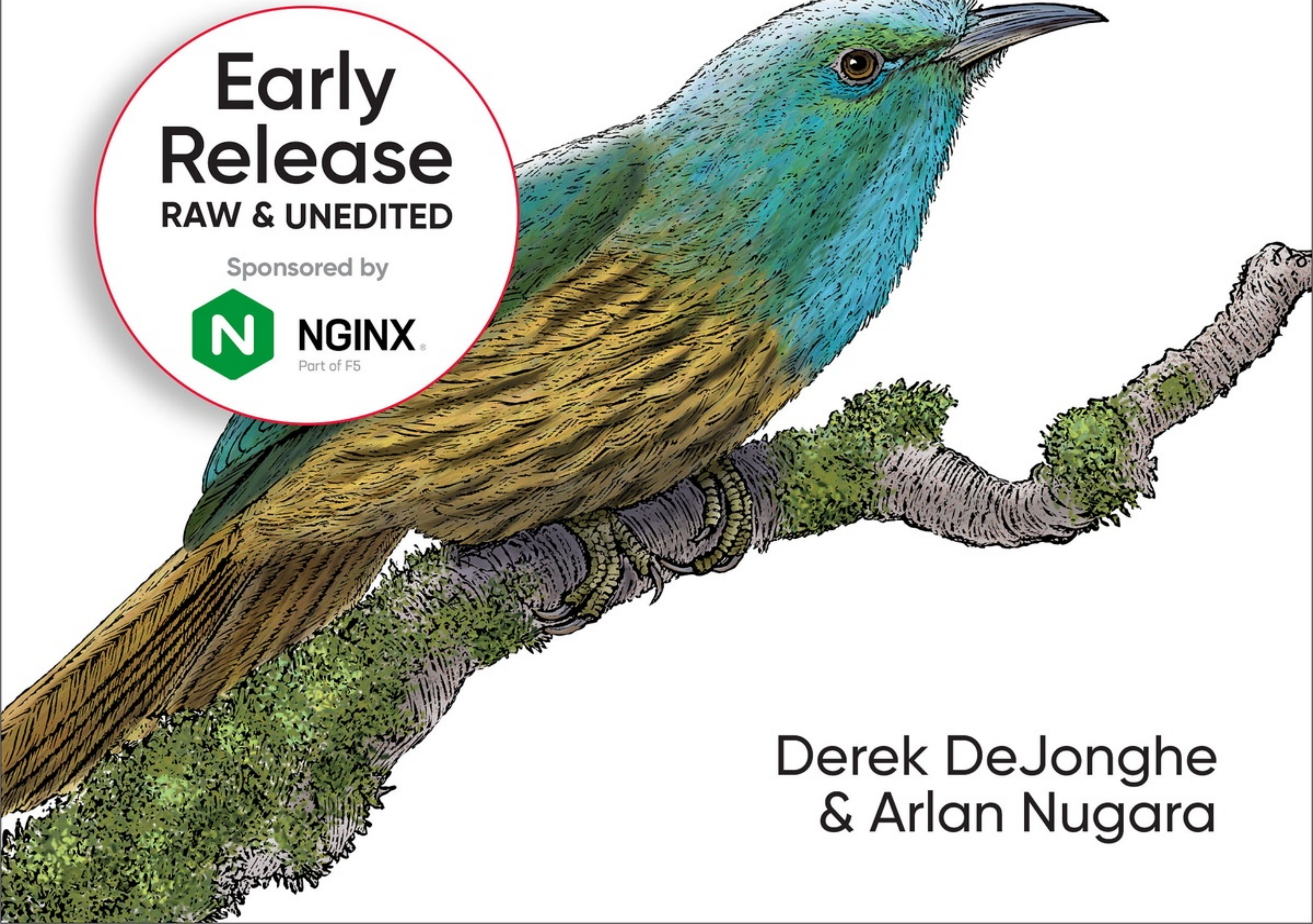
O'REILLY®

Application Delivery & Load Balancing in Microsoft Azure

Practical Solutions with NGINX and Microsoft Azure

Early Release
RAW & UNEDITED
Sponsored by
NGINX®
Part of F5



Derek DeJonghe
& Arlan Nugara

1. 1. What is Application Delivery and Load Balancing and Why It’s Important
    a. Application Delivery Controllers
        i. Load Balancers
        ii. Application Delivery and Load Balancing - A Solution Overview
    b. Conclusion
2. 2. Managed Load-Balancing Options in Azure
    a. Azure Native Load Balancers
        i. Azure Load Balancing Dimensions
    b. Azure Load Balancer
    c. Azure Application Gateway for Load Balancing
    d. Azure Web Application Firewall (WAF) with ModSecurity
    e. Azure Front Door
    f. Azure Traffic Manager for Cloud-Based DNS Load Balancing
        i. Priority traffic-routing

        ii. Weighted traffic-routing
        iii. Performance traffic-routing
        iv. Geographic traffic-routing
    g. Designing Highly Available Systems
        i. Highly Available Multi Region NGINX
3. 3. NGINX and NGINX Plus on Azure
    a. NGINX vs NGINX Plus
    b. Installing NGINX & NGINX Plus
        i. Installing via Azure Marketplace
        ii. Installing Manually on VMs
        iii. Installing via Azure Resource Manager and PowerShell
        iv. Installing NGINX (Open Source) via Terraform
        v. Deploying NGINX in Debian and Ubuntu Linux
        vi. Deploying NGINX in CentOS and RedHat Linux
        vii. Creating the Virtual Machine
        viii. Creating the Input Variables File
        ix. Running Terraform

x. Installing NGINX Plus via Terraform

c. Conclusion

4. 4. NGINX and Microsoft Managed Options

a. Comparing NGINX Plus and Azure Load Balancing Services

i. Use Cases

b. Comparing NGINX and Azure Application Gateway Functionality

c. Comparing NGINX and Azure Web Application Firewall Capabilities

i. ModSecurity

d. NGINX Plus App Protect

e. Conclusion

5. 5. Azure Front Door with NGINX

a. What is the Azure Front Door Service

i. Split TCP

ii. Front Door Features

iii. Front Door's Place in the Stack

b. Benefits of Azure Front Door Service with NGINX
c. Integrating Azure Front Door Service with NGINX
    i. Front Door Features
    ii. Routing Rules
    iii. Optimizing with NGINX
d. Conclusion

NGINX

Try NGINX Plus and NGINX App Protect Free

Get high-performance application delivery and security for microservices. NGINX Plus is a software load balancer, API gateway, and microservices proxy. NGINX App Protect is a lightweight, fast web application firewall (WAF) built on proven F5 technology and designed for modern apps and DevOps environments.

Cost Savings

Over 80% cost savings compared to hardware application delivery controllers and WAFs, with all the performance and features you expect.

Reduced Complexity

The only all-in-one load balancer, API gateway, microservices proxy, and web application firewall helps reduce infrastructure sprawl.

Enterprise Ready

NGINX Plus and NGINX App Protect deliver enterprise requirements for security, scalability, and resiliency while integrating with DevOps and CI/CD environments.

Advanced Security

NGINX App Protect offers 4x the performance and 10x the throughput as open source alternatives like ModSecurity, while providing even more comprehensive controls.

Download at nginx.com/freetrial

NGINX®
Part of F5

Application Delivery and Load Balancing in Microsoft Azure

Practical Solutions with NGINX and Microsoft Azure

Derek DeJonghe and Arlan Nugara

Application Delivery and Load Balancing in Microsoft Azure

by Derek DeJonghe and Arlan Nugara

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Acquisitions Editor: Mary Preap

Development Editor: Gary O’Brien

Production Editor: Daniel Elfanbaum

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: O’Reilly Media

December 2020: First Edition

Revision History for the Early Release

- 2020-09-18: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098115869 for release details.

978-1-098-11579-1

[LSI]

Chapter 1. What is Application Delivery and Load Balancing and Why It’s Important

When anyone uses an application, they expect it to respond quickly, efficiently, and reliably. When a user encounters errors, outages, or over capacity messages, they generally don’t wonder why. They get annoyed, quit using the application, and ultimately complain about the application’s owner on social media. It doesn’t matter that the company was having a great day, and requests to their servers went through the roof. What matters is their request resulted in a failure of the application and, in their eyes, the company providing the application.

As more companies move their applications and services to the web, responding to variable demands and workloads has become increasingly important. In this chapter, we'll introduce the concepts behind application delivery and load balancing by explaining the purpose of application delivery controllers and load balancers and the problems they solve. Subsequent chapters will explore how you can use Microsoft Azure and NGINX for high-performance application delivery and load balancing.

Application Delivery Controllers

Put simply, application delivery is a process that ensures that an application functions properly and efficiently, and the content is delivered no matter how many clients or users are making requests at a time. Servers can fail for any number of reasons, from demand capacity to security breaches or a simple mechanical failure. When that server runs an application that customers or employees are relying upon, then that application also fails. Companies need ways to be adaptable and provide optimal performance and availability in any given situation.

At the heart of application delivery is an advanced load balancer, called an application delivery controller (ADC) that provides full control of application delivery by managing client connections. ADCs help ensure maximum performance and capacity by sitting between the user and the application servers, directly requests only to servers that are currently online. Thus, an ADC reduces the chances of errors due to server outages or other possible interruptions.

Hardware or Software

All ADCs are essentially software solutions. However, there are on-prem hardware/software bundled black box purchased from a vendor. A hardware solution requires the company to keep buying and installing new hardware as traffic increases. Unless you’ve planned ahead, this can result in high costs both in hardware, time, and labor hours, not to mention delays as the ADC must be purchased, delivered, configured, and installed.

Software-based ADCs have the same general feature sets as the hardware versions, but because they are often virtual, cloud, or hybrid solutions, they are much less complicated with a wide range of flexible options. While you can install an ADC on hardware you already own, cloud services like Microsoft Azure, allow greater flexibility and value. By installing an ADC in a cloud environment, you can quickly and efficiently scale to meet demand of applications in various locations, paying only for usage, not peek capacity.

Structure and Function of ADCs

In general, ADCs comprise of two components—a server and a member. The server, whether virtual or on-prem, is where the ADC will route traffic. The member includes the TCP port (for example, 80) of the application that is receiving the traffic and the IP address (for example, 198.51.100.12) to resolve. This combination, 198.51.100.12:80, is referred to as the service, providing a distinction between the server and the applications it is running.

This distinction is essential, because a server may be running multiple services at once (e.g., FTP, HTTPS, etc.) so the ADC can interact with each application service separately rather than the server itself. Unique application identification, like the IP example above, means the ADC can apply load balancing, or other functions, directly to the service itself rather than the server as a whole. Figure 1-1 illustrates a simple ADC solution. You'll still need to deal with the server for routine monitoring and maintenance. But, overall, the server/member approach will improve the efficiency of application delivery and load balance because the ADC provides specifics rather than a broad solution.

NOTE

An ADC may include a node, host, member, or server, sometimes all four, with different meanings. For our purposes we're going to use server and member.

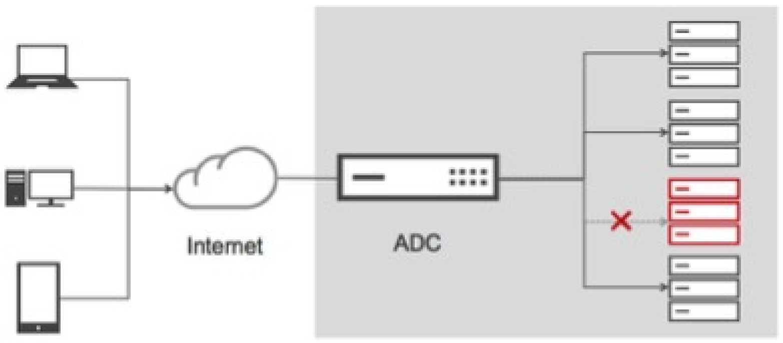


Figure 1-1. Diagram of simple ADC solution

Load Balancers

Optimal load distribution reduces site inaccessibility caused by failure or stress of a single server while assuring consistent performance for all users. Different routing techniques and algorithms ensure optimal performance in varying load-balancing scenarios.

Modern websites must support concurrent connections from clients requesting text, images, video, or application data, all in a fast and reliable manner, while scaling from hundreds of users to millions of users during peak times. Load balancers are a critical part of this scalability.

Load balancers, introduced in the 1990s as hardware-based servers or appliances, have evolved considerably. Cloud load balancing, also called Load Balancing as a Service (LBaaS), is an updated alternative to hardware load balancers. Regardless of the implementation of a load balancer, scalability is still the primary goal of load balancing, even though modern load balancers can do so much more. Figure 1-2 shows a load balancing solution.

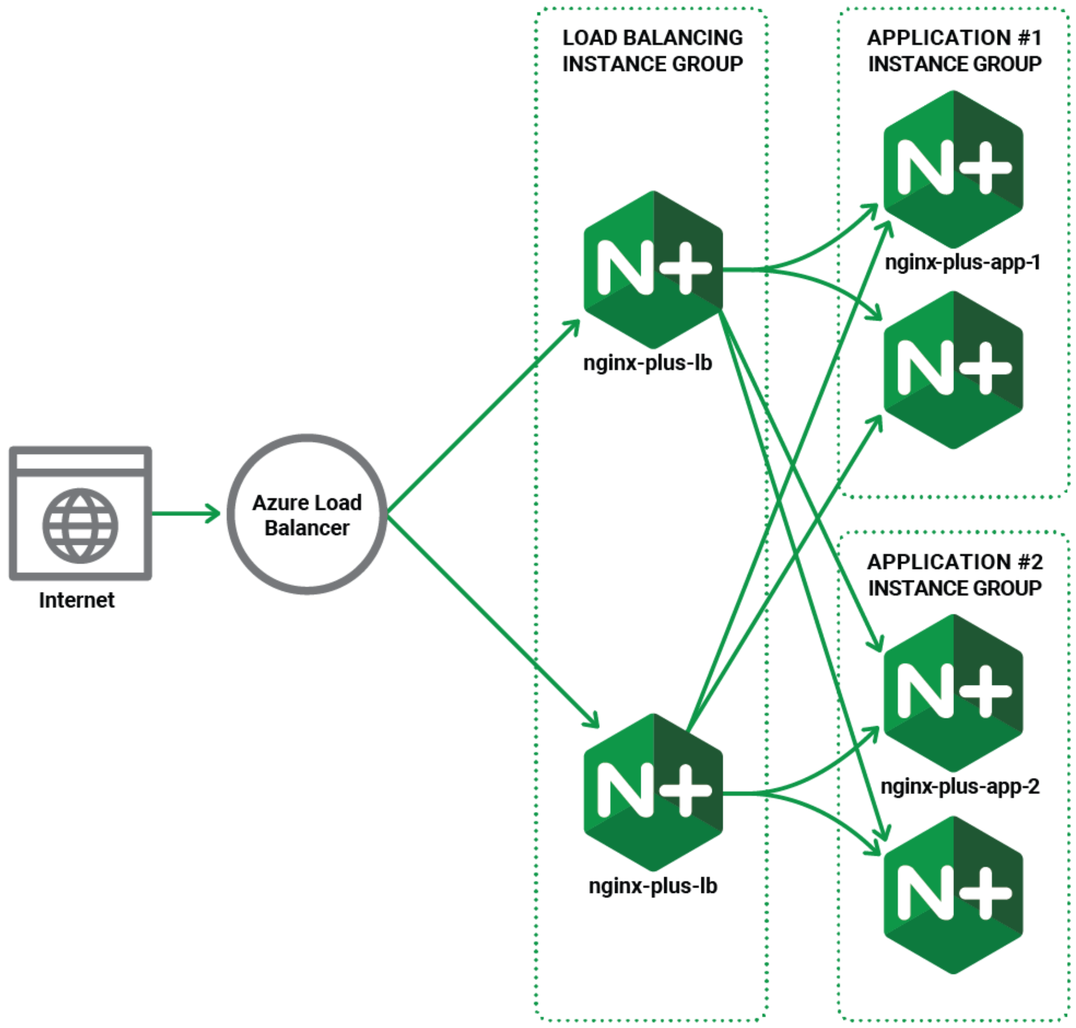

LOAD BALANCING INSTANCE GROUP
APPLICATION #1 INSTANCE GROUP
APPLICATION #2 INSTANCE GROUP
Internet
Azure Load Balancer
nginx-plus-lb
nginx-plus-lb
nginx-plus-app-1
nginx-plus-app-2

Figure 1-2. Diagram of high level Load Balancer Solution

The OSI Model and Load Balancing

Before we discuss load balancing, especially in the cloud, it's important to review The Open System Interconnection (OSI) model. The OSI created the conceptual model to provide a visual representation of the interoperation between systems that is universally applicable no matter what hardware or network characteristics are involved. The OSI model performs no functions in the networking process. It is a conceptual framework to help understand complex interactions that are happening. The model defines a networking framework to implement protocols in seven layers:

- Layer 7: Application layer
- Layer 6: Presentation layer
- Layer 5: Session layer
- Layer 4: Transport layer
- Layer 3: Network layer
- Layer 2: Data-link layer
- Layer 1: Physical layer

Network firewalls are security devices that operate from Layer 1 to Layer 3, whereas load balancing happens from Layer 4 to Layer 7. Load balancers have different capabilities, including the following:

Layer 4 (L4)

Directs traffic based on data from network and transport layer protocols, such as IP address and TCP port.

