API Design Guide for System Design Interviews

Table of Contents

Introduction

API (Application Programming Interface) design is crucial in system design interviews. A well-designed API ensures:

- **Scalability**: Handle growth in users and data
- **Maintainability**: Easy to update and extend
- **Security**: Protect data and prevent unauthorized access
- **Performance**: Fast response times and efficient resource usage
- **Developer Experience**: Intuitive and easy to use

REST API Fundamentals

What is REST?

REST (Representational State Transfer) is an architectural style for designing networked applications.

Key Principles:

1. **Stateless**: Each request contains all information needed

2. **Client-Server**: Separation of concerns
3. **Cacheable**: Responses can be cached
4. **Uniform Interface**: Consistent naming and structure
5. **Layered System**: Client doesn't know if connected to end server
6. **Code on Demand** (optional): Server can send executable code

REST vs RPC vs GraphQL

Feature	REST	RPC	GraphQL
Data Transfer	Resources	Actions	Query-based
Over-fetching	Common	Common	Rare
Under-fetching	Common	Rare	Rare
Learning Curve	Medium	Low	High
Caching	Easy	Hard	Complex
Best For	CRUD ops	Remote procedures	Complex queries

API Design Principles

1. Consistency

- Use consistent naming conventions
- Follow standard HTTP methods
- Maintain uniform response structures

2. Simplicity

- Keep URLs simple and intuitive
- Use standard HTTP features
- Avoid unnecessary complexity

3. Flexibility

- Support multiple formats (JSON, XML)
- Allow filtering, sorting, pagination
- Version your APIs

4. Security

- Use HTTPS everywhere
- Implement proper authentication
- Validate all inputs
- Rate limit requests

5. Documentation

- Provide clear API documentation
- Include examples
- Document error responses

URL Design Best Practices

Resource Naming Conventions

```
✅ GOOD:
/users                    # Collection of users
/users/123                # Specific user
/users/123/orders         # User's orders
/users/123/orders/456     # Specific order

❌ BAD:
/getUsers                 # Don't use verbs
/user                     # Use plural
/users/123/getOrders      # Don't mix conventions
/user_orders              # Use hyphens not underscores
```

URL Structure Rules

1. **Use Nouns, Not Verbs**

   ```
   ✅ GET /articles
   ❌ GET /getArticles
   ```

2. **Use Plural Nouns**

   ```
   ✅ /users
   ❌ /user
   ```

3. **Use Hyphens for Readability**

   ```
   ✅ /user-profiles
   ❌ /user_profiles
   ❌ /userProfiles
   ```

4. **Lowercase Letters**

```
✅ /users/123/orders
❌ /Users/123/Orders
```

5. **Avoid Trailing Slashes**

```
✅ /users/123
❌ /users/123/
```

6. **Use Query Parameters for Filtering**

```
✅ /users?status=active&role=admin
❌ /users/active/admin
```

Nested Resources

```
# Limit nesting to 2-3 levels maximum
✅ /users/123/orders
✅ /users/123/orders/456/items

# For deeper relationships, use query parameters
✅ /items?orderId=456&userId=123
❌ /users/123/orders/456/items/789/reviews
```

HTTP Methods & Status Codes

HTTP Methods

Method	Purpose	Idempotent	Safe	Request Body	Response Body
GET	Retrieve resource	✅	✅	❌	✅
POST	Create resource	❌	❌	✅	✅
PUT	Update/Replace	✅	❌	✅	✅
PATCH	Partial update	❌	❌	✅	✅
DELETE	Delete resource	✅	❌	❌	Optional
HEAD	Get headers	✅	✅	❌	❌
OPTIONS	Get allowed methods	✅	✅	❌	✅

Common Status Codes

Success (2xx)

```
200 OK                    # Request succeeded
201 Created               # Resource created (POST)
202 Accepted              # Request accepted for processing
204 No Content            # Success but no content (DELETE)
```

