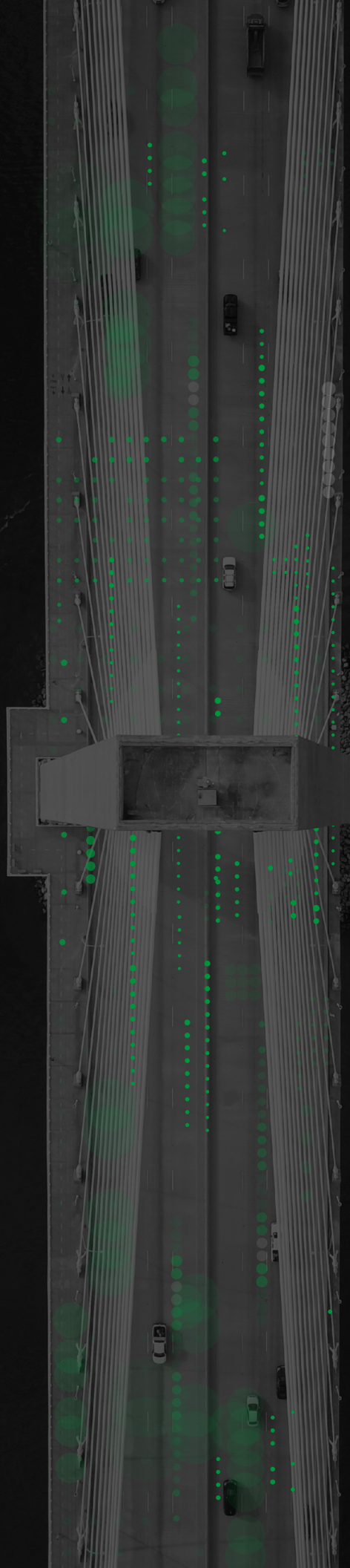


API 게이트웨이로서 NGINX PLUS 구축하기

NGINX

API 게이트웨이로서 NGINX PLUS 구축하기

작성자: Liam Crilly

NGINX

© NGINX, Inc 2018 NGINX 및 NGINX Plus는 NGINX, Inc의 등록 상표입니다.

목차

서문 1

NGINX는 현재 API 게이트웨이로서 어떻게 활용되고 있는가? 2
컨버지드 접근방식의 중요성 3
왜 NGINX가 API 게이트웨이의 역할을 최적으로 수행할 수 있는가? 4
지속적인 NGINX 솔루션의 향상 5
NGINX 컨트롤러 및 API 게이트웨이 기능 5

NGINX Plus 및 API 게이트웨이로 시작하기 6

Warehouse API 소개 7
NGINX 구성 체계화 8
최상위 API 게이트웨이 정의 9
단일 서비스 vs. 마이크로서비스 API 백엔드 11
Warehouse API 정의 12
- API에 대한 광의의 정의 vs. 협의의 정의 13
- 클라이언트 요청 재작성 15

에러에 응답하는 방법 17
인증 구현 19
- API Key 인증 19
- JWT 인증 21

요약 21

백엔드 서비스 보호 22

속도 제한 22
특정 요청 메소드 적용 25
세분화된 액세스 제어 적용 26
- 특정 리소스에 대한 액세스 제어 27
- 특정 메소드에 대한 액세스 제어 28

요청 사이즈 제어 31
요청 본문 유효성 검증 32
- $request_body 변수에 대한 주의 사항 35

요약 36

gRPC 서비스 퍼블리싱 37
gRPC 게이트웨이 정의 38
샘플 gRPC 서비스 실행 40
gRPC 요청 라우팅 42
정교한 라우팅 44
에러에 응답하는 방법 45
gRPC 메타데이터를 이용한 클라이언트 인증 47
헬스체크 구현 47
속도 제한 및 기타 API 게이트웨이 제어 적용 49
요약 49

부록 A: 테스트 환경 설정하기 50
NGINX Plus 설치하기 50
Docker 설치하기 51
RouteGuide Service Container 설치하기 51
helloworld Service Container 설치하기 53
gRPC 클라이언트 애플리케이션 설치하기 55
설정 테스트 56

부록 B: 문서 개정 내역 57

서문

애플리케이션 개발과 딜리버리는 오늘날 대부분 기업들에게 매우 중요하며 특히, 기존의 대규모 비즈니스와 모든 분야의 혁신적인 스타트업들에게는 더욱 그렇습니다.

애플리케이션 개발 및 딜리버리 분야에서 다양한 과제들이 대두하면서, 특정 문제를 위해 개발된 다양한 유형의 솔루션들이 주목을 끌었습니다. 이러한 포인트 솔루션의 예로 F5 및 NetScaler 등의 하드웨어 로드 밸런서 – 일명 ADC(Application Delivery Controller) –, Akamai를 포함한 CDN(content distribution networks), 그리고 Kong과 같은 API 관리 툴 등을 들 수 있습니다.

이러한 포인트 솔루션의 확산과 함께 NGINX를 필두로 보다 가볍고 유연한 접근방식들이 개발되었습니다. 경제적인 범용 서버 하드웨어의 용량이 증가하면서 기업들은 단일 창구에서 손쉽게 작업을 관리하기 위해 간편하고 유연한 소프트웨어를 활용하고 있습니다.

따라서, 오늘날 많은 기업들은 NGINX 로드 밸런싱 기능을 이용해 기존 하드웨어 ADC를 보완하거나, 심지어 이를 완전히 대체하고 있습니다. 다양한 레벨의 캐싱을 위해 NGINX를 사용하거나, 자체적으로 고유의 CDN을 개발해 상용 CDN 사용을 보완 또는 대체하고 있습니다. (대부분의 상용 CDN의 핵심에는 NGINX가 있음)

API 게이트웨이 활용 사례는 이제 범용 하드웨어의 성능 향상과 NGINX의 지속적인 기능 증가의 조합인 거스를 수 없는 추세에 편승하기 시작했습니다. CDN과 마찬가지로, 많은 기존 API 관리 툴들이 NGINX를 기반으로 구축되었습니다.

이 ebook에서는 기존 NGINX Open Source 또는 NGINX Plus 구성을 가져와 API 트래픽도 관리할 수 있도록 확장하는 방법을 설명합니다. API 관리를 위해 NGINX를 사용하는 경우, NGINX의 강점인 고성능, 신뢰성, 강력한 커뮤니티 지원 및 전문가 지원 (NGINX Plus 고객)을 활용할 수 있습니다.

이를 염두에 두고 가능하면, 이 ebook에서 설명한 기법들을 활용해 NGINX 기능들에 접근하는 것을 권장합니다. 그 다음, 고유하게 제공하는 모든 기능들에 대한 보완 솔루션들을 활용해 보십시오.

이 서문에서는 기존 API 게이트웨이 및 API 관리 접근방식에 대한 잠재적 보완 및/대체 기술로서 NGINX가 적합한 이유를 살펴볼 것입니다. 그런 다음, ebook의 나머지 부분에서는 많은 중요한 API 게이트웨이 기능들을 NGINX로 구현하는 방법을 설명할 것입니다.

NGINX는 현재 API 게이트웨이로서 어떻게 활용되고 있는가?

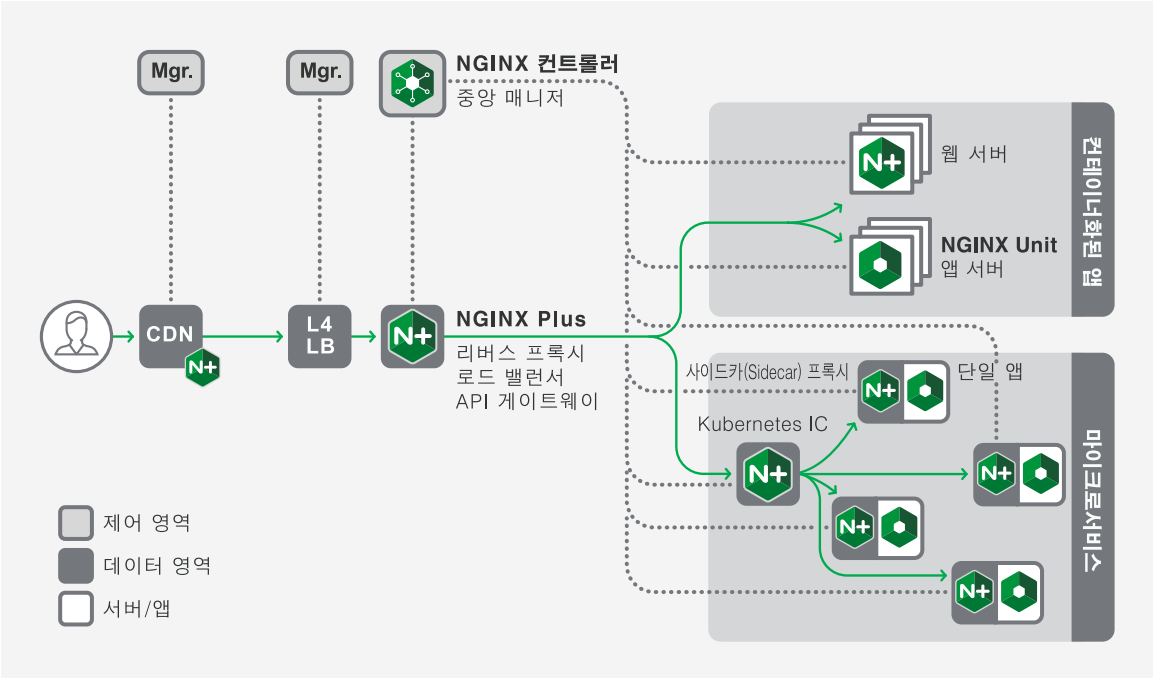
오늘날, NGINX는 3가지 방법으로 API 게이트웨이로서 구축됩니다.

- **네이티브 NGINX 기능** – NGINX를 이용해 API 트래픽을 직접 관리하는 경우가 많습니다. API 트래픽을 HTTP 또는 gRPC라고 인식하고 API 관리 요구 사항을 NGINX 구성으로 변환하여 API 요청을 수신 및 라우팅하고 속도 제한을 적용하며 안전하게 보호합니다.
- **Lua를 이용한 NGINX 확장** – NGINX OpenResty 빌드는 Lua 인터프리터를 NGINX 코어에 추가해 사용자들이 NGINX 위에서 다양한 기능들을 개발할 수 있도록 합니다. Lua 모듈은 vanilla NGINX Open Source와 NGINX Plus 빌드로 컴파일 및 로딩될 수 있습니다. 현재 GitHub에는 수십 개의 오픈 소스 NGINX 기반 API 게이트웨이 구현이 있으며 상당 수가 Lua와 OpenResty를 사용합니다.
- **타사 독립형 API 게이트웨이** – 독립형 API 게이트웨이는 API 트래픽에만 초점을 맞춘 단일 목적(single-purpose) 제품입니다. 대부분의 독립형 API 게이트웨이 제품들은 전용 오픈소스 솔루션이든, 범용 제품이든 관계없이 코어에서 NGINX와 Lua를 사용합니다.

이들 옵션을 감안해 다음 2가지 사항을 권장합니다.

