Complete Backend & System Design Interview Guide

Your complete roadmap to crushing backend and system design interviews

---

Part 1: Backend Fundamentals

1. How the Internet Works

Q: What happens when you type google.com in your browser?

Answer:

1. **DNS Resolution**: Browser finds IP address (142.250.190.46)
2. **TCP Handshake**: 3-way handshake (SYN, SYN-ACK, ACK)
3. **TLS Handshake**: Negotiate encryption, verify certificate
4. **HTTP Request**: Send GET request
5. **Server Processing**: Load balancer routes to server
6. **HTTP Response**: Server sends HTML back
7. **Rendering**: Browser parses HTML, loads assets, renders page

---

2. HTTP Methods

Q: When to use each HTTP method?

Answer:

- **GET**: Retrieve data. Idempotent, cacheable, no body
- **POST**: Create new resource. Not idempotent
- **PUT**: Replace entire resource. Idempotent
- **PATCH**: Partial update
- **DELETE**: Remove resource. Idempotent

---

3. HTTP Status Codes

Answer:

- **2xx**: Success (200 OK, 201 Created, 204 No Content)
- **3xx**: Redirect (301 Permanent, 302 Temporary, 304 Not Modified)
- **4xx**: Client Error (400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 429 Rate Limit)
- **5xx**: Server Error (500 Internal Error, 502 Bad Gateway, 503 Unavailable, 504 Timeout)

---

4. REST API Design

Q: Design a blog API

Answer:

```
GET    /posts              - List posts
GET    /posts/{id}         - Get post
POST   /posts              - Create post
PUT    /posts/{id}         - Update post
DELETE /posts/{id}         - Delete post
GET    /posts/{id}/comments - Get comments
POST   /posts/{id}/comments - Add comment
```

Best practices: Use nouns, plurals, proper HTTP methods, versioning (/v1/)

---

5. Authentication vs Authorization

Answer:

- **Authentication**: Who are you? (Login, JWT, OAuth)
- **Authorization**: What can you do? (RBAC, permissions)

Example: User logs in (auth) → tries to delete post (authz checks if they own it)

---

6. Authentication Methods

Session-Based:

python

```python
# Server stores session, returns session_id in cookie
login() → session_id → cookie
request() → cookie → validate session
```

Pros: Can revoke instantly Cons: Stateful, doesn't scale easily

Token-Based (JWT):

python

```
# Server generates signed token
login() → JWT token → client stores
request() → JWT in header → validate signature
```

Pros: Stateless, scales well Cons: Can't revoke until expiry

---

7. SQL vs NoSQL

Use SQL when:

- Need ACID transactions (banking)
- Complex queries with JOINs
- Clear relationships
- Schema is stable

Use NoSQL when:

- Massive scale needed
- Schema changes frequently
- Unstructured data
- Eventual consistency OK

NoSQL Types:

- **Document**: MongoDB (JSON docs)
- **Key-Value**: Redis (cache)
- **Column-Family**: Cassandra (time series)
- **Graph**: Neo4j (social networks)

---

8. Database Indexing

Q: When to use indexes?

Answer: Index = Table of contents for database

sql

```sql
CREATE INDEX idx_email ON users(email);
-- Fast: WHERE email = 'john@example.com'

CREATE INDEX idx_name_age ON users(last_name, first_name);
-- Fast: WHERE last_name = 'Smith'
-- Slow: WHERE first_name = 'John' (skips first column)
```

Trade-offs:

- Speeds up reads
- Slows down writes
- Uses storage

---

9. ACID Properties

Answer:

- **Atomicity**: All or nothing
- **Consistency**: Valid state always
- **Isolation**: Concurrent transactions don't interfere
- **Durability**: Committed data survives crashes

Isolation Levels (weak → strong):

1. Read Uncommitted (dirty reads possible)
2. Read Committed (most common)
3. Repeatable Read
4. Serializable (slowest, safest)

---

10. N+1 Query Problem

Problem:

python

```python
posts = db.query("SELECT * FROM posts")  # 1 query
for post in posts:
    author = db.query("SELECT * FROM users WHERE id = ?", post.author_id)  # N queries
# Total: 1 + N queries!
```

Solution:

python

```python
# Use JOIN
posts = db.query("""
    SELECT posts.*, users.name
    FROM posts JOIN users ON posts.author_id = users.id
""")  # 1 query only
```

---

11. Caching Strategies

Cache-Aside:

python

```python
data = cache.get(key)
if not data:
    data = db.get(key)
    cache.set(key, data)
return data
```

