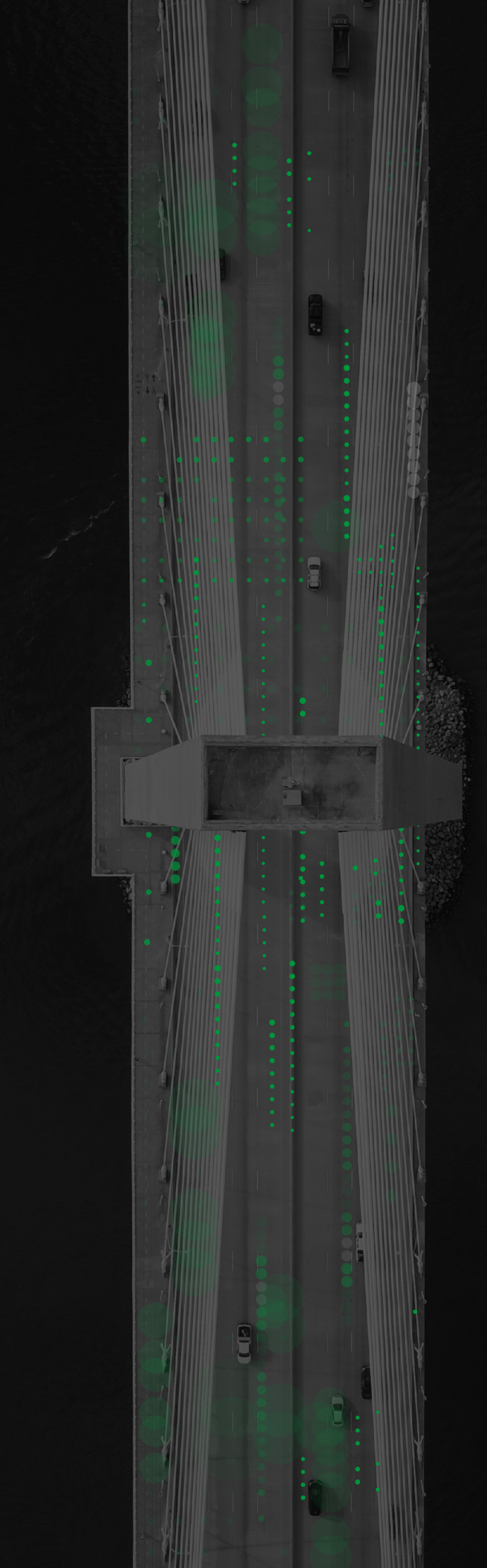


DEPLOYING NGINX PLUS as an API GATEWAY

NGINX

DEPLOYING NGINX PLUS as an API GATEWAY

by Liam Crilly

NGINX

Table of Contents

Foreword 1
- How Is NGINX Currently Used as an API Gateway? 2
- The Importance of a Converged Approach 3
- Why Is NGINX Ideally Suited to Be the Enabler of an API Gateway? 4
- Further Enhancing NGINX Solutions 5
- NGINX Controller and API Gateway Functionality 5

Getting Started with NGINX Plus and your API Gateway 6
- Introducing the Warehouse API 7
- Organizing the NGINX Configuration 8
- Defining the Top-Level API Gateway 9
- Single-Service vs. Microservice API Backends 11
- Defining the Warehouse API 12
  - Choosing Broad vs. Precise Definition for APIs 13
  - Rewriting Client Requests 15
- Responding to Errors 17
- Implementing Authentication 19
  - API Key Authentication 19
  - JWT Authentication 21
- Summary 21

Protecting Backend Services 22
- Rate Limiting 22
- Enforcing Specific Request Methods 25
- Applying Fine-Grained Access Control 26
  - Controlling Access to Specific Resources 27
  - Controlling Access to Specific Methods 28
- Controlling Request Sizes 31
- Validating Request Bodies 32
  - A Note About the `$request_body` Variable 35
- Summary 36

Publishing gRPC Services 37
Defining the gRPC Gateway 38
Running Sample gRPC Services 40
Routing gRPC Requests 42
Precise Routing 44
Responding to Errors 45
Authenticating Clients with gRPC Metadata 47
Implementing Health Checks 47
Applying Rate Limiting and Other API Gateway Controls 49
Summary 49

Appendix A: Setting Up the Test Environment 50
Installing NGINX Plus 50
Installing Docker 51
Installing the RouteGuide Service Containers 51
Installing the helloworld Service Containers 53
Installing the gRPC Client Applications 55
Testing the Setup 56

Appendix B: Document Revision History 57

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 and NetScaler, content distribution networks (CDNs), which include Akamai, and API management tools such as Kong.

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 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'll take a look at how NGINX fits as a potential complement and/or replacement for existing API gateway and API management approaches. We'll 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?

