**What is an API Gateway?**

**APIs**, or **Application Programming Interfaces**, are a set of rules and protocols that allows two software applications or services to communicate with each other.

As applications grow in size, the number of APIs increases too. Without the right tools and infrastructure, managing these APIs can quickly become a challenge.

This is where **API Gateway** comes into play.

An API Gateway acts as a **central server** that sits between clients (e.g., browsers, mobile apps) and backend services.

Instead of clients interacting with multiple microservices directly, they send their requests to the API Gateway. The gateway processes these requests, enforces security, and forwards them to the appropriate microservices.

In this article, we will explore why do we need an API gateway, the key features it provides and how it works step by step.

**1. Why Do We Need an API Gateway?**

Modern applications, especially those built using microservices architecture, have multiple backend services managing different functionalities.

For example, in an e-commerce service:

* One service handles **user accounts**.
* Another handles **payments**.
* Another manages **product inventory**.

**Without an API Gateway:**

[[A diagram of a computer

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!Yp-h!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fc6da835a-257c-4247-a4b9-d7297c289fce_1044x718.png)](https://substackcdn.com/image/fetch/$s_!Yp-h!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fc6da835a-257c-4247-a4b9-d7297c289fce_1044x718.png" \t "_blank)

* Clients would need to know the location and details of all backend services.
* Developers would need to manage authentication, rate limiting, and security for each service individually.

**With an API Gateway:**

[[A diagram of a api gateway

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!_CtQ!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F6e41325f-2b6b-4f2a-8ca6-d622413ad64c_1692x720.png)](https://substackcdn.com/image/fetch/$s_!_CtQ!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F6e41325f-2b6b-4f2a-8ca6-d622413ad64c_1692x720.png" \t "_blank)

* Clients send all requests to one place – the API Gateway.
* The API Gateway takes care of routing, authentication, security, and other operational tasks, simplifying both client interactions and backend management.

**2. Core Features of an API Gateway**

[[A diagram of a computer network

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!_ZNj!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fcb0c8192-4b1b-4834-b068-20d8cfb65050_1856x1412.png)](https://substackcdn.com/image/fetch/$s_!_ZNj!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fcb0c8192-4b1b-4834-b068-20d8cfb65050_1856x1412.png" \t "_blank)

**1. Authentication and Authorization**

API Gateway secures the backend systems by ensuring only authorized users and clients can access backend services.

It handles tasks like:

* **Authentication:** Verifying the identity of the client using tokens (e.g., OAuth, JWT), API keys, or certificates.
* **Authorization:** Checking the client’s permissions to access specific services or resources.

By centralizing these tasks, the API gateway eliminates the need for individual services to handle authentication, reducing redundancy and ensuring consistent access control across the system.

**2. Rate Limiting**

To prevent abuse and ensure fair usage of resources, most API Gateways implement **rate limiting**.

This feature:

* Controls the frequency of requests a client can make within a given timeframe.
* Protects backend services from being overwhelmed by excessive traffic or potential denial-of-service (DoS) attacks.

For example, a public API might allow a maximum of 100 requests per minute per user. If a client exceeds this limit, the API Gateway will block additional requests until the rate resets.

**3. Load Balancing**

High-traffic applications rely on **load balancing** to distribute incoming requests evenly across multiple instances of a service.

The API Gateway can:

* Redirect requests to healthy service instances while avoiding ones that are down or overloaded.
* Use algorithms like round-robin, least connections, or weighted distribution to manage traffic intelligently.

**4. Caching**

To improve response times and reduce the strain on backend services, most API Gateways provide **caching**.

They temporarily store frequently requested data, such as:

* Responses to commonly accessed endpoints (e.g., product catalogs or weather data).
* Static resources like images or metadata.

Caching helps in reducing latency and enhancing user experience while lowering the operational cost of backend services.

**5. Request Transformation**

In systems with diverse clients and backend services, **request transformation** is essential for compatibility.

An API Gateway can:

* Modify the structure or format of incoming requests to match the backend service requirements.
* Transform responses before sending them back to the client, ensuring they meet the client’s expectations.

For instance, it might convert XML responses from a legacy service into JSON for modern frontend applications.

**6. Service Discovery**

Modern systems often involve microservices that scale dynamically.

The **service discovery** feature of an API Gateway dynamically identifies the appropriate backend service instance to handle each request.

This ensures seamless request routing even in environments where services frequently scale up or down.

