in both directions)
- Efficient binary data format
- Automated code generation for many programming languages, enabling a true polyglot development environment without introducing interoperability problems

DEFINING THE gRPC GATEWAY

Chapters 1 and 2 describe how multiple APIs can be delivered through a single entry point (for example, **https://api.example.com**). The default behavior and characteristics of gRPC traffic lead us to take the same approach when NGINX is deployed as a gRPC gateway. While it is possible to share both HTTP and gRPC traffic on the same hostname and port, there are several reasons it is preferable to separate them:

- API clients for REST and gRPC applications expect error responses in different formats
- The relevant fields for access logs vary between REST and gRPC
- Because gRPC never deals with legacy web browsers, it can have a more rigorous TLS policy

To achieve this separation, we put the configuration for our gRPC gateway in its own `server{}` block in the main gRPC configuration file, **grpc_gateway.conf**, located in the **/etc/nginx/conf.d** directory.

```
log_format grpc_json escape=json '{"timestamp": "$time_iso8601",'
            '"client":"$remote_addr","uri":"$uri","http-status":$status,'
            '"grpc-status":$grpc_status,"upstream":"$upstream_addr"'
            '"rx-bytes":$request_length,"tx-bytes":$bytes_sent}';

map $upstream_trailer_grpc_status $grpc_status {
    default $upstream_trailer_grpc_status; # grpc-status is usually a trailer
    ''      $sent_http_grpc_status; # Else use the header, whatever its source
}

server {
    listen 50051 http2; # In production, comment out to disable plaintext port
    listen 443   http2 ssl;
    server_name  grpc.example.com;
    access_log   /var/log/nginx/grpc_log.json grpc_json;

    # TLS config
    ssl_certificate         /etc/ssl/certs/grpc.example.com.crt;
    ssl_certificate_key     /etc/ssl/private/grpc.example.com.key;
    ssl_session_cache       shared:SSL:10m;
    ssl_session_timeout     5m;
    ssl_ciphers             HIGH:!aNULL:!MD5;
    ssl_protocols           TLSv1.2 TLSv1.3;
}
```

View raw on GitHub

We start by defining the format of entries in the access log for gRPC traffic (lines 1–4). In this example, we use a JSON format to capture the most relevant data from each request. Note, for example, that the HTTP method is not included, as all gRPC requests use POST. We also log the gRPC status code along with the HTTP status code. However, the gRPC status code can be generated in different ways. Under normal conditions, `grpc-status` is returned as an HTTP/2 trailer from the backend, but for some error conditions it might be returned as an HTTP/2 header, either by the backend or by NGINX itself. To simplify the access log, we use a `map` block (lines 6–9) to evaluate a new variable `$grpc_status` and obtain the gRPC status from wherever it originates.

This configuration contains two `listen` directives (lines 12 and 13) so that we can test both plaintext (port 50051) and TLS-protected (port 443) traffic. The `http2` parameter configures NGINX to accept HTTP/2 connections – note that this is independent of the `ssl` parameter. Note also that port 50051 is the conventional plaintext port for gRPC, but is not suitable for use in production.

The TLS configuration is conventional, with the exception of the `ssl_protocols` directive (line 23), which specifies TLS 1.2 as the weakest acceptable protocol. The HTTP/2 specification mandates the use of TLS 1.2 (or higher), which guarantees that all clients support the Server Name Indication (SNI) extension to TLS. This means that the gRPC gateway can share port 443 with virtual servers defined in other `server{}` blocks.

RUNNING SAMPLE gRPC SERVICES

To explore the gRPC capabilities of NGINX, we're using a simple test environment that represents the key components of a gRPC gateway, with multiple gRPC services deployed. We use two sample applications from the official gRPC guides: helloworld (written in Go) and RouteGuide (written in Python). The RouteGuide application is especially useful because it includes each of the four gRPC service methods:

- Simple RPC (single request-response)
- Response-streaming RPC
- Request-streaming RPC
- Bidirectional-streaming RPC

Both gRPC services are installed as Docker containers on our NGINX host. For complete instructions on building the test environment, see the Appendix.