Layer 7 (L7)

Adds context switching to load balancing, allowing routing decisions based on attributes like HTTP header, URL, Secure Sockets Layer (SSL) session ID, and HTML form data.

Global Server Load Balancing (GSLB)

GSLB extends L4 and L7 capabilities to servers in different geographic locations. The Domain Name System (DNS) is also used in certain solutions and this topic is addressed when Azure Traffic Manager is used as an example of such an implementation.

As more enterprises seek to deploy cloud-native applications in public clouds, it is resulting in significant changes in the capability of load balancers.

Problems Load Balancers Solve

In cloud computing, load balancers solve three issues that fall under the following categories:

- Cloud bursting
- Local load balancing
- Global load balancing

Cloud bursting is a configuration between a private cloud (i.e., on-premises compute environment) and a public cloud, that uses a load balancer to redirect overflow traffic from a private cloud that has reached 100% of resource capacity to a public cloud to avoid decreases in performance or an interruption of service.

The critical advantage of cloud bursting is economic, in the respect that companies do not need to provision or license excess capacity to meet limited-time peak loads or unexpected fluctuations in demand. This flexibility and the automated self-service model of the cloud allow you to rent resources for any length of time, and be billed monthly based on consumption.

Organizations can use local load balancing within private and public clouds; it is a fundamental infrastructure requirement for any web application that needs high availability and the ability to distribute traffic across several servers.

Global load balancing is much more complex and can involve several layers of load balancers that manage traffic across multiple private clouds, public clouds, and public cloud regions. The greatest challenge is not the distribution of the traffic, but the synchronization of the backend processes and data so that users get consistent and correct data regardless of where the responding server is located. Although state synchronization challenges are not unique to global application delivery, the widely distributed nature of a global-scale solution introduces latency and regional resource resiliency that requires various complex solutions to meet service-level agreements (SLAs).

The Solutions Load Balancers Provide

The choice of a load balancing method depends on the needs of your application to serve clients. Different load-balancing algorithms provide different solutions based on application and client needs:

Round robin

The default load-balancing solution where requests are distributed through the list of servers sequentially.

Weighted round robin

Round robin for situations in which the capacity of the servers are varied. The servers with the higher weight are favored and receive a greater share of the traffic.

Weighted least connections

Similar to round robin, weighted least connections gives a weighted status to each server, the load balancer routes traffic based on the least number of open connections.

Hashing

A hash key is generated, through an algorithm, from the header or other information to direct traffic to a specific server.

Other than load distribution, a load balancer can enforce session persistence, also referred to as a sticky session. This solution refers to directing incoming client requests to the same backend server for the duration of a client session. This is a solution to a problem of load balancing presents. A backend server may store data locally for a number of reasons, for example if the data set is too large to work with over the network in a timely fashion. In this event the client will want subsequent requests to be directed to the same backend server, hence session persistence.

Load balancers also enable health monitoring to ensure that client requests are not directed to a backend server that is unavailable or unhealthy. A load balancer will either actively or passively monitor it's backend servers and can mark them as unhealthy under certain conditions.

Application Delivery and Load Balancing - A Solution Overview

Load balancing and application delivery are interrelated solutions. To understand their relationship, and how load balancing is key to application delivery, lets quickly review the delivery process without an ADC or load balancing:

- An end-user/client requests to connect with an application on a server

- The request is routed over the Internet to the application server
- The server accepts the connection and responds
- The user/client receives the response to their request

Figure 1-3 illustrates this process. From the user/client perspective, this was a direct connection. The user asked the application to do something and it responded.

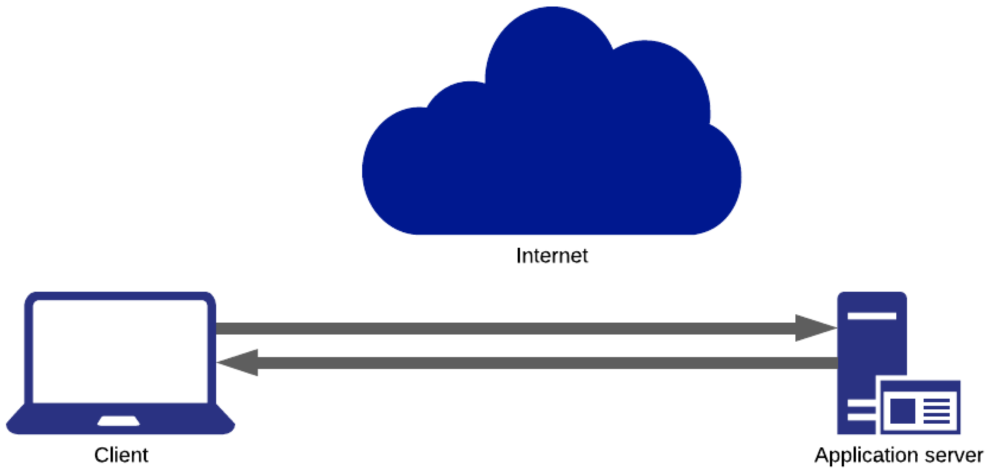


Figure 1-3. Diagram of basic application delivery transaction solution

In an application delivery environment, an ADC sits somewhere between the user/client and the virtual application servers where the requested service resides. The delivery process would look something like:

- An end user/client requests to connect with an application on a server
- The request is routed over the Internet to the ADC

- The ADC determines to accept the connection, makes a match to the selected service's application server, changing the destination of the request appropriately
- The application server response is forwarded back through the ADC
- The user/client receives the response to their request.

Figure 1-4 shows the application delivery process, including an ADC. Again, from the user/client perspective, this was a direct connection. They would have no indication that any action took place between them. During the transaction, the ADC changes the original request to match the IP and port of the virtual server. So each subsequent request returns to that selected virtual server.

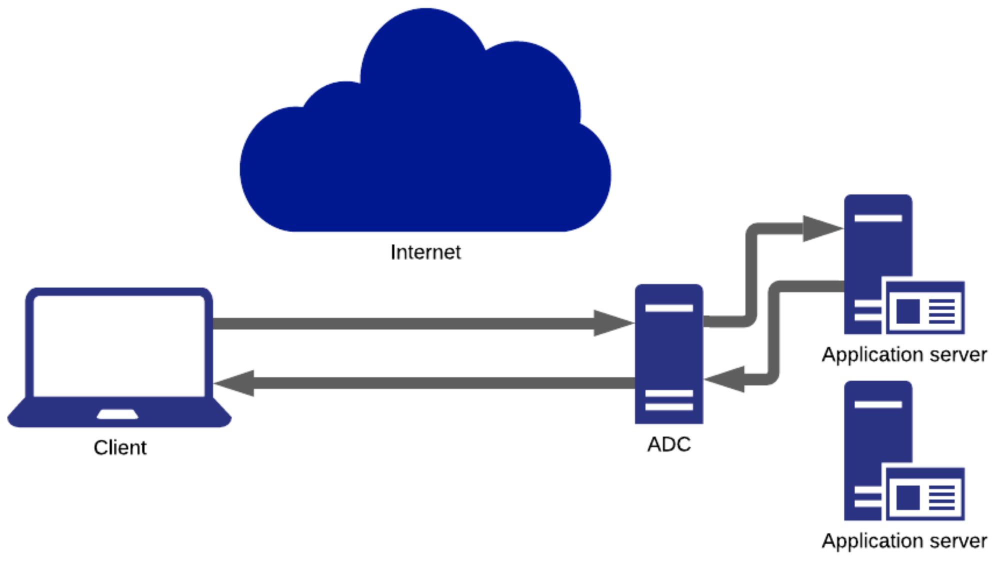


Figure 1-4. Diagram of application delivery with load balancing ADC

There’s no rule of where an ADC sits in the application delivery process. However, best practices generally place it between a firewall and the application server.

In the case of high demand for direction requests, the ADC would balance the load of connections between the assigned virtual servers. As each request is accepted, it is routed to the best possible server determined by the load balancing solution configured in the ADC. This way, the application’s performance and response times are balanced, and the application users are happy.

While basic load balancing is still used, load balancing is now primarily regarded as a function of ADCs. With the rise in cloud services, like Microsoft Azure, and the ability to deploy virtual servers, the ADC’s role in providing improved performance, flexibility, and adaptability is increasing.

Conclusion

The combined technology that ensures applications function efficiently and reliably under varying degrees of load from clients or uses is known as application delivery. The heard of application delivery is the ADC, an advanced load balancer that receives requests and directs them to servers to optimize performance and capacity. Without load balancing, most modern applications would fail. It’s not enough to simply make an application reachable, but it must be dependable, functional, and, most importantly, always available.

As companies move applications from on-prem to the cloud, software architects and cloud solution architects are looking at options to improve application delivery, load balancing, performance, security, and high availability for workloads. This book will provide a meaningful description of application delivery and loading balancing options available natively from Microsoft Azure and the role NGINX can provide in a comprehensive solution.

In chapter 2, we’ll explore the managed solutions that are available in Azure, including its native load balancers, application gateway, web application firewalls, and introduce Azure Front Door.

Chapter 2. Managed Load-Balancing Options in Azure

Microsoft Azure, like other cloud service providers, offers the ability to instantly provision computing resources on demand. This includes support for fully managed Azure services such as load balancers as well as support for third-party network virtual appliance load balancers such as NGINX. There are four native managed load-balancing services in Azure to ensure always-available applications both on the public Internet and private or virtual networks. This chapter will explore the features and capabilities of these four services to show how to improve load balancing, performance, security, and high availability for workloads on Azure.

Azure Native Load Balancers

The Azure managed services that provide load balancing functionality, enable load balancing in different ways, at different layers of the stack. Each of the services compliment each other and can be layered to provide the intended type of service. Azure Load Balancer is a Layer-4, transport layer, service that handles UDP and TCP protocols. Azure Application Gateway is a Layer-7 HTTP load balancer with an application delivery controller and SSL/TLS termination offload capabilities. Azure Traffic Manager provides DNS-based, domain-level load balancing. Finally, Azure Front Door is a full Layer-7 application delivery network with extensive features, positioned to optimize for network latency and proximity. We review each service to understand when to use them effectively. First, it's important to understand Azure's load balancing dimensions.

Azure Load Balancing Dimensions

Azure breaks its load balancing services out along two dimensions: global or regional, and HTTP(S) or non-HTTP(S). Each of these dimensions, or combination thereof, offer different solutions based on business needs. Table 2-1 shows where each Azure load balancer falls into the dimensions.

Table 2-1. Azure Load Balancers by Dime

n
s
i
o
n

Load Balancing Service	Global or Regional	Traffic Type
Azure Load Balancer	Regional	Non-HTTP(S)
Azure Application Gateway	Regional	HTTP(S)
Azure Front Door	Global	HTTP(S)
Azure Traffic Manager	Global	Non-HTTP(S)

GLOBAL VERSUS REGIONAL LOAD BALANCING

A global load balancer efficiently distributes traffic across distributed server clusters, including localized or regional backends, multiple data centers, public or private clouds, and hybrid/on-premises, no matter what their location globally. Traffic is first routed to the closest available location to reduce latency and increase performance and availability. If the closest available location is unavailable or unreliable, the load balancer will move to the next closest, and so on. Globalized load balancing also allows for serving localized content based on the requests' originating IP address to, for example, meet GDPR compliance.

Regional load balancing distributes traffic across virtual machines in a virtual network or when all your servers are in a single region.

HTTP(S) VERSUS NON-HTTP(S)

HTTP(S) load-balancing services operate on Layer 7, application layer, and only accept HTTP(S) traffic for web applications or other HTTP(S) destinations. They include features such as SSL offload, web application firewalls, and path-based routing. Non-HTTP/S load-balancing services have a broader scope than HTTP(S), as they operate lower in the OSI Model, supporting TCP and UDP.

Now that we understand Azure's approach to load balancing, we'll walk through the purpose, use, and features of all four load balancing services.

LOAD BALANCER DECISION DIAGRAM

Below is a diagram to help explain which combination of Azure managed offerings are useful for a given scenario.

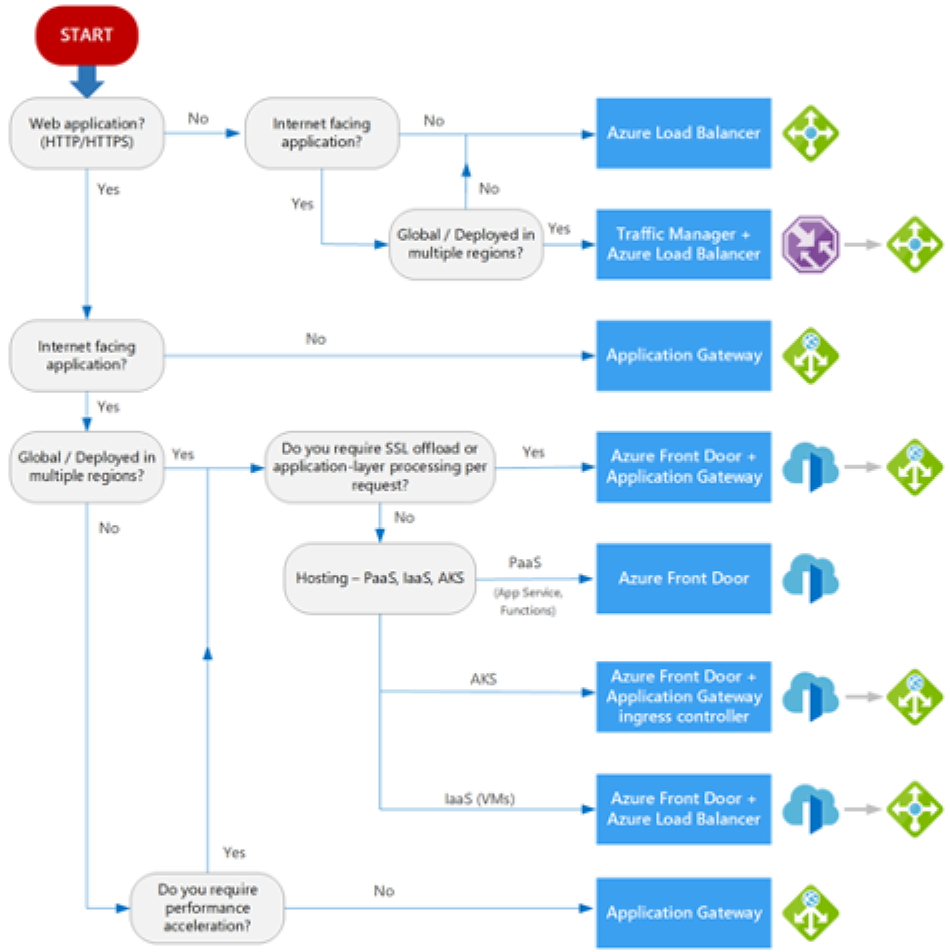

START
Web application? (HTTP/HTTPS)
No
Internet facing application?
No
Azure Load Balancer
Yes
Yes
No
Global / Deployed in multiple regions?
Yes
Traffic Manager + Azure Load Balancer
Internet facing application?
No
Application Gateway
Yes
Global / Deployed in multiple regions?
Yes
Do you require SSL offload or application-layer processing per request?
Yes
Azure Front Door + Application Gateway
No
No
Hosting – PaaS, IaaS, AKS
PaaS
(App Service, Functions)
Azure Front Door
AKS
Azure Front Door + Application Gateway ingress controller
IaaS (VMs)
Azure Front Door + Azure Load Balancer
Yes
Do you require performance acceleration?
No
Application Gateway

Figure 2-1. Update diagram and include reference to original MS Docs version.

Azure Load Balancer

A load balancer resource is either a public load balancer (also called an external load balancer) or an internal load balancer (also called a private load balancer) within the context of the virtual network[1]. Azure Load Balancer has an inbound and an outbound feature set. The Load Balancer resource functionality is expressed with a couple concepts: a frontend, a rule, a health probe, and a backend pool definition. Azure load balancer maps new flows to healthy backend instances.

Azure Load Balancer operates on Layer 4 and is available in two versions. The Standard load balancer enables you to scale your applications and create high availability for small-scale deployments to large and complex multi-zone architectures, allowing for up to 1000 backend virtual machines. The Basic load balancer does not support HTTPS health probes, is limited to only up to 300 backend virtual machines, and is not suitable for production workloads.

As the Azure Load Balancer is a layer 4 device, the connection between a client and a backend server is a direct connection. In the configuration of a public Load Balancer in front of private VMs, the backend VM sees a connection coming from a public IP address, to it's private IP, the Load Balancer is acting as a inbound NAT device, that has load balancing capabilities. The frontend IP address together with the port number of incoming traffic is mapped by the Load Balancer to a backend pool. The backend pool may consist of a number of direct VM IP addresses or a scale set, and a corresponding destination port. By applying load-balancing rules, you can distribute specific types of traffic across multiple VMs or services. For example, you can use an Azure Load Balancer to handle traffic for multiple types of connections for different application types to distribute load amongst horizontally scaled machines with a single load balancer.