**7. Circuit Breaking**

Circuit breaking is a mechanism that temporarily stops sending requests to a backend service when it detects persistent failures, such as:

* Slow responses or timeouts.
* Server errors (e.g., HTTP 500 status codes).
* High latency or unavailability of a service.

The API Gateway continuously monitors the health and performance of backend services and uses circuit breaking to block requests to a failing service.

**8. Logging and Monitoring**

API Gateways provide robust **monitoring and logging** capabilities to track and analyze system behavior.

These capabilities include:

* Logging detailed information about each request, such as source, destination, and response time.
* Collecting metrics like request rates, error rates, and latency.

This data helps system administrators detect anomalies, troubleshoot issues, and optimize the system’s performance. Many API Gateways also integrate with monitoring tools like Prometheus, Grafana, or AWS CloudWatch.

**3. How Does an API Gateway Work?**  
Imagine you're using a food delivery app to order dinner. When you tap "Place Order" your phone makes an API request. But instead of talking directly to various backend services, it communicates with an API Gateway first.

**Step 1: Request Reception**

When you tap "Place Order," the app sends a request to the **API Gateway**, asking it to process your order.

This request includes things like:

* Your user ID
* Selected restaurant and menu items
* Delivery address
* Payment method
* Authentication tokens

The API Gateway receives the request as the single entry point to the backend system.

**Step 2: Request Validation**

Before forwarding the request, the API Gateway validates it to ensure:

* The required parameters or headers are present.
* The data is in the correct format (e.g., JSON).
* The request conforms to the expected structure or schema.

// Example of initial request handling

app.post('/api/v1/orders', async (req, res) => {

// Check if request has required headers

if (!req.headers['content-type'].includes('application/json')) {

return res.status(400).send('Invalid content type');

}

// Continue processing...

});

If any information is missing or incorrect, the gateway immediately rejects the request and notifies the app with an appropriate error message.

**Step 3: Authentication & Authorization**

The gateway now verifies your identity and permissions to ensures only legitimate users can place orders:

* It forwards your authentication token (e.g., OAuth or JWT) to an identity provider to confirm your identity.
* It checks your permissions to ensure you’re authorized to use the app for placing an order.

const authenticateRequest = async (req) => {

// Extract JWT token from header

const token = req.headers.authorization?.split(' ')[1];

// Verify token and get user details

const user = await verifyToken(token);

// Check if user has permission to place orders

return user.permissions.includes('place\_orders');

};

If authentication or authorization fails, the API Gateway sends a 401 Unauthorized or 403 Forbidden error back to the app.

**Step 4: Rate Limiting**

To prevent abuse, the API Gateway checks how many requests you’ve made recently. For example:

* If you’ve made 10 "Place Order" requests in the last minute (maybe by accident), the gateway might block additional requests temporarily and return 429 Too Many Requests response.

const checkRateLimit = async (userId) => {

const key = `rate\_limit:order:${userId}`;

const current = await redis.incr(key);

// If first request in window, set expiry

if (current === 1) {

await redis.expire(key, 60); // 1 minute window

}

return current <= 10; // Allow 10 order requests per minute

};

This ensures the system remains stable and fair for all users specially during traffic spikes or malicious attacks, such as distributed denial-of-service (DDoS) attempts.

**Step 5: Request Transformation (if needed)**

If any of these backend services require specific data formats or additional details, the API Gateway transforms the request.

For example:

* The app sends the delivery address in plain text, but the Delivery Service expects GPS coordinates. The API Gateway converts the address into coordinates before forwarding the request.

const transformRequest = async (originalRequest) => {

const address = originalRequest.deliveryAddress;

// Convert address to GPS coordinates using a geocoding API

const coordinates = await getCoordinatesFromAddress(address);

if (!coordinates) {

throw new Error('Failed to fetch GPS coordinates');

}

// Transform the request for the Delivery Service

return {

orderId: originalRequest.orderId,

customerName: originalRequest.customerName,

deliveryLocation: {

latitude: coordinates.lat,

longitude: coordinates.lng

},

deliveryInstructions: originalRequest.instructions || ""

};

};

**Step 6: Request Routing**

The API Gateway now needs to coordinate several backend services to process your order.

Using **service discovery**, it identifies:

* **Order Service:** To create a new order record.
* **Inventory Service:** To check if the restaurant has your selected items available.
* **Payment Service:** To process your payment.
* **Delivery Service:** To assign a delivery driver to your order.