- **컨버지드 접근방식을 취할 것** – NGINX는 일반적인 웹 트래픽과 함께 API 트래픽을 관리할 수 있습니다. API 게이트웨이는 NGINX 기능의 서브셋입니다. (일부의 경우, API 게이트웨이 오퍼링에 NGINX에서 볼 수 있는 기능을 복제한 Lua 코드가 포함되어 있지만, 신뢰성과 성능이 저하될 수 있습니다.) 독립형 단일 목적 API 게이트웨이를 선택하면, 웹 및 API 트래픽 관리를 위해 별도의 제품을 사용해야 합니다. 이에 따라 DevOps, CI/CD, 모니터링, 보안, 그리고 기타 애플리케이션 개발 및 딜리버리 부서의 복잡성이 가중되면서 아키텍처 측면에서 유연성이 제한됩니다.

- **성능 및 요청 지연 시간에 유의할 것** – Lua에 NGINX를 확장하는 효과적인 방법이지만, NGINX의 성능을 저하시킬 수 있습니다. 단순한 Lua 스크립트에 대해 실시한 자체 테스트에서 50%에서 90%의 성능 저하가 나타났습니다. 선택한 솔루션이 Lua를 크게 의존한다면, 요청 처리에 지연 시간이 늘어나지 않으면서 피크 성능 요구 사항을 충족하는 지를 확인해야 합니다.



NGINX Plus를 API 게이트웨이로 사용한 애플리케이션 아키텍처 예제

컨버지드 접근방식의 중요성

컨버지드 접근방식은 특히 최신 분산형 애플리케이션에서 중요합니다. NGINX가 웹 및 API 트래픽을 위한 리버스 프록시로서만이 아니라, 캐시, 인증 게이트웨이, 웹 애플리케이션 방화벽 및 애플리케이션 게이트웨이로서 실행된다는 사실에 유념하십시오.

분산 환경에서 이기종 구성 요소를 선택하는 경우, 다루기 힘든 "sidecar-on-sidecar" 안티패턴(antipattern)을 인스턴스화 할 위험이 있습니다.

왜 NGINX가 API 게이트웨이의 역할을 최적으로 수행할 수 있는가?

아래 표에서는 외부 소스에서 발생한 API 요청을 관리하고 이를 내부 서비스로 라우팅하는 API 게이트웨이 활용 사례를 보여주고 있습니다.

	API 게이트웨이 활용 사례	NGINX 리버스 프록시
핵심 프로토콜	REST (HTTPS), gRPC	HTTP, HTTPS, HTTP/2, gRPC
추가 프로토콜	TCP 전달 메시지 대기열	WebSocket, TCP, UDP
라우팅 요청 requests	요청은 서비스(호스트 헤더), API 메소드(HTTP URL) 및 매개변수를 기준으로 라우팅됨	호스트 헤더, URL 및 요청 헤더를 기반으로 하는 매우 유연한 라우팅 요청
API 라이프사이클 관리	레거시 API 요청 재작성, deprecated API에 대한 호출 거부	직접 요청을 라우팅하거나 요청에 응답하는 포괄적인 요청 재작성 및 다양한 의사 결정 엔진(decision engine)
취약한 애플리케이션 보호	API 및 메소드별 속도 제한	소스 주소, 요청 매개변수 등 다양한 기준에 따른 속도 제한; 백엔드 서비스에 대한 연결 제한
오프로딩 인증	수신 요청 내 인증 토큰 조회	JWT, API key, OpenID Connect 및 기타 외부 인증 서비스 등을 포함한 다수의 인증 방법
변화하는 애플리케이션 토폴로지 관리	구성 변경 수용 및 blue-green 워크플로우 지원을 위한 다양한 API 구현	엔드포인트를 찾기 위한 API 및 서비스 검색(NGINX Plus); blue-green 구축 및 기타 활용 사례를 위한 API 오케스트레이션

NGINX는 컨버지드 솔루션으로서, 프로토콜(HTTP/2 및 HTTP, FastCGI, uwsgi) 간을 변환하고 일관된 구성 및 모니터링 인터페이스를 제공하여 웹 트래픽을 쉽게 관리할 수 있습니다. NGINX는 컨테이너 환경 내에, 또는 사이드카(sidecar)로서 구축될 수 있을만큼 가벼우며, 최소한의 리소스를 사용합니다.

지속적인 NGINX 솔루션의 향상

NGINX는 당초 HTTP(웹) 트래픽을 위한 게이트웨이로서 개발되었으며 이를 구성하는 기본요소는 HTTP 요청으로 표시됩니다. 이들은 API 게이트웨이를 구성하는 방식과 유사하지만, 동일하지는 않기 때문에 DevOps 엔지니어는 HTTP 요청에 API 정의를 매핑하는 방법을 이해해야 합니다.

단순한 API의 경우, 이는 간단하게 수행될 수 있습니다. 보다 복잡한 상황과 관련해, 이 ebook의 제3장에서 NGINX 내 서비스에 복합 API를 매핑하는 방법과 다음과 같은 일반적인 작업을 수행하는 방법을 설명합니다.

- API 요청 재작성
- 에러에 올바르게 응답
- 액세스 제어를 위한 API key 관리
- 사용자 인증을 위한 JWT 토큰 조회
- 속도 제한
- 특정 요청 메소드 적용
- 세분화된 액세스 제어 적용
- 요청 사이즈 제어
- 요청 본문 유효성 검증

NGINX 컨트롤러 및 API 게이트웨이 기능

NGINX 컨트롤러는 NGINX Application Platform을 위한 제어 영역으로서 NGINX 및 NGINX Plus의 다기능성을 기반으로 개발되어 단순하고 일관된 방식으로 강력한 애플리케이션 딜리버리 기능을 제공합니다.

NGINX 컨트롤러는 API에 최적화된 방식으로 API 관리 정책을 정의하고 이를 애플리케이션 플랫폼 전반에 구축된 다기능 NGINX 게이트웨이로 푸시할 수 있습니다. NGINX 컨트롤러에 대한 자세한 정보와 무료 체험판을 다운로드 받으시려면, NGINX 웹 사이트를 방문해 주십시오.

1 NGINX Plus 및 API 게이트웨이로 시작하기

최신 애플리케이션 아키텍처의 핵심은 HTTP API입니다. HTTP는 애플리케이션을 신속하게 개발하고 손쉽게 유지할 수 있도록 합니다. HTTP API는 단일 목적 마이크로서비스에서 포괄적인 모놀리식에 이르는 애플리케이션의 규모에 관계없이 공통의 인터페이스를 제공합니다. HTTP를 이용함으로써 하이퍼스케일(hyperscale) 인터넷 속성들을 지원하는 웹 애플리케이션 딜리버리의 개선 기능들이 안정적인 고성능 API 딜리버리를 수행하는 데 사용될 수 있습니다.

마이크로서비스 애플리케이션을 위한 API 게이트웨이의 중요성은 NGINX 블로그의 마이크로서비스 구축: API 게이트웨이 활용하기 (Microservices: Using an API Gateway)에서 자세하게 소개하고 있습니다.

업계 선도적인 고성능, 경량의 리버스 프록시 및 로드 밸런서인 NGINX Plus는 API 트래픽을 처리하는 데 필요한 고급 HTTP 처리 기능들을 제공합니다. 이를 통해 NGINX Plus는 API 게이트웨이를 개발하기 위한 최적의 플랫폼으로 자리매김하고 있습니다. 이 블로그 게시글에서는 많은 일반적인 API 게이트웨이 활용 사례를 설명하고, 효율적이고 확장 가능하며 쉽게 유지할 수 있는 방식으로 이를 처리하도록 NGINX Plus를 구성하는 방법을 소개합니다. 또한, 운영 환경의 토대를 형성할 수 있는 완벽한 구성을 설명합니다.

주: 별도로 언급한 경우를 제외하고 이 장에 포함된 모든 정보는 NGINX Plus 및 NGINX Open Source 모두에 해당됩니다.

Warehouse API 소개

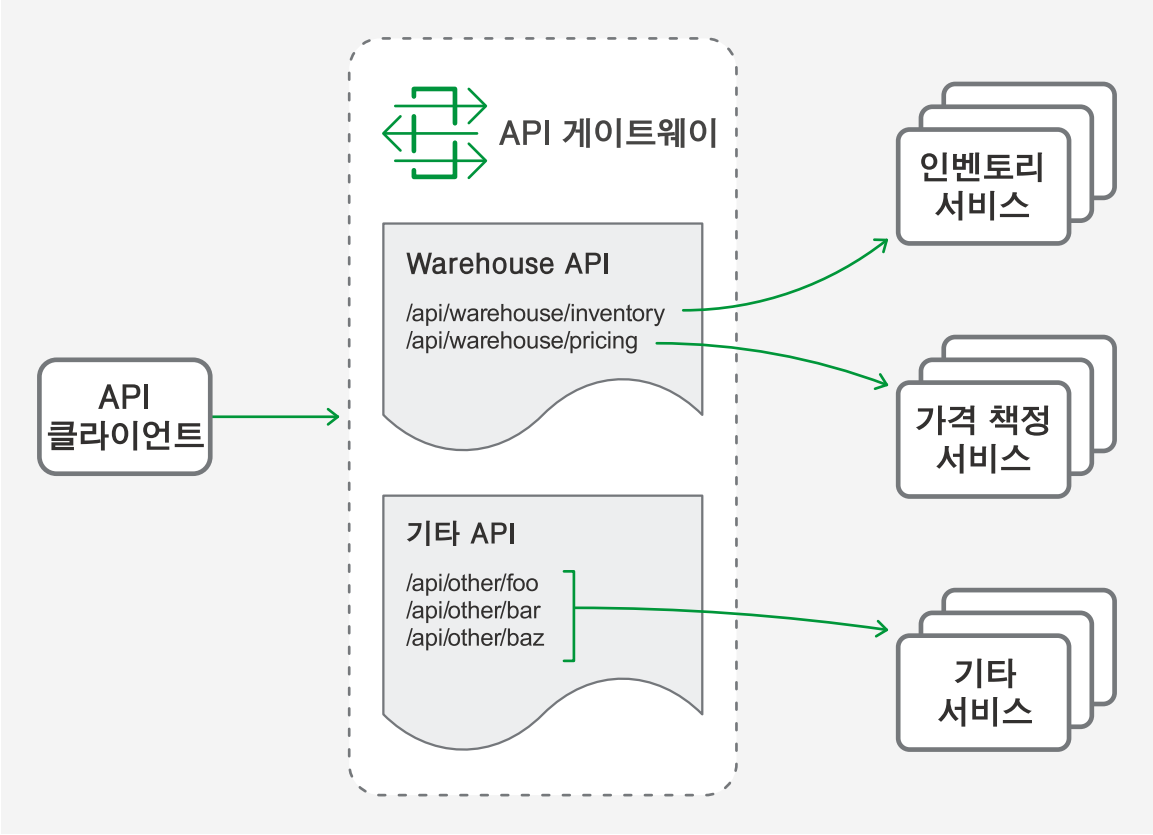
API 게이트웨이의 주된 기능은 백엔드에서 구현 또는 구축된 방법에 상관없이 다수의 API를 위한 일관된 단일 진입점을 제공하는 것입니다. 모든 API들은 마이크로서비스 애플리케이션이 아닙니다. API 게이트웨이는 마이크로서비스로의 부분적인 전환이 진행되는 기존 API, 모놀리식 및 애플리케이션을 관리해야 합니다.