Redirection (3xx)

```
301 Moved Permanently     # Resource permanently moved
302 Found                 # Temporary redirect
304 Not Modified          # Cached version is still valid
```

Client Errors (4xx)

```
400 Bad Request           # Invalid request syntax
401 Unauthorized          # Authentication required
403 Forbidden             # Authenticated but no permission
404 Not Found             # Resource doesn't exist
405 Method Not Allowed    # HTTP method not supported
409 Conflict              # Conflict with current state
422 Unprocessable Entity # Validation errors
429 Too Many Requests     # Rate limit exceeded
```

Server Errors (5xx)

```
500 Internal Server Error # Server encountered error
502 Bad Gateway           # Invalid response from upstream
503 Service Unavailable   # Server temporarily unavailable
504 Gateway Timeout       # Upstream server timeout
```

Request & Response Design

Request Structure

```
// POST /users
{
  "firstName": "John",
  "lastName": "Doe",
  "email": "john.doe@example.com",
```

```
  "age": 30
}
```

Response Structure

```
// Successful Response (200 OK)
{
  "data": {
    "id": "123",
    "firstName": "John",
    "lastName": "Doe",
    "email": "john.doe@example.com",
    "age": 30,
    "createdAt": "2024-01-15T10:30:00Z",
    "updatedAt": "2024-01-15T10:30:00Z"
  },
  "meta": {
    "timestamp": "2024-01-15T10:30:00Z",
    "version": "v1"
  }
}
```

Collection Response with Pagination

```
// GET /users?page=2&limit=10
{
  "data": [
    {
      "id": "123",
      "firstName": "John",
      "lastName": "Doe"
    }
  ],
  "meta": {
    "page": 2,
    "limit": 10,
    "total": 100,
    "totalPages": 10
  },
  "links": {
    "self": "/users?page=2&limit=10",
    "first": "/users?page=1&limit=10",
    "prev": "/users?page=1&limit=10",
    "next": "/users?page=3&limit=10",
    "last": "/users?page=10&limit=10"
  }
}
```

Error Response

```
// Error Response (400 Bad Request)
{
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Invalid input data",
    "details": [
      {
        "field": "email",
        "message": "Email is required"
      },
      {
        "field": "age",
        "message": "Age must be at least 18"
      }
    ]
  },
  "meta": {
    "timestamp": "2024-01-15T10:30:00Z",
    "requestId": "req-abc-123"
  }
}
```

Authentication & Authorization

Authentication Methods

1. API Keys

```
GET /users
X-API-Key: abc123def456

Pros: Simple, easy to implement
Cons: Less secure, difficult to revoke
Use Case: Public APIs, server-to-server
```

2. Bearer Tokens (JWT)

```
GET /users
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...

Pros: Stateless, scalable, contains claims
Cons: Cannot revoke easily, token size
Use Case: Modern web/mobile apps
```

3. OAuth 2.0

```
GET /users
Authorization: Bearer <access_token>

Pros: Industry standard, delegated access
Cons: Complex implementation
Use Case: Third-party integrations
```

4. Basic Authentication

```
GET /users
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=

Pros: Simple, widely supported
Cons: Credentials in every request
Use Case: Simple internal APIs
```

JWT Structure

```
Header:
{
  "alg": "HS256",
  "typ": "JWT"
}

Payload:
{
  "sub": "user123",
  "name": "John Doe",
  "iat": 1516239022,
  "exp": 1516242622,
  "roles": ["admin", "user"]
}

Signature:
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
)
```

Authorization Patterns

1. Role-Based Access Control (RBAC)

```
{
  "user": "john@example.com",
  "roles": ["admin", "editor"],
  "permissions": {
    "users": ["read", "write", "delete"],
    "posts": ["read", "write"]
  }
}
```

2. Attribute-Based Access Control (ABAC)

```
{
  "user": "john@example.com",
  "attributes": {
    "department": "engineering",
    "level": "senior",
    "region": "us-west"
  },
  "rules": [
    "department == engineering AND level == senior"
  ]
}
```

API Versioning

Why Version APIs?