Write-Through:

python

```python
db.update(key, data)
cache.set(key, data)  # Update cache immediately
```

Cache Eviction:

- **LRU**: Remove least recently used
- **LFU**: Remove least frequently used
- **TTL**: Expire after time

---

12. Message Queues

Why use them:

- Decouple services
- Handle traffic spikes
- Async processing
- Retry failures

Example:

python

```python
# Producer
queue.publish('send_email', {'to': 'user@example.com'})

# Consumer
@worker('send_email')
def send_email(message):
    email.send(message['to'])
```

---

13. API Rate Limiting

Token Bucket:

python

```python
class RateLimiter:
    def allow_request(self):
        # Refill tokens over time
        # Check if tokens available
        if tokens >= 1:
            tokens -= 1
            return True
        return False
```

Return headers:

```
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 87
```

Part 2: System Design Principles

1. Scalability

Vertical Scaling (Scale Up):

- Add more CPU/RAM to one machine
- Simple but has limits
- Single point of failure

Horizontal Scaling (Scale Out):

- Add more machines
- Needs load balancing
- Nearly unlimited scaling

---

2. CAP Theorem

You can only have 2 of 3:

- **Consistency**: All nodes see same data
- **Availability**: Always get response
- **Partition Tolerance**: Works despite network failures

In practice:

- **CP**: MongoDB (sacrifice availability)
- **AP**: Cassandra (sacrifice consistency)

---

3. Load Balancing

Algorithms:

- **Round Robin**: Request 1→A, 2→B, 3→C, 4→A
- **Least Connections**: Route to server with fewest connections
- **IP Hash**: Same client → same server (session affinity)

Layer 4 vs Layer 7:

- **L4**: Routes by IP/port (fast)
- **L7**: Routes by URL/headers (flexible)

---

4. Caching Layers

```
User → Browser Cache → CDN → App Cache (Redis) → DB Cache → Database
```

Each layer stores subset of data, getting faster but smaller.

---

5. Database Sharding

Split data across multiple databases.

Hash-Based:

python

```python
shard_id = hash(user_id) % num_shards
```

Even distribution

Range-Based:

```
Shard 1: user_id 1-1M
Shard 2: user_id 1M-2M
```

Can have hotspots

---

6. Replication

Master-Slave:

```
Master (writes) → Slave1, Slave2, Slave3 (reads)
```

Scales reads, single point for writes

Master-Master:

```
Master1 ↔ Master2 (both accept writes)
```

No single point of failure, needs conflict resolution

---

7. Microservices vs Monolith

Monolith:

- Single codebase/deployment
- Good for: Small teams, early stage
- Scales as one unit

Microservices:

- Independent services
- Good for: Large teams, need independent scaling
- Complex infrastructure

Start with modular monolith, split later if needed

---

8. Service Communication

Sync (HTTP):

python

```python
payment = http.post("payment-service/charge", data)
```

Simple, but tight coupling

Async (Message Queue):

python

```python
queue.publish("order.created", data)
# Payment service subscribes and processes
```

Loose coupling, eventual consistency

---

9. Circuit Breaker

Prevents cascading failures.

States:

- **Closed**: Normal (requests pass)
- **Open**: Service failing (fail fast)
- **Half-Open**: Testing recovery

python

```
if circuit_breaker.is_open():
    return error("Service unavailable")
try:
    return call_service()
except:
    circuit_breaker.record_failure()
```

---

10. Consistency Models

Strong Consistency:

- After write, all reads see new value immediately
- Slower
- Use for: Banking, inventory

Eventual Consistency:

- After write, reads eventually see new value
- Faster
- Use for: Social media, analytics

Part 3: System Design Questions

Design 1: URL Shortener

Requirements: Shorten URLs, redirect, 100M URLs/month

Solution:

API:

```
POST /shorten → {short_url}
GET /{code} → redirect
```

Generate short code:

python

```python
def encode(id):
    # Base62 encoding (0-9, a-z, A-Z)
    chars = "0-9a-zA-Z"
    result = ""
    while id > 0:
        result = chars[id % 62] + result
        id //= 62
    return result  # e.g., "dnh9"
```

Architecture:

```
User → LB → App Servers → Redis (cache) → DB
```

Read flow:

python

```python
url = cache.get(code)
if not url:
    url = db.get(code)
    cache.set(code, url)
return redirect(url)
```

Scale: 40 writes/sec, 4000 reads/sec. Cache hot URLs (80/20 rule).