In the scenario where a backend service needs to connect outbound, it's connections are routed out through the Load Balancers public frontend IP address. This functionality is called Source Network Address Translation, SNAT. It’s in a way a reverse load balancer, connections from multiple machines are coming from the same public IP. This is called masquerading, where the machine in the middle hides the real client IP from the destination of the connection. Masquerading is not done on inbound connections, which is why the backend VM sees the connection directly from the client. If a backend VM needs to provide a return port to the client to reconnect, Azure Load Balancers can be configured to enable port access translation, PAT, and allocates of ephemeral ports to each backend VM in a pool for use that will direct the client directly at a specific backend VM.

A few example scenarios of Azure Load Balancer usage:

- Non-HTTP(S) Load Balancing: Load Balancing of any protocols built on top of TCP/UDP, including but not limited to HTTP(S).
- Public Facing Load Balancer: Horizontally scaled or HA application servers hosted in a virtual network that do not have public IP addresses need to have incoming load from the internet distributed between them.
- Private Load Balancer: Provides a single private endpoint to load balance over an HA service or one that is scaled horizontally.
- Outbound Connections: When a HA or horizontally scaled application needs to initiate connections outbound but maintain a single ingress and egress point.

The following diagram depicts the Public Facing Load Balancer scenario, in which an Azure Load Balancer is distributing load for a pool of NGINX servers acting as API gateways.

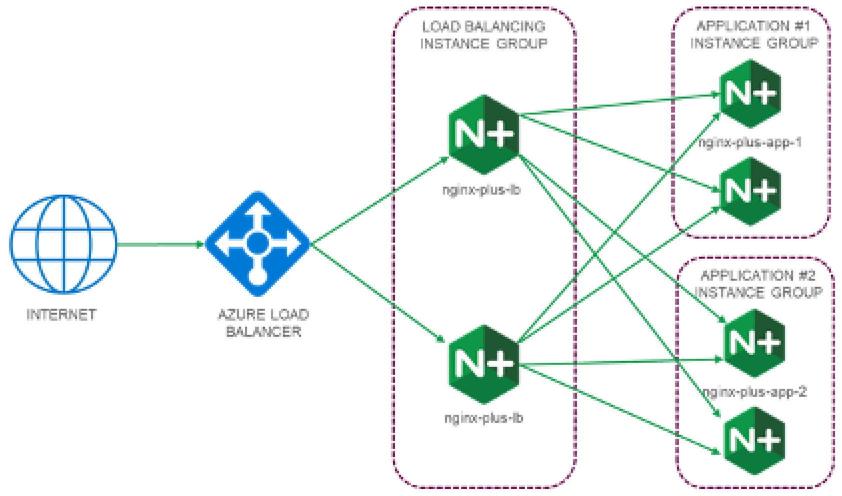

LOAD BALANCING INSTANCE GROUP
APPLICATION #1 INSTANCE GROUP
APPLICATION #2 INSTANCE GROUP
INTERNET
AZURE LOAD BALANCER
nginx-plus-lb
nginx-plus-lb
nginx-plus-app-1
nginx-plus-app-2

Figure 2-2. Visual of code build of solution using Azure LB and NGINX Plus

Azure Application Gateway for Load Balancing

The Azure Application Gateway is a layer 7, application layer, request router. The Azure Application Gateway is a proxy, it receives the request from the client, and makes a request to the backend service on the clients behalf. Unlike the Azure Load Balancer, there are two connections in this scenario, between the client and Application Gateway, and between Application Gateway and the backend. By receiving the request from the client, and understanding the HTTP protocol, the Application Gateway can block, redirect, and route requests accordingly.

Routing rules are applied which can direct the request to different backend pools based on HTTP information such as URI. This is perfect for the micro service architecture, it enables you to have one endpoint listening resource route prefixes, and directing the request to the correct micro service. The backend pool can consist of IP addresses, Full Qualified Domain Names (FQDNs), VMs, Scale Sets, and additionally App Services. When multiple entities exist within a single pool they're load balanced over with a round-robin algorithm. Only entities considered healthy will be sent requests

Health is checked for a backend by use of a health probe. A health probe is a configuration that tells the Application Gateway to periodically make requests to all entities in a backend pool with a specific request, and expect an specific response code, and optionally response body match in return. If a entity in a backend pool fails it's health probe, the Application Gateway will no longer send it client requests, but will continue to monitor with a health probe.

Session affinity can be configured for the Application Gateway. Session affinity will cause Application Gateway to generate a cookie for the client that will instruct the Application Gateway to request subsequent request from this client to a specific entity within a backend pool. This affinity happens at the pool layer, meaning if a request from the same client is made that routes to a different backend pool, they would receive another cookie, binding them to a separate backend entity in the pool the request is served from.

Rules are mapped to listeners, which receive incoming requests. Listeners can be mapped by IP or hostname. Each listener can be attached to many rules, and those rules map to backend pools. By using separate listeners, you can route the same URI path for two different hostnames, to two separate backend pools with one Application Gateway. It also means that you can use the same backend pool for different offerings listening on separate hostnames.

The Application Gateway provides the very valuable feature of layer 7 request routing, through a scalable managed regional service. With a vast amount of configuration possibilities, Azure Application Gateway is meant to be the traffic cop that stops, and directs traffic between your client and application.

Azure Web Application Firewall (WAF) with ModSecurity

One feature of Application Gateway is the ability to apply Web Application Firewall policies, which provides your applications with protection from common vulnerabilities and exploits like SQL injection or cross site scripting (XSS) attacks. A WAF policy requires at least one Azure managed rule set, and optionally can be configured with custom rules. Within a managed rule set, particular rules can be turned on or off. WAF policies can be set to enforce and actively block violating request, or passively monitor and record security events. Individual custom rules can be set to block or monitor as well.

Azure WAF policies utilize the open source, cross platform WAF ModSecurity. The Azure WAF policies ModSecurity uses the core rule set of the Open Web Application Security Project (OWASP). The OWASP rule set is community supported and regularly updated. You will see in the WAF policies that there are multiple version of the OWASP rule set.

Azure WAF policies do not expose the full functionality of ModSecurity, but does allow you to filter requests through rules to match variables from a request to policies to block, flag, or deny requests. A layer 7 router is the perfect point in the stack to block malicious requests, before they ever reach the application server. This protects your resources and reduces stress on your business logic layer.

Figure 2-3. ILLUSTRATION RATHER THAN TERRAFORM

Azure Front Door

Azure Front Door provides the ability to configure, manage, and monitor global routing for all your web traffic by optimizing performance and failover for high availability at a global scale. Using something called Split TCP anycast protocol, the service ensures that your users always connect to the nearest Front Door Point of Presence (POP). Split TCP works by breaking a connection with a high round-trip in smaller segments. Front Door environments are placed closer to end-users, and the connections are terminated inside the Front Door, which means a TCP with an extensive round trip to a backend is split into two TCP connections. Anycast is a network addressing and routing methodology where a destination has multiple routing paths to two or more endpoint destinations – the ideal approach is determined based on the number of hops, distance, or network latency. Front Door leverages Anycast for DNS and HTTP traffic to ensure user traffic goes to the environment with the fewest hops. The DNS layer being how a client selects a Front Door endpoint, and the HTTP layer being how Front Door selects a backend to serve the request.

To do this, Front Door organizes environments into rings. The outer ring has environments that are closer to users, which means they offer lower latency while the inner rings can handle the failover of the outer ring environment if an issue were to occur. This configuration ensures that end-user requests always reach the closest Front Door environment but that traffic gets moved to a healthy environment if the outer ring environment fails.

Front Door automatically triggers health probes to check on your backend services for latency and availability – if a health probe detects that a backend is down, it will quickly provide automatic failover. The load balancing performed by Front Door is like a least latency algorithm. Front Door is useful when clients are spread out globally, because of it's split TCP functionality, it's load balancing is useful when you're running the same service in multiple regions. The failover capabilities allows for planned maintenance without downtime by switching load to another installation of the service. We'll discuss further applications of Front Door and integration with NGINX in chapter 5.

Azure Traffic Manager for Cloud-Based DNS Load Balancing

Azure Traffic Manager is a DNS-based load manager that directs the request to the proper endpoints based on method and endpoint health. Traffic Manager offers multiple routing methods and endpoint monitoring tools suitable for a wide range of application or failover needs. Table 2-1 shows what each method offers.

Table 2-2. Azure Traffic Manager R

outing Methods

Method	Description
Priority	Uses a primary service endpoint for all traffic with a set of backups for failover
Weighted	Distributes traffic across all endpoints based on configured weights
Performance	For endpoints in different geographic locations, routes traffic to the closest endpoint to a user based on latency
Geographic	Routes based on the geographic location of DNS query origin
Multivalue	When selected, returns all healthy endpoints; used for profiles limited to IPv4/IPv6 addresses as endpoints
Subn	User IP address ranges mapped to a specific endpoint within a

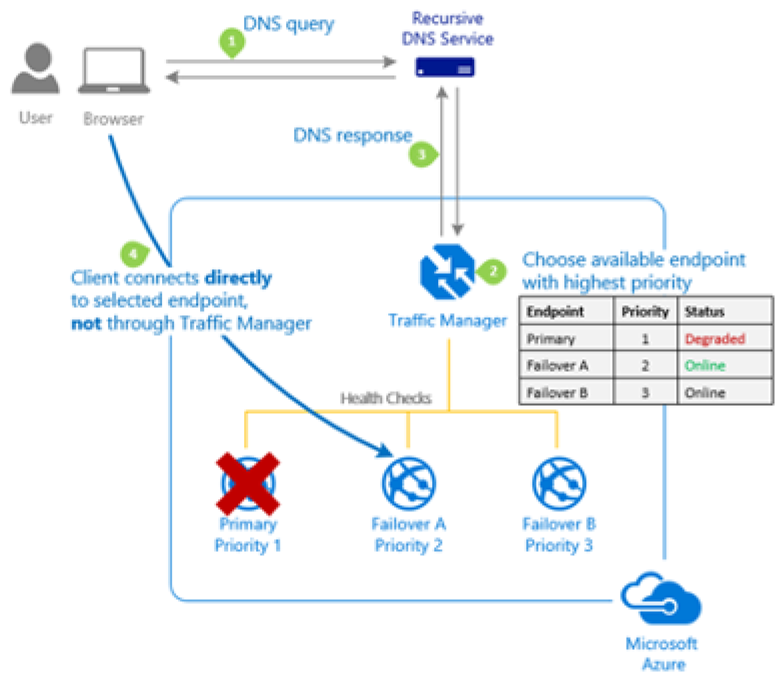
et	profile; endpoint returned is the one mapped to source IP of the request

Endpoint health and automatic endpoint failover monitoring are included in all Traffic Manager profiles. Only one routing method can be used by a single profile but can be changed at any time without downtime. You can combine routing profiles with nested Traffic Manager profiles.

Priority traffic-routing

For a high-reliability service deployment, you should configure one or more services as Failover endpoints. That way, if your Primary service goes down, traffic is routed to the Failover endpoints based on the priority set to each of the secondary (Failover) services. In Figure 2-X, when 'Primary Priority 1 fails', all traffic is automatically redirected to 'Failover A Priority 2'.

The priority can be any value between 1 and 1000, and lower values represent higher priorities. Remember that if you do not define a priority, a default priority will be automatically assigned based on endpoint order.



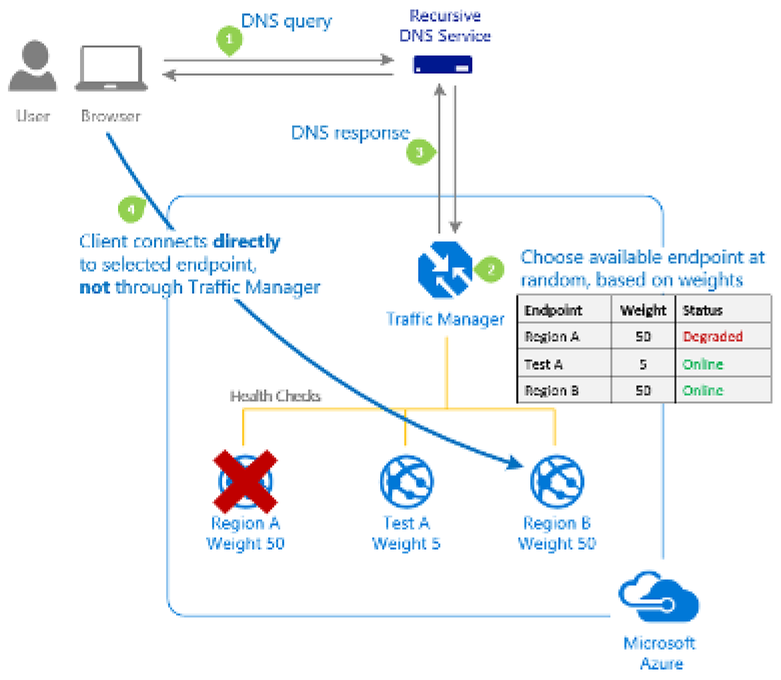
Endpoint	Priority	Status
Primary	1	Degraded
Failover A	2	Online
Failover B	3	Online

Figure 2-4. Update diagram and include reference to original MS Docs version.

Weighted traffic-routing

With this method, a weight is pre-assigned to each endpoint, and every incoming request is routed based on these values. For example, if Endpoint 1 Weight = 30%, Endpoint 2 Weight = 20%, and Endpoint 3 Weight = 50%, 30% of the traffic will be routed to Endpoint 1, 20% to endpoint and finally, half of all traffic to Endpoint 3. If you assign an equal weight distribution, every endpoint will receive the same amount of traffic.

Pilot scenarios where you would like to test a new deployment of your application (let’s say including some new features or an upgrade) frequently use this method. Here, you can start by deploying the new feature or upgrade to one of your endpoints, set a lower weight such as 10-20%, and monitor how it behaves from a systems and user behavior standpoint. If everything is running smoothly, you can then increase the weight and once it is confirmed, the release succeeds, deploy to all other nodes. Intrinsically, deploying in this phased manner, avoids Big Bang approaches and diminishes severity of issues.



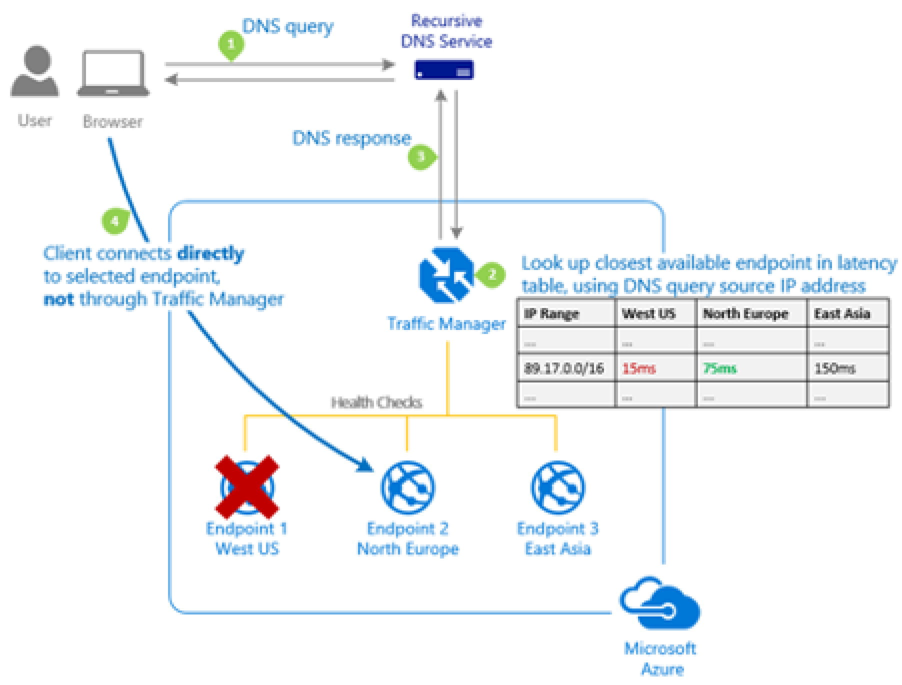
Endpoint	Weight	Status
Region A	50	Degraded
Test A	5	Online
Region B	50	Online

Figure 2-5. Update diagram and include reference to original MS Docs version.

Performance traffic-routing

The performance routing method is for when application responsiveness is a top priority. Here, you would deploy endpoints in two or more locations, and traffic is routed to the closest location. Closest doesn't necessarily mean closest in terms of distance but by measuring network latency. To determine which endpoint has the lowest network latency, the Traffic Manager service maintains an Internet Latency Table to track round-trip time between IP address ranges and each datacenter.

Remember, you can use a Nested Traffic Manager profile to define a different traffic distribution within a region or to define a failover sequence in the case that all endpoints in the closest region are degraded.


DNS query
1
Recursive DNS Service
User
Browser
DNS response
3
4
Client connects **directly** to selected endpoint, **not** through Traffic Manager
2
Look up closest available endpoint in latency table, using DNS query source IP address
Traffic Manager
Health Checks
Endpoint 1 West US
Endpoint 2 North Europe
Endpoint 3 East Asia
Microsoft Azure

IP Range	West US	North Europe	East Asia
...	...	...	...
89.17.0.0/16	15ms	75ms	150ms
...	...	...	...