- **Breaking Changes**: Modify response structure
- **New Features**: Add new endpoints
- **Deprecation**: Remove old functionality
- **Backward Compatibility**: Support old clients

Versioning Strategies

1. URI Versioning (Recommended)

```
✅ Most Common
GET /v1/users
GET /v2/users

Pros: Clear, easy to implement, cacheable
Cons: Duplicates code, URL pollution
```

2. Header Versioning

```
GET /users
Accept: application/vnd.api+json; version=1

Pros: Clean URLs
Cons: Harder to test, less discoverable
```

3. Query Parameter Versioning

```
GET /users?version=1

Pros: Easy to implement
Cons: Not RESTful, caching issues
```

4. Content Negotiation

```
GET /users
Accept: application/vnd.company.v1+json

Pros: RESTful, flexible
Cons: Complex, requires client support
```

Versioning Best Practices

```
1. Version only when breaking changes occur
2. Support at least N-1 versions
3. Clearly document deprecation timeline
4. Use semantic versioning (v1, v2, v3)
5. Default to latest stable version
6. Provide migration guides
```

Deprecation Example

```
// Response with deprecation warning
{
  "data": {...},
  "meta": {
    "deprecated": true,
    "deprecationDate": "2024-12-31",
```

```
    "sunsetDate": "2025-06-30",
    "alternativeEndpoint": "/v2/users",
    "migrationGuide": "https://api.example.com/docs/migration-v2"
  }
}
```

Rate Limiting

Why Rate Limit?

- Prevent abuse
- Ensure fair usage
- Protect infrastructure
- Control costs

Rate Limiting Algorithms

1. Token Bucket

```
- Bucket holds tokens
- Each request consumes token
- Tokens refill at fixed rate
- Request blocked if no tokens

Pros: Handles bursts, smooth rate
Cons: Complex implementation
```

2. Leaky Bucket

```
- Requests enter bucket
- Leave at constant rate
- Overflow requests rejected

Pros: Smooth output rate
Cons: May reject valid bursts
```

3. Fixed Window

```
- Count requests per time window
- Reset at window boundary
- Simple counter
```

```
Pros: Easy to implement
Cons: Burst at boundaries
```

4. Sliding Window

```
- Track requests in rolling window
- More accurate than fixed window

Pros: Prevents boundary bursts
Cons: More complex
```

Rate Limit Headers

```
HTTP/1.1 200 OK
X-RateLimit-Limit: 1000          # Max requests per window
X-RateLimit-Remaining: 997       # Remaining in window
X-RateLimit-Reset: 1640995200    # Unix timestamp of reset
Retry-After: 3600                # Seconds to wait
```

Rate Limit Response

```
// 429 Too Many Requests
{
  "error": {
    "code": "RATE_LIMIT_EXCEEDED",
    "message": "Rate limit exceeded",
    "retryAfter": 3600
  },
  "meta": {
    "limit": 1000,
    "remaining": 0,
    "resetAt": "2024-01-15T12:00:00Z"
  }
}
```

Rate Limiting Strategies

```
Tier-based Limits:
- Free: 100 requests/hour
- Basic: 1,000 requests/hour
- Premium: 10,000 requests/hour
- Enterprise: 100,000 requests/hour
```

```
Per-Endpoint Limits:
- GET /users: 1000/hour
- POST /users: 100/hour
- DELETE /users: 50/hour
```

Pagination

Why Paginate?