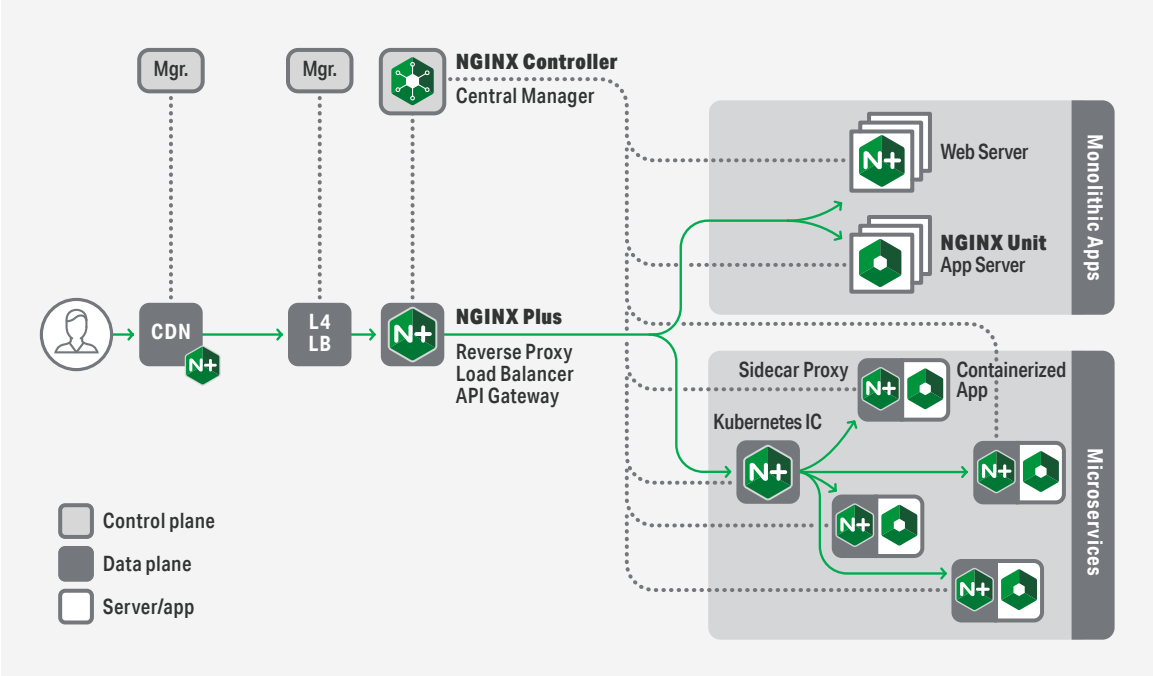
Today, NGINX is deployed as an API gateway in three different manners:

- **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 OpenResty build 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 standalone API gateways** – Standalone API gateways are single-purpose products that focus on API traffic alone. The majority of standalone API gateway products use NGINX and Lua at their core, whether they are dedicated open source solutions or commercial products.

Given these options, we have two recommendations:

- **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 standalone, 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.

- **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 anything from a 50% to a 90% decrease in performance. If your chosen solution relies heavily on Lua, make sure to verify that it meets your peak performance requirements without adding latency to requests.



Example application architecture using NGINX Plus as an API gateway

The Importance of a Converged Approach

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.

If you select discrete components in a distributed environment, you risk instantiating the ungainly "sidecar-on-sidecar" antipattern.

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 – source address, 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	Implements 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 HTTP (web) 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

NGINX Controller and API Gateway Functionality

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 Plus 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.

As the leading high-performance, lightweight reverse proxy and load balancer, NGINX Plus has the advanced HTTP processing capabilities needed for handling API traffic. This makes NGINX Plus 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 Plus 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 Plus and NGINX Open Source.

Introducing the Warehouse API

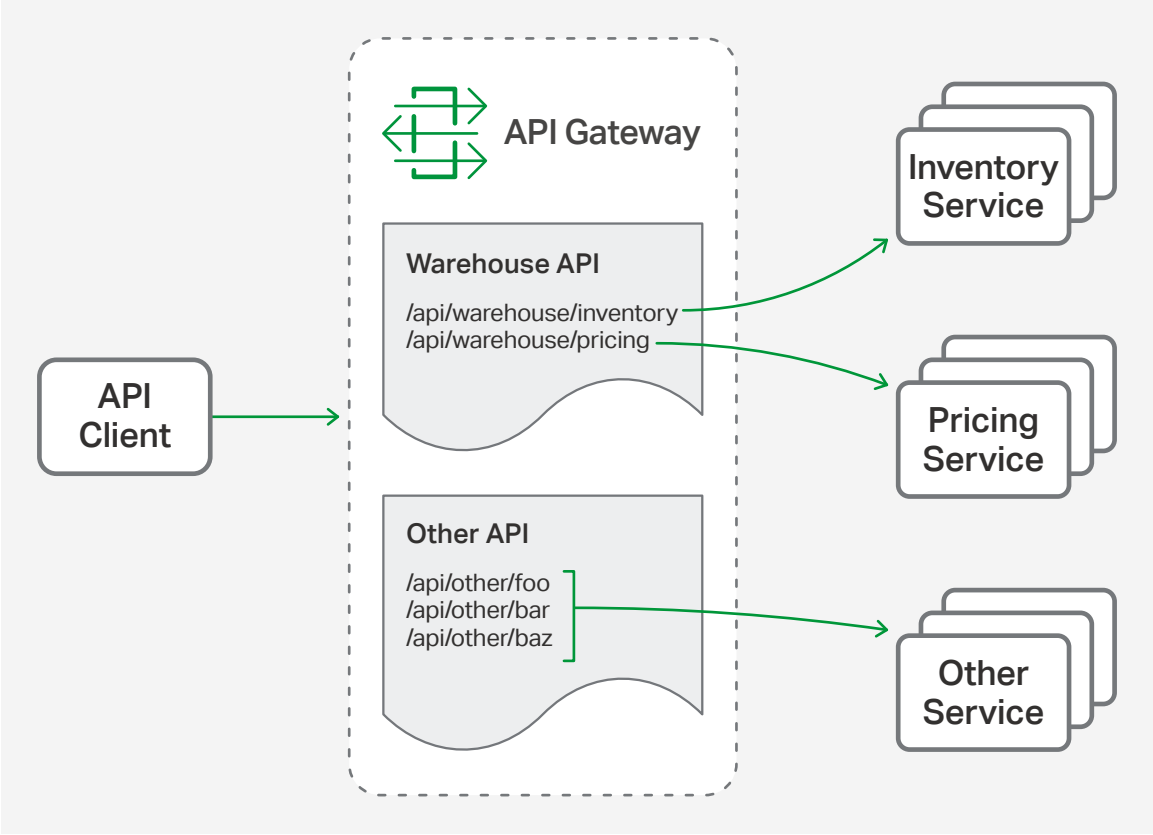
The primary function of the API gateway is to provide a single, consistent entry point for multiple APIs, regardless of how they are implemented or deployed at the backend. Not all APIs are microservices applications. Our API gateway needs to manage existing APIs, monoliths, and applications undergoing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