The gateway dynamically routes the request to these services using a **load balancing** algorithm, ensuring it connects to available and healthy service instances.

const routeRequest = async (req, serviceType) => {

// Get service registry

const services = await serviceDiscovery.getServices(serviceType);

// Select instance

const targetService = selectServiceInstance(services);

// Forward request

return await axios.post(

`${targetService.url}/api/orders`,

req.body,

{ headers: req.headers }

);

};

**Step 7: Response Handling**

Once the API Gateway receives the response(s) from the backend service(s), it performs the following tasks:

* **Transformation:** Adjusts the response format or structure to match the client’s requirements.
* **Caching (Optional):** Stores the response temporarily for frequently accessed data, reducing future latency.

const handleResponse = async (serviceResponse) => {

// Transform response if needed

const transformedResponse = {

orderId: serviceResponse.order\_reference,

estimatedDelivery: serviceResponse.eta,

status: serviceResponse.current\_status

};

// Cache response if applicable

if (serviceResponse.cacheable) {

await cacheResponse(

transformedResponse.orderId,

transformedResponse

);

}

return transformedResponse;

};

Finally, the API Gateway sends the processed response back to the client in a format they can easily understand.

**Step 8: Logging & Monitoring**

Throughout this process, the gateway records important metrics to track each request:

const logRequest = async (req, res, timing) => {

await logger.log({

timestamp: new Date(),

path: req.path,

method: req.method,

responseTime: timing,

statusCode: res.statusCode,

userId: req.user?.id

});

};

Thank you for reading!

**1. What is an API Gateway?**

* An **API Gateway** is a single entry point for client requests to multiple backend services.
* It handles **routing**, **authentication**, **rate limiting**, **logging**, **load balancing**, and sometimes **response transformation**.

**2. Why do we use an API Gateway instead of calling microservices directly?**

* **Simplifies client logic** — one endpoint instead of many.
* Handles **cross-cutting concerns** (security, throttling, monitoring).
* Can transform requests/responses without changing backend services.
* Improves security by **hiding internal service endpoints**.

**3. Give examples of popular API Gateway solutions.**

* AWS API Gateway
* Kong
* NGINX
* Apigee
* Azure API Management
* Netflix Zuul / Spring Cloud Gateway

**4. What are some common features of an API Gateway?**

* Request routing
* Protocol translation (HTTP ↔ WebSocket, gRPC ↔ REST)
* Authentication & Authorization (OAuth 2.0, JWT)
* Rate limiting & throttling
* Logging & Monitoring
* Request/response transformation
* Caching

**5. How does an API Gateway help in a microservices architecture?**

* Acts as a **reverse proxy** to route requests to the right service.
* Reduces **chattiness** between client and services by aggregating multiple requests.
* Implements **service discovery** and **versioning** without changing the client.

**6. What is request throttling in an API Gateway?**

* Limiting the number of requests a client can send in a given time.
* Prevents **overloading backend services** and protects against DDoS attacks.
* Example: **100 requests per minute per client**.

**7. How does authentication work in an API Gateway?**

* The gateway validates incoming requests using:
  + **JWT tokens**
  + **OAuth 2.0**
  + API keys
* Once verified, it forwards requests to backend services with security context.

**8. What’s the difference between authentication and authorization in an API Gateway?**

* **Authentication**: Verifying identity (e.g., checking token validity).
* **Authorization**: Checking permissions for the requested resource.

**9. How does an API Gateway handle protocol translation?**

* Converts incoming request protocol to the format backend expects.
* Examples:
  + REST → gRPC
  + HTTP → WebSocket
  + JSON → XML

**10. What is response caching in an API Gateway?**

* The gateway stores responses for repeated requests to avoid hitting the backend every time.
* Reduces latency and improves performance.

**11. How can an API Gateway help improve security?**

* Hides internal service endpoints from the public.
* Enforces **TLS/HTTPS** for all traffic.
* Adds **IP whitelisting/blacklisting**.
* Integrates with **WAF (Web Application Firewall)** for filtering.

**12. What’s the difference between an API Gateway and a Load Balancer?**

* **Load Balancer**: Distributes traffic among multiple servers.
* **API Gateway**: Adds routing, transformation, authentication, caching, and more — **can include load balancing** as part of its job.

**13. How do you implement rate limiting in an API Gateway?**

* Use **token bucket** or **leaky bucket** algorithms.
* Configure limits like:
  + X requests per second per client.
  + Burst capacity handling.

**14. What is an API Gateway pattern, and what are its drawbacks?**

* **Pattern**: Single entry point for all client requests in a distributed system.
* **Drawbacks**:
  + Single point of failure if not highly available.
  + Can become a performance bottleneck if not scaled properly.
  + Adds extra network hop → increases latency slightly.