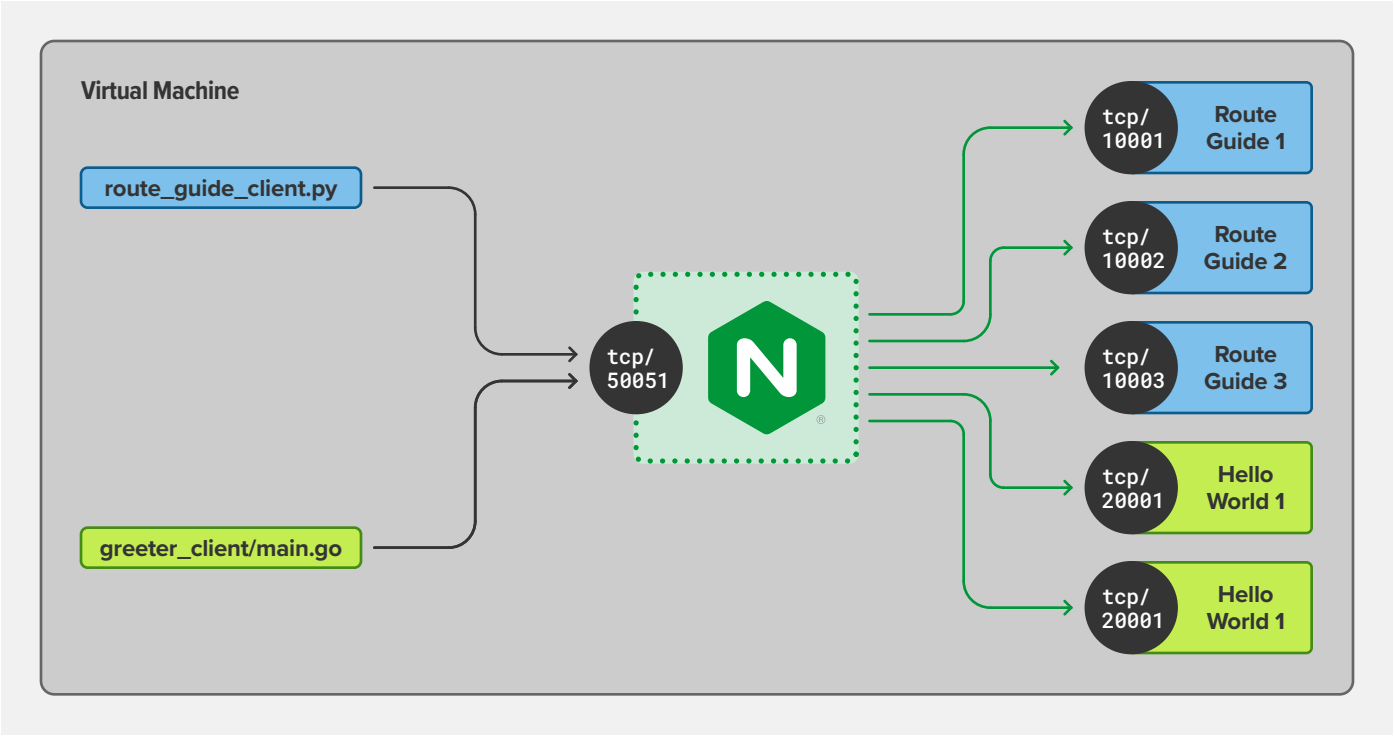

Figure 3: Test Environment for NGINX as a gRPC Gateway

We configure NGINX to know about the RouteGuide and helloworld services, along with the addresses of the available containers.

```
40  upstream routeguide_service {
41      zone routeguide_service 64k;
42      server 127.0.0.1:10001;
43      server 127.0.0.1:10002;
44      server 127.0.0.1:10003;
45  }
46
47  upstream helloworld_service {
48      zone helloworld_service 64k;
49      server 127.0.0.1:20001;
50      server 127.0.0.1:20002;
51  }
```

View raw on GitHub

We add an `upstream` block for each of the gRPC services (lines 40–45 and 47–51) and populate them with the addresses of the individual containers that are running the gRPC server code.