API gateway architecture for multiple applications

Organizing the NGINX Configuration

One advantage of using NGINX Plus 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 Plus 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 Plus 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 enable different features and capabilities of the API gateway and are explained in detail below.

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). In this chapter we make 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 **api_gateway.conf** file defines the virtual server that exposes NGINX Plus 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 13), protected by TLS as configured on lines 16 through 21. 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.

```
1  log_format api_main '$remote_addr - $remote_user [$time_local] "$request"'
2                      '$status $body_bytes_sent "$http_referer" "$http_user_agent"'
3                      '"$http_x_forwarded_for" "$api_name"';
4
5  include api_backends.conf;
6  include api_keys.conf;
7
8  server {
9      set $api_name -; # Start with undefined API name; it's updated by each API
10     access_log /var/log/nginx/api_access.log api_main; # Each API may also log
                                                          # to a separate file
11
12     listen 443 ssl;
13     server_name api.example.com;
14
15     # TLS config
16     ssl_certificate      /etc/ssl/certs/api.example.com.crt;
17     ssl_certificate_key  /etc/ssl/private/api.example.com.key;
18     ssl_session_cache    shared:SSL:10m;
19     ssl_session_timeout  5m;
20     ssl_ciphers          HIGH:!aNULL:!MD5;
21     ssl_protocols        TLSv1.1 TLSv1.2;
22
23     # API definitions, one per file
24     include api_conf.d/*.conf;
25
26     # Error responses
27     error_page 404 = @400;          # Invalid paths are treated as bad requests
28     proxy_intercept_errors on;      # Do not send backend errors to the client
29     include api_json_errors.conf;   # API client friendly JSON error responses
30     default_type application/json;  # If no content-type then assume JSON
31 }
```

View raw on GitHub

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 24. Lines 27 through 30 deal with logging defaults and error handling, and are discussed later in this chapter in Responding to Errors.

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 }
```

View raw on GitHub

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

This part of the configuration first defines the valid URIs for the Warehouse API and then defines a common policy for handling requests to the Warehouse API.

```
1  # API definition
2  #
3  location /api/warehouse/inventory {
4      set $upstream warehouse_inventory;
5      rewrite ^ /_warehouse last;
6  }
7
8  location /api/warehouse/pricing {
9      set $upstream warehouse_pricing;
10     rewrite ^ /_warehouse last;
11 }
12
13 # Policy section
14 #
15 location = /_warehouse {
16     internal;
17     set $api_name "Warehouse";
18
19     # Policy configuration here (authentication, rate limiting, logging, more...)
20
21     proxy_pass http://$upstream$request_uri;
22 }
```

View raw on GitHub

The Warehouse API is defined with a number of `location` blocks. NGINX Plus has a highly efficient and flexible system for matching the request URI to a section of the configuration. In general, a request is matched by the most specific path prefix and the order of the `location` directives is not important. Here, on lines 3 and 8 we define two path prefixes. In each case the `$upstream` variable is set to the name of the `upstream` block that represents the backend API service or services for the inventory and pricing services respectively.

The goal of this configuration is to separate API definition from the policy governing how the API is delivered. To achieve this, we minimize the configuration that appears in the API definition section. After determining the appropriate upstream group for each location, we stop processing and use a `rewrite` directive to find the policy for the API (line 10).

```
rewrite   ^   /_warehouse   last;
```

Change the request URI	Match any URI	Rewrite the URI to the name of the API (Underscore prefix indicates "internal")	Look for a location that matches the new URI

Using the `rewrite` directive to move processing to the API policy section

The result of the `rewrite` directive is that NGINX Plus searches for a `location` block that matches URIs starting with **`/_warehouse`**. The `location` block on line 15 uses the = modifier to perform an exact match, which speeds up processing.

At this stage our policy section is very simple. The `location` block itself is marked as internal on line 16, meaning that clients cannot make requests to it directly. The `$api_name` variable is redefined to match the name of the API so that it appears correctly in log files. Finally, the request is proxied to the upstream group specified in the API definition section, using the `$request_uri` variable – which contains the original request URI, unmodified.

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**, we use the broad approach for the Warehouse API by defining URI prefixes on lines 3 and 8. This means that any URI that begins with either prefix is proxied to the appropriate backend service. With 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, the 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 the Warehouse API uses a combination of exact matching (=) and regular expressions (~) to define each and every URI.