Design 2: Twitter Feed

Requirements: Post tweets, follow users, view timeline

Schema:

sql

```
tweets(id, user_id, content, created_at)
follows(follower_id, followee_id)
```

Timeline Approaches:

Pull (read-time):

python

```python
following_ids = get_following(user_id)
tweets = db.query("SELECT * FROM tweets WHERE user_id IN (?)", following_ids)
```

Slow for many followings

Push (write-time):

python

```python
# When posting
followers = get_followers(user_id)
for follower in followers:
    cache.lpush(f"timeline:{follower}", tweet_id)
```

Fast reads, expensive for celebrities

Hybrid (Twitter's approach):

- Push for regular users
- Pull for celebrities
- Merge at read time

Design 3: Instagram

Requirements: Upload photos, view feed

Upload flow:

python

```
# 1. Save original to S3
s3.upload(photo_id, file)

# 2. Queue processing
queue.publish('resize_photo', photo_id)

# 3. Worker resizes (thumbnail, medium, large)
# 4. Upload all sizes to S3/CDN
```

Feed: Same as Twitter (pre-compute timelines)

Optimization:

- Multiple image sizes
- CDN for global distribution
- Lazy loading

Design 4: Uber

Requirements: Match riders with drivers, real-time location

Key component: Find nearby drivers

Using Redis Geo:

python

```python
# Driver updates location
redis.geoadd('drivers', lng, lat, driver_id)

# Find nearby
drivers = redis.georadius('drivers', lng, lat, 5km)
```

Ride flow:

```
1. Rider requests → find nearby drivers
2. Notify drivers
3. Driver accepts
4. Real-time tracking via WebSocket
5. Complete ride → calculate fare
```

Fare calculation:

python

```python
fare = base + (distance * per_km) + (time * per_min)
fare *= surge_multiplier
```

Design 5: Netflix

Requirements: Stream videos, multiple quality levels

Video processing:

python

```python
# Upload → transcode to 360p, 720p, 1080p, 4K
# Create HLS segments (10 sec chunks)
# Store on S3 → serve via CDN
```

Adaptive streaming (HLS):

- Client downloads manifest
- Picks quality based on bandwidth
- Switches quality dynamically

Architecture:

```
User → CDN (edge) → Origin → S3
```

Scale: Cache video segments (immutable), use geo-distributed CDN

Design 6: WhatsApp

Requirements: Real-time messaging, delivery guarantee

Architecture:

```
User → WebSocket → Chat Server → Kafka → Cassandra
```

Message flow:

python

```python
# Send
kafka.publish(recipient_id, message)

# Receive
@websocket.on_message
def handle(msg):
    # If recipient online
    websocket.send(recipient_id, msg)
    # Else store for later
    db.save(msg)
```

Delivery guarantee:

- Message ID for deduplication
- ACK from recipient
- Retry with exponential backoff

Design 7: Google Drive

Requirements: Upload files, share, sync

Upload:

python

```
# Split file into chunks
chunks = split_file(file, chunk_size=4MB)

# Upload in parallel
for chunk in chunks:
    s3.upload(chunk_id, chunk)

# Save metadata
db.save(file_id, chunk_ids)
```

Sync:

- Client polls for changes (or WebSocket)
- Download only changed chunks
- Merge locally

Sharing:

sql

```sql
CREATE TABLE permissions (
    file_id, user_id, access_level
)
```

Design 8: Notification System

Requirements: Send email, SMS, push notifications

Architecture:

```
API → Queue → Workers (Email/SMS/Push) → Provider APIs
```

Priority queue:

python

```python
# High priority (OTP)
queue.publish('notifications_high', msg)

# Low priority (marketing)
queue.publish('notifications_low', msg)
```

Retry logic:

python

```python
@worker
def send(msg):
    try:
        provider.send(msg)
    except:
        if retry_count < 3:
            queue.publish_delayed(msg, delay=2^retry_count)
```

Design 9: Rate Limiter

Requirements: Limit API requests per user

Token Bucket (Redis):

python

```python
key = f"rate:{user_id}"
count = redis.incr(key)
if count == 1:
    redis.expire(key, window)
if count > limit:
    return 429  # Too many requests
```

Distributed: Use Redis cluster, handle race conditions with Lua scripts

---

Design 10: Search Engine

Requirements: Index web pages, search fast

Crawling:

python

```python
queue = ['seed_urls']
while queue:
    url = queue.pop()
    page = fetch(url)
    index(page)
    queue.extend(extract_links(page))
```

Indexing (Inverted Index):

```
"hello" → [doc1, doc5, doc9]
"world" → [doc1, doc3, doc9]
```