Routing gRPC Requests

With NGINX listening on the conventional plaintext port for gRPC (50051), we add routing information to the configuration, so that client requests reach the correct backend service. But first we need to understand how gRPC method calls are represented as HTTP/2 requests. The following diagram shows an abbreviated version of the **route_guide.proto** file for the RouteGuide service, illustrating how the package, service, and RPC method form the URI, as seen by NGINX.

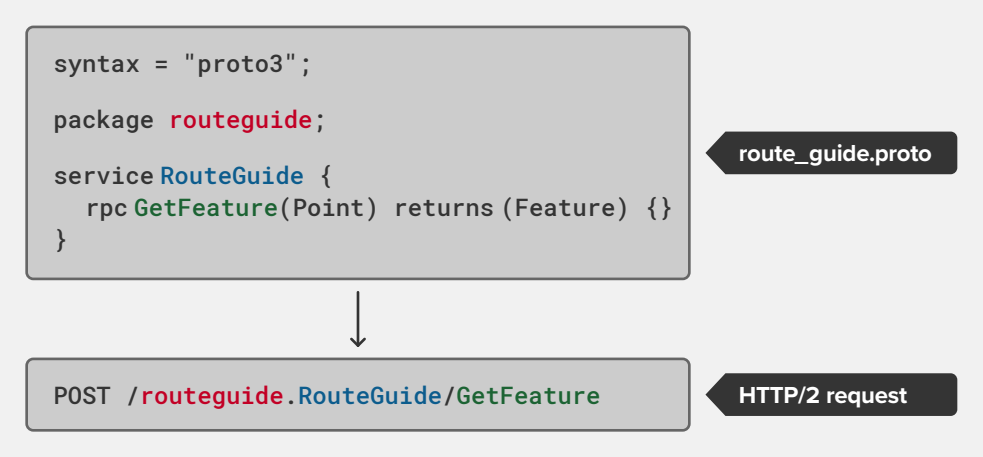

Figure 4: How Protocol Buffer RPC Methods Translate to HTTP/2 requests

The information carried in the HTTP/2 request can therefore be used for routing purposes by simply matching on the package name (here, `routeguide` or `helloworld`).

```
25      # Routing
26      location /routeguide. {
27          grpc_pass grpc://routeguide_service;
28      }
29      location /helloworld. {
30          grpc_pass grpc://helloworld_service;
31      }
```

View raw on GitHub

The first `location` block (line 26), without any modifiers, defines a prefix match such that `/routeguide.` matches all of the services and RPC methods defined in the corresponding **.proto** file for that package. The `grpc_pass` directive (line 27) therefore passes all requests from the RouteGuide client to the upstream group **routeguide_service**. This configuration (and the parallel one for the helloworld service on lines 29 and 30) provides a simple mapping between a gRPC package and its backend services.

Notice that the argument to the `grpc_pass` directives starts with the `grpc://` scheme, which proxies requests using a plaintext gRPC connection. If the backend is configured for TLS, we can use the `grpcs://` scheme to secure the gRPC connection with end-to-end encryption.