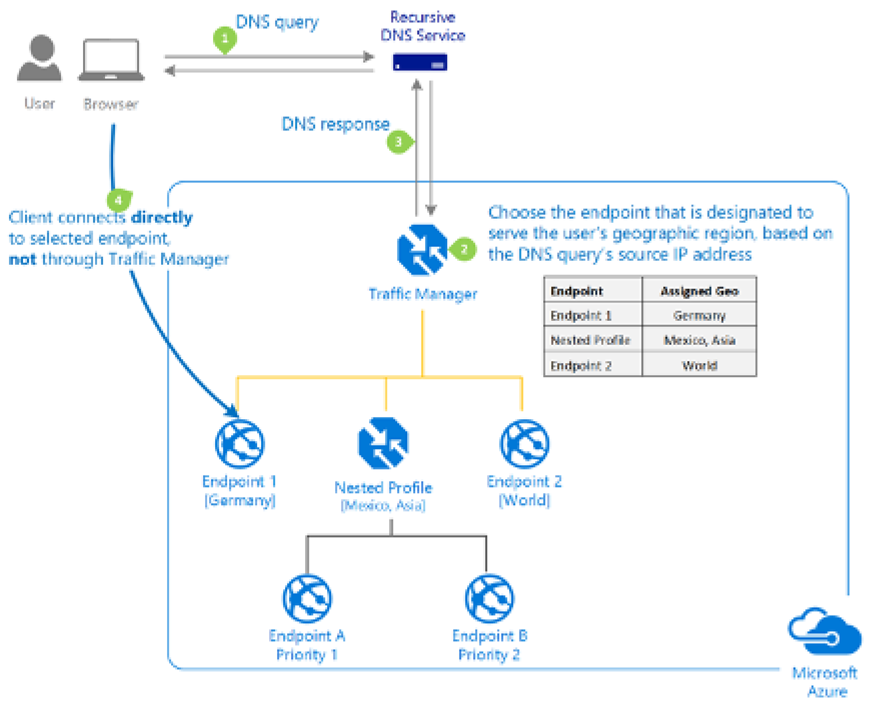
Figure 2-6. Update diagram and include reference to original MS Docs version.

GEOGRAPHIC TRAFFIC-ROUTING

Geographic traffic-routing applies rules which map client IP addresses to geographical regions. The routing is done by applying rules to match specific regions and send them to the appropriate endpoint. This is helpful for keeping requests and data within geographies, complying with GDPR for example.

The following are the types of rule matches that can be used:

- Any region
- Specific region: Africa, Middle East, Australia/Pacific
- Country/Region: Ireland, Peru
- State/Province: United States-Florida, Australia-Sydney, Canada-Toronto – this granularity level is only supported for USA, Canada, and Australia)


DNS query
1
Recursive DNS Service
User
Browser
DNS response
3
4
Client connects **directly** to selected endpoint, **not** through Traffic Manager
2
Choose the endpoint that is designated to serve the user's geographic region, based on the DNS query's source IP address
Traffic Manager
Endpoint 1 (Germany)
Nested Profile (Mexico, Asia)
Endpoint 2 (World)
Endpoint A Priority 1
Endpoint B Priority 2
Microsoft Azure

Endpoint	Assigned Geo
Endpoint 1	Germany
Nested Profile	Mexico, Asia
Endpoint 2	World

Figure 2-7. This is an image caption

With Geographic routing, if an endpoint is assigned a region or set of regions, that endpoint will receive all requests coming from those regions. Keep in mind that since a region can be mapped only to an endpoint, the Traffic Manager service will return it regardless of endpoint health. You should always use this method with Nested type endpoints that have child profiles of at least two endpoints each; otherwise, you can suffer downtime.

Designing Highly Available Systems

Remember that when designing large scale applications that are highly available, you will need several of these load balancing components together. As an example, Figure 2-X shows a geographically distributed and load balanced application. Here, the Traffic Manager will route any incoming request to the appropriate region based on performance. Once the request is received at the nearest data center, the Application Gateway will choose which application server to fetch the response from. Once the application has received the request, it needs data to be able fulfill it, from where a Load Balancer distributes the load to the Database tier.

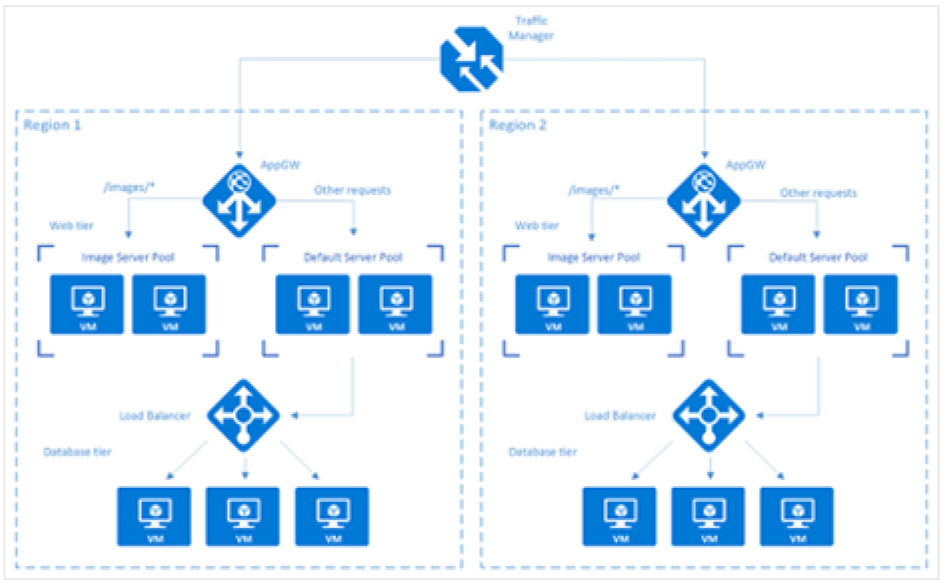


Figure 2-8. Update diagram and include reference to original MS Docs version.

Highly Available Multi Region NGINX

All of the same concepts apply when using NGINX as a load balancer or API gateway over the Azure managed offerings. Because of distribution and point of presence locations that Azure managed services provide, you should utilize the global managed services from Azure to distribute load and route client request to the correct environment region.

Figure 2-x shows a multi region deployment, using NGINX Plus as an API gateway in both regions. NGINX Plus is also being used to load balance over a database tier. Traffic is routed through Traffic Manager using the Performance algorithm, to provide our clients with responses from the least latent region.

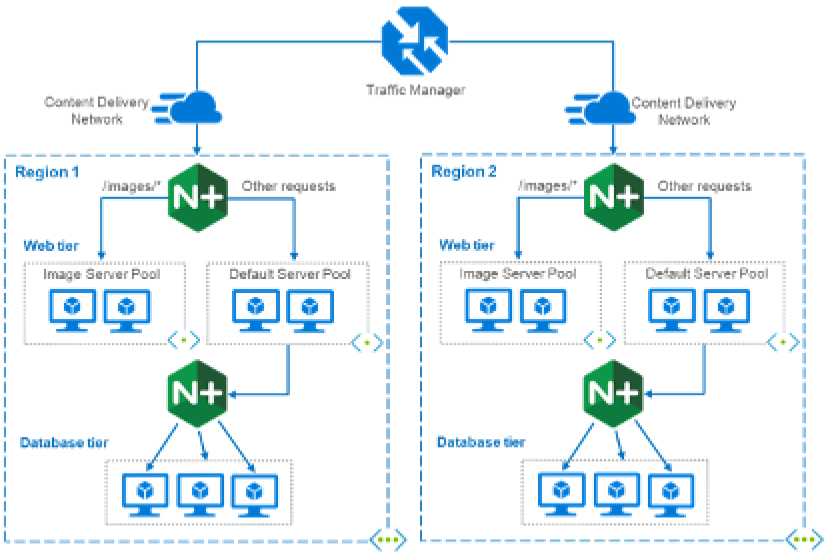


Figure 2-9. View of code solution with NGINX Plus

Figure 2-x shows a multi region deployment, using NGINX Plus as an API gateway in both regions. Traffic is routed through Traffic Manager using the Geographic rules, to direct clients in California to the US West region, and clients in New York to the US East region.

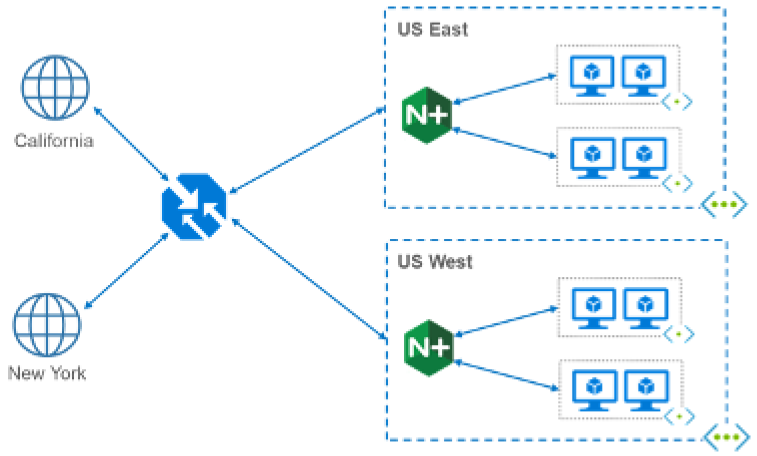


Figure 2-10. NGINX Plus along with GeoDNS creates a globally distributed application, When redrawnig remove NGINX Traffic manager arrow and add one direct from globe location to NGINX

In these scenarios Traffic Manager is directing our client to an available region that best fits our clients needs or the needs of our regulation. NGINX is providing the API gateway functionality, as well as internal load balancing. These solutions together enable High Availability failover, as well as highly configurable traffic routing within our environment.

1

Chapter 3. NGINX and NGINX Plus on Azure

NGINX the company, now part of F5 Inc., is also the name of the company's two leading products called NGINX and NGINX Plus. These two products dominate the world of application delivery controllers, both on-prem and in the cloud. For Azure, many companies are trying to decide between Azure native managed services discussed in the previous chapter and solutions they already use and trust from their on-prem environments. This chapter will explore the similarities and differences between NGINX and NGINX Plus and how to deploy them using the Azure Portal, PowerShell and Terraform. In the next chapter we'll cover comparisons between NGINX solutions and Azure managed solutions.

NGINX vs NGINX Plus

NGINX Open Source Software (OSS) is free, whereas NGINX Plus is a commercial product that offers advanced features and enterprise-level support as licensed software by NGINX.

NGINX combines the functionality of a high-performance web server, a powerful load balancer, and a highly scalable caching layer to create the ideal end-to-end platform for your web applications. NGINX Plus is built on top of NGINX OSS.

For organizations currently using NGINX OSS, NGINX Plus eliminates the complexity of managing a "do-it-yourself" chain of proxies, load balancers, and caching servers in a mission-critical application environment.

For organizations currently using hardware-based load balancers, NGINX Plus provides a full set of ADC features in a much more flexible software form factor, on a cost-effective subscription.

NGINX Plus provides enterprise-ready features such as intelligent session persistence, JWT and OpenID Connect integration, advanced monitoring statistics, and clustering abilities to your data plane.

Table 3-1 shows the NGINX Plus feature sets compared to NGINX OSS. You can get more information on the differences between NGINX products at nginx.com.

Table3-1.ComparisonofHighlightedN

G
I
N
X

O
S
S
a
n
d
N
G
I
N
X

P
l
u
s
F
e
a
t
u
r
e
s

Feature type	Feature	OSS	NGINX Plus

Load Balancer			
	HTTP/TCP/UDP support	X	X
	Layer 7 request routing	X	X
	Active health checks		X
	Sophisticated Session persistence		X
	DNS SRV support (Service Discovery)		X
Content cache			
	Static/dynamic content caching	X	X
	Cache-purging API		X
Web server/Reverse Proxy			
	Origin server for static content	X	X
	Reverse proxy protocols: TCP, UDP, HTTP, FastCGI, uwsgi, gRPC	X	X
	HTTP/2 gateway	X	X
	HTTP/2 server push	X	X
Security controls			
	HTTP Basic Authentication	X	X
	HTTP authentication subrequests	X	X
	IP address-based subrequests	X	X
	Rate limiting	X	X
	Dual-stack RSA/ECC SSL/TLS offload	X	X

	ModSecurity 3.0 support	X	X
	TLS 1.3 support	X	X
	JWT authentication		X
	OpenID Connect SSO		X
	NGINX App Protect (WAF)		X
Monitoring			
	Syslog	X	X
	AppDynamics, Datadog, Dynatrace plug-ins	X	X
	Basic Status Metrics	X	X
	Advanced Metrics with Dashboard		X
High availability			
	Behind Azure Load Balancer	X	X
	Configuration synchronization		X
	State sharing: sticky-learn session persistence, rate limiting, key-value stores		X
	Dynamic reconfiguration without process reloads		X
Progamability			
	NGINX JavaScript module	X	X
	Lua and Perl modules	X	X
	Custom C module	X	X
	NGINX Plus API dynamic configuration		X
	Key Value store		X

Installing NGINX & NGINX Plus

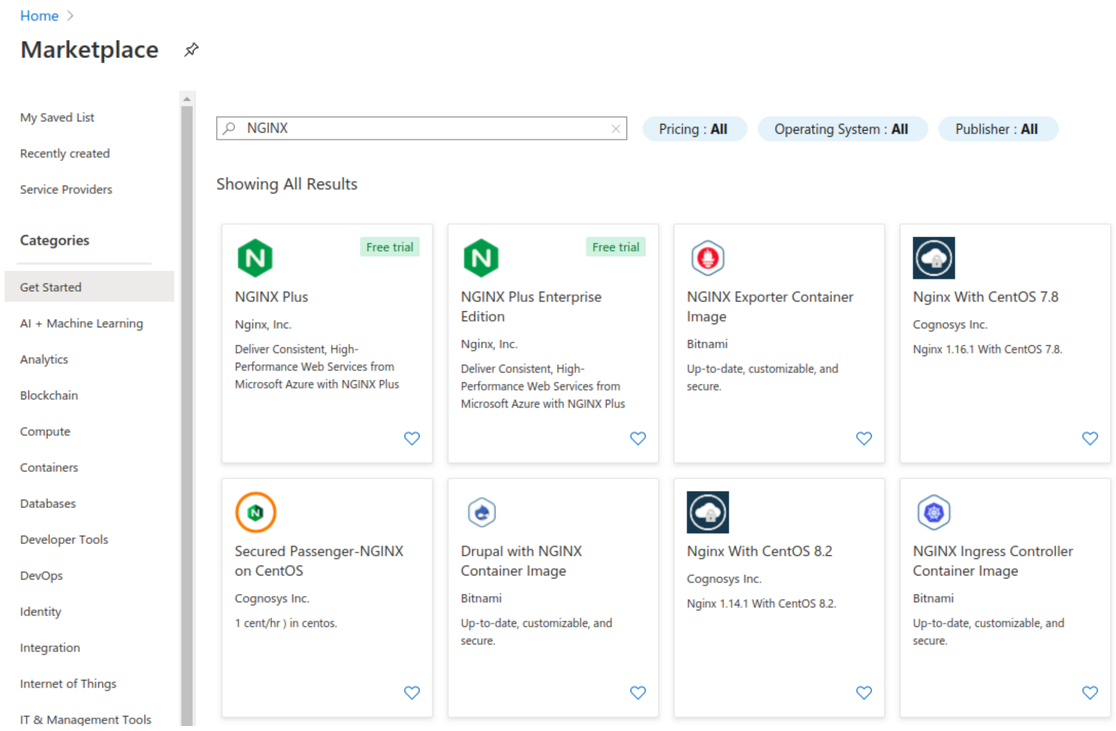
Both NGINX and NGINX Plus are widely available to download and install from a variety of sources. This flexibility allows for a number of deployment options to best suit your needs. While, you can install via pre-built Azure Virtual Machine Images available in the Azure Marketplace, manually on a virtual machine, through the Azure Resource Center with PowerShell, and more. We'll walk through the installation process for these settings below.

Installing via Azure Marketplace

Azure Marketplace is a software repository for pre-built and configured Azure resources from independent software vendors (ISVs). You will find open source and enterprise applications that have been certified and optimized to run on Azure.

NGINX, Inc. provides the latest release of NGINX Plus in Azure Marketplace as a virtual machine (VM) image. NGINX OSS is not available from NGINX, Inc., but there are several options available from other ISVs in Azure Marketplace.

Searching for “NGINX” in Azure Marketplace will produce several results, as shown in Figure 3-1.



NGINX Plus
Nginx, Inc.
Deliver Consistent, High-Performance Web Services from Microsoft Azure with NGINX Plus

NGINX Plus Enterprise Edition
Nginx, Inc.
Deliver Consistent, High-Performance Web Services from Microsoft Azure with NGINX Plus

NGINX Exporter Container Image
Bitnami
Up-to-date, customizable, and secure.

Nginx With CentOS 7.8
Cognosys Inc.
Nginx 1.16.1 With CentOS 7.8.

Secured Passenger-NGINX on CentOS
Cognosys Inc.
1 cent/hr) in centos.

Drupal with NGINX Container Image
Bitnami
Up-to-date, customizable, and secure.

