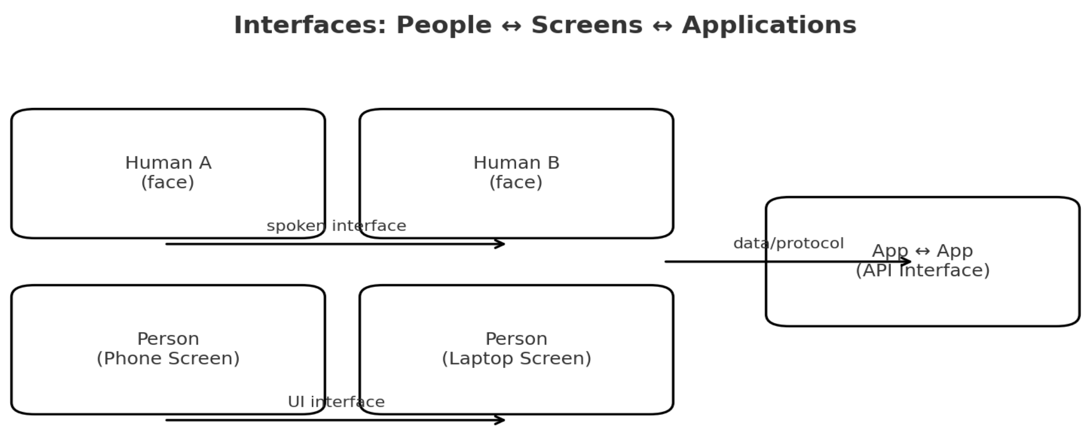
APIs — From Interface to Internet■Scale

Turning the ‘face■to■face’ analogy into production■grade interfaces

Plain■English Analogy → Technical Reality
People can talk face■to■face; our phones and laptops are *interfaces* we use to communicate. Likewise, applications talk to each other through a well■defined **Application Programming Interface (API)** — a contract that specifies *what* you can ask for and *how* to ask. This document expands your short reel into a deep, technical guide.



Interfaces, everywhere — humans, screens, and apps.

1) What is an Interface? What is an API?

An **interface** defines the surface for interaction: the allowed operations, inputs, and outputs. An **API** is an interface for software components, typically across a network. It is both a *contract* (specification) and a *mechanism* (how to call it). Good APIs are predictable, consistent, and versioned.

Common API Styles

- **REST** (HTTP/JSON over URLs): resource■oriented, cache■friendly, human■readable.
- **GraphQL**: one endpoint; clients specify the exact fields; reduces over/under■fetching.
- **gRPC**: Protocol Buffers over HTTP/2; strongly typed, fast, great for microservice RPC.
- **WebSockets**: full■duplex, low■latency bidirectional streams (chats, dashboards).
- **SOAP**: XML■based, WS-* standards; enterprise legacy but still used in some domains.
- **Webhooks / Event APIs**: push■based callbacks; decouple producers/consumers.

2) HTTP Basics: Methods, URIs, Status Codes

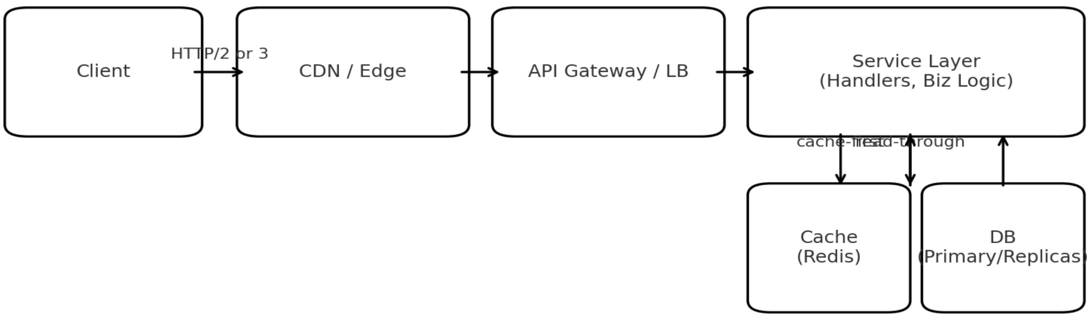
Most public APIs use HTTP. The request line is METHOD /path?query HTTP/1.1 and responses carry a status code.
Safe methods don’t mutate state (GET, HEAD, OPTIONS); **idempotent** methods can be retried without side■effects (GET, PUT, DELETE); POST is neither safe nor idempotent by default.

Method	Semantics	Typical Use

GET	Read (safe, idempotent)	Fetch a resource or list; support pagination, filters
POST	Create / action (not idempotent)	Create resource, submit commands, login
PUT	Replace (idempotent)	Full update of a resource
PATCH	Partial update (idempotent-ish)	Update a subset of fields
DELETE	Remove (idempotent)	Delete a resource
HEAD/OPTIONS	Metadata / CORS preflight	Headers only / capability discovery

Request/Response Anatomy

HTTP Request → Response Path



Request: URL, method, headers (auth, content-type, idempotency■key), and body (usually JSON).
Response: status code (2xx/4xx/5xx), headers (caching, rate■limit), and body (JSON/protobuf).
Auth: API keys, OAuth2, signed tokens (JWT), mTLS for service■to■service.