```
3  location = /api/warehouse/inventory { # Complete inventory
4      set $upstream inventory_service;
5      rewrite ^ /_warehouse last;
6  }
7
8  location ~ ^/api/warehouse/inventory/shelf/[^/]*$ { # Shelf inventory
9      set $upstream inventory_service;
10     rewrite ^ /_warehouse last;
11 }
12
13 location ~ ^/api/warehouse/inventory/shelf/[^/]*/box/[^/]*$ { # Box on shelf
14     set $upstream inventory_service;
15     rewrite ^ /_warehouse last;
16 }
17
18 location ~ ^/api/warehouse/pricing/[^/]*$ { # Price for specific item
19     set $upstream pricing_service;
20     rewrite ^ /_warehouse last;
21 }
```

View raw on Gihub

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 Plus 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 Plus API definitions from the OpenAPI Specification (formerly Swagger). A sample script for this purpose is provided among the Gists for this chapter.

Rewriting Client Requests

As APIs evolve, breaking changes sometimes occur that 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`) 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 can see on line 3 that the pricing service was previously implemented as part of the inventory service: the **rewrite** directive converts requests made to the old pricing resource to the new pricing service.

```
1  # Rewrite rules
2  #
3  rewrite ^/api/warehouse/inventory/item/price/(.*) /api/warehouse/pricing/$1;
4
5  # API definition
6  #
7  location /api/warehouse/inventory {
8      set $upstream inventory_service;
9      rewrite ^(.*)$ /_warehouse$1 last;
10 }
11
12 location /api/warehouse/pricing {
13     set $upstream pricing_service;
14     rewrite ^(.*)$ /_warehouse$1 last;
15 }
16
17 # Policy section
18 #
19 location /_warehouse {
20     internal;
21     set $api_name "Warehouse";
22
23     # Policy configuration here (authentication, rate limiting, logging, more...)
24
25     rewrite ^/_warehouse/(.*)$ /$1 break;    # Remove /_warehouse prefix
26     proxy_pass http://$upstream;             # Proxy the rewritten URI
27 }
```

View raw on GitHub

Rewriting the URI on the fly means that we can no longer use the `$request_uri` variable when we ultimately proxy the request at line 26 (as we did on line 21 of **warehouse_api_simple.conf**). This means we need to use slightly different `rewrite` directives on lines 9 and 14 of the API definition section in order to preserve the URI as processing switches to the policy section.

```
rewrite ^(.*)$ /_warehouse$1 last;
```

Change the request URI	Capture the entire original URI	Rewrite the URI so that the original URI ($1) is prefixed by /_warehouse to identify the URI	Look for a location that matches the new URI

Using the `rewrite` directive to move processing to the API policy section while preserving the URI

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 Plus 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.

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

View raw on GitHub

The `error_page` directive on line 27 specifies that when a request does not match any of the API definitions, NGINX Plus 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 28 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 to the client.

The complete list of error responses is defined in a separate configuration file referenced by the **`include`** directive on line 29, the first few lines of which are shown below. This file can be modified if a different error format is preferred, and by changing the `default_type` value on line 30 of **api_gateway.conf** to match. You can also have a separate **`include`** directive in each API's policy section to define a different set of error responses which override the default.

```
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/1.13.10
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 Plus offers several approaches for protecting APIs and authenticating API clients. See the documentation for information about 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. They are 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 6 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.

```
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. API key authentication is enforced by the policy section for each API.

```
14 # Policy section
15 #
16 location = /_warehouse {
17     internal;
18     set $api_name "Warehouse";
19
20     if ($http_apikey = "") {
21         return 401; # Unauthorized (please authenticate)
22     }
23     if ($api_client_name = "") {
24         return 403; # Forbidden (invalid API key)
25     }
26
27     proxy_pass http://$upstream$request_uri;
28 }
```

View raw on GitHub

Clients are expected to present their API key in the `apikey` HTTP header. If this header is missing or empty (line 20), we send a `401` response to tell the client that authentication is required. Line 23 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` response to tell the client that authentication failed.

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 Plus as an API gateway. The complete set of files discussed in this blog can be reviewed and downloaded from our GitHub Gist repo.

2 Protecting Backend Services

Rate Limiting

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 Plus 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.

