**Load Balancer vs Reverse Proxy vs API Gateway**

In system design discussions, the terms **Load Balancer**, **API Gateway**, and **Reverse Proxy** often come up.

Although they are sometimes used interchangeably and their functionalities sometimes overlap, each serves a distinct purpose in **backend architecture**.

Understanding the differences between these components, their strengths, and when to use them can help you design more scalable, efficient, and maintainable systems.

In this post, we’ll cover:

* What each component is and how it works
* Core capabilities that set them apart
* When to choose a load balancer, reverse proxy, or API gateway
* How they can work together

1. **Load Balancer**

[[A diagram of a computer

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!SvwJ!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F555c4580-aba8-48e3-9bbd-f14e3372e898_1304x1044.png)](https://substackcdn.com/image/fetch/$s_!SvwJ!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F555c4580-aba8-48e3-9bbd-f14e3372e898_1304x1044.png" \t "_blank)

A **load balancer** distributes incoming network traffic across multiple backend servers (nodes) to ensure no single server is overwhelmed.

By balancing the load, it helps improve the overall performance, availability, fault tolerance, and scalability of applications.

**Examples:**AWS Elastic Load Balancer (ELB), Google Cloud Load Balancing, HAProxy (TCP/HTTP mode), Azure Load Balancer.

**Key Features of a Load Balancer**

**1. Traffic Distribution**

The core function of a load balancer is to evenly distribute traffic across multiple backend servers. The decision to determine which backend server should handle the request is based on predefined algorithms like:

* Round Robin: Requests are sent to servers in a fixed, rotating order
* Least Connections: Requests are directed to the server with the fewest active connections
* Weighted Distribution: Assigns more traffic to more powerful servers based on configured weights

**2. Health Checks**

A good load balancer continuously monitors the health and responsiveness of backend servers. If a server is found to be unresponsive, slow, or returning errors, the load balancer temporarily removes it from the rotation.

**3. Session Persistence**

In some scenarios, it is important for all requests from a particular user to go to the same server during a session. This is known as session persistence or sticky sessions.

Load balancers can use:

* IP-based affinity
* Cookie-based tracking

to consistently route user requests to the same backend server.

**4. SSL Termination**

Handling SSL/TLS encryption and decryption can be computationally expensive. A load balancer can offload this task by decrypting incoming HTTPS traffic and forwarding unencrypted requests to the backend servers.

**5. High Availability and Failover**

Load balancers provide high availability by ensuring that requests are only sent to operational servers. If a server crashes or is taken down for maintenance, the load balancer reroutes traffic to the remaining healthy servers without affecting users.

Many load balancers themselves are deployed in a redundant setup, often with multiple instances across availability zones or regions, to eliminate single points of failure.

**Types of Load Balancers**

Load balancers can be classified based on the network layer at which they operate in the OSI model. The two most common types are:

**1. Layer 4 Load Balancer (Transport Layer)**

A Layer 4 load balancer makes routing decisions based on data from the transport layer of the OSI model. It uses information such as:

* Source and destination IP addresses
* TCP/UDP ports

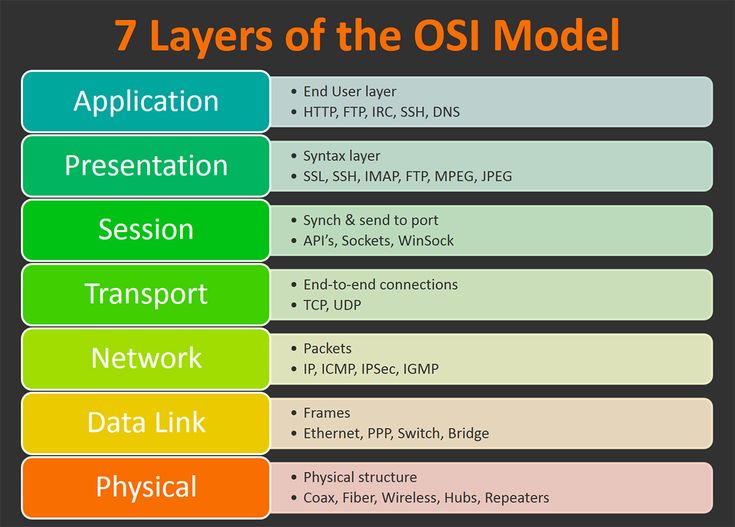
These load balancers are protocol-agnostic and forward packets without inspecting the content. They are faster and more efficient but less flexible when it comes to intelligent routing.