- Reduce response size
- Improve performance
- Better user experience
- Lower bandwidth

Pagination Methods

1. Offset-Based (Page Number)

```
GET /users?page=2&limit=20

Response:
{
  "data": [...],
  "meta": {
    "page": 2,
    "limit": 20,
    "total": 100,
    "totalPages": 5
  }
}

Pros: Simple, can jump to page
Cons: Inconsistent with new data, expensive OFFSET
```

2. Cursor-Based

```
GET /users?cursor=eyJpZCI6MTIzfQ&limit=20

Response:
{
  "data": [...],
  "meta": {
    "nextCursor": "eyJpZCI6MTQzfQ",
    "prevCursor": "eyJpZCI6MTAzfQ",
    "hasMore": true
  }
```

```
}

Pros: Consistent results, better performance
Cons: Can't jump to specific page
```

3. Keyset Pagination

```
GET /users?after_id=123&limit=20

Response:
{
  "data": [...],
  "meta": {
    "nextId": 143,
    "prevId": 103
  }
}

Pros: Fast, consistent
Cons: Requires indexed column
```

Pagination Best Practices

```
1. Default and maximum limits
   - Default: 20-50 items
   - Maximum: 100-500 items

2. Include pagination metadata
   - Total count
   - Current page/cursor
   - Next/previous links

3. Use HATEOAS links
   - self, first, last, next, prev

4. Consider performance
   - Cursor for real-time data
   - Offset for static data
```

Filtering, Sorting & Searching

Filtering

```
Single field:
GET /users?status=active

Multiple fields:
GET /users?status=active&role=admin

Ranges:
GET /users?age_min=18&age_max=65

Arrays:
GET /users?roles=admin,editor

Date ranges:
GET /orders?created_after=2024-01-01&created_before=2024-12-31
```

Sorting

```
Single field:
GET /users?sort=createdAt

Descending:
GET /users?sort=-createdAt

Multiple fields:
GET /users?sort=lastName,firstName

Complex:
GET /users?sort=-createdAt,lastName
```

Searching

```
Simple search:
GET /users?q=john

Field-specific:
GET /users?email=*@example.com

Full-text search:
GET /users?search=john%20doe&fields=firstName,lastName,email
```

Advanced Query Examples

```
// Complex filtering
GET /products?
  category=electronics&
```

```
  price_min=100&price_max=1000&
  brand=apple,samsung&
  rating_min=4&
  inStock=true&
  sort=-rating,price&
  page=1&limit=20

// Sparse fieldsets (return only specified fields)
GET /users?fields=id,firstName,email

// Including related resources
GET /users?include=orders,profile

// Excluding fields
GET /users?exclude=password,internalNotes
```

Error Handling

Error Response Structure

```
{
  "error": {
    "code": "RESOURCE_NOT_FOUND",
    "message": "User not found",
    "details": "No user exists with id: 123",
    "field": "userId"
  },
  "meta": {
    "timestamp": "2024-01-15T10:30:00Z",
    "requestId": "req-abc-123",
    "documentation":
"https://api.example.com/docs/errors/RESOURCE_NOT_FOUND"
  }
}
```

Standard Error Codes

```
Business Logic Errors:
- RESOURCE_NOT_FOUND
- RESOURCE_ALREADY_EXISTS
- INVALID_STATE_TRANSITION
- BUSINESS_RULE_VIOLATION

Validation Errors:
- VALIDATION_ERROR
- MISSING_REQUIRED_FIELD
- INVALID_FIELD_FORMAT
```

```
- INVALID_FIELD_VALUE

Authentication/Authorization:
- AUTHENTICATION_REQUIRED
- INVALID_CREDENTIALS
- TOKEN_EXPIRED
- INSUFFICIENT_PERMISSIONS

Rate Limiting:
- RATE_LIMIT_EXCEEDED
- QUOTA_EXCEEDED

System Errors:
- INTERNAL_SERVER_ERROR
- SERVICE_UNAVAILABLE
- DATABASE_ERROR
- EXTERNAL_SERVICE_ERROR
```