```
log_format api_main '$remote_addr - $remote_user [$time_local] "$request"'
                    '$status $body_bytes_sent
                    "$http_referer" "$http_user_agent"'
                    '"$http_x_forwarded_for" "$api_name"';

include api_backends.conf;
include api_keys.conf;

limit_req_zone $binary_remote_addr  zone=client_ip_10rs:1m rate=10r/s;
limit_req_zone $http_apikey         zone=apikey_200rs:1m   rate=200r/s;

server {
    set $api_name -; # Start with an undefined API name, each API will update this value
    access_log /var/log/nginx/api_access.log api_main; # Each API may also log
                                                       # to a separate file

    listen 443 ssl;
    server_name api.example.com;

    # TLS config
    ssl_certificate      /etc/ssl/certs/api.example.com.crt;
    ssl_certificate_key  /etc/ssl/private/api.example.com.key;
    ssl_session_cache    shared:SSL:10m;
    ssl_session_timeout  5m;
    ssl_ciphers          HIGH:!aNULL:!MD5;
    ssl_protocols        TLSv1.1 TLSv1.2;

    # API definitions, one per file
    include api_conf.d/*.conf;

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

View raw on GitHub

In this example, the `limit_req_zone` directive on line 8 defines a rate limit of 10 requests per second for each client IP address (`$binary_remote_addr`), and the one on line 9 defines a limit of 200 requests per second for each authenticated client ID (`$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 Plus 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.

```
21 # Policy section
22 #
23 location = /_warehouse {
24     internal;
25     set $api_name "Warehouse";
26
27     limit_req zone=client_ip_10rs;
28     limit_req_status 429;
29
30     proxy_pass http://$upstream$request_uri;
31 }
```

View raw on GitHub

You can use additional parameters to the `limit_req` directive to fine-tune how NGINX Plus 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 (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 definition of 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).

```
1  # API definition
2  #
3  location /api/warehouse/pricing {
4     limit_except GET PATCH {
5           deny all;
6     }
7     error_page 403 = @405; # Convert response from '403 (Forbidden)' to
                             # '405 (Method Not Allowed)'
8     set $upstream pricing_service;
9     rewrite ^ /_warehouse last;
10 }
11
12 location /api/warehouse/inventory {
13    limit_except GET {
14          deny all;
15    }
16    error_page 403 = @405;
17    set $upstream inventory_service;
18    rewrite ^ /_warehouse last;
19 }
```

View raw on GitHub

With this configuration in place, requests to the pricing service other than those listed on line 4 (and to the inventory service other than those on line 13) are rejected and are not passed to the backend services. NGINX Plus 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 described 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 extends a configuration presented in Chapter 1 and uses a whitelist of API clients to control access to a specific API resource, based on API key authentication. The second example implements the JWT authentication method mentioned in Chapter 1, using a custom claim to control which HTTP methods NGINX Plus accepts from the client. Of course, all of the NGINX Plus authentication methods are applicable to these examples.

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 a whitelist 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 on lines 13–19. The `if` block ensures that only infrastructure clients can access the resource.

```
1  # API definition
2  #
3  location /api/warehouse/pricing {
4      set $upstream pricing_service;
5      rewrite ^ /_warehouse last;
6  }
7
8  location /api/warehouse/inventory {
9      set $upstream inventory_service;
10     rewrite ^ /_warehouse last;
11 }
12
13 location = /api/warehouse/inventory/audit {
14     if ($is_infrastructure = 0) {
15         return 403; # Forbidden (not infrastructure)
16     }
17     set $upstream inventory_service;
18     rewrite ^ /_warehouse last;
19 }
```

View raw on GitHub

Note that the `location` directive on line 13 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 whitelist 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

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 issue. 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`.

```
52 map $request_method $request_type {
53     "GET"     "READ";
54     "HEAD"    "READ";
55     "OPTIONS" "READ";
56     default   "WRITE";
57 }
```

View raw on GitHub

This `map` block, added to the bottom of **api_gateway.conf**, coalesces all of the possible HTTP methods into a new variable, **`$request_type`**, which evaluates to either `READ` or `WRITE`. In the following snippet, we use the **`$request_type`** variable to direct requests to the appropriate Warehouse API policy, `/_warehouse_READ` or `/_warehouse_WRITE`.

```
1  # API definition
2  #
3  location /api/warehouse/pricing {
4      limit_except GET PATCH DELETE {
5                       deny all;
6      }
7      error_page 403 = @405; # Convert response from '403 (Forbidden)'
                              # '405 (Method Not Allowed)'
8      set $upstream pricing_service;
9      rewrite ^ /_warehouse_$request_type last;
10 }
11
12 location /api/warehouse/inventory {
13     set $upstream inventory_service;
14     rewrite ^ /_warehouse_$request_type last;
15 }
```

View raw on GitHub

The `rewrite` directives on lines 9 and 14 append the **`$request_type`** variable to the name of the Warehouse API policy, thereby splitting the policy section into two. Now different policies apply to read and write operations.

```
17 # Policy section
18 #
19 location = /_warehouse_READ {
20     internal;
21     set $api_name "Warehouse";
22 
23     auth_jwt $api_name;
24     auth_jwt_key_file /etc/nginx/jwk.json;
25 
26     proxy_pass http://$upstream$request_uri;
27 }
28 
29 location = /_warehouse_WRITE {
30     internal;
31     set $api_name "Warehouse";
32 
33     auth_jwt $api_name;
34     auth_jwt_key_file /etc/nginx/jwk.json;
35     if ($jwt_claim_admin != "true") { # Write operations must have "admin":true
36         return 403; # Forbidden
37     }
38 
39     proxy_pass http://$upstream$request_uri;
40 }
```

View raw on GitHub

Both the **/_warehouse_READ** and **/_warehouse_WRITE** policies require the client to present a valid JWT. However, for requests that use a WRITE method (**POST**, **PATCH**, or **DELETE**), we further require that the JWT includes the claim **"admin":true** (line 35). This approach of having separate policies for different request methods is not limited to authentication. Other controls can also be applied on a per-method basis, such as rate limiting, logging, and routing to different backends.