**2. Layer 7 Load Balancer (Application Layer)**

A Layer 7 load balancer operates at the application layer and makes decisions based on the content of the request. It can inspect:

* HTTP methods
* URLs and query strings
* Headers and cookies
* Application-specific logic

This allows for advanced routing, such as sending requests for /api to one server group and /images to another.



**2. Reverse Proxy**

[[A diagram of a network

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!bO8H!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F91a2f014-ad4e-40d6-a576-5a01af14a83a_1874x916.png)](https://substackcdn.com/image/fetch/$s_!bO8H!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F91a2f014-ad4e-40d6-a576-5a01af14a83a_1874x916.png" \t "_blank)

A **reverse proxy** is a server that sits between clients and one or more backend services When a client sends a request, the reverse proxy intercepts it. Based on predefined rules, it decides which internal service should handle the request. The proxy then forwards the request to the backend service and returns the response to the client.

To the client, it appears as if all content comes from a single server.

**Examples:,**NGINX, Apache HTTP Server, HAProxy (Layer 7), Traefik

**Key Features of a Reverse Proxy**

**1. Security and Abstraction:** One of the most important roles of a reverse proxy is to shield backend servers from direct access.It hides IP addresses, port configurations, and other identifying information, making backend services less vulnerable to attacks such as:

* DDoS (Distributed Denial of Service) attacks
* Port scanning
* Application fingerprinting

**2. Centralized SSL/TLS Termination:** A reverse proxy can handle SSL/TLS encryption and decryption, allowing all traffic between the client and the proxy to be secure, while traffic from the proxy to backend servers can remain unencrypted (if acceptable within a trusted internal network).

**3. Caching of Static and Dynamic Content:** Reverse proxies can cache frequently accessed content such as images, JavaScript files, CSS stylesheets, and even HTML pages. By serving cached responses directly, the proxy reduces:

* Load on backend servers
* Response times for clients
* Network bandwidth usage

This is particularly effective for high-traffic websites with large amounts of static or infrequently changing content.

**4. Compression:** To improve performance and reduce bandwidth usage, reverse proxies often compress server responses before sending them to the client. Common algorithms like Gzip or Brotli shrink the size of the response, speeding up page loads, especially for users on slow connections.

**5. Load Balancing Capabilities:** Although load balancing is typically considered a separate function, many reverse proxies support built-in load balancing across multiple backend servers.This overlap allows a reverse proxy to distribute traffic efficiently in addition to routing and caching.

**6. URL Rewriting and Routing:** Reverse proxies can rewrite incoming URLs before they are forwarded to backend services. This enables:

* Clean, user-friendly URLs
* Internal path mapping
* Seamless routing to microservices

For example, a request to /products could be internally routed to <http://product-service.internal/api/v1/items>.

**Using Both Reverse Proxy and Load Balancer**

In real-world systems, reverse proxies and load balancers are often used **together**, and sometimes a single tool like **NGINX** can perform both functions.

[[A diagram of a server

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!vpbQ!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F342c837c-25c4-4aa9-908d-a4ca0be87cc4_1728x1570.png)](https://substackcdn.com/image/fetch/$s_!vpbQ!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F342c837c-25c4-4aa9-908d-a4ca0be87cc4_1728x1570.png" \t "_blank)

A typical pattern involves placing a **reverse proxy (e.g., NGINX)** in front of several **load balancers**, each of which manages a dedicated group of backend servers:

* A client request (e.g., from a browser or mobile app) hits the **reverse proxy**.
* Based on the request path (e.g., /orders), the proxy routes the traffic to the appropriate **load balancer**.
* The load balancer then forwards the request to one of the healthy servers in its **assigned server group**.
* The selected server processes the request and sends the response back through the chain to the client.