이 블로그 게시글에서는 인벤토리 관리를 위한 가상 API를 "Warehouse API"라고 부릅니다. 여러 다양한 활용 사례를 설명하기 위해 샘플 구성 코드를 사용합니다. Warehouse API는 JSON 요청을 수신(comsume)하고 JSON 응답을 생성(produce)하는 RESTful API입니다. 하지만, API 게이트웨이로서 구축할 때, JSON 사용이 NGINX Plus의 제약 조건이나 요구 사항은 아닙니다. NGINX Plus는 API 자체에 의해 사용되는 아키텍처 스타일 및 데이터 형식에 독립적입니다.

Warehouse API는 개별적인 마이크로서비스의 집합으로 구현되며 단일 API로서 퍼블리싱됩니다. 인벤토리 및 가격 책정 리소스는 별도의 서비스로서 구현되고 다른 백엔드에 구축됩니다. 따라서, API의 path 구조는 다음과 같습니다.

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

예를 들어, 현재의 물류 창고 인벤토리를 쿼리하기 위해 클라이언트 애플리케이션은 **/api/warehouse/inventory**에 대한 HTTP GET 요청을 만듭니다.



다수의 애플리케이션을 위한 API 게이트웨이 아키텍처

NGINX 구성 체계화

NGINX Plus를 API 게이트웨이로서 사용하는 데 따른 이점 중 하나는 그 역할을 수행하는 동시에 리버스 프록시, 로드 밸런서 및 기존 HTTP 트래픽을 위한 웹 서버의 역할도 수행할 수 있다는 것입니다. 만약 NGINX Plus가 이미 애플리케이션 딜리버리 스택에 포함되어 있다면, 일반적으로 별도의 API 게이트웨이를 구축할 필요가 없습니다. 하지만, API 게이트웨이에서 기대하는 몇몇 기본 동작들은 브라우저 기반 트래픽에서 기대하는 것과는 다릅니다. 이 때문에 API 게이트웨이 구성과 브라우저 기반 트래픽을 위한 모든 기존(또는 향후) 구성을 구분하고 있습니다.

이러한 분리를 위해 다목적 NGINX Plus 인스턴스를 지원하는 구성 레이아웃을 작성하고 CI/CD 파이프라인을 통해 구성 배포를 자동화하는 편리한 구조를 제공합니다. **/etc/nginx** 아래의 최종 디렉토리 구조는 다음과 같습니다.

```
etc/
└── nginx/
    ├── api_conf.d/ ..................................  API별 구성을 위한 하위 디렉토리
    │   └── warehouse_api.conf ......                   Warehouse API의 정의 및 정책
    ├── api_backends.conf ...................           백엔드 서비스 (upstream)
    ├── api_gateway.conf ....................           API 게이트웨이 서버의 최상위 레벨 구성
    │
    ├── api_json_errors.conf ...........                JSON 형식의 HTTP 에러 응답
    ├── conf.d/
    │   ├── ...
    │   └── existing_apps.conf
    └── nginx.conf
```

모든 API 게이트웨이 구성과 관련한 디렉토리 및 파일명 앞에는 **api_**가 붙어 있습니다. 이들 각 파일 및 디렉토리는 API 게이트웨이의 여러 다양한 특징 및 기능을 지원하며 이는 아래에서 자세하게 설명하고 있습니다.

Top-Level API 게이트웨이 정의

모든 NGINX 구성은 메인 구성 파일인 **nginx.conf**에서 시작합니다. API 게이트웨이 구성을 입력하기 위해 게이트웨이 구성을 포함한 파일인 **api_gateway.conf**(바로 아래 라인 28)를 참조하는 **nginx.conf**의 **http** 블록에 `include` 지시문(directive)을 추가합니다. 기본 **nginx.conf** 파일은 **conf.d** 하위 디렉토리(라인 29)에서 브라우저 기반 HTTP 구성을 가져오기 위해 include 지시문(directive)을 사용한다는 점에 유념하십시오. 이 장에서는 가독성을 돕고 구성의 일부를 자동화하기 위해 include 지시문(directive)을 광범위하게 사용합니다.

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

GitHub의 raw 보기

api_gateway.conf 파일은 NGINX Plus를 API 게이트웨이로서 클라이언트에게 표시하는 가상 서버를 정의합니다. 이 구성은 단일 진입점인 **https://api.example.com/** (라인 13)에서 API 게이트웨이에 의해 퍼블리싱되고 라인 16에서 21까지에서 구성된 TLS에 의해 보호되는 모든 API를 표시합니다. 이 구성은 순수하게 HTTPS이며 평문(plaintext) HTTP 리스너는 없습니다. 여기에서는 API 클라이언트가 정확한 진입점을 알고 있으며 기본값으로 HTTPS 연결이 설정되어 있다고 간주합니다.

```
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 }
```

GitHub의 raw 보기

이 구성은 정적(static) 모드를 위한 것입니다. 개별 API와 그 백엔드 서비스의 세부 사항은 라인 24의 **include** 지시문(directive)에 의해 참조된 파일에 명시되어 있습니다. 라인 27에서 30까지는 로깅 기본값과 에러 처리에 대해 다루며, 이 장의 뒷부분에 나오는 에러에 응답하는 방법에서 더 자세하게 다룰 것입니다.

단일 서비스 vs. 마이크로서비스 API 백엔드

탄력성이나 로드 밸런싱을 이유로 일반적으로 2개 이상의 백엔드에 구현될 것으로 예상하지만, 일부 API는 단일 백엔드에서 구현될 수도 있습니다. 마이크로서비스 API를 이용해 각 서비스를 위한 개별 백엔드를 정의하고 이들은 함께 완벽한 API의 기능을 수행합니다. 여기에서 Warehouse API는 2개의 개별 서비스로서 구축되며 각각 다수의 백엔드를 갖습니다.

```
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;
}
```

GitHub의 raw 보기

API 게이트웨이에 의해 퍼블리싱된 모든 API를 위한 모든 백엔드 API 서비스들은 **api_backends.conf**에서 정의됩니다. 여기에서는 각 `upstream` 블록에서 다수의 IP 주소-포트 쌍을 이용해 API 코드가 배포된 위치를 표시하지만, 호스트 이름(hostname) 또한 사용될 수 있습니다. NGINX Plus 가입자들은 동적 DNS 로드 밸런싱을 활용해 런타임 구성에 새로운 백엔드를 자동으로 추가할 수 있습니다.

Warehouse API 정의

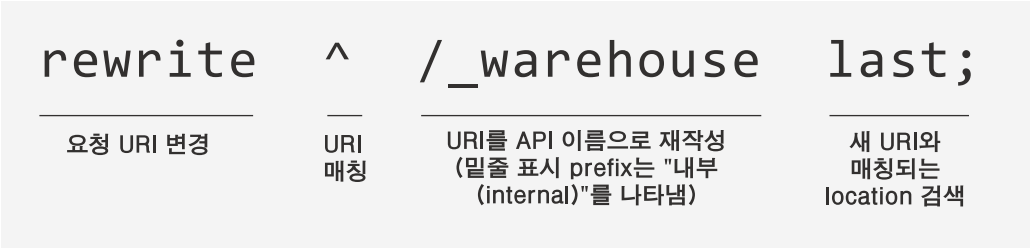
구성에서 먼저 Warehouse API에 유효한 URI를 정의한 다음, Warehouse API에 대한 요청을 처리하는 공통의 정책을 정의합니다.

```
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 }
```

GitHub의 raw 보기

Warehouse API는 많은 location 블록으로 정의됩니다. NGINX Plus는 요청 URI를 구성의 한 섹션에 매칭시킬 수 있는 매우 효율적이고 유연한 시스템입니다. 일반적으로 요청은 가장 구체적인 path prefix와 매칭되며 location 지시문(directive)의 순서는 중요하지 않습니다. 여기에서는 라인 3과 8에서 2개의 path prefix를 정의합니다. 각각의 경우, $upstream 변수는 백엔드 API 서비스 또는 인벤토리 및 가격 책정 서비스 각각을 위한 서비스를 나타내는 upstream 블록의 이름으로 설정됩니다.

이 구성의 목표는 API 정의와 API 제공 방법을 관리하는 정책을 분리시키는 것입니다. 이를 위해 API 정의 섹션에 표시되는 구성을 최소화합니다. 각 location에 대해 적절한 upstream 그룹을 결정한 다음, 처리를 중단하고 `rewrite` 지시문(directive)을 사용해 해당 API에 대한 정책을 찾습니다(라인 10).



API 정책 섹션으로 처리하기 위해 rewrite 지시문(directive) 사용

`rewrite` 지시문(directive)의 결과는 NGINX Plus가 **/_warehouse**로 시작하는 URI와 매칭되는 `location` 블록을 검색하는 것입니다. 라인 15의 `location` 블록은 완전 일치 매칭(exact match)을 실행하기 위해 = 수정자(modifier)를 사용하며, 이를 통해 처리 속도가 높아집니다.

이 단계에서 정책 섹션은 매우 단순합니다. `location` 블록 자체는 라인 16에서 내부(internal)로 표시되며, 이는 클라이언트가 이를 직접 요청할 수 없다는 것을 의미합니다. `$api_name` 변수는 로그 파일에서 올바르게 나타나도록 하기 위해 API 이름과 일치하도록 재정의됩니다. 마지막으로 요청은 수정되지 않은 원래의 요청 URI를 포함하고 있는 `$request_uri` 변수를 사용해 API 정의 섹션에 지정된 upstream 그룹으로 프록시됩니다.

API에 대한 광역의(Broad) 정의 vs. 정교한(Precise) 정의

API에 대한 정의에는 광역의 정의와 정교한 정의의 2가지 접근방식이 있습니다. 각 API에 가장 적합한 접근방식은 API의 보안 요구 사항과 백엔드 서비스가 유효하지 않은 URI을 처리하는 것이 바람직한 지 여부에 따라 결정됩니다.

warehouse_api_simple.conf의 라인 3과 8에서 URI prefix를 정의함으로써 Warehouse API에 대한 광역의 접근방식을 사용합니다. 이는 이들 두 prefix로 시작하는 모든 URI는 적합한 백엔드 서비스로 프록시된다는 것을 의미합니다. prefix 기반의 location 매칭을 실행하는 경우, 다음 URI에 대한 API 요청은 모두 유효합니다.

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

각 요청을 정확한 백엔드 서비스로 프록시하는 것이 유일한 고려 사항이라면, 광역의 접근방식이 가장 빠른 처리와 가장 간결한 구성을 제공합니다. 반면에, 정교한 접근방식은 각 가용 API 리소스에 대한 URI path를 명시적으로 정의함으로써 API 게이트웨이가 API의 전체 URI 공간을 이해할 수 있도록 합니다. 정교한 접근방식을 취하는 경우, Warehouse API를 위한 다음 구성은 완전 일치 매칭(=) 및 정규식(~)의 조합을 이용해 각각의 혹은 모든 URI를 정의합니다.

```
location = /api/warehouse/inventory { # Complete inventory
    set $upstream inventory_service;
    rewrite ^ /_warehouse last;
}

location ~ ^/api/warehouse/inventory/shelf/[^/]*$ { # Shelf inventory
    set $upstream inventory_service;
    rewrite ^ /_warehouse last;
}

location ~ ^/api/warehouse/inventory/shelf/[^/]*/box/[^/]*$ { # Box on shelf
    set $upstream inventory_service;
    rewrite ^ /_warehouse last;
}

location ~ ^/api/warehouse/pricing/[^/]*$ { # Price for specific item
    set $upstream pricing_service;
    rewrite ^ /_warehouse last;
}
```

GitHub의 raw 보기

이 구성은 다소 장황하지만, 백엔드 서비스에 의해 구현된 리소스를 보다 정확하게 설명합니다. 이는 정규식 매칭을 위한 약간의 추가 오버헤드를 대가로 이상(malformed) 클라이언트 요청으로부터 백엔드 서비스를 보호할 수 있다는 이점이 있습니다.
이 구성을 적용하면, NGINX Plus는 일부 URI만을 허용하고 유효하지 않은 URI는 거부합니다.