JWT authentication is exclusive to NGINX Plus.

Controlling Request Sizes

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 Plus 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.

```
13 # Policy section
14 #
15 location = /_warehouse {
16     internal;
17     set $api_name "Warehouse";
18
19     client_max_body_size 16k;
20
21     proxy_pass http://$upstream$request_uri;
22 }
```

View raw on GitHub

The `client_max_body_size` directive on line 19 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/1.13.10
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/1.13.10
Content-Type: application/json
Content-Length: 45
Connection: close

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

Validating Request Bodies

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_include` directive to reference the file containing the JavaScript code for the function that validates JSON data.

```
42 js_include json_validator.js;
43 js_set $validated json_validator;
```

View raw on GitHub

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

```
1  function json_validator(req) {
2      try {
3          if ( req.variables.request_body.length > 0 ) {
4              JSON.parse(req.variables.request_body);
5          }
6          return req.variables.upstream;
7      } catch (e) {
8          req.log('JSON.parse exception');
9          return '127.0.0.1:10415'; // Address for error response
10     }
11 }
```

View raw on GitHub

The **json_validator** function attempts to parse the request body using the **JSON.parse** method. If successful, then the name of the intended upstream group for this request is returned (line 6). If the request body cannot be parsed (causing an exception), then a local server address is returned (line 9). The **return** directive populates the **$validated** variable so that we can use it to determine where to send the request.

```
13 # Policy section
14 #
15 location = /_warehouse {
16     internal;
17     set $api_name "Warehouse";
18
19     mirror /_NULL;                      # Create a copy of the request to
                                           # capture request body
20     client_body_in_single_buffer on;    # Minimize memory copy operations
                                           # on request body
21     client_body_buffer_size      16k;   # Largest body to keep in memory
                                           # (before writing to file)
22     client_max_body_size         16k;
23
24     proxy_pass http://$validated$request_uri;
25 }
```

View raw on GitHub

In the policy section for the Warehouse API, we modify the `proxy_pass` directive on line 24. It passes the request to the backend API service as before, but now uses the **`$validated`** variable as the destination address. If the client body was successfully parsed as JSON then we proxy to the upstream group as normal. 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.

```
45 server {
46     listen 127.0.0.1:10415; # This is the error response of json_validator()
47     return 415; # Unsupported media type
48     include api_json_errors.conf;
49 }
```

View raw on GitHub

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

With this complete configuration in place, NGINX Plus 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/1.13.10
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 `json_validator` uses the `$request_body` variable to perform JSON parsing. However, NGINX Plus does not populate this variable by default, and simply streams the request body to the backend without making intermediate copies. By using the `mirror` directive inside the Warehouse API policy section (line 19) we create a copy of the client request, and consequently populate the `$request_body` variable.

```
13 # Policy section
14 #
15 location = /_warehouse {
16     internal;
17     set $api_name "Warehouse";
18
19     mirror /_NULL;                          # Create a copy of the request to
                                               # capture request body
20     client_body_in_single_buffer on;        # Minimize memory copy operations on
                                               # request body
21     client_body_buffer_size      16k;       # Largest body to keep in memory
                                               # (before writing to file)
22     client_max_body_size         16k;
23
24     proxy_pass http://$validated$request_uri;
25 }
```

View raw on GitHub

The directives on lines 20 and 21 control how NGINX Plus 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 above, 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 (**`/_NULL`**) that we define in the top-level API gateway entry point.

```
35      # Dummy location used to populate $request_body for JSON validation
36      location = /_NULL {
37          internal;
38          return 204;
39      }
```

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 Plus uses the same technology for managing API traffic that is used to power and protect the busiest sites on the Internet today.

3 Publishing gRPC Services

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 the previous two chapters in this ebook use a hypothetical REST API to illustrate how NGINX Plus 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 number 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, 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