**3. API Gateway**

[[A diagram of a computer

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!IhU3!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F6d4f4233-2703-4997-a207-64e9197937e4_2218x1152.png)](https://substackcdn.com/image/fetch/$s_!IhU3!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F6d4f4233-2703-4997-a207-64e9197937e4_2218x1152.png" \t "_blank)

An **API Gateway** is a server that functions as a **central entry point** for all client interactions with backend services.

It is especially valuable in **microservices architectures**, where multiple services exist and client requests need to be managed, secured, routed, and orchestrated efficiently. **Examples:**Amazon API Gateway, Apigee (Google Cloud), Kong Gateway, Zuul (Netflix)

**Key Features of an API Gateway**

**1. Single Point of Entry:** The gateway provides a unified interface for all backend APIs. Instead of exposing every service endpoint to the client, you only expose the gateway’s endpoint. This simplifies client logic, reduces surface area for attacks, and allows you to evolve internal services without impacting external users.

2. Request Routing: The API Gateway can route incoming requests to the correct backend service based on:

* URL path (e.g., /api/users → user service)
* HTTP method (e.g., POST vs GET)
* Request headers or query parameters
* Service discovery metadata

**3. Authentication and Authorization:** An API gateway can centralize security across the system. It can:

* Integrate with identity providers (e.g., OAuth 2.0, OpenID Connect)
* Validate tokens (e.g., JWT)
* Enforce role-based or scope-based access controls

This ensures consistent enforcement of security policies across all backend services, reducing the risk of misconfigured individual services.

**4. Rate Limiting and Throttling:** An API gateway can protect backend services from overuse or abuse by restricting the number of requests a client can make per time unit. This prevents individual services from being overwhelmed and ensures fair usage.

**5. Request and Response Transformation:** An API Gateway can transform the format or structure of data as it passes through.For example:

* Add or remove headers
* Convert between formats (e.g., XML ↔ JSON)
* Filter or reshape response payloads
* Modify query parameters or request bodies

This is especially useful when backend services and clients follow different protocols or formats.

**6. API Composition and Aggregation:** The gateway can aggregate responses from multiple microservices into a single response, reducing the number of round-trips the client needs to make.For example, a /user-dashboard endpoint can fetch user profile, orders, and notifications from three different services and return them as one payload.

**7. Caching:** To reduce backend load and improve latency, gateways can cache:

* Frequently requested data (e.g., public product listings)
* Authentication token validations
* Static or slow-changing API responses

This reduces redundant calls and significantly improves scalability.

**8. Logging, Monitoring, and Analytics:** Gateways provide centralized logging and metrics for all API traffic, including:

* Request counts and response times
* Error rates and status codes
* User behavior and usage patterns
* Latency bottlenecks

These insights help with observability, debugging, and performance optimization.

**9. Protocol Translation:** An API Gateway can translate between different protocols, such as:

* HTTP ↔ gRPC
* WebSocket ↔ REST
* SOAP ↔ REST

This allows clients to use simpler protocols while internal services operate using more efficient ones.

**4. When to Choose a Load Balancer, Reverse Proxy, or API Gateway**

Below is a breakdown of when to use each based on the goals and requirements of your application:

**When to Use a Load Balancer**

Use a load balancer when your primary goal is to ensure high availability, scalability, and fault tolerance by distributing incoming traffic across multiple servers or services.

**Ideal scenarios:**

* You are hosting multiple identical instances of a web application or microservice and need to distribute traffic among them.
* You want to scale your application horizontally to handle increased traffic.
* You need automatic failover so that requests are only routed to healthy backend nodes.
* You want to implement traffic distribution strategies (e.g., round-robin, least connections, IP-hash).
* You are serving stateless applications, where any instance can handle any request.
* You are running containerized services behind an orchestrator (e.g., Kubernetes, ECS) and need internal or external traffic distribution.