Ranking:

python

```python
score = TF-IDF * PageRank * freshness
```

Architecture:

```
User → LB → Query Servers → Index Shards
```

Part 4: Key Concepts Summary

Performance Metrics

- **Latency**: Time to complete one request
- **Throughput**: Requests per second
- **Availability**: Uptime percentage (99.9% = 43 min downtime/month)

Back-of-Envelope Calculations

```
1 million requests/day = 12 requests/sec
1 billion requests/day = 12K requests/sec

1 KB * 1 million users = 1 GB
1 MB * 1 million users = 1 TB
```

Storage

```
Text: ~1 KB per message
Image: ~500 KB
Video: ~50 MB (compressed)
```

Common Numbers

```
L1 cache: 1 ns
RAM: 100 ns
SSD: 100 μs
Network within DC: 500 μs
HDD: 10 ms
Network cross-continent: 100 ms
```

Part 5: Interview Tips

How to Approach System Design

1. **Clarify requirements** (5 min)
   - Functional: What features?
   - Non-functional: Scale, latency, availability?
   - Constraints: Budget, time?
2. **High-level design** (10 min)
   - Draw basic components
   - API design

    - Database schema
3. **Deep dive** (20 min)
    - Focus on 2-3 components
    - Discuss trade-offs
    - Address bottlenecks
4. **Wrap up** (5 min)
    - Monitoring/alerts
    - Future improvements

Things to Mention

- **Bottlenecks**: Where will system fail at scale?
- **Trade-offs**: Why choose X over Y?
- **Monitoring**: How to detect issues?
- **Failure scenarios**: What if DB goes down?

Red Flags to Avoid

- Don't jump to solution immediately
- Don't over-engineer for day 1
- Don't ignore scale numbers
- Don't forget about failures

Part 6: Common Patterns

API Design Patterns

Pagination:

```
GET /posts?page=2&limit=20
GET /posts?cursor=abc123&limit=20  (preferred)
```

Filtering:

```
GET /posts?status=published&author=john
```

Sorting:

```
GET /posts?sort=created_at&order=desc
```

Versioning:

```
/v1/posts
Accept: application/vnd.api.v2+json
```

Database Patterns

Soft Delete:

sql

```sql
UPDATE users SET deleted_at = NOW() WHERE id = ?
```

Optimistic Locking:

sql

```sql
UPDATE orders SET status = 'shipped', version = version + 1
WHERE id = ? AND version = ?
```

Denormalization: Store computed values to avoid JOINs

sql

```sql
posts(id, title, author_name)  -- Duplicate author name
```

Caching Patterns

Cache warming: Pre-populate cache before launch

Cache stampede prevention:

python

```python
lock = redis.lock(key)
if lock.acquire():
    data = compute_expensive()
    cache.set(key, data)
    lock.release()
```

Part 7: Technology Choices

Databases

PostgreSQL: Strong ACID, complex queries **MySQL:** Proven, good for reads **MongoDB:** Flexible schema, document storage **Cassandra:** Write-heavy, time series **Redis:** In-memory cache, pub/sub **Elasticsearch:** Full-text search, analytics

Message Queues

RabbitMQ: Traditional, reliable **Kafka:** High throughput, event streaming **SQS:** Managed, AWS **Redis Pub/Sub:** Simple, fast

Caching

Redis: Most popular, rich features **Memcached:** Simple, fast **CDN:** CloudFlare, CloudFront

Storage

S3: Object storage, cheap **EBS:** Block storage for servers **Glacier:** Archive, very cheap

---

Part 8: Final Checklist

Before your interview, make sure you can:

Explain clearly:

- [ ] HTTP request lifecycle
- [ ] Database indexes
- [ ] Caching strategies
- [ ] CAP theorem
- [ ] Sharding vs replication
- [ ] Microservices trade-offs
- [ ] Message queues

Design from scratch:

- [ ] URL shortener

- [ ] Social media feed
- [ ] Chat application
- [ ] Video streaming
- [ ] Ride sharing

Calculate:

- [ ] Storage needs
- [ ] Bandwidth needs
- [ ] QPS (queries per second)
- [ ] Number of servers needed

Discuss trade-offs:

- [ ] SQL vs NoSQL
- [ ] Sync vs Async
- [ ] Consistency vs Availability
- [ ] Monolith vs Microservices

You're Ready!

You now have comprehensive knowledge of: ✓ Backend fundamentals ✓ System design principles ✓ Real-world design patterns ✓ Interview strategies

Practice: Draw architectures on paper, explain out loud, time yourself.

Remember: There's no single "correct" answer. Show your thought process, discuss trade-offs, and be ready to defend your choices.

Go crush those interviews! 🚀