유효한 URI	유효하지 않은 URI
/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

정교한 API 정의를 사용하면, 기존 API 문서 형식으로 API 게이트웨이의 구성을 실행할 수 있습니다. OpenAPI Specification(이전의 Swagger)에서 NGINX Plus API 정의를 자동화할 수 있습니다. 이를 위한 샘플 스크립트가 이 장을 통해 제공됩니다.

클라이언트 요청 재작성

API가 발전하면서 클라이언트 업데이트가 필요한 중요한 변경 사항들이 때때로 발생합니다. 그 예로, API 리소스의 이름이 변경되거나 제거된 경우를 들 수 있습니다.
웹 브라우저와 달리, API 게이트웨이는 자체 클라이언트에 새 location을 지정하는 redirect (코드 `301`)를 보낼 수 없습니다. 다행히 API 클라이언트를 수정하는 것이 비현실적인 경우, 즉시 클라이언트 요청을 재작성할 수 있습니다.

아래 예제에서 라인 3을 보면 가격 책정 서비스가 인벤토리 서비스의 일부로서 먼저 구현되어 있는 것을 볼 수 있습니다. **rewrite** 지시문(directive)은 이전의 가격 책정 리소스에 대한 요청을 새 가격 책정 서비스에 대한 요청으로 변환합니다.

```
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 }
```

GitHub의 raw 보기

즉시 URI를 재작성할 수 있다는 것은 궁극적으로 라인 26의 요청을 프록시하는 경우 (warehouse_api_simple.conf의 라인 21에서 수행한 것처럼), 더 이상 `$request_uri` 변수를 사용할 수 없다는 것을 의미합니다. 따라서, 처리를 정책 섹션으로 전환할 때 URI을 보존하기 위해서는 API 정의의 라인 9와 14에서 약간 다른 rewrite 지시문(directive)을 사용해야 합니다.

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

요청 URI 변경	원본 URI 전체 캡처	URI를 식별하기 위해 원본 URI($1) 앞에 /_warehouse를 추가해 URI 재작성	새 URI와 매칭되는 location 검색

URI를 보존하면서 API 정책 섹션으로 처리를 위한 *rewrite* 지시문(directive) 사용

에러에 응답하는 방법

HTTP API와 브라우저 기반 트래픽 간의 중요한 차이점 중 하나는 에러를 클라이언트에 알리는 방법입니다. NGINX Plus가 API 게이트웨이로서 구축된 경우, API 클라이언트에 가장 적합한 방식으로 에러를 반환하도록 구성합니다.

최상위 API 게이트웨이 구성은 에러 응답을 어떻게 처리하는지에 대한 방법을 정의한 섹션을 포함하고 있습니다.

```
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
```

GitHub의 raw 보기

라인 27의 `error_page` 지시문(directive)은 요청이 API 정의와 일치하지 않는 경우, NGINX Plus가 default **404** (Not Found) 에러가 아니라 **400** (Bad Request) 에러를 반환하도록 지정합니다. 이러한 (선택적) 동작을 위해서는 API 클라이언트가 API 문서에 포함된 유효한 URI만 요청해야 하며 승인받지 않은 클라이언트가 API 게이트웨이를 통해 퍼블리싱된 API의 URI 구조를 발견하지 못하도록 해야 합니다.

라인 28은 백엔드 서비스 자체에서 생성된 에러를 나타냅니다. 처리되지 않은 예외(Unhandled exception)는 스택 트레이스(stack trace)나 클라이언트에게 전송되기를 원치않는 다른 민감한 데이터를 포함할 수 있습니다. 이 구성은 클라이언트에 표준화된 에러를 전송함으로써 보안 수준을 한층 강화합니다.

에러 응답의 전체 리스트는 라인 29의 `include` 지시문(directive)이 참조하는 별도의 구성 파일에 정의되어 있으며, 처음 몇 라인은 다음과 같습니다. 다른 에러 형식을 선호하는 경우, **api_gateway.conf**의 라인 30에서 `default_type` 값이 일치하도록 변경해 이 파일을 수정할 수 있습니다. 또한 각 API의 정책 섹션에 별도의 `include` 지시문(directive)을 사용함으로써 기본 값과는 다른 다양한 에러 응답들을 정의할 수 있습니다.

```
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'; }
```

GitHub의 raw 보기

이 구성을 적용하면, 유효하지 않은 URI에 대한 클라이언트 요청은 다음과 같은 응답을 받게 됩니다.

```
$ 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"}
```

인증 구현

어떤 형태로든 API를 보호하는 인증 절차 없이 API를 퍼블리싱하는 경우는 거의 없습니다. NGINX Plus는 API를 보호하고 API 클라이언트를 인증하는 여러 접근방식을 제공합니다. IP 주소 기반 ACL(Access Control Lists), 디지털 증명서 인증 및 HTTP 기본 인증 등에 대한 정보는 문서를 참조하십시오. 여기에서는 API 기반의 인증 방식에 중점을 두고 있습니다.

API Key 인증

API key는 클라이언트와 API 게이트웨이가 알고 있는 공유된 비밀입니다. 이는 장기적인 인증 정보로서 기본적으로 API 클라이언트에 발행된 길고 복잡한 암호입니다. API key를 생성하는 것은 간단합니다. 아래 예제와 같이 난수를 인코딩하면 됩니다.

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

최상위 API 게이트웨이 구성 파일인 **api_gateway.conf**의 라인 6에는 **api_keys.conf** 라는 파일이 있습니다. 이 파일은 각 API 클라이언트를 위한 API key를 포함하고 있으며 클라이언트의 이름이나 다른 설명을 통해 식별됩니다.

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

GitHub의 raw 보기

API key는 `map` 블록 안에 정의됩니다. `map` 지시문(directive)에는 2개의 매개변수가 있습니다. 첫번째 매개변수는 `$http_apikey` 변수에 캡처된 클라이언트 요청의 `apikey` HTTP 헤더에서 API key를 찾을 위치를 정의합니다. 두번째 매개변수는 새 변수(`$api_client_name`)를 생성하고, 첫번째 매개변수가 이 key와 일치하는 라인에서 두번째 매개변수의 값으로 설정합니다.

예를 들어, 클라이언트가 API key `7B5zIqmRGXmrJTFmKa99vcit`를 제시하면, `$api_client_name` 변수는 `client_one`으로 설정됩니다. 이 변수는 인증된 클라이언트를 검사하는 데 사용될 수 있으며 보다 상세한 감사(auditing)를 위해 로그 항목에 포함됩니다.

`map` 블록의 형식은 단순하며 기존 인증 정보 저장소에서 **api_keys.conf**를 생성하는 자동화 워크플로우로 쉽게 통합됩니다. API key 인증은 각 API의 정책 섹션에 의해 실행됩니다.

```
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 }
```

GitHub의 raw 보기

클라이언트는 `apikey` HTTP 헤더에 고유의 API key를 나타내야 합니다. 이 헤더가 누락되거나 공백(라인 20)인 경우, 클라이언트에 `401` 응답을 전송해 인증이 필요하다는 것을 알립니다. 라인 23은 API key가 `map` 블록 내 그 어떤 key와도 일치하지 않는 경우를 처리합니다. 이 경우, **api_keys.conf**에서 라인 2의 **기본** 매개변수가 `$api_client_name`을 빈 문자열로 설정하고 403 응답을 보내 클라이언트에게 인증이 실패했음을 알립니다.

이 구성을 적용하면, Warehouse API는 이제 API key 인증을 실행합니다.

```
$ 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 인증

JSON Web Token(JWT)은 API 인증에 점차 많이 사용되고 있습니다. 네이티브 JWT 지원은 NGINX Plus에서만 제공되며 NGINX 블로그에 게재된 "JWT 및 NGINX Plus를 이용한 API 클라이언트 인증"에서 설명한 대로 JWT의 유효성을 검사할 수 있습니다. 샘플 구현의 경우, 제2장 특정 메소드에 대한 액세스 제어를 참조하십시오.

요약

이 장에서는 NGINX Plus를 API 게이트웨이로서 구축하는 완벽한 해법을 자세하게 설명했습니다. 이 블로그에서 논의된 전체 파일들은 NGINX의 GitHub Gist repo에서 검토하고 다운로드할 수 있습니다.

2 백엔드 서비스 보호

속도 제한

브라우저 기반 클라이언트와 달리, 개별 API 클라이언트는 API에 막대한 부하를 가중시킬 수 있으며, 심지어 시스템 리소스의 상당 부분을 소모하여 다른 API 클라이언트들은 사실상 lock out 상태에 빠지게 될 수도 있습니다. 악의적인 클라이언트들만이 이러한 위협을 가하는 것은 아닙니다. 오작동하거나 버그가 있는 API 클라이언트들이 백엔드를 마비시키는 루프를 시작할 수 있습니다. 이를 방지하기 위해 각 클라이언트의 정당한 사용을 보장하고 백엔드 서비스의 리소스를 보호할 수 있도록 속도 제한을 적용합니다.

NGINX Plus는 요청의 모든 속성을 기초로 속도 제한을 적용할 수 있습니다. 클라이언트 IP 주소가 일반적으로 사용되지만, API에 대한 인증이 활성화되면, 인증된 클라이언트 ID는 더 신뢰할 수 있고 정확한 속성입니다.

속도 제한 자체는 최상위 레벨 API 게이트웨이 구성에서 정의되며 이후 전역적으로 적용되거나, API 또는 심지어 URI별로 적용될 수 있습니다.

```
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
```

GitHub의 raw 보기

이 예제에서, 라인 8의 `limit_req_zone` 지시문(directive)은 각 클라이언트 IP 주소 (`$binary_remote_addr`)에 대해 초당 10회 속도의 요청 제한을 정의하며 라인 9에서는 각 인증 클라이언트 ID (`$http_apikey`)에 대해 초당 200개의 요청 제한을 정의합니다. 이는 적용되는 위치에 관계없이 다수의 속도 제한을 정의하는 방법을 보여줍니다. 한 API가 동시에 다수의 속도 제한을 적용하거나 여러 리소스에 대해 서로 다른 속도 제한을 적용할 수 있습니다.

아래의 구성 스니펫(snippet)에서는 `limit_req` 지시문(directive)을 이용해 제1장에서 설명한 "Warehouse API"의 정책 섹션에 첫번째 속도 제한을 적용합니다. 기본적으로 NGINX Plus는 속도 제한을 초과하는 경우, `503 (Service Unavailable)` 응답을 전송합니다. 하지만, 이는 API 클라이언트가 속도 제한을 초과했다는 것을 분명하게 이해하고 자신의 행동을 수정할 수 있도록 한다는 점에서 유용합니다. 이를 위해 `limit_req_status` 지시문(directive)을 사용해 `429 (Too Many Requests)` 응답을 대신 전송합니다.