After running the RouteGuide client, we can confirm the routing behavior by reviewing the log file entries. Here we see that the RouteChat RPC method was routed to the container running on port 10002.

```
$ python route_guide_client.py
...
$ tail -1 /var/log/nginx/grpc_log.json | jq
{
     "timestamp": "2021-10-20T12:17:56+01:00",
     "client": "127.0.0.1",
     "uri": "/routeguide.RouteGuide/RouteChat",
     "http-status": 200,
     "grpc-status": 0,
     "upstream": "127.0.0.1:10002",
     "rx-bytes": 161,
     "tx-bytes": 212
}
```

Precise Routing

As shown above, the routing of multiple gRPC services to different backends is simple, efficient, and requires very few lines of configuration. However, the routing requirements in a production environment might be more complex and require routing based on other elements in the URI (the gRPC service or even individual RPC methods).

The following configuration snippet extends the previous example so that the bidirectional-streaming RPC method `RouteChat` is routed to one backend and all other `RouteGuide` methods to a different backend.

```
1  # Service-level routing
2  location /routeguide.RouteGuide/ {
3      grpc_pass grpc://routeguide_service_default;
4  }
5
6  # Method-level routing
7  location = /routeguide.RouteGuide/RouteChat {
8      grpc_pass grpc://routeguide_service_streaming;
9  }
```

View raw on GitHub

The second `location` directive (line 7) uses the = (equals sign) modifier to indicate that this is an exact match on the URI for the `RouteChat` RPC method. Exact matches are processed before prefix matches, which means that no other `location` blocks are considered for the `RouteChat` URI.

RESPONDING TO ERRORS

gRPC errors are somewhat different from those for conventional HTTP traffic. Clients expect error conditions to be expressed as gRPC responses, which makes the default set of NGINX error pages (in HTML format) unsuitable when NGINX is configured as a gRPC gateway. We address this by specifying a set of custom error responses for gRPC clients.

```
33      # Error responses
34      include conf.d/errors.grpc_conf;          # gRPC-compliant error responses
35      default_type application/grpc;            # Ensure gRPC for all error responses
```

View raw on GitHub

The full set of gRPC error responses is a relatively long and largely static configuration, so we keep them in a separate file, **errors.grpc_conf**, and use the `include` directive (line 34) to reference them. Unlike HTTP/REST clients, gRPC client applications are not expected to handle a wide range of HTTP status codes. The gRPC documentation specifies how an intermediate proxy such as NGINX must convert HTTP error codes into gRPC status codes so that clients always receive a suitable response. We use the `error_page` directive to perform this mapping.

```
1   # Standard HTTP-to-gRPC status code mappings
2   # Ref: https://github.com/grpc/grpc/blob/master/doc/http-grpc-status-mapping.md
3   #
4   error_page 400 = @grpc_internal;
5   error_page 401 = @grpc_unauthenticated;
6   error_page 403 = @grpc_permission_denied;
7   error_page 404 = w@grpc_unimplemented;
8   error_page 429 = @grpc_unavailable;
9   error_page 502 = @grpc_unavailable;
10  error_page 503 = @grpc_unavailable;
11  error_page 504 = @grpc_unavailable;#
```

View raw on GitHub

Each of the standard HTTP status codes are passed to a named location using the `@` prefix so a gRPC-compliant response can be generated. For example, the HTTP `404` response is internally redirected to the `@grpc_unimplemented` location, which is defined later in the file:

```
49 location @grpc_unimplemented {
50     add_header grpc-status 12;
51     add_header grpc-message unimplemented;
52     return 204;
53 }
```

View raw on GitHub

The `@grpc_unimplemented` named location is available only to internal NGINX processing – clients cannot request it directly, as no routable URI exists. Within this location, we construct a gRPC response by populating the mandatory gRPC headers and sending them, without a response body, using HTTP status code `204` (`No Content`).

We can use the `curl(1)` command to mimic a badly behaved gRPC client requesting a non-existent gRPC method. Note, however, that `curl` is not generally suitable as a gRPC test client because protocol buffers use a binary data format. To test gRPC on the command line, consider using `grpc_cli`.

```
$ curl -i --http2 -H "Content-Type: application/grpc" -H "TE: trailers" -X POST
https://grpc.example.com/does.Not/Exist
HTTP/2 204
server: nginx/1.21.3
date: Wed, 20 Oct 2021 15:03:41 GMT
grpc-status: 12
grpc-message: unimplemented
```

The **grpc_errors.conf** file referenced above also contains HTTP-to-gRPC status code mappings for other error responses that NGINX might generate, such as timeouts and client certificate errors.