**When to Use a Reverse Proxy**

Use a reverse proxy when you need a flexible traffic router, performance optimizer, or security layer in front of your backend services or web servers.

**Ideal scenarios:**

* You want to hide your backend infrastructure from external clients.
* You need SSL/TLS termination to reduce the burden on application servers.
* You want to cache static or dynamic content to improve performance.
* You need to compress server responses before sending them to the client.
* You want to rewrite URLs, add custom headers, or apply routing logic based on the request path.
* You are hosting multiple services or applications behind the same domain or IP address.
* You want to provide basic load balancing along with caching and routing features.

**When to Use an API Gateway**

Use an API Gateway when your application exposes multiple APIs or microservices and you need centralized control over security, routing, and API lifecycle management.

**Ideal scenarios:**

* You have a microservices architecture and need a single entry point for all APIs.
* You need to enforce authentication and authorization using tokens (e.g., JWT, OAuth2).
* You want to apply rate limiting, request quotas, or throttling to protect backend APIs.
* You want to transform requests and responses (e.g., modify headers, convert formats).
* You want to aggregate responses from multiple microservices into a single API call.
* You are exposing APIs to external developers or partners and need API documentation, key management, or a developer portal.
* You want to version APIs, manage deprecated routes, or route traffic based on API versions.

**5. Can They Work Together?**

Yes, absolutely. In modern, large-scale web architectures, Load Balancer, Reverse Proxy, and API Gateway are often used together as part of a layered infrastructure.

For example:

* Applications deployed on AWS often use Amazon API Gateway to manage and secure API traffic, while using Elastic Load Balancer (ELB) to distribute incoming HTTP or TCP traffic across EC2 instances or containers.
* NGINX, a highly flexible tool, is frequently used both as a reverse proxy (for SSL termination, caching, and request forwarding) and as a Layer 7 load balancer that distributes traffic based on content-aware routing.

**Typical Architecture Using All Three**

Here’s how a modern web application can incorporate all three components in a single architecture:

A diagram of a computer

AI-generated content may be incorrect.

**1. Client Request:** The journey begins when a user’s browser or mobile application sends a request to your system.

**2. Edge Load Balancer (Optional):** A global or regional load balancer (e.g., AWS Route 53 + Global Accelerator, Cloudflare Load Balancer) receives the request. It routes the traffic to the nearest or most responsive data center or region, based on geolocation, latency, or health checks.

**3. Reverse Proxy / Web Tier Load Balancer:** The request then arrives at a reverse proxy or web-tier load balancer. At this layer, the component may perform several critical tasks:

* SSL/TLS termination: Decrypts HTTPS traffic before forwarding it internally
* Caching static content: Serves HTML, CSS, JavaScript, and images without hitting backend services
* Logging and header manipulation: Adds, removes, or rewrites headers for security or analytics
* Load balancing: Distributes traffic to web servers or directly to the API Gateway

**4. API Gateway**

The request is then forwarded to an API Gateway, which handles API-specific responsibilities, including:

* Routing the request to the appropriate microservice or backend API
* Authentication and authorization, using mechanisms like OAuth 2.0 or JWT
* Rate limiting and throttling to prevent abuse or overload
* Request and response transformation, such as reformatting JSON or XML payloads
* API version routing to support multiple client versions
* Response aggregation from multiple services into a single response payload

**5. Internal Load Balancer**

Once the API Gateway determines the appropriate microservice, it may forward the request through an internal load balancer or service mesh.

This layer is responsible for:

* Distributing traffic evenly across multiple instances of the target microservice
* Failover and retry logic in case a particular instance is unhealthy or unresponsive

**6. Backend Services / Microservices**

Finally, the designated backend service or microservice receives the request. It:

* Executes the business logic
* Interacts with databases, caches, or other services as needed
* Returns the response to the API Gateway, which may further process or format it before sending it back to the client