```
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 }
```

GitHub의 raw 보기

`limit_req` 지시문(directive)에 추가 매개변수를 사용하여 NGINX Plus가 속도 제한을 적용하는 방법을 세밀하게 조정할 수 있습니다. 예를 들어, 속도 제한을 초과하면, 바로 요청을 거부하는 것이 아니라 대기열에 추가함으로써, 요청 속도가 지정된 제한 아래로 떨어질 때까지 기다리도록 합니다. 속도 제한을 세밀하게 조정하는 데 대한 자세한 정보는 NGINX 블로그의 "NGINX 및 NGINX Plus를 이용한 속도 제한"을 참조하십시오.

특정 요청 메소드 적용

RESTful API에서 HTTP 메소드(또는 verb로도 불림)는 각 API 호출의 중요한 부분이며, API 정의에 있어 매우 중요합니다. 웨어하우스 API의 가격 책정 서비스를 예로 들어 보겠습니다.

- **`GET /api/warehouse/pricing/item001`**의 가격 반환
- **`PATCH /api/warehouse/pricing/item001`**의 가격 변경

가격 책정 서비스에 대한 요청에서 이들 2개의 HTTP 메소드만(그리고 인벤토리 서비스에 대한 요청에서 `GET` 메소드만) 수용하도록 Warehouse API의 정의를 업데이트할 수 있습니다.

```
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 }
```

GitHub의 raw 보기

이 구성을 적용하면, 라인 4에 나열된 것 이외의 가격 책정 서비스에 대한 요청(및 라인 13에 나열된 것 이외의 나머지 서비스에 대한 요청)은 거부되고 백엔드 서비스로 전달되지 않습니다. 아래 콘솔 트레이스(console trace)에서 볼 수 있듯이 NGINX Plus는 `405 (Method Not Allowed)` 응답을 전송해 API 클라이언트에게 에러의 정확한 특성을 알립니다.

최소 공개 보안 정책이 필요한 경우, error_page 지시문(directive)을 사용해 이러한 응답을 400 (Bad Request)와 같은 less informative 에러로 변환할 수 있습니다.

```
$ 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"}
```

세분화된 액세스 제어 적용

제1장에서는 API key 및 JSON Web Token(JWT) 등과 같은 인증 옵션을 실행함으로써 승인받지 않은 액세스로부터 API를 보호하는 방법을 설명했습니다. 인증된 ID 또는 인증된 ID의 속성을 사용해 세분화된 액세스 제어를 실행할 수 있습니다.

여기에서는 2개 예제를 들어 보겠습니다. 첫번째 예제는 제1장에서 제공된 구성을 확장하고, API 클라이언트 화이트리스트를 이용해 API key 인증을 기준으로 특정 API 리소스에 대한 액세스를 제어하는 것입니다. 두번째 예제는 NGINX Plus가 클라이언트에서 허용하는 HTTP 메소드를 제어하기 위해 커스텀 클레임(custom claim)을 사용해 제1장에서 언급한 JWT 인증 메소드를 구현하는 것입니다. 물론, 모든 NGINX Plus 인증 메소드를 이들 예제에 적용할 수 있습니다.

특정 리소스에 대한 액세스 제어

오직 "인프라 클라이언트"만 Warehouse API 인벤토리 서비스의 감사 리소스에 액세스하는 것을 허용하려고 한다고 가정해 보겠습니다. API key 인증이 활성화된 경우, `map` 블록을 사용해 인프라 클라이언트 이름의 화이트리스트를 작성함으로써 해당 API key가 사용될 때 변수 `$is_infrastructure`의 값이 `1`이 되도록 합니다.

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

GitHub의 raw 보기

Warehouse API의 정의에서, 라인 13부터 19까지에 인벤토리 감사 리소스를 위한 `location` 블록을 추가합니다. `if` 블록은 인프라 클라이언트만 해당 리소스에 액세스할 수 있다는 것을 보장합니다.

```
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 }
```

GitHub의 raw 보기

라인 13의 `location` 지시문(directive)은 = (등호) 수정자를 사용해 감사 리소스에서 완전 일치 매칭(exact match)을 실행합니다. 완전 일치 매칭은 다른 리소스에 사용되는 기본 path-prefix 정의보다 우선합니다. 아래 트레이스(trace)는 이 구성을 적용하여 화이트리스트에 포함되어 있지 않은 클라이언트는 인벤토리 감사 리소스에 액세스할 수 없도록 하는 방법을 보여줍니다. 제시된 API key는 `client_tow` (제1장에서 정의한 바와 같이)에 속합니다.

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

특정 메소드에 대한 액세스 제어

위에서 정의한 바와 같이 가격 책정 서비스는 각각 클라이언트가 특정 품목의 가격을 입수하고 수정할 수 있도록 하는 GET 및 PATCH 메소드를 수용합니다. (또한, POST 및 DELETE 메소드를 허용하도록 설정해 가격 책정 데이터의 전체 라이프사이클 관리를 수행할 수 있습니다.) 이 섹션에서는 활용 사례를 확장해 특정 사용자가 발행할 수 있는 메소드를 제어합니다. Warehouse API에 JWT 인증을 활성화하면, 각 클라이언트에 대한 권한은 커스텀 클레임으로서 인코딩됩니다. 가격 책정 데이터를 변경할 수 있는 권한을 가진 관리자들에게 발행된 JWT는 클레임 `"admin":true`를 포함하고 있습니다.

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

GitHub의 raw 보기

api_gateway.conf의 하단에 추가된 이 `map` 블록은 모든 가능한 HTTP 메소드를 `READ` 또는 `WRITE`로 평가하는 새로운 변수인 `$request_type`으로 결합시킵니다. 아래 스니펫에서는 $request_type 변수를 사용해 요청을 적절한 Warehouse API 정책, `/_warehouse_READ` 또는 `/_warehouse_WRITE`로 보냅니다.

```
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 }
```

GitHub의 raw 보기

라인 9와 14의 `rewrite` 지시문(directive)은 `$request_type` 변수를 Warehouse API 정책 이름에 추가하며, 그 결과, 정책 섹션이 2개로 나뉩니다. 이제 read 및 write 작업에 서로 다른 정책이 적용됩니다.

```
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 }
```

GitHub의 raw 보기

/_warehouse_READ 및 /_warehouse_WRITE 정책 모두 클라이언트가 유효한 JWT를 제시하도록 요구합니다. 하지만, WRITE 메소드(POST, PATCH 또는 DELETE)를 사용하는 요청의 경우, JWT에 클레임 "admin":true (라인 35)이 포함되어야 합니다. 각 요청 메소드마다 개별 정책을 갖는 이러한 접근 방식은 인증에만 국한되지 않습니다. 속도 제한, 로깅 및 다른 백엔드로의 라우팅 등 메소드별로 다른 컨트롤을 적용할 수 있습니다.

JWT 인증은 NGINX Plus에서만 제공됩니다.

요청 사이즈 제어

HTTP API는 일반적으로 요청 본문을 사용해 백엔드 API 서비스가 처리할 명령과 데이터를 포함합니다. 이는 XML/SOAP API는 물론, JSON/REST API에도 적용됩니다. 결과적으로 요청 본문은 백엔드 API 서비스에 대한 공격 경로를 제공할 수 있으며, 이는 대용량 요청 본문을 처리할 때 버퍼 오버플로우 공격에 취약할 수 있습니다.

기본적으로, NGINX Plus는 1 MB를 넘는 본문을 포함한 요청을 거부합니다. 이미지 처리와 같이 대용량 페이로드를 처리하는 API의 경우에는 이 값을 늘릴 수 있지만, 대부분의 API들은 낮은 값으로 설정하고 있습니다.

```
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 }
```

GitHub의 raw 보기

라인 19의 `client_max_body_size` 지시문(directive)은 요청 본문의 사이즈를 제한합니다. 이 구성을 적용하면, 가격 책정 서비스에 대한 2개의 서로 다른 PATCH 요청을 수신하는 API 게이트웨이의 동작을 비교할 수 있습니다. 첫번째 `curl` 명령은 작은 크기의 JSON 데이터를 전송하는 반면, 두번째 명령은 대용량 컨텐츠 파일 (**/etc/services**)을 전송하려고 시도합니다.

```
$ 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"}
```

요청 본문 유효성 검증

백엔드 API 서비스는 대용량 요청 본문을 포함한 버퍼 오버플로우 공격에 취약할 뿐만 아니라 유효하지 않거나 예상하지 못한 데이터가 본문에 포함될 가능성이 높습니다. 요청 본문에 올바른 형식의 JSON을 요구하는 애플리케이션의 경우, NGINX JavaScript 모듈을 사용해 백엔드 API 서비스로 프록시하기 전에 JSON 데이터가 에러 없이 파싱되는지를 확인할 수 있습니다.

JavaScript 모듈을 설치한 경우, `js_include` 지시문(directive)을 사용해 JSON 데이터의 유효성을 검증하는 함수를 위한 JavaScript 코드가 포함된 파일을 참조합니다.

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

GitHub의 raw 보기

`js_set` 지시문(directive)은 새로운 변수 `$validated`를 정의하며 이는 `json_validator` 함수를 호출해 평가됩니다.

```
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 }
```

GitHub의 raw 보기

`json_validator` 함수는 `JSON.parse` 메소드를 이용해 요청 본문을 파싱하려고 시도합니다. 성공하면, 이 요청을 위해 의도한 upstream 그룹의 이름이 반환됩니다(라인 6). 요청 본문을 파싱할 수 없는 경우(예외로 인해), 로컬 서버 주소가 반환됩니다(라인 9). `return` 지시문(directive)은 `$validated` 변수를 입력합니다. 따라서 이 변수를 이용해 요청을 어디로 보내야 하는지 결정할 수 있습니다.

```
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 }
```

GitHub의 raw 보기

Warehouse API의 정책 섹션에서 라인 24의 `proxy_pass` 지시문(directive)을 수정합니다. 이는 이전과 같이 요청을 백엔드 API 서비스로 전달하지만, 이제 `$validated` 변수를 목적지 주소로서 사용합니다. 클라이언트 본문이 JSON으로 파싱되면, 평상시처럼 upstream 그룹을 프록시합니다. 하지만, 예외가 있는 경우, `127.0.0.1:10415`의 반환된 값을 이용해 클라이언트에게 에러 응답을 전송합니다.

```
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 }
```

GitHub의 raw 보기

요청이 이 가상 서버로 프록시되면, NGINX Plus는 클라이언트에게 `415 (Unsupported Media Type)` 응답을 전송합니다.

이러한 전체 구성을 적용하면, NGINX Plus는 올바른 형식의 JSON 본문을 포함한 경우에만 요청을 백엔드 API 서비스로 프록시합니다.

```
$ 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"}
```

$request_body 변수에 대한 주의 사항

JavaScript 함수 `json_validator`는 `$request_body` 변수를 사용해 JSON 파싱을 실행합니다. 하지만, NGINX Plus는 이 변수를 기본적으로 입력하지는 않으며 중간 사본을 만들지 않고 해당 요청을 바로 백엔드로 스트리밍합니다. Warehouse API 정책 섹션의 `mirror` 지시문(directive)(라인 19)을 이용해 클라이언트 요청의 사본을 작성하면, `$request_body` 변수가 입력됩니다.