Nginx With CentOS 8.2
Cognosys Inc.
Nginx 1.14.1 With CentOS 8.2.

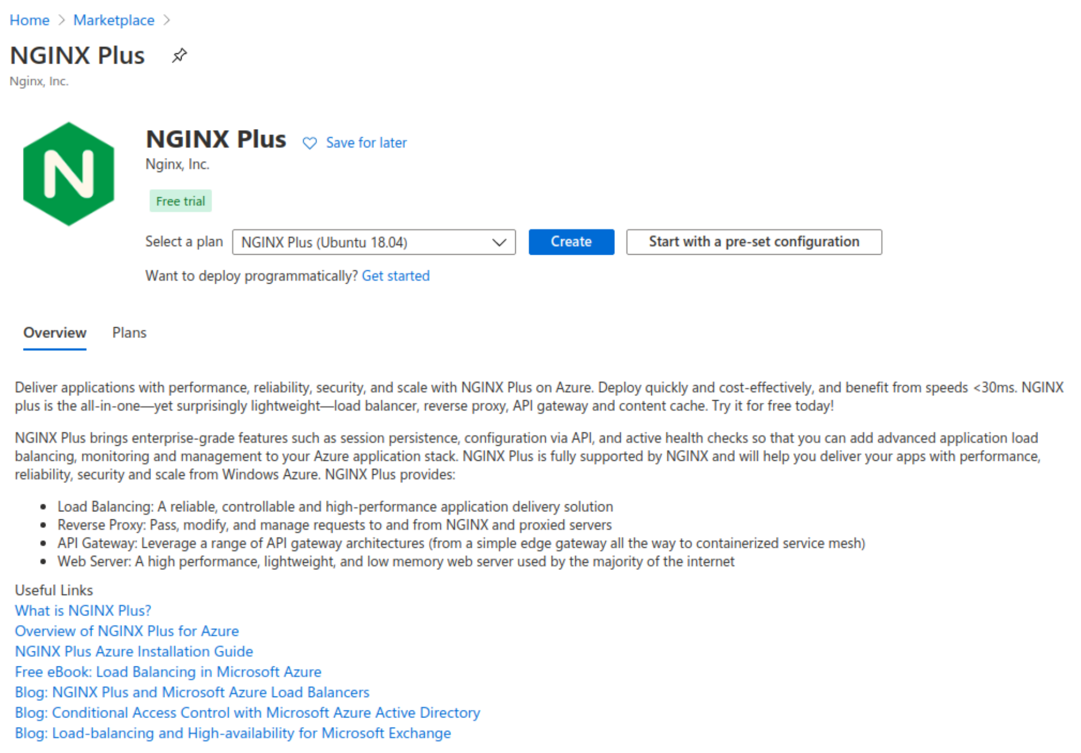
NGINX Ingress Controller Container Image
Bitnami
Up-to-date, customizable, and secure.

Figure 3-1. Searching for “NGINX” in Azure Marketplace

You will see several results besides the official NGINX Plus VM image from NGINX, Inc., such as the following examples from other ISVs for NGINX OSS:

- NGINX Web Server (Centos 7)
- NGINX Web Server on Windows Server 2016
- NGINX Ingress Controller Container Image

If you search for NGINX Plus in Azure Marketplace, there is only one option available from NGINX, Inc., as shown in Figure 3-2.



Deliver applications with performance, reliability, security, and scale with NGINX Plus on Azure. Deploy quickly and cost-effectively, and benefit from speeds <30ms. NGINX plus is the all-in-one—yet surprisingly lightweight—load balancer, reverse proxy, API gateway and content cache. Try it for free today!

NGINX Plus brings enterprise-grade features such as session persistence, configuration via API, and active health checks so that you can add advanced application load balancing, monitoring and management to your Azure application stack. NGINX Plus is fully supported by NGINX and will help you deliver your apps with performance, reliability, security and scale from Windows Azure. NGINX Plus provides:

- Load Balancing: A reliable, controllable and high-performance application delivery solution
- Reverse Proxy: Pass, modify, and manage requests to and from NGINX and proxied servers
- API Gateway: Leverage a range of API gateway architectures (from a simple edge gateway all the way to containerized service mesh)
- Web Server: A high performance, lightweight, and low memory web server used by the majority of the internet

Useful Links
What is NGINX Plus?
Overview of NGINX Plus for Azure
NGINX Plus Azure Installation Guide
Free eBook: Load Balancing in Microsoft Azure
Blog: NGINX Plus and Microsoft Azure Load Balancers
Blog: Conditional Access Control with Microsoft Azure Active Directory
Blog: Load-balancing and High-availability for Microsoft Exchange

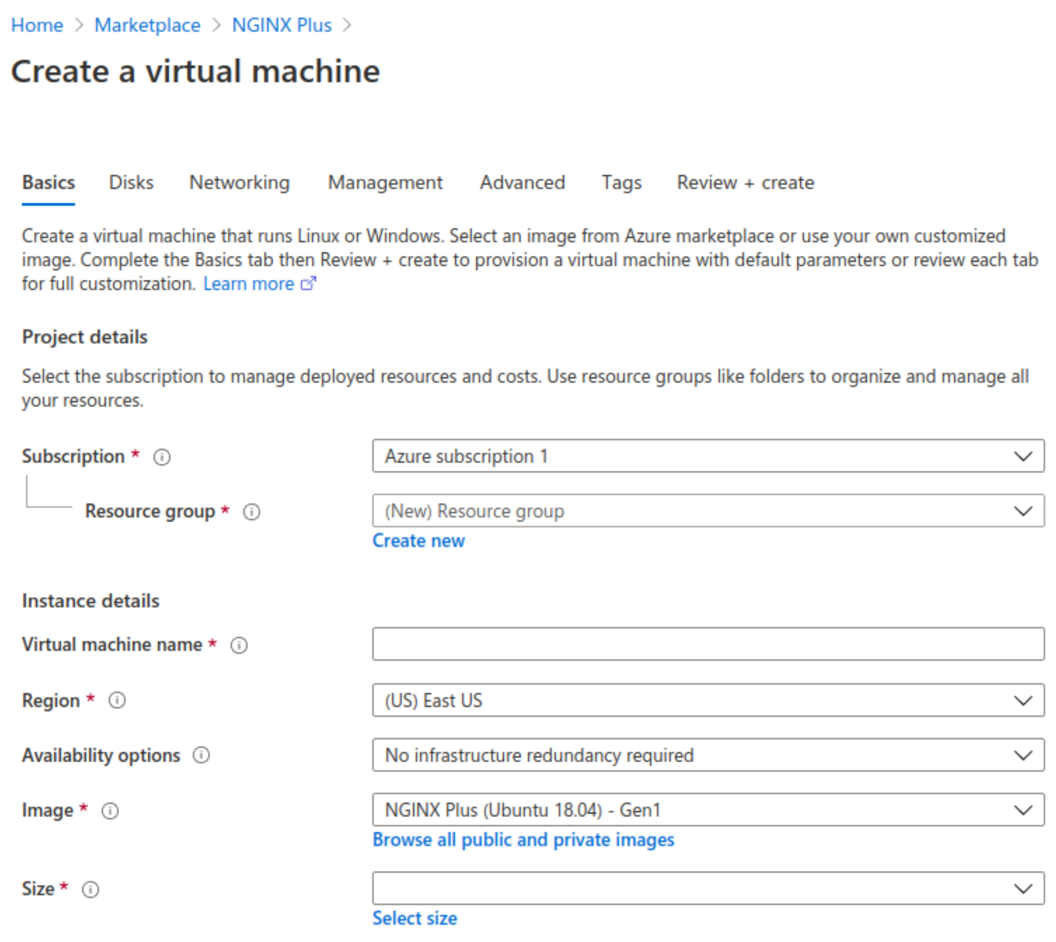
Figure 3-2. NGINX Plus in the Azure Marketplace

The initial page presented is the Overview page, which summarizes the NGINX Plus software functionality and pricing. For more details, click the “Plans” link. There are a number of plans. The plans simply provide a way to select the base OS you’d like NGINX Plus to run on. Your options are Ubuntu (18.04, 16.04), RHEL (8, 7), and CentOS 7. Select a plan and press the Create button to be taken to the Azure virtual machine creation process.

The Azure virtual machine (VM) creation process using the Azure Portal follows seven standard steps with explanations for each step on the Azure Portal page in the following areas: Basics, Disks, Networking, Management, Advanced (Settings) and Tags. The final step allows you to review and approve any associated costs before the VMs are built.

When selecting a size for your VM, a cost will be associated. This includes the cost for the NGINX Plus software.

It is recommended that an Azure availability set of two or more VMs be used to provide high-availabilty in the case of planned system maintenance by Azure or as a safe guard against one VM becoming unavailable. These seven steps are displayed in Figure 3-3.


Home > Marketplace > NGINX Plus >

Create a virtual machine

Basics | Disks | Networking | Management | Advanced | Tags | Review + create

Create a virtual machine that runs Linux or Windows. Select an image from Azure marketplace or use your own customized image. Complete the Basics tab then Review + create to provision a virtual machine with default parameters or review each tab for full customization. Learn more

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *	Azure subscription 1
Resource group *	(New) Resource group
	Create new

Instance details

Virtual machine name *	
Region *	(US) East US
Availability options	No infrastructure redundancy required
Image *	NGINX Plus (Ubuntu 18.04) - Gen1
	Browse all public and private images
Size *	
	Select size

Figure 3-3. Creating a virtual machine using the Azure Portal

You will need to manually create endpoints to support HTTPS (port 443) and HTTP traffic (port 80) in the Azure Portal to enable access to the NGINX Plus VM. For more information, see "How to set up endpoints on a Linux classic virtual machine in Azure" in the Azure documentation.

NGINX Plus will start automatically and load its default start page once the VM starts. You can use a web browser to navigate to the VM's IP address or DNS name. You can also check the running status of NGINX Plus by logging into the VM and running the following command:

```
$ /etc/init.d/nginx status
```

Azure virtual machine scale sets (VMSSs) let you create and manage a group of identical, load-balanced VMs. VMSSs provide redundancy and improved performance by automatically scaling up or down based on workloads or a predefined schedule.

To scale NGINX Plus, create a public or internal Azure load balancer with a VMSS. You can deploy the NGINX Plus VM to the VMSS and then configure the Azure load balancer for the desired rules, ports, and protocols for allowed traffic to the backend pool.

The cost of running NGINX Plus is a combination of the selected software plan charges plus the Azure infrastructure costs for the VMs on which you will be running the software. There are no additional costs for VMSSs, but you do pay for the underlying compute resources. The actual Azure infrastructure price might vary if you have enterprise agreements or other discounts.

Using NGINX Plus from the Azure Marketplace enables you to scale your NGINX Plus layer on demand without having to procure more licenses, as the software cost is built into the

Marketplace with a pay per usage model. You may want to procure a couple machine licenses for your base footprint to enter the support contract with NGINX Inc. and then use the Marketplace for burst capacity.

Installing Manually on VMs

There are cases where you may want to install NGINX manually on an Azure VM. Example use cases include, needing modules not included in a Marketplace image, extra packages, advanced configurations, latest version of NGINX, or for bootstrapping to be tightly controlled by configuration management.

The process to install NGINX or NGNIX Plus on a Azure VM, is no different than installing on any other hosting platform because NGINX is software that runs on top of any linux distribution.

In Azure your configuration should be repeatable through automation so that you can scale as necessary. You can either manually build a VM and take an image of it, so that you can use the image in an Azure Scale Set, or automate the installation through scripting or configuration management. You can also combine the two methods, so that automation builds VM images.

VM images will be ready to serve the client faster because the software is already installed. Installing at boot time provides flexibility as the configuration can change without having to create images, but takes longer to become ready as it has to install the software. A hybrid approach is to be considered, where an image is made with the software installed, but configuration management brings the configuration up to date at boot time.

When installing NGINX, I always make sure to use the NGINX official package repository for the Linux distribution that I'm using. This ensure's I've always got the latest version with the most up to date features and security fixes. You can learn how to install from the official repository by visiting the Supported Distributions and Versions page of the NGINX Documentation.

Installing via Azure Resource Manager and PowerShell

Azure Resource Manager (ARM) Templates are the native Azure automation process that uses declarative state JSON objects to build resources within Azure. It is the default option for Azure Infrastructure as Code (IaC) and allows you to check your templates into source control.

There are currently no prebuilt ARM templates or PowerShell scripts available from NGINX. However, there is nothing preventing the creation of a Resource Manager template and PowerShell script based on your custom deployment requirements for Azure using your previously created custom VM images.

The following provides an example of creating an Ubuntu 16.04 LTS marketplace image from Canonical along with the NGINX web server using Azure Cloud Shell and the Azure PowerShell module.

Open Azure Cloud Shell and perform the following steps in Azure PowerShell:

First, let's use ssh-keygen to create a Secure Shell (SSH) key pair. Accept all the defaults by pressing the Enter key:

```
# Created in directory: '/home/azureuser/.ssh'
# RSA private key will be saved as id_rsa
# RSA public key will be saved as id_rsa.pub
ssh-keygen -t rsa -b 2048
```

Before we run any Azure CLI commands we'll need to be logged in. Use the following command to receive a link and an access code that you past into a browser to verify your Azure identity:

```
Connect-AzAccount
```

Then you will create an Azure resource group by using New-AzResourceGroup:

```
New-AzResourceGroup `
```

```
-Name "nginx-rg" `
-Location "EastUS2"
```

You can now create a virtual network (New-AzVirtualNetwork), subnet (New-AzVirtualNetworkSubnetConfig), and a public IP address (New-AzPublicIpAddress):

```
# Create a subnet configuration
$subnetConfig = New-AzVirtualNetworkSubnetConfig `
-Name "nginx-Subnet" `
-AddressPrefix 192.168.1.0/24

# Create a virtual network
$vnet = New-AzVirtualNetwork `
-ResourceGroupName "nginx-rg" `
-Location "EastUS2" `
-Name "nginxVNET" `
-AddressPrefix 192.168.0.0/16 `
-Subnet $subnetConfig

# Create a public IP address
# and specify a DNS name
$pip = New-AzPublicIpAddress `
-ResourceGroupName "nginx-rg" `
-Location "EastUS2" `
-AllocationMethod Static `
-IdleTimeoutInMinutes 4 `
-Name "nginxpublicdns$(Get-Random)"
```

While optional, it is a best practice to an Azure Network Security Group (NSG) (New-AzNetworkSecurityGroup) and traffic rules using New-AzNetworkSecurityRuleConfig:

```
# Create an inbound NSG rule for port 22
$nsgRuleSSH = New-AzNetworkSecurityRuleConfig `
-Name "nginxNSGRuleSSH" `
-Protocol "Tcp" `
-Direction "Inbound" `
-Priority 1000 `
-SourceAddressPrefix * `
-SourcePortRange * `
-DestinationAddressPrefix * `
-DestinationPortRange 22 `
-Access "Allow"

# Create an inbound NSG rule for port 80
$nsgRuleWeb = New-AzNetworkSecurityRuleConfig `
-Name "nginxNSGRuleWWW" `
-Protocol "Tcp" `
-Direction "Inbound" `
-Priority 1001 `
-SourceAddressPrefix * `
-SourcePortRange * `
-DestinationAddressPrefix * `
-DestinationPortRange 80 `
-Access "Allow"

# Create a network security group (NSG)
$nsg = New-AzNetworkSecurityGroup `
  -ResourceGroupName "nginx-rg" `
  -Location "EastUS2" `
  -Name "nginxNSG" `
  -SecurityRules $nsgRuleSSH,$nsgRuleWeb

# Create a virtual network card and
# associate it with the public IP
# address and NSG
$nic = New-AzNetworkInterface `
-Name "nginxNIC" `
-ResourceGroupName "nginx-rg" `
```

```
-Location "EastUS2" `
-SubnetId $vnet.Subnets[0].Id `
-PublicIpAddressId $pip.Id `
-NetworkSecurityGroupId $nsg.Id
```

PowerShell allows you to quickly build a VM while specifying VM attributes such memory, vCPUs, disks, network cards based on the VM image options available on Azure. The following is the configuration of the VM suitable for our example:

```
# Define a credential object
$securePassword = ConvertTo-SecureString `
' ' -AsPlainText -Force
$cred = New-Object `
System.Management.Automation.PSCredential("azureuser",
$securePassword)

# Create a virtual machine configuration
$vmConfig = New-AzVMConfig `
-VMName "nginxVM" `
-VMSize "Standard_B1s" | `
Set-AzVMOperatingSystem `
-Linux `
-ComputerName "nginxVM" `
-Credential $cred `
-DisablePasswordAuthentication | `
Set-AzVMSourceImage `
-PublisherName "Canonical" `
-Offer "UbuntuServer" `
-Skus "16.04-LTS" `
-Version "latest" | `
Add-AzVMNetworkInterface `
-Id $nic.Id
```

```
# Configure the SSH key
$sshPublicKey = cat ~/.ssh/id_rsa.pub
Add-AzVMSshPublicKey `
-VM $vmconfig `
-KeyData $sshPublicKey `
-Path "/home/azureuser/.ssh/authorized_keys"
```

Now, combine the previous configuration definitions to create a new VM by using New-AzVM:

```
New-AzVM `
-ResourceGroupName "nginx-rg" `
-Location eastus2 -VM $vmConfig
```

Connect using SSH to the VM after it is created by using the public IP displayed by the following code:

```
Get-AzPublicIpAddress `
-ResourceGroupName "nginx-rg" | `
Select "IpAddress"
```

In the Azure Cloud Shell or your local bash shell, paste the SSH connection command into the shell to create an SSH session using the login user name azureuser when prompted. If an optional passphrase is used, please enter it when prompted:

```
ssh azureuser@<vm-public-ip>
```

From your SSH session, update your package sources and then install the latest NGINX package by running the following as root or with `sudo`:

```
echo \
 "deb http://nginx.org/packages/mainline/ubuntu/ xenial
nginx" \
 > /etc/apt/sources.list.d/nginx.list
echo \
 "deb-src http://nginx.org/packages/mainline/ubuntu/
xenial nginx" \
 >> /etc/apt/sources.list.d/nginx.list
wget http://nginx.org/keys/nginx_signing.key
apt-key add nginx_signing.key
apt-get update
apt-get -y install nginx

# Test NGINX is installed
nginx -v

# Start NGINX - it's enable to start at boot by default
/etc/init.d/nginx start
```

You will need to use a web browser to test the loading of the default NGINX start page which is the public IP address of the VM created. To exit the SSH session, type `exit` when done.