AUTHENTICATING CLIENTS WITH gRPC METADATA

gRPC metadata allows clients to send additional information alongside RPC method calls, without requiring that data to be part of the protocol buffers specification (**.proto** file). Metadata is a simple list of key-value pairs, with each pair transmitted as a separate HTTP/2 header. Metadata is therefore easily accessible to NGINX.

Of the many use cases for metadata, client authentication is the most common for a gRPC API gateway. The following configuration snippet shows how NGINX Plus can use gRPC metadata to perform JWT authentication (JWT authentication is exclusive to NGINX Plus). In this example, the JWT is sent in the `auth-token` metadata.

```
1  location /routeguide. {
2      auth_jwt realm=routeguide token=$http_auth_token;
3      auth_jwt_key_file my_idp.jwk;
4      grpc_pass grps://routeguide_service;
5  }
```

View raw on GitHub

Every HTTP request header is available to NGINX Plus as a variable called $`http_`*header*. Hyphens (-) in the header name are converted to underscores (_) in the variable name, so the JWT is available as $`http_auth_token` (line 2).

If API keys are used for authentication, perhaps with existing HTTP/REST APIs, then these can also be carried in gRPC metadata and validated by NGINX. A configuration for API key authentication is provided in Chapter 1.

IMPLEMENTING HEALTH CHECKS

When load balancing traffic to multiple backends, it is important to avoid sending requests to backends that are down or otherwise unavailable. With NGINX Plus, we can use active health checks to proactively send out-of-band requests to backends and remove them from the load-balancing rotation when they don't respond to health checks as expected. In this way we ensure that client requests never reach backends that are out of service.

The following configuration snippet enables active health checks for the RouteGuide and helloworld gRPC services; to highlight the relevant configuration, it omits some directives that are included in the **grpc_gateway.conf** file used in previous sections.

```
11 server {
12     listen 50051 http2;    # Plaintext
13
14     # Routing
15     location /routeguide. {
16         grpc_pass grpc://routeguide_service;
17         health_check type=grpc grpc_status=12;      # 12=unimplemented
18     }
19     location /helloworld. {
20         grpc_pass grpc://helloworld_service;
21         health_check type=grpc grpc_status=12;      # 12=unimplemented
22     }
23 }
```

View raw on GitHub

For each route we now also specify the `health_check` directive (lines 17 and 21). As specified by the `type=grpc` argument, NGINX Plus uses the gRPC health checking protocol to send a health check to every server in the upstream group. However, our simple gRPC services do not implement the gRPC health checking protocol and so we expect them to respond with the status code that means "unimplemented" (`grpc_status=12`). When they do, that is enough to indicate that we are communicating with an active gRPC service.

With this configuration in place, we can take down any of the backend containers without gRPC clients experiencing delays or timeouts. Active health checks are exclusive to NGINX Plus; read more about gRPC health checks on our blog.

APPLYING RATE LIMITING AND OTHER API GATEWAY CONTROLS

The sample configuration in **grpc_gateway.conf** is suitable for production use, with some minor modifications for TLS. The ability to route gRPC requests based on package, service, or RPC method means that existing NGINX functionality can be applied to gRPC traffic in exactly the same way as for HTTP/REST APIs, or indeed as for regular web traffic. In each case, the relevant `location` block can be extended with further configuration, such as rate limiting or bandwidth control.

SUMMARY

This chapter focused on gRPC as a cloud-native technology for building microservices applications. We demonstrated how NGINX is able to deliver gRPC applications as effectively as it does HTTP/REST APIs, and how both styles of API can be published through NGINX as a multi-purpose API gateway.