**15. How would you troubleshoot high latency in an API Gateway?**

* Check **gateway logs** for slow downstream calls.
* Analyze **metrics** (CPU, memory, request counts).
* Use **distributed tracing** (Jaeger, Zipkin).
* Test latency between gateway and backend services.
* Check **rate limiting**, caching, and authentication overhead.

**Scenario-Based API Gateway Interview Questions & Answers**

**1. You are designing an API Gateway for a high-traffic e-commerce platform. How would you ensure high availability?**

* Deploy **multiple gateway instances** across different regions/zones.
* Use a **global load balancer** (e.g., AWS Route 53, Azure Traffic Manager).
* Enable **auto-scaling** based on traffic spikes.
* Store config in a **centralized, replicated store** (e.g., Consul, etcd).

**2. How would you secure an API Gateway for a banking application?**

* **Mutual TLS** between clients and gateway.
* JWT with short expiration times.
* IP whitelisting for partner APIs.
* Integrate with a **WAF** to block SQL injection, XSS, etc.
* Strict **rate limits** to prevent brute-force attacks.

**3. Your gateway must support both REST and gRPC clients. How do you handle protocol differences?**

* Use **protocol translation** features:
  + REST → gRPC via transformation layer.
  + Define mapping rules for headers, payload format.
* Maintain **versioned APIs** for each protocol.
* Use a **polyglot gateway** like Envoy or Kong.

**4. You need to aggregate data from 3 microservices before responding. How do you do it efficiently?**

* Use **request aggregation** at the gateway.
* Make **parallel calls** to services.
* Implement **circuit breakers** to avoid blocking on slow services.
* Return partial data with fallback values if one service fails.

**5. You observe that 70% of gateway traffic is fetching the same product catalog. What’s your approach?**

* Enable **response caching** in the gateway with short TTL (e.g., 5 min).
* Use **ETags** or conditional requests.
* Pre-warm cache during off-peak hours.

**6. Your API Gateway adds 100ms latency in every request. How do you debug?**

* Profile gateway execution time (auth, transformation, routing).
* Check **network RTT** between gateway and services.
* Disable features (auth, logging, caching) one by one to isolate the cause.
* Switch to a **lighter gateway implementation** if possible.

**7. How would you handle authentication for both public and private APIs on the same gateway?**

* Use **multiple routes with different security policies**.
* Public APIs → API key or none (read-only).
* Private APIs → OAuth2/JWT with scopes.
* Separate rate limits for each category.

**8. You need zero downtime deployments for the API Gateway. What’s your strategy?**

* Use **blue-green deployments** or **canary releases**.
* Keep old version running until new version passes health checks.
* Gradually shift traffic using load balancer weights.

**9. How do you prevent one slow backend service from affecting the whole gateway?**

* **Timeouts** and **retry policies** for backend calls.
* **Circuit breakers** to stop calls to failing services.
* Return cached or default response if service is down.

**10. Your API Gateway integrates with multiple third-party APIs. How do you ensure resilience?**

* Implement **failover routes** to backup providers.
* Use **asynchronous queuing** for non-critical calls.
* Log and monitor third-party latency separately.

**11. How would you implement versioning in an API Gateway?**

* Path versioning (/v1/orders, /v2/orders).
* Header-based versioning (X-API-Version).
* Allow clients to **opt-in to newer versions** gradually.

**12. Your analytics team needs request/response logs from the gateway without affecting performance.**

* Use **asynchronous logging** to a log aggregator (e.g., Kafka, Fluentd).
* Avoid synchronous disk writes.
* Sample logs (e.g., log 1% of requests) for high-traffic endpoints.

**13. You are migrating from monolith to microservices. How can the API Gateway help?**

* Act as a **facade** — clients keep calling the same API while internally routing to microservices.
* Gradually replace monolith endpoints with microservice endpoints behind the same gateway.

**14. How do you monitor and troubleshoot API Gateway performance?**

* Metrics: Latency, request rate, error rate.
* Distributed tracing (Jaeger, OpenTelemetry).
* Alert on sudden traffic spikes or error surges.

**15. Your API Gateway is becoming a bottleneck under peak load. How do you scale it?**

* **Horizontal scaling** with stateless gateway instances.
* Use **API caching** to offload backends.
* Optimize transformations and remove unnecessary processing.
* Upgrade hardware/network bandwidth.