Once you have completed this process, you can remove the Azure resources by using the Remove-AzResourceGroup cmdlet to remove the resource group, VM, virtual network and all other Azure resources to avoid incurring ongoing charges:

```
Remove-AzResourceGroup `
-Name "nginx-rg"
```

Installing NGINX (Open Source) via Terraform

In this section, we will deploy a Linux Virtual Machine with NGINX using Terraform. We will show two examples, one for Debian and Ubuntu and another for CentOS & RedHat. There are items common to both of them: the provider and the network that will provide the starting point for installing NGINX.

IF YOU'RE UNFAMILIAR WITH TERRAFORM

You can learn about Terraform by reading the Introduction to Terraform document, or going through the Getting Started Guide.

THE PROVIDER

The first step is to create the provider file. The provider is used to interact with the Azure APIs.

We create a file called provider-main.tf used to create the interaction with Terraform and the Azure providers.

```
# Define Terraform provider
terraform {
  required_version = ">= 0.12"
}

# Configure the Azure provider
provider "azurerm" {
  environment = "public"
  version     = ">= 2.15.0"
  features {}
  subscription_id = var.azure-subscription-id
  client_id       = var.azure-client-id
  client_secret   = var.azure-client-secret
  tenant_id       = var.azure-tenant-id
}
```

Next, we create a file called provider-variables.tf used to manage the authentication variables of the Azure provider.

```
variable "azure-subscription-id" {
  type        = string
  description = "Azure Subscription ID"
}

variable "azure-client-id" {
  type        = string
  description = "Azure Client ID"
}

variable "azure-client-secret" {
  type        = string
  description = "Azure Client Secret"
}
```

```
variable "azure-tenant-id" {
  type        = string
  description = "Azure Tenant ID"
}
```

THE NETWORK

The next step is to create the Resource Group that will host all of our Azure resources. A VNET, and a subnet within the VNET will also be created. The subnet will be host our virtual machine.

We will create a file called *network-main.tf* to describe these resources.

```
# Create a resource group
resource "azurerm_resource_group" "network-rg" {
  name     = "nginx-network-rg"
  location = var.location
}

# Create the network VNET
resource "azurerm_virtual_network" "network-vnet" {
  name                = "nginx-network-vnet"
  address_space       = [var.network-vnet-cidr]
  resource_group_name = azurerm_resource_group.network-
rg.name
  location            = azurerm_resource_group.network-
rg.location
}

# Create a subnet for VM
resource "azurerm_subnet" "vm-subnet" {
  name                 = "nginx-vm-subnet"
  address_prefixes     = [var.vm-subnet-cidr]
  virtual_network_name =
azurerm_virtual_network.network-vnet.name
  resource_group_name  =
azurerm_resource_group.network-rg.name
}
```

Then, we will create the file network-variables.tf for manage network variables.

```
variable "location" {
  type        = string
  description = "Azure Region"
  default     = "eastus"
}

variable "network-vnet-cidr" {
  type        = string
  description = "The CIDR of the network VNET"
}

variable "vm-subnet-cidr" {
  type        = string
  description = "The CIDR for the vm subnet"
}
```

SECURITY

In this section, we will create an Azure NSG (Network Security Group) to protect our virtual machine.

First, we create a file called security-main.tf and add the following code:

```
# Create Network Security Group
resource "azurerm_network_security_group" "nginx-vm-
nsg" {
  depends_on=[azurerm_resource_group.network-rg]

  name                = "nginxvm-nsg"
  location            = azurerm_resource_group.network-
rg.location
  resource_group_name = azurerm_resource_group.network-
rg.name

  security_rule {
    name                       = "Allow-SSH"
    description                = "Allow SSH"
    priority                   = 100
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "Tcp"
    source_port_range          = "*"
    destination_port_range     = "22"
    source_address_prefix      = "Internet"
    destination_address_prefix = "*"
  }

  security_rule {
    name                       = "Allow-HTTP"
    description                = "Allow HTTP"
    priority                   = 110
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "Tcp"
    source_port_range          = "*"
    destination_port_range     = "80"
    source_address_prefix      = "Internet"
    destination_address_prefix = "*"
  }
}
```

```
# Associate the web NSG with the subnet
resource
"azurerm_subnet_network_security_group_association"
"nginx-vm-nsg-association" {
  depends_on=[azurerm_resource_group.network-rg]

  subnet_id                 = azurerm_subnet.vm-
subnet.id
  network_security_group_id =
azurerm_network_security_group.nginx-vm-nsg.id
}
```

Deploying NGINX in Debian and Ubuntu Linux

In this section, we are going to learn how to deploy a virtual machine with NGINX running Ubuntu Linux. This code will work without major changes on Debian, we just need to update the source_image_reference section (instructions are at the end of this chapter).

If you are using CentOS or RedHat, please jump ahead to the next section, Deploying NGINX in CentOS and RedHat Linux.

BOOTSTRAPPING SCRIPT TO INSTALL NGINX

In this step, we will create a Bash script called install-nginx.sh to install NGINX in the virtual machine.

```
#! /bin/bash
echo \
 "deb http://nginx.org/packages/mainline/ubuntu/ xenial
nginx" \
 > /etc/apt/sources.list.d/nginx.list
echo \
 "deb-src http://nginx.org/packages/mainline/ubuntu/
xenial nginx" \
 >> /etc/apt/sources.list.d/nginx.list
wget http://nginx.org/keys/nginx_signing.key
apt-key add nginx_signing.key
apt-get update
apt-get -y install nginx

# Test NGINX is installed
nginx -v

# Start NGINX - it's enable to start at boot by default
/etc/init.d/nginx start
```

CREATING THE VIRTUAL MACHINE

Here, we will create a file called vm-nginx-main.tf, this file will load the bootstrapping script, get a public IP address and create a virtual machine.

```
# Bootstrapping Template File
data "template_file" "nginx-vm-cloud-init" {
  template = file("install-nginx.sh")
}

# Generate random password
resource "random_password" "nginx-vm-password" {
  length           = 16
  min_upper        = 2
  min_lower        = 2
  min_special      = 2
  number           = true
  special          = true
  override_special = "!@#$%&"
}

# Get a Static Public IP
resource "azurerm_public_ip" "nginx-vm-ip" {
  depends_on=[azurerm_resource_group.network-rg]

  name                = "nginxvm-ip"
  location            = azurerm_resource_group.network-
rg.location
  resource_group_name = azurerm_resource_group.network-
rg.name
  allocation_method   = "Static"
}

# Create Network Card for the VM
resource "azurerm_network_interface" "nginx-nic" {
  depends_on=[azurerm_resource_group.network-rg]

  name                = "nginxvm-nic"
  location            = azurerm_resource_group.network-
rg.location
  resource_group_name = azurerm_resource_group.network-
rg.name

  ip_configuration {
    name                          = "internal"
    subnet_id                     = azurerm_subnet.vm-
subnet.id
    private_ip_address_allocation = "Dynamic"
    public_ip_address_id          =
azurerm_public_ip.nginx-vm-ip.id
  }
}

# Create NGINX VM
resource "azurerm_linux_virtual_machine" "nginx-vm" {
  depends_on=[azurerm_network_interface.nginx-nic]
  name                  = "nginxvm"
  location              =
azurerm_resource_group.network-rg.location
  resource_group_name   =
azurerm_resource_group.network-rg.name
  network_interface_ids =
[azurerm_network_interface.nginx-nic.id]
  size                  = var.nginx_vm_size

  source_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "18.04-LTS"
    version   = "latest"
  }

  os_disk {
    name                 = "nginxvm-osdisk"
    caching              = "ReadWrite"
    storage_account_type = "Standard_LRS"
  }

  computer_name  = "nginxvm"
  admin_username = var.nginx_admin_username
  admin_password = random_password.nginx-vm-
password.result
  custom_data    =
base64encode(data.template_file.nginx-vm-cloud-
init.rendered)

  disable_password_authentication = false
}
```

Then, create a file called vm-nginx-variables.tf to manage variables for virtual machines:

```
variable "nginx_vm_size" {
  type        = string
  description = "Size (SKU) of the virtual machine to
create"
}

variable "nginx_admin_username" {
  description = "Username for Virtual Machine
administrator account"
  type        = string
  default     = ""
}

variable "nginx_admin_password" {
  description = "Password for Virtual Machine
administrator account"
  type        = string
  default     = ""
}
```

Deploying NGINX in CentOS and RedHat Linux

In this section, we will deploy a virtual machine with NGINX running CentOS Linux. If you prefer Ubuntu, you can skip these next two sections as they overwrite the files created above. This code will work without major changes on a RedHat system, we just need to update the NGINX package repository, replacing `centos` with `rhel`, and the `source_image_reference` section in the *vm-nginx-main.tf* file.

BOOTSTRAPPING SCRIPT TO INSTALL NGINX

In this step, we will overwrite the Bash script used in the Ubuntu section above to install NGINX through yum during the bootstrapping of the virtual machine. Replace the *install-nginx.sh* file with the following:

```
#! /bin/bash
echo "[nginx]
name=nginx repo
baseurl=http://nginx.org/packages/mainline/centos/7/$basearch/

gpgcheck=0
enabled=1" > /etc/yum.repos.d/nginx.repo

yum -y install nginx
systemctl enable nginx
systemctl start nginx
firewall-cmd --permanent --zone=public --add-
port=80/tcp
firewall-cmd --reload
```

Creating the Virtual Machine

Here, we will replace the file called *vm-nginx-main.tf*, this file will load the bootstrapping script, get a public IP address, and create a CentOS based virtual machine that runs the bash shell at boot.

```
# Bootstrapping Template File
data "template_file" "nginx-vm-cloud-init" {
  template = file("install-nginx.sh")
}
```

```
# Generate random password
resource "random_password" "nginx-vm-password" {
  length           = 16
  min_upper        = 2
  min_lower        = 2
  min_special      = 2
  number           = true
  special          = true
  override_special = "!@#$%&"
}

# Get a Static Public IP
resource "azurerm_public_ip" "nginx-vm-ip" {
  depends_on=[azurerm_resource_group.network-rg]

  name                = "nginxvm-ip"
  location            = azurerm_resource_group.network-
rg.location
  resource_group_name = azurerm_resource_group.network-
rg.name
  allocation_method   = "Static"
}

# Create Network Card for the VM
resource "azurerm_network_interface" "nginx-nic" {
  depends_on=[azurerm_resource_group.network-rg]

  name                = "nginxvm-nic"
  location            = azurerm_resource_group.network-
rg.location
  resource_group_name = azurerm_resource_group.network-
rg.name

  ip_configuration {
    name                          = "internal"
    subnet_id                     = azurerm_subnet.vm-
subnet.id
    private_ip_address_allocation = "Dynamic"
    public_ip_address_id          =
azurerm_public_ip.nginx-vm-ip.id
  }
}

# Create NGINX VM
resource "azurerm_linux_virtual_machine" "nginx-vm" {
  depends_on=[azurerm_network_interface.nginx-nic]

  name                  = "nginxvm"
  location              =
azurerm_resource_group.network-rg.location
  resource_group_name   =
azurerm_resource_group.network-rg.name
  network_interface_ids =
[azurerm_network_interface.nginx-nic.id]
  size                  = var.nginx_vm_size

  source_image_reference {
    publisher = "OpenLogic"
    offer     = "CentOS"
    sku       = "7_8-gen2"
    version   = "latest"
  }

  os_disk {
    name                 = "nginxvm-osdisk"
    caching              = "ReadWrite"
    storage_account_type = "Standard_LRS"
  }

  computer_name  = "nginxvm"
  admin_username = var.nginx_admin_username
  admin_password = random_password.nginx-vm-
password.result
  custom_data    =
base64encode(data.template_file.nginx-vm-cloud-
init.rendered)
```

```
    disable_password_authentication = false
  }
```

Creating the Input Variables File

We can provide values to our variables through the *terraform.tfvars* file, this will make calling the `terraform` command line tool simpler.

```
location          = "eastus"
network-vnet-cidr = "10.0.0.0/24"
vm-subnet-cidr    = "10.0.0.0/26"

nginx_vm_size        = "Standard_B1s"
nginx_admin_username = "tfadmin"

azure-subscription-id = "complete-here"
azure-client-id       = "complete-here"
azure-client-secret   = "complete-here"
azure-tenant-id       = "complete-here"
```

Running Terraform

Before we run `terraform`, it's a good idea to use the `plan` command to discover what Terraform intends on doing in our Azure account.

```
terraform plan
```

If you approve the plan, you can apply the Terraform to your Azure account by running the following. You will be prompted to approve, you have to type `yes`.

```
terraform apply
```

After Terraform runs, you can go find your newly created resources in Azure, and use the IP address to view the default NGINX landing page.

Installing NGINX Plus via Terraform

In this section, we will deploy a Linux Virtual Machine with NGINX Plus using Terraform. Unlike the open-source version, in this section we will deploy a virtual machine image preinstalled with NGINX Plus from the Azure Marketplace. Currently, NGINX Plus suggested Azure VM sizes are:

- Standard_A1
- Standard_A2
- Standard_A3

Before we get started with the Terraform, we need to accept Azure Marketplace Terms, using the following PowerShell script:

```
Get-AzMarketplaceTerms -Publisher "nginxinc" -Product
"nginx-plus-v1" -Name "nginx-plus-ub1804" | Set-
AzMarketplaceTerms -Accept
```

HOW TO FIND AZURE NGINX PLUS VM IMAGES FOR TERRAFORM USING POWERSHELL

To deploy a NGINX Plus virtual machine, we will need to find the publisher, offer and sku parameters of the Azure MarketPlace source image, using PowerShell.

Start by defining the Azure region you'd like to provision into using a variable.

```
$Location = “East US”
```

Then set the a variable to hold the name of the publisher, `nginxinc`, and query the list of Offers. For NGINX images, the publisher is called nginxinc.

```
$publisher = “nginxinc”
Get-AzVMImageOffer -Location $location -PublisherName
$publisher | Select Offer
```

These are the Results:

```
Offer
-----
nginx-plus-ent-v1
nginx-plus-v1
```

Then, we will list SKUs for NGINX Plus. We do not want the enterprise agreement because that requires use to bring our own license. We'll instead use the standard offering to pay for the software license by hour:

```
$offer = "nginx-plus-v1"
Get-AzVMImageSku -Location $location -PublisherName
$publisher -Offer $offer | Select Skus
```

These are the resulting Skus:

```
Skus
----
nginx-plus-centos7
nginx-plus-q1fy17
nginx-plus-rhel7
nginx-plus-rhel8
nginx-plus-ub1604
nginx-plus-ub1804
```

As we can see there are several options of operating systems to deploy NGINX Plus on Azure: CentOS Linux 7, RedHat Enterprise Linux 7 & 8 and Ubuntu Linux 16.04 & 18.04.

If we want to use the Enterprise version of NGINX Plus, we can use the following code to list SKUs.

```
$offer = "nginx-plus-ent-v1"
Get-AzVMImageSku -Location $location -PublisherName
$publisher -Offer $offer | Select Skus
```

The result will be.

```
Skus
----
nginx-plus-ent-centos7
nginx-plus-ent-rhel7
nginx-plus-ent-ub1804
```

THE PROVIDER

The first step is to create the provider file for Terraform. The provider is used to interact with APIs.

We create a file called *provider-main.tf*, used to create the interaction with Terraform and Azure providers.

```
# Define Terraform provider
terraform {
  required_version = ">= 0.12"
}

# Configure the Azure provider
```

```
provider "azurerm" {
  environment = "public"
  version     = ">= 2.15.0"
  features {}
  subscription_id = var.azure-subscription-id
  client_id       = var.azure-client-id
  client_secret   = var.azure-client-secret
  tenant_id       = var.azure-tenant-id
}
```

Next, we create a file called *provider-variables.tf*, used to manage the authentication variables of the Azure provider.

```
variable "azure-subscription-id" {
  type        = string
  description = "Azure Subscription ID"
}

variable "azure-client-id" {
  type        = string
  description = "Azure Client ID"
}

variable "azure-client-secret" {
  type        = string
  description = "Azure Client Secret"
}

variable "azure-tenant-id" {
  type        = string
  description = "Azure Tenant ID"
}
```

THE NETWORK

The next step is to create the Resource Group that will host all of our Azure resources. A VNET, and a subnet within the VNET will also be created. The subnet will be host our virtual machine.