Instructions for setting up the test environment used in this blog post are in the Appendix, and you can download all of the files from our GitHub Gist repo.

APPENDIX

Setting Up the gRPC Test Environment

The following instructions install the gRPC test environment used in Chapter 3 on a virtual machine so that it is isolated and repeatable. However, there is no reason why it can't be installed on a physical, "bare metal" server.

To simplify the test environment, we use Docker containers to run the gRPC services. This means that we don't need multiple hosts for the test environment, but can still have NGINX make proxied connections with a network call, as in a production environment.

Using Docker also allows us to run multiple instances of each gRPC service on a different port without requiring code changes. Each gRPC service listens on port 50051 within the container which is mapped to a unique localhost port on the virtual machine. This in turn frees up port 50051 so that NGINX can use it as its listen port. Therefore, when the test clients connect using their preconfigured port of 50051, they reach NGINX.

Installing NGINX Open Source or NGINX Plus

1. Install NGINX Open Source or NGINX Plus according to the instructions in the NGINX Plus Admin Guide.
2. Copy the following files from the GitHub Gist repo to **/etc/nginx/conf.d**:
   - **grpc_gateway.conf**
   - **errors.grpc_conf**

   Note: If not using TLS, comment out the `ssl_*` directives in **grpc_gateway.conf**.
3. Start NGINX Open Source or NGINX Plus.

```
$ sudo nginx
```

Installing Docker

For Debian and Ubuntu, run:

```
$ sudo apt-get install docker.io
```

For CentOS, RHEL, and Oracle Linux, run:

```
$ sudo yum install docker
```

Installing the RouteGuide Service Containers

1. Build the Docker image for the RouteGuide containers from the following Dockerfile.

```
# This Dockerfile runs the RouteGuide server from
# https://grpc.io/docs/tutorials/basic/python.html

FROM python
RUN pip install grpcio-tools
RUN git clone -b v1.14.x https://github.com/grpc/grpc
WORKDIR grpc/examples/python/route_guide

EXPOSE 50051
CMD ["python", "route_guide_server.py"]
```

View raw on GitHub

You can either copy the Dockerfile to a local subdirectory before the build, or specify the URL of the Gist for the Dockerfile as an argument to the `docker build` command:

```
$ sudo docker build -t routeguide
https://gist.githubusercontent.com/nginx-gists/
87ed942d4ee9f7e7ebb2ccf757ed90be/raw/ce090f92f3bbcb5a94bbf8de
d4d597cd47b43cbe/routeguide.Dockerfile
```

It may take a few minutes to download and build the image. The appearance of the message `Successfully built` and a hexadecimal string (the image ID) signal the completion of the build.

2. Confirm that the image was built by running `docker images`.

```
$ sudo docker images
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
routeguide    latest    63058a1cf8ca   1 minute ago    1.31 GB
python        latest    825141134528   9 days ago      923 MB
```

3. Start the RouteGuide containers.

```
$ sudo docker run --name rg1 -p 10001:50051 -d routeguide
$ sudo docker run --name rg2 -p 10002:50051 -d routeguide
$ sudo docker run --name rg3 -p 10003:50051 -d routeguide
```

As each command succeeds, a long hexadecimal string appears, representing the running container.

4. Check that all three containers are up by running `docker ps`. (The sample output is split across multiple lines for ease of reading.)

```
$ sudo docker ps
CONTAINER ID   IMAGE       COMMAND              STATUS         ...
d0cdaaeddf0f   routeguide  "python route_g..."  Up 2 seconds   ...
c04996ca3469   routeguide  "python route_g..."  Up 9 seconds   ...
2170ddb62898   routeguide  "python route_g..."  Up 1 minute    ...

...  PORTS                    NAMES
...  0.0.0.0:10003->50051/tcp  rg3
...  0.0.0.0:10002->50051/tcp  rg2
...  0.0.0.0:10001->50051/tcp  rg1
```

The `PORTS` column in the output shows how each of the containers has mapped a different local port to port 50051 inside the container.