Validation Error Example

```
{
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Request validation failed",
    "details": [
      {
        "field": "email",
        "code": "INVALID_FORMAT",
        "message": "Email format is invalid",
        "value": "invalid-email"
      },
      {
        "field": "age",
        "code": "OUT_OF_RANGE",
        "message": "Age must be between 18 and 120",
        "value": 15,
        "constraints": {
          "min": 18,
          "max": 120
        }
      },
      {
        "field": "password",
        "code": "TOO_SHORT",
        "message": "Password must be at least 8 characters",
        "constraints": {
          "minLength": 8
        }
      }
    ]
```

```
  },
  "meta": {
    "timestamp": "2024-01-15T10:30:00Z",
    "requestId": "req-abc-123"
  }
}
```

Caching Strategies

Cache-Control Headers

```
Response Headers:
Cache-Control: public, max-age=3600        # Cache for 1 hour
Cache-Control: private, max-age=300        # User-specific, 5 mins
Cache-Control: no-cache                     # Revalidate before use
Cache-Control: no-store                     # Don't cache at all
Cache-Control: must-revalidate             # Always check if stale

ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4"
Last-Modified: Wed, 15 Jan 2024 10:30:00 GMT
```

Conditional Requests

```
Request:
GET /users/123
If-None-Match: "33a64df551425fcc55e4d42a148795d9f25f89d4"
If-Modified-Since: Wed, 15 Jan 2024 10:30:00 GMT

Response (Not Modified):
304 Not Modified
ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4"
```

Caching Layers

```
1. Browser Cache
   - Client-side caching
   - Cache-Control headers

2. CDN Cache
   - Edge locations
   - Static assets
   - Public endpoints

3. API Gateway Cache
```

```
   - Query results
   - Authentication tokens

4. Application Cache (Redis/Memcached)
   - Database queries
   - Computed results
   - Session data

5. Database Cache
   - Query results
   - Connection pooling
```

Cache Invalidation Strategies

```
1. Time-based (TTL)
   - Set expiration time
   - Simple but may serve stale data

2. Event-based
   - Invalidate on updates
   - Complex but always fresh

3. Cache Tagging
   - Tag related resources
   - Invalidate by tags

4. Versioned URLs
   - Include version in URL
   - Change URL on updates
```

Security Best Practices

1. Use HTTPS Everywhere

```
- Encrypt all communications
- Prevent man-in-the-middle attacks
- Use TLS 1.2 or higher
```

2. Input Validation

```
- Validate all inputs
- Sanitize user data
- Use whitelist approach
- Prevent injection attacks
```

3. Authentication & Authorization

```
- Use strong authentication
- Implement proper RBAC
- Validate permissions per request
- Use secure token storage
```

4. Rate Limiting

```
- Prevent brute force attacks
- Limit by IP, user, endpoint
- Implement exponential backoff
```

5. CORS Configuration

```
Access-Control-Allow-Origin: https://trusted-domain.com
Access-Control-Allow-Methods: GET, POST, PUT, DELETE
Access-Control-Allow-Headers: Content-Type, Authorization
Access-Control-Max-Age: 86400
```

6. Security Headers

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block
Content-Security-Policy: default-src 'self'
```

7. API Key Security

```
- Never expose in URLs
- Use environment variables
- Rotate regularly
- Hash before storing
- Use different keys per environment
```

8. SQL Injection Prevention

```
// ❌ BAD - String concatenation
query = "SELECT * FROM users WHERE id = " + userId;

// ✅ GOOD - Parameterized query
query = "SELECT * FROM users WHERE id = ?";
db.execute(query, [userId]);
```

9. Sensitive Data

```
- Don't return passwords
- Mask sensitive fields (SSN, credit cards)
- Log carefully (no PII)
- Use field-level encryption
```

API Documentation

Essential Documentation Elements

1. Overview

```
# Users API

## Base URL
https://api.example.com/v1

## Authentication
All requests require Bearer token authentication

## Rate Limits
- 1000 requests per hour for authenticated users
- 100 requests per hour for unauthenticated users
```

2. Endpoint Documentation

```
### Get User

Retrieve a specific user by ID.