We will create a file called *network-main.tf* to describe these resources.

```
# Create a resource group
resource "azurerm_resource_group" "network-rg" {
  name     = "nginx-network-rg"
  location = var.location
}

# Create the network VNET
resource "azurerm_virtual_network" "network-vnet" {
  name                = "nginx-network-vnet"
  address_space       = [var.network-vnet-cidr]
  resource_group_name = azurerm_resource_group.network-
rg.name
  location            = azurerm_resource_group.network-
rg.location
}

# Create a subnet for VM
resource "azurerm_subnet" "vm-subnet" {
  name                 = "nginx-vm-subnet"
  address_prefixes     = [var.vm-subnet-cidr]
  virtual_network_name =
azurerm_virtual_network.network-vnet.name
  resource_group_name  =
azurerm_resource_group.network-rg.name
}
```

Then, we will create the file *network-variables.tf* for manage network variables.

```
variable "location" {
  type        = string
  description = "Azure Region"
  default     = "eastus"
}

variable "network-vnet-cidr" {
  type        = string
  description = "The CIDR of the network VNET"
}

variable "vm-subnet-cidr" {
  type        = string
  description = "The CIDR for the vm subnet"
}
```

SECURITY

In this section, we will create an Azure NSG (Network Security Group) to protect our virtual machine. The security group will allow inbound traffic in ports 22 (SSH), 80 (HTTP) and 443 (HTTPS).

We create a file called *security-main.tf* and add the following code:

```
# Create Network Security Group
resource "azurerm_network_security_group" "nginx-vm-
```

```
nsg" {
  depends_on=[azurerm_resource_group.network-rg]

  name                = "nginxvm-nsg"
  location            = azurerm_resource_group.network-
rg.location
  resource_group_name = azurerm_resource_group.network-
rg.name

  security_rule {
    name                       = "Allow-SSH"
    description                = "Allow SSH"
    priority                   = 100
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "Tcp"
    source_port_range          = "*"
    destination_port_range     = "22"
    source_address_prefix      = "Internet"
    destination_address_prefix = "*"
  }

  security_rule {
    name                       = "Allow-HTTP"
    description                = "Allow HTTP"
    priority                   = 110
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "Tcp"
    source_port_range          = "*"
    destination_port_range     = "80"
    source_address_prefix      = "Internet"
    destination_address_prefix = "*"
  }

  security_rule {
    name                       = "Allow-HTTPS"
    description                = "Allow HTTPS"
    priority                   = 120
```

```
      direction                  = "Inbound"
      access                     = "Allow"
      protocol                   = "Tcp"
      source_port_range          = "*"
      destination_port_range     = "443"
      source_address_prefix      = "Internet"
      destination_address_prefix = "*"
    }
  }

  # Associate the web NSG with the subnet
  resource
  "azurerm_subnet_network_security_group_association"
  "nginx-vm-nsg-association" {
    depends_on=[azurerm_resource_group.network-rg]

    subnet_id                 = azurerm_subnet.vm-
  subnet.id
    network_security_group_id =
  azurerm_network_security_group.nginx-vm-nsg.id
  }
```

DEFINE A NGINX PLUS VIRTUAL MACHINE

In this section, we will define a virtual machine with NGINX Plus.

First, we are going to create a file called *vm-nginx-main.tf* and add code to generate a random password and a random virtual machine name.

```
  # Generate random password
  resource "random_password" "nginx-vm-password" {
    length           = 16
    min_upper        = 2
    min_lower        = 2
    min_special      = 2
    number           = true
    special          = true
    override_special = "!@#$%&"
  }

  # Generate a random vm name
  resource "random_string" "nginx-vm-name" {
    length  = 8
    upper   = false
    number  = false
    lower   = true
    special = false
  }
```

Then to the same file, we are going to add the code to request a public IP address, generate a Network Card, and assign the public ip address to it.

```

# Get a Static Public IP
resource "azurerm_public_ip" "nginx-vm-ip" {
  depends_on=[azurerm_resource_group.network-rg]

  name                = "nginx-${random_string.nginx-
vm-name.result}-ip"
  location            = azurerm_resource_group.network-
rg.location
  resource_group_name = azurerm_resource_group.network-
rg.name
  allocation_method   = "Static"
}

# Create Network Card for the VM
```

```
resource "azurerm_network_interface" "nginx-nic" {
  depends_on=[azurerm_resource_group.network-rg]

  name                = "nginx-${random_string.nginx-
vm-name.result}-nic"
  location            = azurerm_resource_group.network-
rg.location
  resource_group_name = azurerm_resource_group.network-
rg.name

  ip_configuration {
    name                          = "internal"
    subnet_id                     = azurerm_subnet.vm-
subnet.id
    private_ip_address_allocation = "Dynamic"
    public_ip_address_id          =
azurerm_public_ip.nginx-vm-ip.id
  }
}
```

Next, we will add the definition to create the virtual machine with the NGINX Plus.

```
# Create NGINX VM
resource "azurerm_linux_virtual_machine" "nginx-vm" {
  depends_on=[azurerm_network_interface.nginx-nic]

  name                  = "nginx-${random_string.nginx-
vm-name.result}-vm"
  location              =
azurerm_resource_group.network-rg.location
  resource_group_name   =
azurerm_resource_group.network-rg.name
  network_interface_ids =
[azurerm_network_interface.nginx-nic.id]
```

```
  size                  = var.nginx_vm_size

  source_image_reference {
    publisher = var.nginx-publisher
    offer     = var.nginx-plus-offer
    sku       = "nginx-plus-ub1804"
    version   = "latest"
  }

  plan {
    name      = "nginx-plus-ub1804"
    publisher = var.nginx-publisher
    product   = var.nginx-plus-offer
  }

  os_disk {
    name                 =
"nginx-${random_string.nginx-vm-name.result}-osdisk"
    caching              = "ReadWrite"
    storage_account_type = "Standard_LRS"
  }

  computer_name  = "nginx-${random_string.nginx-vm-
name.result}-vm"
  admin_username = var.nginx_admin_username
  admin_password = random_password.nginx-vm-
password.result

  disable_password_authentication = false
}
```

Finally, we will create a file called *vm-nginx-variables.tf* to manage variables for virtual machines:

```
variable "nginx_vm_size" {
  type        = string
  description = "Size (SKU) of the virtual machine to
create"
}

variable "nginx_admin_username" {
  description = "Username for Virtual Machine
administrator account"
  type        = string
  default     = ""
}

variable "nginx_admin_password" {
  description = "Password for Virtual Machine
administrator account"
  type        = string
  default     = ""
}

variable "nginx-publisher" {
  type        = string
  description = "Publisher ID for NGINX"
  default     = "nginxinc"
}

variable "nginx-plus-offer" {
  type        = string
  description = "Offer ID for NGINX"
  default     = "nginx-plus-v1"
}
```

CREATING THE INPUT VARIABLES FILE

We can provide values to our variables through the *terraform.tfvars* file, this will make calling the `terraform` command line tool simpler.

```
location          = "eastus"
network-vnet-cidr = "10.0.0.0/24"
vm-subnet-cidr    = "10.0.0.0/26"

nginx_vm_size        = "Standard_B1s"
nginx_admin_username = "tfadmin"

azure-subscription-id = "complete-here"
azure-client-id       = "complete-here"
azure-client-secret   = "complete-here"
azure-tenant-id       = "complete-here"
```

Running Terraform

Before we run `terraform`, it's a good idea to use the `plan` command to discover what Terraform intends on doing in our Azure account.

```
terraform plan
```

If you approve the plan, you can apply the Terraform to your Azure account by running the following. You will be prompted to approve, you have to type `yes`.

```
terraform apply
```

After Terraform runs, you can go find your newly created resources in Azure, and use the IP address to view the default

NGINX landing page.

Conclusion

This chapter was a chance to deploy both NGINX and NGINX Plus and to explore the levels of functionality available from both products as well as the differences. NGINX Open Source Software (OSS) is free but requires a better understanding of how to deploy it and how to make best use of its feature set. NGINX Plus has several varied and convenient options for deployment and is a commercial product that offers advanced features and enterprise-level support as licensed software by NGINX, Inc.

We deployed NGINX and NGINX Plus using a combination of the Azure Portal, PowerShell, and Terraform to see the options available. Terraform provided the most complete solution for NGINX and NGINX Plus allowing the greatest levels of automation and integration into a full Azure deployment scenario.

Chapter 4. NGINX and Microsoft Managed Options

Microsoft Azure provides a number of different proxy-like, data plane level services that forward a request or connection through different networking layers, load balancing and applying rules along the way. NGINX provides much of the same functionality as these services but can reside deeper in the stack, and has less configuration limitation. When delivering applications hosted in Microsoft Azure, you need to determine what controls are needed where, and how best to provide those.

Most of the time the right answer is not, one service or the other, but a mix. By layering proxy like services in front of your application, you're able to maintain more control, and distribute the incoming load. The Azure services are meant to compliment each other, by layering them throughout the stack. NGINX can be interchanged with the Azure services that reside with in the Azure Virtual Network. A major value add of Azure managed services, is that they're manged and do not require maintenance and care on your part.

The Azure managed services that provide data plane, proxy like services are: Azure Front Door, CDN Profiles, Application Gateway, and Load Balancers. All of them have valid use cases, some have overlapping features, all of them can be front ends for NGINX. Azure Front Door is covered in depth in chapter 5, this chapter will focus on Azure Load Balancers, the Application Gateway, and the integration with Azure WAF policies. CDN Profiles, while they do act as a proxy, are not designed for Load Balancing, therefore are not in focus for this book.

Comparing NGINX Plus and Azure Load Balancing Services

Azure Load Balancers operate at layer 4 of the OSI model, the transport layer. This means that the Azure Load Balancer is chauffeuring the connection from the client to a backend server. As the connection is direct between the client and the server, Azure Load Balancer is not considered a proxy. Only data within the connection headers is used or updated by Azure Load Balancer, it does not and can not use or manipulate the data within the packets.

Using information from the connection headers, Azure Load Balancer can determine to which backend server it should route the request. Load balancing of a connection is performed by a hash algorithm that uses the connection headers to place a connection across the backend pool. There are five pieces of information used from the connection header to generate the hash:

- Source IP
- Source Port
- Destination IP
- Destination Port
- IP Protocol

Azure Load Balancers call the connection sessions flows, because a flow may consist of multiple connections. As source port usually changes between connections, Azure Load Balancer creates an affinity, or rudimentary session persistence between client and server by hashing only of a portion of the connection header information that is used for initial distribution. By doing so connections from a given source IP, destination IP, and destination port, will be pinned to the same backend server.

This operating model is different from NGINX because NGINX operates at layer 7 of the OSI model, the application layer. With NGINX there are two connections, one between the client and

NGINX and another between NGINX and the server. By acting as an intermediary in the connection, makes NGINX a proxy.

Operating at layer 7, NGINX has the ability to read and manipulate the data packet from the client destine for the server, and the response from the server destine to the client. In this way, NGINX can understand higher level application protocols such as HTTP and use that information for routing, where layer 4 load balancers just enable the transport of a connection.

Use Cases

There are valid use cases for both. With layer 4 load balancing, the connection being directly between the client and server has benefits. The server is receives the direct connection and has all of the original connection information without having to understand proxy protocol. This is especially important for legacy applications that depend on a direct connection with the client. A proxy scenario also has it's own benefits, as it's able to control the backend connections, and therefore optimize optimize or manipulate those in any way it may need. If a application relies on client information in the connection headers, it would simply just need to understand the proxy protocol. The proxy protocol appends information about proxies the request has passed through on it's way to the server. The proxy protocol is an addendum to layer 7 protocols, which means the information goes in the application layer protocol headers, and not in the connection headers.

The mentioned differences, make the solutions quite different. Their main relation is that they're able to load balance traffic, and can route based on connection information. Both NGINX and Azure Load Balancer are able listen for incoming traffic on one port and direct the request to the backend service that may be on a different port, in the layer 4 scenario this is considered Network Access Translation, NAT, in a proxy scenario this doesn't have a name, it's just part of the nature of creating the second connection. Both Azure Load Balancer and NGINX can perform TCP and UDP load balancing.

While the solutions are different, and serve their own use case, it's not uncommon to see them working together. The Azure Load Balancer compliments NGINX well when multiple NGINX machines are deployed and the traffic bound for them needs to be balanced with something more sophisticated than DNS. Don't be surprise to see an Azure Load Balancer in front of a fleet of NGINX nodes.

Comparing NGINX and Azure Application Gateway Functionality

Where Azure Load Balancers and NGINX differ, Azure Application Gateway and NGINX have more commonalities. The Azure Application Gateway operates at layer 7, like NGINX. To operate at this layer, Azure Application Gateway must and does act as proxy. One major difference being that

NGINX is able to do this for all TCP and UDP protocols, where Application Gateway is concentrated only on HTTP(S).

By receiving the request, and understanding the HTTP protocol, Application Gateway is able to use HTTP header information to make decisions about how to route or respond to requests. The idea of an Application Gateway, or API gateway, is to consolidate multiple micro services that up a service or product offering under a single API endpoint. This API endpoint understands the services that it’s providing for, and it’s own over all API spec.

CONCEPT VS PRODUCT TERMINOLOGY

The Azure Application Gateway is an API gateway, as is NGINX. Throughout this book, when referring to the concept, I’ll use API gateway, and when referring to the Azure service, I’ll use Azure Application Gateway or Application Gateway to limit confusion.

By having a full understanding the API spec, an API gateway can validate requests on their way to your application. If a request is invalid it can be denied at the API gateway. Basic matching of requests for redirection is also possible. The power and necessity of an API gateway is in it’s ability to route traffic to different backend services based on the URI path. Microservices for RESTful APIs are typically broken up by the sets of API resources they handle, and that’s reflected by the APIs path. In this way, we can use URI path matching to direct requests for specific API resources, based on information in the URI, to the correct microservice.

An example of URI based routing would be if we had two services, authentication, and users. Our API gateway handles request for both but routes based on the URI. Resource requests for authentication are behind a URI path prefix of `/auth/` and the users service behind URI path prefixes of `/users/`. The following diagram depicts this scenario.

Figure 4-1. URI Based Routing with Azure Application Gateway

Once the API gateway has validated the request and has matched a URI path routing rule, it can manipulate the request it makes to the backend service, by altering headers, or URI paths. It can perform these actions because it’s a proxy, and is making it’s own connection and request to the backend service, on behalf of the request it received from the client. This is important to note because there are cases where you may have headers that are only used internally, or paths on the backend services may not match exactly what your frontend API provides. By virtue of being a proxy, it’s also able to terminate SSL/TLS, meaning not use encrypted connection between the API gateway and the backend service, or use a different certificate for that connection.

Once the request is ready to be sent to a backend service, the feature sets of what could be considered an API gateway vs what more advanced data plane services provide starts to differ. Both Azure Application Gateway, and NGINX are able to provide load balancing, where some API gateways would simply pass the request to a load balancer. Having load balancing built into an API gateway solution is nice because it saves a hop in the connection and provides a complete control in between client server communication and routing in a single system.

CONNECTION DRAINING

A useful feature that both Azure Application Gateway and NGINX Plus provides is connection draining, which allows live connection to finish before removing a backend node from the load balancing pool. This feature is not available in the open source version of NGINX.

When load balancing requests, there is some times a need for session persistence. In Azure it’s referred to as Session affinity. When a backend service does not share session state between horizontally scaled nodes, subsequent requests need to be routed to the same backend node. The most common case of the need for session persistence is when legacy applications were ported to cloud environment, and session state has not yet moved off local disk or memory to storage that is network addressable, like Redis or Memcached. In regards to an API gateway this is less common as API gateways were built around more modern day web architecture. A scenario where an API gateway may require session persistence might be when the session data being worked with is too large to be performant over the network.

An important feature to point out that both NGINX and Azure API Gateway provide, is the support of Websockets and HTTP/2 traffic. HTTP/2 enables the client server connection to pass multiple requests through a single connection, cutting down on handshakes and SSL/TLS negotiation. The server in the case of HTTP/2 is the API gateway. A Websocket enables bi-directional communication between the client and server over a long standing connection. The server in this case being the backend application server.

A feature NGINX does provide that Azure Application Gateway does not is HTTP/2 push, were the server can push extra documents that it knows are going to be subsequent requests. A common example would be in response to a request for `index.html` and the server knows that the browser will also need some CSS and JavaScript documents. The server can push those documents with the response for the `index.html` to save on round trip requests.