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

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 Plus 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 a number of reasons why 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**.

```
1  log_format grpc_json escape=json '{"timestamp":"$time_iso8601","client":
                                    "$remote_addr",'
2                                   '"uri":"$uri","http-status":$status,'
3                                   '"grpc-status":$grpc_status,"upstream":
                                    "$upstream_addr"'
4                                   '"rx-bytes":$request_length,"tx-bytes":
                                      $bytes_sent}';
5
6  map $upstream_trailer_grpc_status $grpc_status {
7      default $upstream_trailer_grpc_status; # We normally expect to receive
8                                             # grpc-status as a trailer
9      ''      $sent_http_grpc_status;        # Else use the header, regardless of
10                                            # who generated it
11 }
12
13 server {
14     listen 50051 http2; # In production, comment out to disable plaintext port
15     listen 443   http2 ssl;
16     server_name  grpc.example.com;
17     access_log   /var/log/nginx/grpc_log.json grpc_json;
18
19     # TLS config
20     ssl_certificate      /etc/ssl/certs/grpc.example.com.crt;
21     ssl_certificate_key  /etc/ssl/private/grpc.example.com.key;
22     ssl_session_cache    shared:SSL:10m;
23     ssl_session_timeout  5m;
24     ssl_ciphers          HIGH:!aNULL:!MD5;
25     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 as well as 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 Plus itself. To simplify the access log, we use a **map** block (lines 6–11) to evaluate a new variable **$grpc_status** and obtain the gRPC status from wherever it originates.

This configuration contains two `listen` directives (lines 14 and 15) so that we can test both plaintext (port 50051) and TLS-protected (port 443) traffic. The `http2` parameter configures NGINX Plus 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.

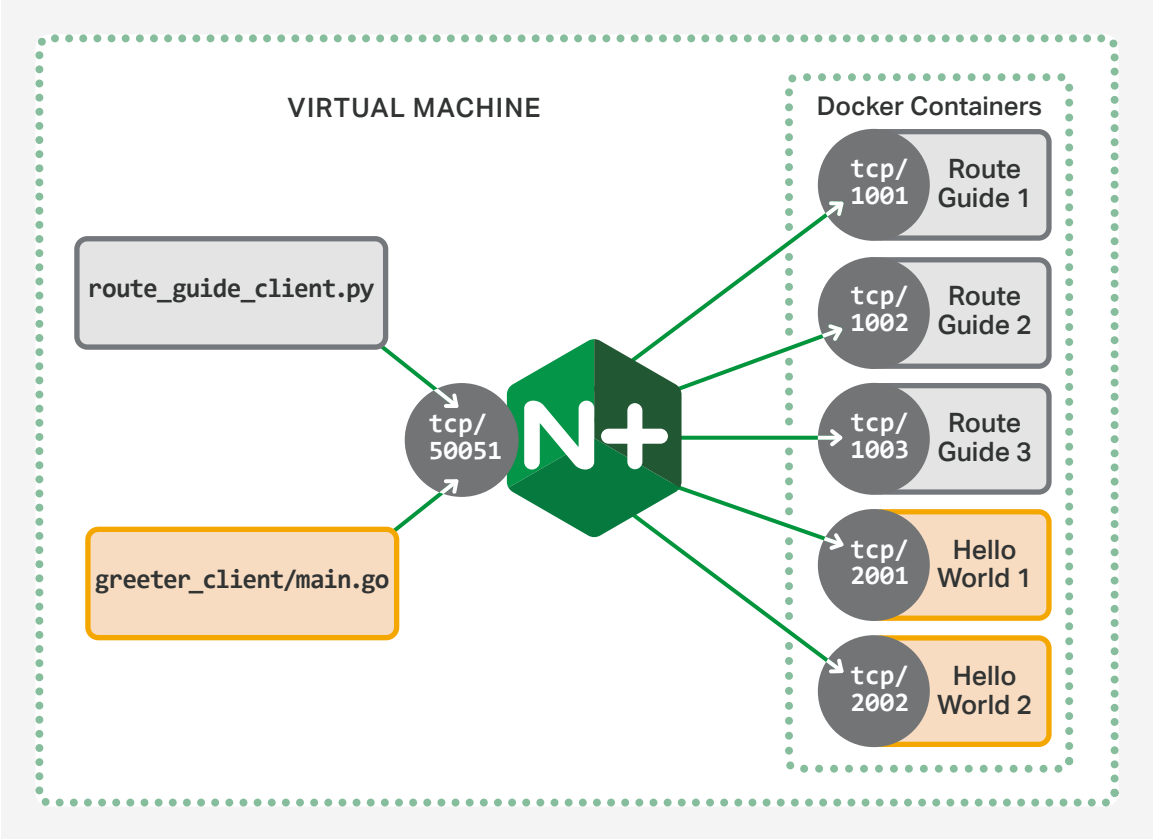
The TLS configuration is conventional, with the exception of the `ssl_protocols` directive (line 25), which specifies TLS 1.2 as the weakest acceptable protocol. The HTTP/2 specification mandates the use of TLS 1.2 (or higher), which guarantees that all clients support the Server Name Indication (SNI) 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 Plus, 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 Plus host. For complete instructions on building the test environment, see Appendix A.



Test environment for NGINX Plus as a gRPC gateway

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

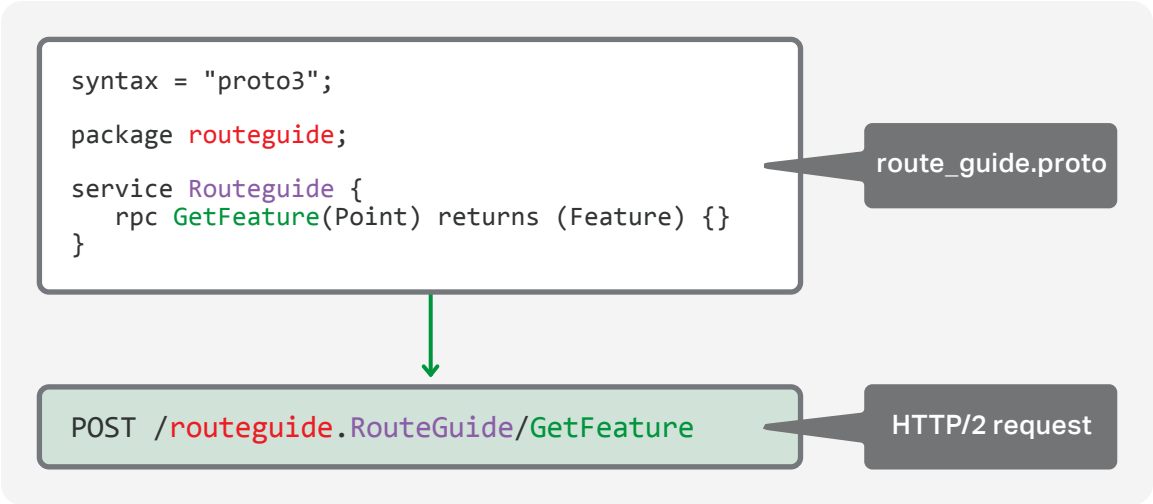
```
40 # Backend gRPC servers
41 #
42 upstream routeguide_service {
43     zone routeguide_service 64k;
44     server 127.0.0.1:10001;
45     server 127.0.0.1:10002;
46     server 127.0.0.1:10003;
47 }
48
49 upstream helloworld_service {
50 zone helloworld_service 64k;
51     server 127.0.0.1:20001;
52     server 127.0.0.1:20002;
53 }
```

View raw on GitHub

We add an `upstream` block for each of the gRPC services (lines 42–47 and 49–53) and populate them with the addresses of the individual containers that are running the gRPC server code.

Routing gRPC Requests

With NGINX Plus 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 Plus.



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 `helloword`).