3) Designing & Writing an API

- **Model resources** and nouns: /dogs, /dogs/{id}, /owners/{id}/dogs
- **Versioning**: /v1 prefix or Accept: application/vnd.app.v1+json
- **Validation** & contracts: JSON Schema / Protobuf; reject bad input early
- **Pagination**: *cursor*-based over *offset* for scale; include next_cursor
- **Filtering & sorting**: explicit query params; document allowed fields
- **Errors**: consistent error envelope with codes and correlation IDs
- **Idempotency**: Idempotency-Key header for POST that can be retried safely
- **Observability**: request IDs, structured logs, metrics, traces

Minimal FastAPI Example

```
from fastapi import FastAPI, HTTPException, Query
from pydantic import BaseModel
from typing import Optional, List

app = FastAPI(title="Dogs API", version="1.0")

class Dog(BaseModel):
    id: int
```

```python
    name: str
    breed: Optional[str] = None

DB: dict[int, Dog] = {}

@app.get("/v1/dogs", response_model=List[Dog])
def list_dogs(limit: int = Query(10, le=100), cursor: Optional[int] = None):
    # pretend cursor is last-id-seen
    ids = sorted(DB.keys())
    start = ids.index(cursor) + 1 if (cursor in ids) else 0
    page = ids[start:start+limit]
    next_cursor = page[-1] if len(page) == limit else None
    return [DB[i] for i in page]

@app.post("/v1/dogs", response_model=Dog)
def create_dog(d: Dog):
    if d.id in DB:
        raise HTTPException(409, "Dog exists")
    DB[d.id] = d
    return d
```

Spring Boot (Java) — Controller Snippet

```java
@RestController
@RequestMapping("/v1/dogs")
public class DogController {
  private final DogService service;
  public DogController(DogService service) { this.service = service; }

  @GetMapping
  public ResponseEntity<List<Dog>> list(@RequestParam(defaultValue="10") int limit,
                                        @RequestParam(required=false) String cursor) {
    Page<Dog> page = service.list(limit, cursor);
    return ResponseEntity.ok().body(page.items());
  }

  @PostMapping
  public ResponseEntity<Dog> create(@RequestBody @Valid Dog d) {
    Dog saved = service.create(d);
    return ResponseEntity.status(HttpStatus.CREATED).body(saved);
  }
}
```

4) Optimizing APIs: Latency, Throughput & Cost

- **Keep connections alive**: HTTP/2 or HTTP/3; TCP reuse; gRPC for internal RPC
- **Minimize payloads**: field selection, compression (gzip/br), binary (protobuf)
- **Cache aggressively**: CDN for static/GET, Redis for hot keys, client■side caching with ETags
- **Avoid N+1**: batch endpoints; server■side joins; precompute/materialize
- **Parallelize**: concurrent downstream calls; async IO; reactive (Project Reactor, asyncio)
- **Backpressure**: bounded queues; shed load early; tail■latency awareness
- **Database hygiene**: proper indexes, read replicas, partitioning/sharding

Caching Example (Spring)

```
@Service
public class DogService {
  @Cacheable(value="dogById", key="#id", unless="#result == null")
  public Dog getDog(long id) { /* load from DB */ }
}
```

Nginx Rate Limit Snippet

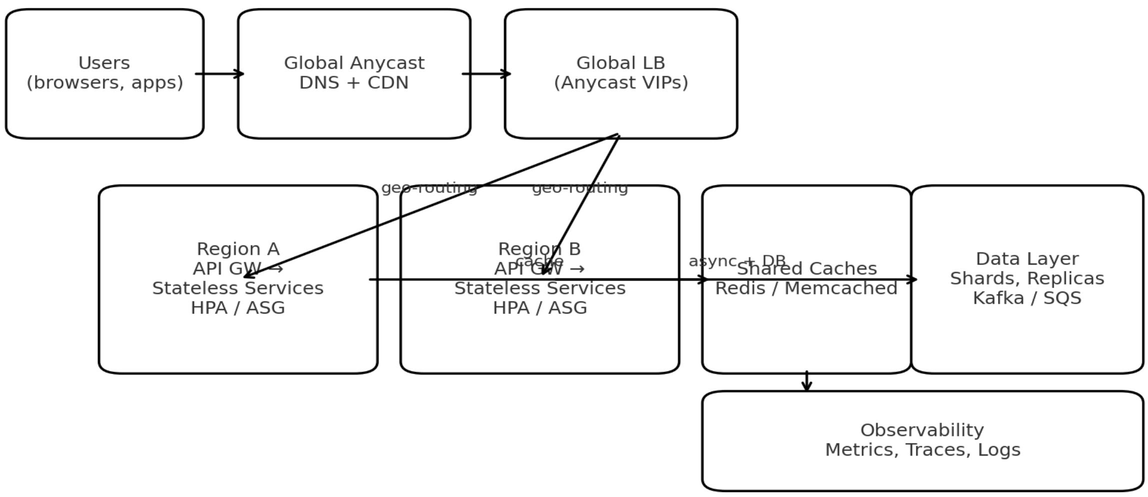
```
http {
  limit_req_zone $binary_remote_addr zone=api_limit:10m rate=100r/s;
  server {
    location /api/ {
      limit_req zone=api_limit burst=200 nodelay;
      proxy_pass http://upstream_pool;
    }
  }
}
```