Azure Application Gateway and NGINX are a lot alike, however, Azure Application Gateway is missing one important nice to have of an API gateway, which is the ability to validate authentication. The web has began to standardize on JSON Web Tokens, JWTs, that use asymmetric encryption to validate identity claims. Standard authentication workflows such as OAUTH/2 and OpenId Connect utilize JWTs, which enables services that can validate JWTs to take part in the authentication validation process. NGINX Plus is able to validate JWTs out of the box, where with the open-source NGINX it requires a bit of work through extendable programming. Both NGINX and NGINX Plus can also perform authentication sub requests where the request, or portion of the request can be sent to an authentication service for validation before NGINX proxies the request to a backend service. Azure Application Gateway does not offer any authentication validation, which means your services will need to validate the request once received, where this action could, and should be offloaded to the API gateway where possible.

Comparing NGINX and Azure Web Application Firewall Capabilities

A Web Application Firewall, WAF, is a layer 7 proxy that specifically reviews the request for security exploits and determined if the request should be proxied to the backend service or denied. The request is evaluated against a number of rules, that look for things like cross site scripting, SQL injection, known bad bots, protocol violations, application language and framework specific vulnerabilities, and size limit abuse. Azure services and NGINX are able to act as Web Application Firewalls.

Azure provides Web Application Firewall capabilities in the form of policies that can be attached to the Azure Front Door, and Application Gateway services. Azure WAF policies, are made up of a number of rules. The rules take the form of managed rule sets provided by Azure, and custom rules defined by you. Managed rule sets are supplied by Azure, and at least one must be configured. Individual rules within a managed rule set can be disabled if necessary. The managed rule sets provide protection out of the gate, but you can also build and apply your own custom rules on top of the managed rule set. You can set specific custom WAF rules or the entire policies to block, or passively monitor and record events.

When using custom rules, you can match on a number of different conditions gleaned from the request. A rule is made up of a number of components such as, the type of match, where to find the variable, an operator, and our matching pattern. The following describes the type of rules that can be setup and they’re different options:

- IP Address: The source IP Address of the request is matched inclusively or exclusively against a CIDR range, or specific IP address.
- Number: A variable numeric value derived from the query string, request URI, headers, arguments, body, or

cookies, is or is not less than, greater to, or equal to a specific value.

- String: A variable string value that is derived from the query string, request URI, headers, arguments, body, or cookies. The value is evaluated by an operator such as, contains, equal, begins with, ends with, or matches a regular expression.
- Geo Location: The variable derived from the source IP or request header is compared against an array of country or region codes. The rule allows the provided country or region code list to be inclusive or exclusive.

Azure WAF policies logs and produces a metric for blocked requests. The log has metadata about the request, and the rule that blocked it. The metric can be filtered by rule name and action type. Metrics can be found in Azure Monitor. Logs are able to be streamed to an Azure Storage Account, Event Hub, or Log Analytics. This monitoring information allows you to analyse how your WAF rules are performing, and if they're flagging false positives. With any WAF you should monitor live traffic with your rule set in a mode that passively monitors for rule violations, review the information and determine if the WAF is working appropriately before enabling it to actively block traffic.

The Azure WAF policies are a great addition to the Azure Managed Service offerings. WAF policies should be enabled at any layer of your Azure environment that they can be applied.

Being that these are fully managed and come with default rule sets, there’s no reason not to take advantage of them.

ModSecurity

The aforementioned functionality provides the basis of what would be considered a WAF, evaluating requests to block based on matching rules. These rules be configured to be extremely versatile and specific to block all sorts of attacks. This type of functionality can be found in the Open Source Web Application Firewall ModSecurity, which integrates directly into NGINX. ModSecurity is a rule engine specifically for matching web requests attributes. It could be speculated that ModSecurity is what Azure WAF rules are using under the hood, and only exposing a controlled portion of the functionality for custom rules, but we can’t be sure.

Installing ModSecurity for NGINX provides the same type of plugin option as the Azure WAF policies do to the Application Gateway. With ModSecurity, you can find a number of community maintained rule sets ready for use, plug them in and get going. ModSecuritys configuration capabilities go extremely deep to which entire books are written on the topic. One of the most popular community maintained rule sets is the OWASP ModSecurity Core Rule Set (CSR), which is provided by the OWASP project. The OWASP CSR is one of the two managed rule sets provided by Azure WAF policies, the other is a list specifically about bots. The OWASP CSR is versioned, the latest public rule set version is 3.2 at time of writing, the latest offered by Azure is 3.1.

Another extremely popular rule set is from Trustwave SpiderLabs, it requires a commercial subscription, but is updated daily so you're ModSecurity rules are always up to date with the latest discovered vulnerabilities and web attack patterns. The increased rate of updates on current web attacks is worth a premium over waiting for Azure to update their managed rule sets.

If you look at this comparison as a this or that, you're weighing a fully managed solution with a do it your self open source solution. There are clear pros and cons here, fully managed with simplified configuration is a clear pro for Azure WAF policies. Bleeding edge updates to security patterns and advanced configuration is a clear win for NGINX with ModSecurity. The cons are the exact reverse of the pros, NGINX must be managed by you, and is more complicated to configure, where Azure is not bleeding edge on security updates, but easy to configure and doesn't require management on your part. This however does not have to be the comparison. You can use a mix of the two, you could use apply Azure WAF Polices to Azure Front Door, and use NGINX as a WAF at the API gateway layer. Evaluating whats best for your situation depends on circumstantial conditions in your organization.

NGINX Plus App Protect

A commercial WAF option for NGINX Plus called NGINX Plus App Protect exists. After NGINX was acquired by F5, they integrated the F5 WAF with NGINX Plus to create the NGINX App Protect module. The App Protect module is more advanced than ModSecurity, it receives updated signatures from F5 to keep the rules up to date with the latest security policies.

To use NGINX App Protect, you need a subscription to NGINX Plus, and a subscription to NGINX App Protect. The installation is done through a private NGINX Plus repository for the package manager being used by your system. After the module is installed, it can be dynamically loaded into NGINX Plus, enabled, and provided a policy file. A passive mode can be enabled by turning on the modules logging directive and providing a log location. The log location consists of a JSON configuration file, and a destination. The destination may be the local or remote syslog receiver, a file, or `/dev/stderr`. The JSON configuration file enables filtering of which events are logged. An example follows:

```
{
    "filter":{
        "request_type":"all"
    },
    "content":{
        "format":"default",
        "max_request_size":"any",
        "max_message_size":"5k"
    }
}
```

As mentioned before, monitoring a rule set before enabling it is suggested, to understand the pattern of what will be blocked or allowed.

Once logging is setup, the App Protect module is open to a vast amount of configuration through the policy file. NGINX and F5 have provided a number of different templates to enable you to protect your apps with high level definitions rather than building your own rules, however that is an option. Each policy provides the ability to put set an `enforcementMode` attribute to `transparent`, or `blocking`. This is an advantage over turning the entire WAF on or off because you can test certain policies while keeping your known good policies enforcing.

The attribute names of a policy file speak for themselves. The following is an example of a policy:

```
{
    "policy": {
        "name": "blocking_policy",
        "template": { "name": "POLICY_TEMPLATE_NGINX_BASE" },
        "applicationLanguage": "utf-8",
        "enforcementMode": "blocking",
        "blocking-settings": {
            "violations": [
                {
                    "name": "VIOL_JSON_FORMAT",
                    "alarm": true,
                    "block": true
                },
                {
                    "name": "VIOL_PARAMETER_VALUE_META CHAR",
                    "alarm": true,
                    "block": false
                }
            ]
        }
    }
}
```

At the core, App Protect is still using the same information from the request to look for malicious requests based on a number of filters, but the funding behind it has enabled it to advance past what's going on in the open source WAF options. One of the most valuable features, in my opinion, of the App Protect module is the ability to filter responses. Which enables us to filter data going outbound to prevent sensitive data from leaving the system. Credit card information is an example of data that should never be returned to the user, with the ability to filter responses we can ensure it doesn't. When dealing with sensitive information, risk reduction of data leaks are of the most importance.

App Protect is, in a way a managed service because of the updated signatures, and the amount of high level features. Prebuilt parsers for application data transfer standards like JSON and XML, SQL and Mogno syntaxes, Linux and Windows Commands enable higher level controls. Signature updates take a whole load of security management off of an organization. It takes a certain type of skill and effort to be build complex filter rules to block only bad requests, while staying up to date with the landscape of active new threats.

NGINX Plus with the App Protect module flips the management vs configurability scenario. The rules are tightly managed by the subscription and configuration options are more in depth, but you have to manage the hosting. Hosting and ensuring availability is par for the course in cloud environments, and therefore if you build and configure your NGINX Plus layer as you do your application code, it's no more than another app on the stack. This makes a solid case for distributing your data plane technologies, by layering *fully managed* with *highly configurable* and *up to date*, you build towards the highest levels of security and availability.

Conclusion

Microsoft Azure provides a number of different data plane managed services to aid in stronger, more reliable delivery of your application. Throughout this chapter you learned about how the Azure Load Balancer, Application Gateway, and WAF Policies work, and how they differ and can be used with NGINX or NGINX Plus. The knowledge of where these different services fit in the stack, enables you to make informed architecture decisions for load balancing and application delivery for your cloud platform.

In the next chapter we will add another layer to the managed Azure data plane service looking at the Azure Front Door service.

Chapter 5. Azure Front Door with NGINX

What is the Azure Front Door Service

Azure Front Door is a service that provides global web traffic routing, optimized for performance and high availability failover. Front Door is useful in all types of scenarios, especially multi-regional deployments. As the name implies, Front Door is meant to be the first and primary access point between your client and your web service. You use Front Door as a way to always answer your client with a local connection, then work as an operator to connect the client to the closest available next hop for that request. By acting as a configurable distributed proxy, Front Door enables you to have layer 7 routing and network control at the edge.

Front Door uses latency based DNS routing to connect the client to the nearest Front Door POP, Point of Presence. Latency based routing is used for the backend connection as well, routing client requests to the nearest installation to a given service, where ever in the world it may be. Being health aware, Front Door automatically enables multi-region failover. Backend Pools can be constructed with a number of different Azure services, and internet standard endpoints. What makes Azure Front Door special, is that it’s network optimized by use of a split TCP-based anycast protocol.

Split TCP

Split TCP is a technique aimed at reducing latency packet delivery issues, by separating the TCP connections between client and backend. This makes an impact on distant and dynamic mobile connections, optimizing at the TCP and TLS handshake layers, by use of proximity to reduce network latency. The Front Door Service takes on the role of a proxy, the client’s TCP connection to the Front Door is quick because of the network proximity of the Front Door POP. On the backend, Front Door can pre-establish, and reuse TCP connections. By reusing connections to secure backends, Front Door also reduces the amount of TCP and TLS negotiations that need to happen. The anycast aspect, happens at the DNS level to determine the closest POP, as well as the backend, to determine the closest endpoint. This use of anycast connects global users to a healthy service with the least number of network hops, therefore lowering network latency.

Front Door Features

Azure Front Door Service includes a number of other useful features you would want at the edge layer; caching and Web Application Firewall (WAF), for example. If a managed edge service can validate the request and respond to your client, the user experience is enhanced by the response time, and your business logic receives only valid, necessary requests. Better response times and less stress on your resources is a clear win win.

Other Azure Front Door features are what you would expect of a proxy, such as TLS termination, session persistence, redirects and rewrites. In my experience in the industry, the term TLS termination is some times used to describe what I would consider necessary to layer 7 routing. An application aware router, must decrypt the packet to operate on layer 7. I and others use the term TLS termination in regards to when a proxy receives a secure connection then proxies the request over an insecure connection, effectively terminating the encryption. Both are acceptable, just make sure you know how the term is being used. Front Door must decrypt the request in order for it to route appropriately. Front Door may route the request to a secure or insecure upstream endpoint. In the event you have a number of backend pools that consist of varying TLS certificates, and wish to consolidate, or extrapolate their frontend presence, Azure Front Door is a great choice.

In the event that backend services rely on session persistence and are incapable of sharing session data, Front Door can create a cookie for the client to bind a client to a backend endpoint. Based on the client cookie, Front Door will continually route specific client requests to the same backend endpoint.

When using a redirect, Azure Front Door simply provides a 3xx level HTTP response to the client for a given rule match. Rewrites however, enable you to alter the request URI. If you're familiar with NGINX rewrites, Azure Front Door's rewrites are much simpler. In NGINX, the notion of a rewrite triggers reprocessing of the request through the proxy rules. A rewrite in Azure Front Door enables you to modify the requested URI path, before making an upstream request to the Backend Pool endpoint. Think of a rewrite with Front Door, as the functionality NGINX provides by allowing manipulation of the request URI before using the `proxy_pass` directive.

Front Door's Place in the Stack

In the end Azure Front Door can listen for multiple frontend configuration endpoints, and directly respond with blocks, redirects, and cached content directly. Requests not served directly have routing rules applied to direct the request to the intended healthy backend service. Network optimization for latency happens by proximity and connection management. Request routing with Front Door is at the macro level, connecting frontend listeners with backend service providers whether by hostname or URI path. Using Azure Front Door provides enables the usefulness of a proxy in close proximity to the client, which you can configure but do not have to actively manage.

Benefits of Azure Front Door Service with NGINX

Azure Front Door and NGINX have a lot of overlapping functionality, however, they compliment each other by being at different layers of the stack. The Front Door being the first receiver of the request, doing high level routing and optimization closest to the consumer, before proxying requests on to a more centralized location for more specific routing and load balancing.

Front Door's physical distribution adds a lot of benefits itself, but when used with NGINX it can be optimized even further. By knowing the way Front Door operates, NGINX can be optimized to to keep client connections and TLS sessions open longer. Usually client connection and TLS sessions would be kept fairly low when serving directly to the end consumer to limit memory and file handle limitations of a server. In a scenario where you do control the client, Front Door, we can lengthen the amount of time client connections are open to optimize connections safely. This optimization can also be made between NGINX and the upstream application server as we have complete control over that client server communication as well.

Some of the overlapping functionality, you may only want to employ at a one layer. Any processing and responses you can offload to a managed edge service you should. Limiting the amount of processing for a given request will result in optimizing usage costs. When using Azure Front Door in conjunction with NGINX, focus caching, macro level routing, and regional failover at the edge, the Front Door. Use NGINX as the reverse proxy load balancer at the micro level, routing between internal micro services that make up a service offering. In regards to blocking, security is done in layers, enabling the Front Door WAF, along with NGINX working as a WAF, provides a second layer of protection, and is an example of where overlapping functionality would be desired at both layers.

NGINX fits as the last layer between the public internet, or other Azure services, and your internal applications on the private network. In that regard, NGINX is much more intimate with the applications that make up the service offering and is prime for private communication between machines on the internal network. Where Azure provides the buildings Front Door, NGINX acts as the office suite door. Once the request has entered the office, it’s directed to a specific conference room or personal office for fulfillment. If a posted sign or lock on the outside of a building does not meet the need, and office door or front desk attendant may, and if not, only then can internal resources be utilized.

This sort of layering of application delivery controllers, or layer 7 proxies, provides security, as well as thorough customization for web request routing.

Integrating Azure Front Door Service with NGINX

To integrate Azure Front Door with NGINX, you need to configure NGINX as a backend pool to Front Door.

If you have not already you'll need to build a Front Door. Through the Azure Console, or with some other infrastructure management tool, construct a Front door configured for your Subscription ID, Resource Group, and Region. Next, configure a Frontend Host for Front Door by defining the host name and enabling session affinity or WAF if necessary. To utilize the WAF capabilities, you will need to have pre-existing Azure WAF policies.

Front Door Features

Once you have a Frontend Host configured, you can setup a backend pool. You can add multiple backend pools to a Front Door, so if you've already have a Front Door setup, you can add another for your NGINX backend.

Backends are logical, if a service is distributed regionally, each endpoint should be grouped and added to the same backend pool. In this manner Front Door will enable failover and automatically route to the closest endpoint.

Add NGINX endpoints as backend hosts to a backend pool. The host header used when proxying the request to NGINX is configurable, which may be useful when the NGINX node receives requests for multiple domains then routes appropriately, even if DNS for the hostname is not actually pointed at the NGINX endpoint but rather somewhere else like Front Door. Configure any health probes that determine if the nodes in the backed pool are healthy or not. Azure Front Door will only direct request to backend pools that are passing their health probes.

Routing Rules

Next apply routing rules to map the Frontend Host to the Backend Pool. In this section of the Front Door configuration you can match based on request URI, and choose to redirect or forward the request. To route to NGINX you would forward, at which point you can choose to rewrite the request to manipulate it's URI.

Once this section is configured and the Front Door is deployed, it's passing requests to NGINX.

Optimizing with NGINX

To optimize NGINX for use with Azure Front Door Service, alter the length of time NGINX allows for client connections and TLS sessions. You can do so by use of the `keepalive_timeout` directive, and the `ssl_session_timeout` directives. Azure Front Door keeps connections open for (pending question of Azure) seconds, the following code is set to the same.

```
server {
    # Your server configuration
    keepalive_timeout 300s;
    ssl_session_timeout 300s;
}
```

Conclusion

In this chapter you learned about the Azure Front Door service, and the optimizations it can provide for your application. Now that you have a full view of the services Azure provides for the data plane layer, you can start to see how the pieces fit together for your scenario. This layer of Azure managed service, provides you a way to distribute the endpoint connection for the client to the closest location to lower latency. Together with NGINX, Azure Front Door can handle initial requests, run it through initial managed WAF, then proxy the request over an optimized connection to an NGINX API gateway for more control over delivery of your application.