```
27      # Routing
28      location /routeguide. {
29      grpc_pass grpc://routeguide_service;
30      }
31      location /helloworld. {
32            grpc_pass grpc://helloworld_service;
33      }
```

View raw on GitHub

The first `location` block (line 28), 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 29) therefore passes all requests from the RouteGuide client to the upstream group **routeguide_service**. This configuration provides a simple mapping between a gRPC package and its backend services.

Notice that the argument to the **grpc_pass** directive 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": "2018-08-09T12:17:56+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 Plus error pages (in HTML format) unsuitable when NGINX Plus is configured as a gRPC gateway. We address this by specifying a set of custom error responses for gRPC clients.

```
35      # Error responses
36      include conf.d/errors.grpc_conf; # gRPC-compliant error responses
37      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 36) 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 Plus should 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 = @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 nonexistent 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
"Trailers: TE" -X POST https://grpc.example.com/does.Not/Exist
HTTP/2 204
server: nginx/1.15.2
date: Thu, 09 Aug 2018 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 Plus 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 Plus.

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 grpc://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 Plus. 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.

gRPC is an application protocol that runs on top of HTTP/2, and as such it is not easy for NGINX Plus to simulate a gRPC client. However, if the backend gRPC

service accepts HTTP `GET` requests – as is the case for services written in Go – then we can configure NGINX Plus to test whether a backend gRPC service is up.

The following configuration snippet enables active health checks for the helloworld gRPC service (which is written in Go); to highlight the relevant configuration, it omits some directives that are included in the **grpc_gateway.conf** file used in previous sections.

```
1  server {
2      listen 50051 http2; # Plaintext
3
4      # Routing
5      location /helloworld. {
6          grpc_pass grpc://helloworld_service;
7      }
8
9      # Health-check the helloworld containers
10         location @helloworld_health {
11         health_check mandatory uri=/nginx.health/check match=grpc_unknown;
12         grpc_set_header Content-Type application/grpc;
13         grpc_set_header TE Trailers;
14         grpc_pass grpc://helloworld_service;
15     }
16 }
17
18     # Specify the expected response to the health check (this
19     # assumes that the gRPC service responds to GET requests)
20     match grpc_unknown {
21         header Content-Type = application/grpc;
22         header grpc-status = 12; # unimplemented / unknown method
23 }
```

View raw at GitHub

Following the routing section (lines 4–7) we define the health check in a named location, `@helloworld_health`. This allows us to customize the health check request without overloading the `location` blocks used for request routing.

The `health_check` directive (line 11) specifies a URI known to be invalid, **/nginx.health/check**, which we expect the backend to report as unknown. The expected response is then defined in the `match` block (line 20) with the HTTP headers that indicate we are communicating with an active gRPC service.

With this configuration in place, we can take down one of the **helloworld** containers without gRPC clients experiencing delays or timeouts. Active health checks are exclusive to NGINX Plus.

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 Plus 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

In this chapter, we focused on gRPC as a cloud-native technology for building microservices applications. We demonstrated how NGINX Plus 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 Plus as a multi-purpose API gateway.

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

Appendix A:

Setting Up the Test Environment

The following instructions install the test environment 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 Plus 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 Plus can use it as its listen port. Therefore, when the test clients connect using their preconfigured port of 50051, they reach NGINX Plus.

Installing NGINX Plus

1. Install NGINX Plus. Instructions are 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 Plus.

```
$ sudo nginx
```

Installing Docker

For CentOS, RHEL, and Oracle Linux, run:

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

For CentOS/Redhat/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/
ce090f92f3bbcb5a94bbf8ded4d597cd47b43cbe/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 routeguide https://gist.githubusercontent.com/
nginx-gists/87ed942d4ee9f7e7ebb2ccf757ed90be/raw/
ce090f92f3bbcb5a94bbf8ded4d597cd47b43cbe/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`.

```
$ 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`.

```
$ 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 Plus logs to confirm that the test environment is operational:

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

Appendix B:

Document Revision History

Version	Date	Description
1.0	2018-10-04	Initial release