Installing the helloworld Service Containers

1. Build the Docker image for the helloworld containers from the following Dockerfile.

```
# This Dockerfile runs the helloworld server from
# https://grpc.io/docs/quickstart/go.html

FROM golang
RUN go get -u google.golang.org/grpc
WORKDIR $GOPATH/src/google.golang.org/grpc/examples/helloworld

EXPOSE 50051
CMD ["go", "run", "greeter_server/main.go"]
```

View raw on GitHub

You can either copy the Dockerfile to a local subdirectory before the build, or specify the URL of the Gist for the Dockerfile as an argument to the `docker build` command:

```
$ sudo docker build -t helloworld
https://gist.githubusercontent.com/nginx-gists/
87ed942d4ee9f7e7ebb2ccf757ed90be/raw/ce090f92f3bbcb5a94bbf8de
d4d597cd47b43cbe/helloworld.Dockerfile
```

It may take a few minutes to download and build the image. The appearance of the message `Successfully built` and a hexadecimal string (the image ID) signal the completion of the build.

2. Confirm that the image was built by running docker images.

```
$ sudo docker images
REPOSITORY    TAG        IMAGE ID       CREATED          SIZE
helloworld    latest     e5832dc0884a   10 seconds ago   926MB
routeguide    latest     170761fa3f03   4 minutes ago    1.31GB
python        latest     825141134528   9 days ago       923MB
golang        latest     d0e7a411e3da   3 weeks ago      794MB
```

3. Start the helloworld containers.

```
$ sudo docker run --name hw1 -p 20001:50051 -d helloworld
$ sudo docker run --name hw2 -p 20002:50051 -d helloworld
```

As each command succeeds, a long hexadecimal string appears, representing the running container.

4. Check that the two helloworld containers are up by running `docker ps`.

```
$ sudo docker ps
CONTAINER ID   IMAGE       COMMAND             STATUS         ...
e0d204ae860a   helloworld  "go run greeter..."  Up 5 seconds  ...
66f21d89be78   helloworld  "go run greeter..."  Up 9 seconds  ...
d0cdaaeddf0f   routeguide  "python route_g..." Up 4 minutes  ...
c04996ca3469   routeguide  "python route_g..." Up 4 minutes  ...
2170ddb62898   routeguide  "python route_g..." Up 5 minutes  ...

...  PORTS                    NAMES
...  0.0.0.0:20002->50051/tcp  hw2
...  0.0.0.0:20001->50051/tcp  hw1
...  0.0.0.0:10003->50051/tcp  rg3
...  0.0.0.0:10002->50051/tcp  rg2
...  0.0.0.0:10001->50051/tcp  rg1
```

Installing the gRPC Client Applications

1. Install the programming language prerequisites, some of which may already be installed on the test environment.

   - For Ubuntu and Debian, run:

     ```
     $ sudo apt-get install golang-go python3 python-pip git
     ```

   - For CentOS, RHEL, and Oracle Linux, run:

     ```
     $ sudo yum install golang python python-pip git
     ```

   Note that `python-pip` requires the EPEL repository to be enabled (run `sudo yum install epel-release` first as necessary).

2. Download the helloworld application:

   ```
   $ go get google.golang.org/grpc
   ```

3. Download the RouteGuide application:

   ```
   $ git clone -b v1.14.1 https://github.com/grpc/grpc
   $ pip install grpcio-tools
   ```

Testing the Setup

1. Run the helloworld client:

```
$ go run
go/src/google.golang.org/grpc/examples/helloworld/
greeter_client/main.go
```

2. Run the RouteGuide client:

```
$ cd grpc/examples/python/route_guide
$ python route_guide_client.py
```

3. Check the NGINX logs to confirm that the test environment is operational:

```
$ tail /var/log/nginx/grpc_log.json
```