Kubernetes HPA (autoscaling)

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata: { name: dogs-api }
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: dogs-api
  minReplicas: 4
  maxReplicas: 200
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 60
```

5) From 10 RPS to 1,000,000 RPS: An Engineering Playbook

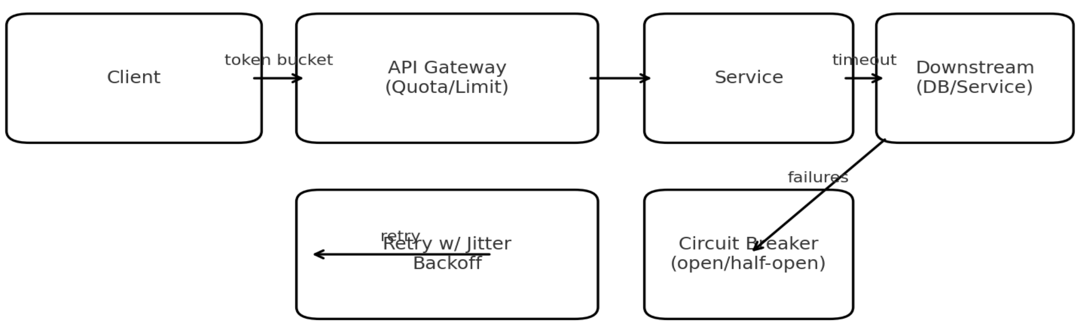
Conceptual: Scaling to ~1M Requests/sec



- **Edge first**: Anycast DNS + CDN to terminate TLS close to users; cache popular GETs
- **Stateless services**: keep servers disposable; store session in tokens or Redis
- **Horizontal scaling**: many small instances behind L4/L7 load balancers
- **Split read vs write**: heavy reads to replicas/cache; writes batched & queued
- **Async pipelines**: Kafka/SQS to absorb spikes; process with consumers
- **Data partitioning**: shard by key; avoid hot partitions with good keys (hash or rendezvous)
- **Rate limits & quotas**: per user/app; protect shared resources
- **Multi■region**: active■active when possible; graceful failover; eventual consistency
- **Capacity planning**: measure p50/p95/p99; use load tests; apply Little's Law ($L = \lambda W$)

Rate Limiting, Retries & Circuit Breakers

Rate Limiting, Retries & Circuit Breakers



Retries must use exponential backoff + jitter; **circuit breakers** stop hammering broken dependencies; **bulkheads** isolate resource pools; **timeouts** bound work; **idempotency keys** make retries safe.

6) Security & Reliability

- **TLS everywhere**: encrypt in transit; pin to TLS1.2+
- **AuthN/AuthZ**: OAuth2/OIDC for users; mTLS/service mesh (Istio/Linkerd) for service■to■service
- **Input validation**: never trust input; enforce schemas; sanitize logs
- **Secrets**: store in a vault (KMS, HashiCorp Vault) — never in code
- **Least privilege**: scoped tokens; resource■level permissions
- **Observability**: traces (W3C Trace Context), metrics (RED/USE), structured logs with IDs
- **Disaster recovery**: backups, PITR, chaos drills, RPO/RTO targets

7) Testing Strategy

- **Unit**: business logic and validators
- **Integration**: against a real DB or containers (Testcontainers)
- **Contract tests**: provider/consumer (e.g., Pact) to prevent breaking clients
- **Performance**: k6/JMeter/Gatling; test p95/p99, steady state and burst
- **Chaos**: inject latency and faults to verify resilience

8) Alternatives to Request/Response APIs

- **Event■driven**: publish/subscribe with Kafka, Pulsar; consumers react asynchronously
- **Batch/File**: scheduled S3/Blob drops with schemas; good for large payloads
- **SDKs**: client libraries wrapping APIs; great DX but maintain per language
- **Direct DB access**: rarely acceptable; security & coupling issues
- **iPaaS/No■code**: Zapier, n8n for simple integrations; limited control at scale

Appendix A — HTTP Status Codes (Quick Reference)

- 200 OK — success
- 201 Created — resource created
- 204 No Content — success, empty body
- 304 Not Modified — use your cache
- 400 Bad Request — validation failed
- 401 Unauthorized — auth required/invalid
- 403 Forbidden — not allowed
- 404 Not Found — resource missing
- 409 Conflict — duplicate or version conflict
- 422 Unprocessable Entity — semantic validation error
- 429 Too Many Requests — throttled
- 500/502/503/504 — server/gateway errors

Appendix B — Error Envelope Example

```
{
  "error": {
    "code": "DOG_ALREADY_EXISTS",
    "message": "Dog exists",
    "correlation_id": "5a9f6c6d-8b32-4db3-9e10-7a95d6fef111",
    "details": { "id": 123 }
  }
}
```