```
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 }
```

GitHub의 raw 보기

라인 20 및 21의 지시문(directive)들은 NGINX Plus가 내부적으로 요청 본문을 처리하는 방법을 제어합니다. 요청 본문이 디스크에 기록되지 않도록 `client_body_buffer_size`를 `client_max_body_size`와 동일한 사이즈로 설정합니다. 이는 I/O 작업을 최소화해 전반적인 성능을 향상시키지만, 메모리 사용량이 증가하게 됩니다. 작은 요청 본문을 이용하는 대부분의 API 게이트웨이 활용 사례의 경우, 이는 적절한 절충안이 될 수 있습니다.

위에서 언급했듯이 `mirror` 지시문(directive)은 클라이언트 요청의 사본을 만듭니다. `$request_body`를 입력하는 것 이외에는 이 사본이 필요하지 않기 때문에 이를 최상위 API 게이트웨이 진입점에서 정의한 "dead end" 위치(`/_NULL`)로 보냅니다.

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

GitHub의 raw 보기

이 위치는 `204 (No Content)` 응답을 전송하는 역할을 수행하는 데 불과합니다. 이 응답은 미러링된 요청과 관련되어 있기 때문에 무시되며, 따라서 원래의 클라이언트 요청을 처리하는 데 발생하는 오버헤드는 무시할 수 있는 수준입니다.

요약

이 장에서는 운영 환경에서 악의적인 클라이언트와 오작동하는 클라이언트로부터 운영 환경의 백엔드 API 서비스를 보호하는 과제에 중점을 두었습니다. NGINX Plus는 오늘날 가장 많은 트래픽이 발생하는 인터넷 사이트의 작동 및 보호에 사용되는 API 트래픽을 관리하는 데 동일한 기술을 적용합니다.

3 gRPC 서비스 퍼블리싱

최근 마이크로서비스 애플리케이션 아키텍처의 개념과 이해에 관련한 많은 자료들이 나와 있으며, 특히 NGINX 블로그에 가장 많은 자료들이 올라와 있습니다.
마이크로서비스 애플리케이션의 핵심은 HTTP API이며 앞서 두 장에 걸쳐 가상 REST API를 사용해 NGINX Plus가 이러한 형태의 애플리케이션을 처리하는 방법을 살펴봤습니다.

최신 애플리케이션을 위한 JSON 메시지 형식의 REST API가 인기를 얻고 있지만, 모든 시나리오, 또는 모든 조직에 적합한 접근 방식은 아닙니다.
가장 일반적인 과제는 다음과 같습니다.

- **표준화된 문서** – 체계적인 개발자 교육이나 필수 문서의 요구 사항이 없다면, 정확한 정의가 결여된 많은 REST API들로 끝나기 마련입니다. Open API Specification은 REST API를 위한 일반적인 인터페이스 설명 언어로서 부상했지만, 그 사용은 선택적이며 개발 조직 내에서 엄격한 관리가 이루어져야 합니다.
- **이벤트 및 오래 유지되는 커넥션(long-lived connection)** – REST API와 전송 방식으로 HTTP를 사용하는 것은 모든 API 호출에 대해 주로 요청–응답 패턴을 요구합니다. 애플리케이션이 서버 생성 이벤트를 요구하는 경우, HTTP long polling과 WebSocket 등의 솔루션을 이용하는 것이 도움이 되지만, 이들 솔루션을 사용하기 위해서는 궁극적으로 별도의 인접 API를 작성해야 합니다.
- **복잡한 트랙잭션** – REST API는 고유한 리소스에 대한 개념을 기반으로 하며 각각 URI를 부여합니다. 애플리케이션 이벤트가 여러 리소스를 업데이트하도록 요청하는 경우, 여러 API 호출이 필요하거나(비효율적), 백엔드에서 복잡한 트랜잭션을 실행해야(REST의 핵심 원칙에 위배) 합니다.

최근 gRPC는 분산 애플리케이션과 특히 마이크로서비스 개발을 위한 대안으로 부상했습니다. 원래 구글(Google)에서 개발된 gRPC는 2015년 오픈소스로 제공되었으며 현재 CNCF(Cloud Native Computing Foundation)의 새로운 프로젝트 중 하나입니다. 특히, gRPC는 HTTP/2를 전송 방식으로 활용하기 때문에 바이너리 데이터 형식의 이점과 다중화된 스트리밍 기능을 활용할 수 있습니다.

gRPC의 주요 이점은 다음과 같습니다.

- 강결합 구조의 인터페이스 정의 언어(프로토콜 버퍼)
- 스트리밍 데이터에 대한 네이티브 지원(양방향으로)
- 효율적인 바이너리 데이터 형식
- 다양한 프로그래밍 언어를 위한 자동 코드 생성을 통해 상호 운영 문제 없이 진정한 다수 프로그래밍 언어 개발 환경 실현

주: 별도로 언급한 경우를 제외하고 이 장에 포함된 모든 정보는 NGINX Plus 및 NGINX Open Source 모두에 해당됩니다.

gRPC 게이트웨이 정의

제1장과 2장에서는 단일 진입 지점(예를 들어, **https://api.example.com**)을 통해 다수의 API들을 제공하는 방법을 설명했습니다. gRPC 트래픽의 기본 동작 및 특성에 따라, NGINX Plus를 gRPC 게이트웨이로서 구축할 때와 동일한 접근 방식을 사용할 수 있습니다. 동일한 호스트 이름 및 포트에서 HTTP 및 gRPC 트래픽 모두 공유하는 것이 가능하지만, 이를 분리하는 것이 바람직함에는 많은 이유들이 있습니다.

- REST 및 gRPC 애플리케이션용 API 클라이언트들은 서로 다른 형식의 에러 응답을 받을 것으로 기대합니다.
- 액세스 로그를 위한 관련 필드는 REST와 gRPC에 따라 다릅니다.
- gRPC는 절대 레거시 웹 브라우저를 다루지 않기 때문에 보다 엄격한 TLS 정책을 가질 수 있습니다.

이러한 분리를 달성하기 위해 **/etc/nginx/conf.d 디렉토리**에 위치한 메인 gRPC 구성 파일, **grpc_gateway.conf**의 고유한 `server{}` 블록 내에 gRPC 게이트웨이를 위한 구성을 추가합니다.

```
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;
```

GitHub의 raw 보기

gRPC 트래픽을 위한 액세스 로그에 항목 형식을 정의하는 것으로 시작합니다 (라인 1–4). 이 예제에서는 JSON 형식을 사용해 각 요청에서 가장 연관성 높은 데이터를 포착합니다. 예를 들어, 모든 gRPC 요청은 POST를 사용하기 때문에, HTTP 메소드는 포함되어 있지 않다는 점을 유의하십시오. 또한, gRPC 상태 코드는 물론, HTTP 상태 코드도 기록합니다. 하지만, gRPC 상태 코드는 여러 다양한 방식으로 생성될 수 있습니다. 정상 조건에서 grpc-status는 백엔드로부터 HTTP/2 트레일러로서 반환되지만, 일부 에러 조건의 경우, 백엔드 또는 NGINX Plus 자체에 의해 HTTP/2 헤더로서 반환될 수 있습니다. 액세스 로그를 단순화하기 위해 map 블록(라인 6–11)을 사용해 새 변수 grpc-status를 평가하고 출처에 관계없이 gRPC 상태를 입수합니다.

이 구성은 평문(포트 50051) 및 TLS로 보호된(포트 443) 트래픽 모두를 테스트할 수 있도록 2개의 `listen` 지시문(directive)(라인 14 및 15)을 포함하고 있습니다. `http2` 매개변수는 NGINX Plus가 HTTP/2 연결을 수용하도록 구성합니다. 이는 `ssl` 매개변수와는 독립적입니다. 또한, 포트 50051은 gRPC을 위한 일반적인 평문 포트이지만, 운영 환경에서 사용하기에는 적합하지 않다는 점도 유념하십시오.

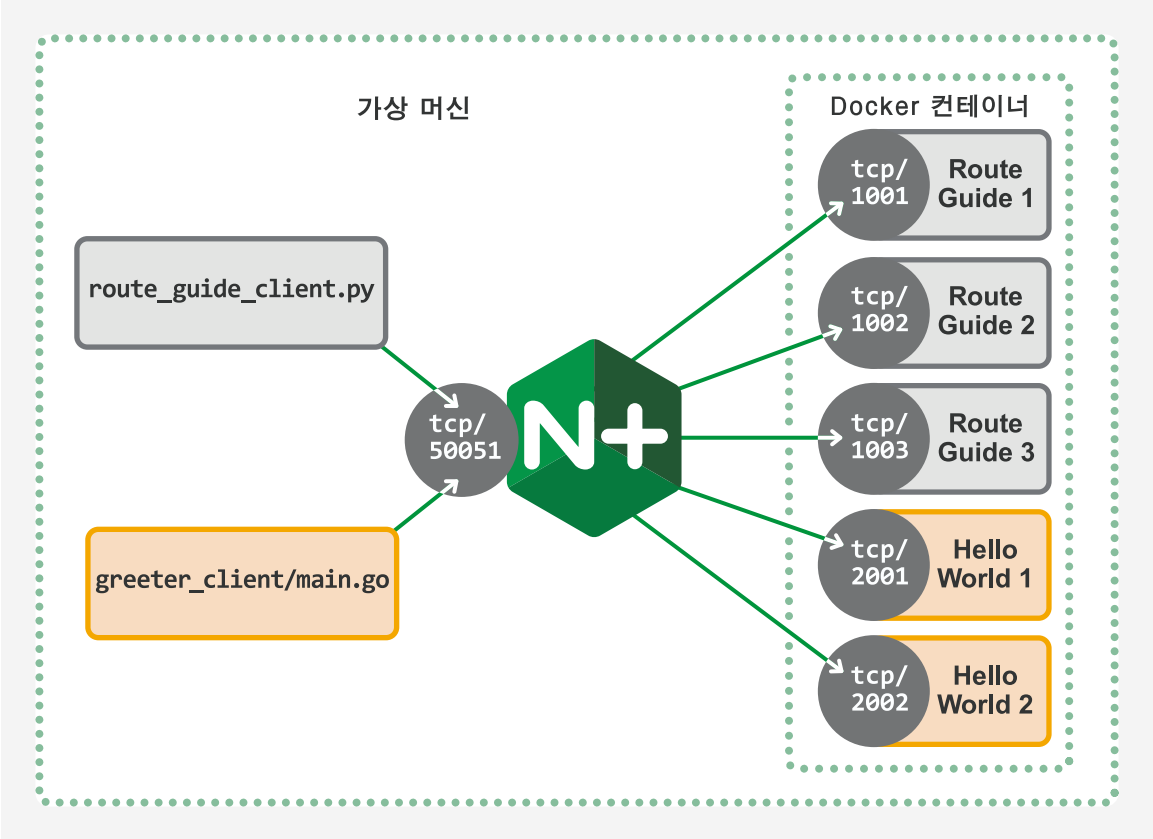
TLS 1.2를 가장 취약한 허용 프로토콜로 지정한 `ssl_protocols` 지시문(directive)(라인 25)을 제외하면, 이 TLS 구성이 일반적입니다. HTTP/2 스펙은 TLS 1.2(또는 상위 버전)의 사용을 의무로 규정하고 있으며, 이에 따라 모든 클라이언트가 TLS에 대한 SNI(Server Name Indication) 확장을 지원하도록 보장합니다. 이는 gRPC 게이트웨이가 다른 `server{}` 블록에서 정의된 가상 서버와 포트 443을 공유할 수 있다는 것을 의미합니다.

샘플 gRPC 서비스 실행

NGINX Plus의 gRPC 기능을 살펴보기 위해 gRPC 게이트웨이의 주요 구성 요소들을 보여주며 여러 gRPC 서비스가 설치된 간단한 테스트 환경을 사용하기로 하겠습니다. 공식 gRPC 가이드에서 helloworld (Go로 작성) 및 RouteGuide (Python으로 작성) 등 2개의 샘플 애플리케이션을 사용합니다. RouteGuide 애플리케이션은 4개의 각 gRPC 서비스 메소드를 포함하고 있기 때문에 매우 유용합니다.

- Simple RPC (단일 요청–응답)
- Response–streaming RPC
- Request–streaming RPC
- Bidirectional–streaming RPC

두 gRPC 서비스 모두 NGINX Plus 호스트상의 Docker 컨테이너로서 설치됩니다. 테스트 환경을 구축하는 전체 지침은 부록 A를 참조하십시오.



gRPC 게이트웨이로서 NGINX Plus를 위한 테스트 환경

가용 컨테이너의 주소와 함께 RouteGuide와 helloworld 서비스에 대해 알 수 있도록 NGINX Plus를 구성합니다.

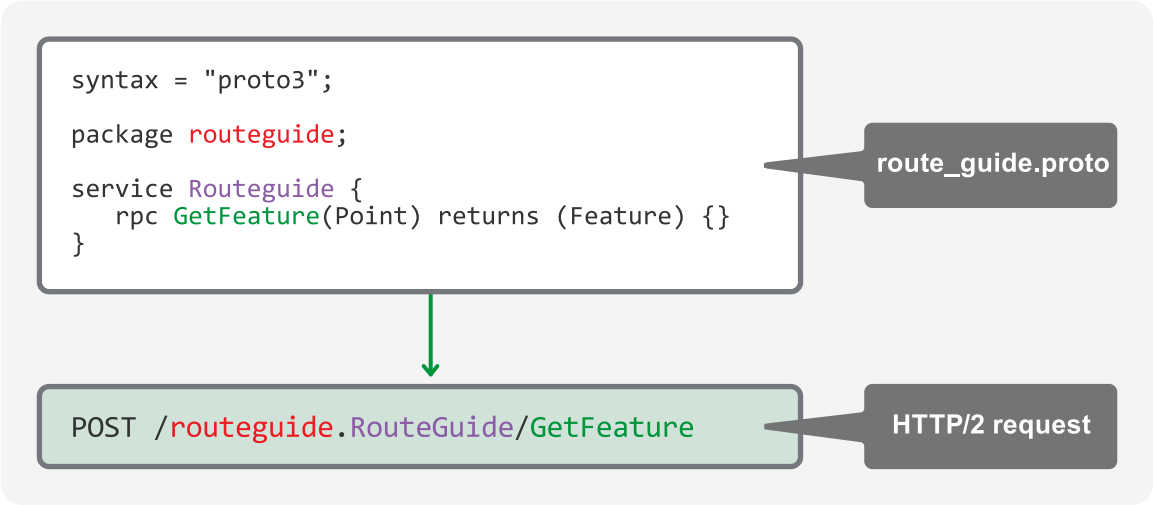
```
# Backend gRPC servers
#
upstream routeguide_service {
    zone routeguide_service 64k;
    server 127.0.0.1:10001;
    server 127.0.0.1:10002;
    server 127.0.0.1:10003;
}

upstream helloworld_service {
zone helloworld_service 64k;
    server 127.0.0.1:20001;
    server 127.0.0.1:20002;
}
```

GitHub의 raw 보기

각 gRPC 서비스를 위한 `upstream` 블록(라인 42–47 및 49–53)을 추가하고 gRPC 서버 코드를 실행하는 개별 컨테이너의 주소와 함께 이를 입력합니다.

gRPC 요청 라우팅

NGINX Plus가 gRPC (50051)를 위한 일반적인 평문 포트에서 수신 대기하는 경우, 클라이언트 요청이 정확한 백엔드 서비스에 도달할 수 있도록 구성에 라우팅 정보를 추가합니다. 하지만, 먼저 gRPC 메소도 호출을 HTTP/2 요청으로 표현하는 방법을 이해해야 합니다. 아래 그림은 RouteGuide 서비스를 위한 **route_guide.proto** 파일의 요약 버전을 보여 주고 있으며 NGINX Plus에서와 같이 패키지, 서비스 및 RPC 메소드가 URI를 구성하는 방법을 나타내고 있습니다.



Protocol Buffer RPC 메소드가 HTTP/2 요청으로 전환하는 방법

따라서 HTTP/2 요청에 포함된 정보는 단순히 패키지 이름(여기에서는 `routeguide` 또는 `helloword`)을 매칭시켜 라우팅하는 목적으로 사용될 수 있습니다.

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

GitHub의 raw 보기

첫번째 `location` 블록(라인 28)은 수정자(modifier) 없이`/routeguide.`가 해당 패키지를 위한 **.proto** 파일에서 정의된 모든 서비스 및 RPC 메소드와 일치하도록 prefix 매치를 정의합니다. 따라서 `grpc_pass directive`(라인 29)은 RouteGuide 클라이언트의 모든 요청을 upstream 그룹 **routeguide_service**로 전달합니다. 이 구성은 gRPC 패키지와 그 백엔드 서비스 간의 단순한 매핑을 제공합니다.

`grpc_pass` 지시문(directive)에 대한 인수는 평문 plaintext gRPC 연결을 이용해 요청을 프록시하는 `grpc://` 구조로 시작합니다. 백엔드가 TLS를 위해 구성된 경우, `grpcs://` 구조를 이용해 엔드 투 엔드 암호화로 gRPC 연결을 보호할 수 있습니다.

RouteGuide 클라이언트를 실행한 후, 로그 파일 항목을 검토함으로써 라우팅 동작을 확인할 수 있습니다. 여기에서 RouteChat RPC 메소드는 포트 10002로 실행되는 컨테이너로 라우팅된 것을 볼 수 있습니다.

```
$ 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
}
```

정교한 라우팅

위에서 볼 수 있듯이, 여러 gRPC 서비스를 서로 다른 백엔드로 라우팅하는 것은 단순하고 효율적이며 단 몇 라인의 구성만을 필요로 합니다. 하지만, 운영 환경의 라우팅 요구 사항은 보다 복잡하고, URI의 다른 요소들(gRPC 서비스 또는 심지어 개별 RPC 메소드)을 기반으로 라우팅해야 할 수도 있습니다.

아래의 구성 스니펫은 앞서 나온 예제를 확장한 것으로 bidirectional streaming RPC 메소드 `RouteChat`가 하나의 백엔드로 라우팅되고 다른 모든 RouteGuide 메소드는 다른 백엔드로 라우팅됩니다.

```
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  }
```

GitHub의 raw 보기

두번째 `location` 지시문(directive)(라인 7)은 = (등호) 수정자를 이용해 이것이 `RouteChat` RPC 메소드를 위한 URI에 완전 일치 매칭(exact match)임을 나타냅니다. 완전 일치 매칭(exact match)은 prefix 매칭 이전에 처리됩니다. 이는 `RouteChat` URI에 다른 그 어떤 `location` 블록도 고려되지 않는다는 것을 의미합니다.

에러에 응답하는 방법

gRPC 에러는 일반적인 HTTP 트래픽 관련 에러와 다소 다릅니다. 클라이언트들은 에러 조건들이 gRPC 응답으로 표현되고, 이는 NGINX Plus가 gRPC 게이트웨이로서 구성된 경우, 기본 NGINX Plus 에러 페이지(HTML 형식)가 적합하지 않을 것으로 예상합니다. gRPC 클라이언트를 위한 커스텀 에러 응답을 지정함으로써 이를 해결합니다.

```
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
```

GitHub의 raw 보기

전체 gRPC 에러 응답은 비교적 길고 대개 정적 구성이기 때문에 별도의 파일인 **errors.grpc_conf**에 보관하고 `include` 지시문(directive)(라인 36)을 이용해 이를 참조합니다. HTTP/REST 클라이언트와 달리, gRPC 클라이언트 애플리케이션은 다양한 HTTP 상태 코드를 처리할 것으로 기대되지 않습니다. gRPC 문서는 NGINX Plus와 같은 중간 프록시가 HTTP 에러 코드를 gRPC 상태 코드로 변환함으로써 클라이언트가 항상 적합한 응답을 받도록 하는 방법을 명시하고 있습니다. `error_page` 지시문(directive)을 이용해 이러한 매핑을 실행합니다.

```
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;
```

GitHub의 raw 보기

각 표준 HTTP 상태 코드는 @ prefix를 이용해 명명된 위치로 전달되며, 따라서 gRPC 규격의 응답이 생성될 수 있습니다. 예를 들어, HTTP **404** 응답은 내부적으로 **@grpc_unimplemented** 위치로 경로가 재지정되며 이는 이 파일의 뒷부분에서 정의됩니다.

```
location @grpc_unimplemented {
    add_header grpc-status 12;
    add_header grpc-message unimplemented;
    return 204;
}
```

GitHub의 raw 보기

@grpc_unimplemented로 명명된 위치는 오직 내부 NGINX 처리에만 사용될 수 있으며 클라이언트는 이를 직접 요청할 수 없습니다. 따라서, 라우팅 가능한 URI가 존재하지 않습니다. 이 위치 내에서 필수 gRPC 헤더를 입력하고 이를 전송함으로써 응답 본문 없이, HTTP 상태 코드 **204 (No Content)**를 이용해 gRPC 응답을 구성합니다.

curl(1) 명령을 이용해 존재하지 않는 gRPC 메소드를 요청한 오작동 gRPC 클라이언트를 복제할 수 있습니다. 하지만, 프로토콜 버퍼가 바이너리 데이터 형식을 사용하기 때문에 **curl** 명령은 일반적으로 gRPC 테스트 클라이언트로서 적합하지 않습니다. 커맨드라인에서 gRPC를 테스트한다면, grpc_cli 사용을 고려하십시오.

```
$ 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/1.15.2
date: Thu, 09 Aug 2018 15:03:41 GMT
grpc-status: 12
grpc-message: unimplemented
```

위에서 참조된 **grpc_errors.conf** 파일은 타임아웃 및 클라이언트 인증서 에러 등 NGINX Plus가 생성하는 다른 에러 응답을 위한 HTTP-to-gRPC 상태 코드 매핑도 포함하고 있습니다.

gRPC 메타데이터를 이용한 클라이언트 인증

gRPC 메타데이터는 클라이언트가 RPC 메소드 호출과 함께 추가 정보를 전송할 수 있도록 하며, 해당 데이터를 프로토콜 버퍼 스펙(**.proto** 파일)에 포함시키도록 요구하지 않습니다. 메타데이터는 단순한 key-value 쌍 리스트이며, 각 쌍은 별도의 HTTP/2 헤더로서 전송됩니다. 따라서, 메타데이터는 NGINX Plus에 쉽게 접근할 수 있습니다.

메타데이터를 위한 많은 활용 사례 중에서 클라이언트 인증이 gRPC API 게이트웨이의 가장 일반적인 사례입니다. 아래 구성 스니펫은 NGINX Plus가 어떻게 gRPC 메타데이터를 이용해 JWT 인증을 실행하는지를 보여줍니다. (JWT 인증은 NGINX Plus에서만 제공됩니다.) 이 예제에서는 JWT가 **auth-token** 메타데이터로 보내집니다.

```
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  }
```

GitHub의 raw 보기

모든 HTTP 요청 헤더는 NGINX Plus에서 **$http_*header***로 불리는 변수로 이용할 수 있습니다. 헤더 이름 내 하이폰(-)은 변수 이름에서 밑줄(_)로 변환됩니다. 따라서 JWT는 **$http_auth_token**(라인 2)으로 나옵니다.

아마도 기존 HTTP/REST API와 함께, API key들이 인증에 이용되는 경우, 이들은 gRPC 메타데이터에 의해 포함되고 NGINX Plus에 의해 검증될 수 있습니다. API key 인증을 위한 구성은 제1장에서 소개했습니다.

헬스체크 구현

트래픽을 다수의 백엔드로 로드 밸런싱하는 경우, 다운되었거나, 여타 다른 이유로 사용 불가능한 백엔드로 요청을 전송하지 않도록 하는 것이 중요합니다. NGINX Plus에서는 능동형 헬스체크 기능을 이용해 사전에 대역외(out-of-band) 요청을 백엔드로 전송하고 헬스체크에 제대로 응답하지 않는 경우, 로드 밸런싱 회전에서 제외시킬 수 있습니다. 이러한 방식으로 클라이언트 요청이 장애가 발생한 백엔드에 전달되지 않도록 합니다.

gRPC는 HTTP/2 위에서 실행되는 애플리케이션 프로토콜이기 때문에 NGINX Plus가 gRPC 클라이언트를 시뮬레이션하는 것은 쉽지 않습니다. 백엔드 gRPC 서비스가

HTTP **GET** 요청을 수용한다면 – Go로 작성된 서비스처럼 – 백엔드 gRPC 서비스가 작동 중인지 여부를 테스트하도록 NGINX Plus를 구성할 수 있습니다.

아래 구성 스니펫으로 helloworld gRPC 서비스(Go로 작성된)에 대한 능동형 헬스체크를 실행할 수 있습니다. 관련 구성을 하이라이트하기 위해 이전 섹션에서 사용되었던 **grpc_gateway.conf** 파일에 포함된 일부 지시문(directive)을 생략합니다.

```
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 }
```

GitHub의 raw 보기

다음 라우팅 섹션(라인 4–7)에서는 명명된 위치, `@helloworld_health`에 헬스체크를 정의합니다. 이를 통해 요청 라우팅에 사용된 `location` 블록에 과부하를 발생시키지 않으면서 헬스체크 요청을 커스터마이즈할 수 있습니다.

`health_check` 지시문(directive)(라인 11)은 유효하지 않은 것으로 알려진 URI, **/nginx.health/check**를 명시하고 있으며, 이 백엔드는 알 수 없는 것(unknown)으로 보고할 것입니다. 활성 상태(active)인 gRPC 서비스와 통신하고 있음을 나타내는 HTTP 헤더를 사용해 예상 응답이 `match` 블록(라인 20)에 정의됩니다.

이 구성을 적용하면, gRPC 클라이언트에서 지연이나 타임아웃 에러를 발생시키지 않으면서, **helloworld** 컨테이너 중 하나의 작동을 중단시킬 수 있습니다. 활성(active) 헬스체크는 NGINX Plus에서만 제공됩니다.

속도 제한 및 기타 API 게이트웨이 제어 적용

grpc_gateway.conf의 샘플 구성은 운영 환경에서 사용하기에 적합하며 TLS에 대한 중요도가 낮은 몇 가지 수정 사항들이 포함되어 있습니다. 패키지, 서비스 또는 RPC 메소드를 기준으로 gRPC 요청을 라우팅할 수 있으며, 이는 기존 NGINX Plus 기능을 HTTP/REST API 또는 실제 일반 웹 트래픽과 완전히 동일한 방식으로 gRPC 트래픽에 적용될 수 있다는 것을 의미합니다. 각각의 경우, 속도 제한이나 대역폭 제어 등과 같은 추가 구성을 통해 관련 `location` 블록을 확장할 수 있습니다.

요약

이 장에서는 마이크로서비스 애플리케이션 개발을 위한 클라우드 네이티브 기술로서 gRPC에 초점을 맞추었습니다. NGINX Plus가 HTTP/REST API만큼 효과적으로 gRPC 애플리케이션을 제공하고 NGINX Plus를 통해 두 스타일의 API를 다목적 API 게이트웨이로서 퍼블리싱하는 방법을 설명했습니다.

이 블로그 게시글에서 사용한 테스트 환경을 설정하는 지침은 부록 A에 소개되어 있으며 NGINX의 GitHub Gist repo에서 모든 파일을 다운로드할 수 있습니다.

부록 A:

테스트 환경 설정하기

아래 지침은 여타 환경과 분리되고 반복할 수 있도록 가상 머신에 테스트 환경을 설치하는 방법을 설명한 것입니다. 하지만, 실제 "베어 메탈" 서버에 설치하지 못할 이유는 없습니다.

테스트 환경을 단순화하기 위해 Docker 컨테이너를 사용해 gRPC 서비스를 실행합니다. 이는 테스트 환경을 위해 다수의 호스트를 사용할 필요는 없지만, NGINX Plus가 운영 환경에서처럼 네트워크 호출을 통해 프록시연결을 할 수 있다는 것을 의미합니다.

Docker를 이용하면, 코드 변경 없이 다수의 포트에서 각 gRPC 서비스의 여러 인스턴스를 실행할 수 있습니다. 각 gRPC 서비스는 컨테이너 내 포트 50051에서 수신 대기하며 이는 가상 머신의 고유한 localhost 포트로 매핑됩니다. 이에 따라 포트 50051이 비게 되며 NGINX Plus는 이를 수신 대기 포트로 사용할 수 있습니다. 따라서 테스트 클라이언트가 사전 구성된 포트 50051을 이용해 연결하면, NGINX Plus에 도달하게 됩니다.

NGINX Plus 설치하기

1. NGINX Plus를 설치합니다. 지침들은 NGINX Plus 운영자 가이드에 나와 있습니다.
2. GitHub Gist repo에서 /etc/nginx/conf.d로 아래 파일을 복사합니다.
   - **grpc_gateway.conf**
   - **errors.grpc_conf**

   주: TLS를 사용하지 않는 경우, **grpc_gateway.conf**의 `ssl_*` 지시문(directive)을 주석 처리합니다.
3. NGINX Plus를 시작합니다.

```
$ sudo nginx
```

Docker 설치하기

CentOS, RHEL 및 Oracle Linux의 경우, 다음을 실행합니다.

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

CentOS/Redhat/Oracle Linux의 경우, 다음을 실행합니다.

```
$ sudo yum install docker
```

RouteGuide Service Container 설치하기

1. 아래 Dockerfile에서 RouteGuide 컨테이너를 위한 Docker 이미지를 빌드합니다.

```
# 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"]
```

GitHub의 raw 보기

빌드 전에 Dockerfile을 로컬 하위 디렉토리로 복사하거나 Dockerfile을 위한 Gist의 URL을 **도커 빌드** 명령에 대한 인수로서 지정할 수 있습니다.

```
$ sudo docker build -t routeguide https://gist.githubusercontent.com/
nginx-gists/87ed942d4ee9f7e7ebb2ccf757ed90be/raw/
ce090f92f3bbcb5a94bbf8ded4d597cd47b43cbe/routeguide.Dockerfile
```

이미지를 다운로드하고 빌드하는 데 몇 분 정도 걸릴 수 있습니다. 성공적으로 빌드되었다는 메시지와 16진수 문자열(이미지 ID)이 표시되어 **빌드가 완료되었음**을 알려 줍니다.

2. **도커 이미지**를 실행해 이미지가 빌드되었는지 확인합니다.

```
$ 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. RouteGuide 컨테이너를 시작합니다.

```
$ 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
```

각 명령이 성공하면, 긴 16진수 문자열이 표시되어 실행 중인 컨테이너를 나타냅니다.

4. docker ps를 실행해 3개 컨테이너 모두 작동되는지 확인합니다. (샘플 출력은 읽기 쉽게 여러 라인으로 나뉩니다.)

```
$ 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
```

출력 내 PORTS 열은 각 컨테이너가 어떻게 여러 로컬 포트를 컨테이너 내부의 포트 50051로 매핑하는지를 보여줍니다.

helloworld Service Container 설치하기

1. 아래 Dockerfile에서 helloworld 컨테이너를 위한 Docker 이미지를 빌드합니다.

```
# 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"]
```

GitHub의 raw 보기

빌드 전에 Dockerfile을 로컬 하위 디렉토리로 복사하거나 Dockerfile을 위한 Gist의 URL을 **도커 빌드** 명령에 대한 인수로서 지정할 수 있습니다.

```
$ sudo docker build -t routeguide https://gist.githubusercontent.com/
nginx-gists/87ed942d4ee9f7e7ebb2ccf757ed90be/raw/
ce090f92f3bbcb5a94bbf8ded4d597cd47b43cbe/routeguide.Dockerfile
```

이미지를 다운로드하고 빌드하는 데 몇 분 정도 걸릴 수 있습니다. **성공적으로 빌드**되었다는 메시지와 16진수 문자열(이미지 ID)이 표시되어 빌드가 완료되었음을 알려 줍니다.

2. **도커 이미지**를 실행해 이미지가 빌드되었는지 확인합니다.

```
$ 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. helloworld 컨테이너를 시작합니다.

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

각 명령이 성공하면, 긴 16진수 문자열이 표시되어 실행 중인 컨테이너를 나타냅니다.

4. `docker ps`를 실행해 2개 helloworld 컨테이너 모두 작동되는지 확인합니다.

```
$ 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
```

gRPC 클라이언트 애플리케이션 설치하기

1. 프로그래밍 언어 필수 요소들을 설치합니다. 일부는 이미 테스트 환경에 설치되어 있을 수 있습니다.
   - Ubuntu 및 Debian의 경우, 다음을 실행합니다.

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

   - CentOS, RHEL 및 Oracle Linux의 경우, 다음을 실행합니다.

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

   `python-pip`을 사용하기 위해서는 EPEL 리포지터리가 활성화되어야 한다는 점에 유념하십시오(필요한 경우, `sudo yum install epel-release`를 먼저 실행합니다).

2. helloworld 애플리케이션을 다운로드합니다.

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

3. RouteGuide 애플리케이션을 다운로드합니다.

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

설정 테스트

1. helloworld 클라이언트를 실행합니다.

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

2. RouteGuide 클라이언트를 실행합니다.

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

3. NGINX Plus 로그를 검사해 테스트 환경이 작동하는지 확인합니다.

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

부록 B:

문서 개정 내역

버전	날짜	설명
1.0	2018-10-04	초판 발행