**Endpoint:** `GET /users/{userId}`

**Parameters:**
- `userId` (path, required): The unique identifier of the user
```

```
**Query Parameters:**
- `include` (optional): Related resources to include (orders, profile)
- `fields` (optional): Specific fields to return

**Request Example:**
GET /users/123?include=orders&fields=id,firstName,email

**Response Example (200 OK):**
{
  "data": {
    "id": "123",
    "firstName": "John",
    "email": "john@example.com",
    "orders": [...]
  }
}

**Error Responses:**
- 404: User not found
- 401: Authentication required
- 403: Insufficient permissions
```

3. Data Models

```
### User Model

| Field | Type | Required | Description |
|-------|------|----------|-------------|
| id | string | Yes | Unique identifier |
| firstName | string | Yes | User's first name |
| lastName | string | Yes | User's last name |
| email | string | Yes | User's email (unique) |
| age | integer | No | User's age (18-120) |
| status | enum | Yes | active, inactive, suspended |
| createdAt | datetime | Yes | Creation timestamp |
| updatedAt | datetime | Yes | Last update timestamp |
```

Documentation Tools

```
1. OpenAPI/Swagger
   - Industry standard
   - Interactive docs
   - Code generation

2. Postman
   - Collection-based
   - Team collaboration
   - Testing support
```

```
3. API Blueprint
   - Markdown-based
   - Simple syntax

4. RAML
   - YAML-based
   - Reusable components
```

Performance Optimization

1. Response Compression

```
Request:
Accept-Encoding: gzip, deflate

Response:
Content-Encoding: gzip
(compressed content)

Benefits:
- 60-80% size reduction
- Faster transfer times
- Lower bandwidth costs
```

2. Connection Pooling

```
- Reuse TCP connections
- Reduce handshake overhead
- Configure pool size
- Set connection timeout
```

3. Database Optimization

```
- Use indexes effectively
- Implement query caching
- Optimize N+1 queries
- Use connection pooling
- Consider read replicas
```

4. Async Processing

```
POST /orders
202 Accepted
Location: /orders/123/status

{
  "orderId": "123",
  "status": "processing",
  "statusUrl": "/orders/123/status"
}

Benefits:
- Fast response times
- Better user experience
- Handle long operations
```

5. Batch Operations

```
// Create multiple users
POST /users/batch
[
  {"firstName": "John", "email": "john@example.com"},
  {"firstName": "Jane", "email": "jane@example.com"}
]

Response:
{
  "created": 2,
  "failed": 0,
  "results": [...]
}
```

6. Field Selection

```
// Return only needed fields
GET /users?fields=id,firstName,email

Benefits:
- Smaller payloads
- Faster serialization
- Lower bandwidth
```

7. CDN Usage

```
- Cache static responses
- Serve from edge locations
```

```
- Reduce origin load
- Lower latency
```

GraphQL vs REST

When to Use GraphQL

Advantages:

- Flexible queries
- Single endpoint
- No over-fetching
- No under-fetching
- Strong typing
- Real-time with subscriptions

Use Cases:

- Complex data requirements
- Mobile applications
- Rapidly changing requirements
- Multiple client types

When to Use REST

Advantages:

- Simple and familiar
- Easy caching
- Better tooling
- Stateless
- HTTP features

Use Cases:

- CRUD operations
- Public APIs
- Microservices
- Simple requirements

Comparison Example

REST

```
// Multiple requests needed
GET /users/123
GET /users/123/orders
```

```
GET /users/123/profile

// May get unnecessary data
{
  "id": "123",
  "firstName": "John",
  "lastName": "Doe",
  "email": "john@example.com",
  "age": 30,
  "address": {...},  // Not needed
  "preferences": {...}  // Not needed
}
```

GraphQL

```
// Single request, exact data needed
query {
  user(id: "123") {
    firstName
    email
    orders {
      id
      total
    }
  }
}

// Get only requested data
{
  "data": {
    "user": {
      "firstName": "John",
      "email": "john@example.com",
      "orders": [...]
    }
  }
}
```

Common Interview Scenarios

1. Social Media Feed API

```
GET /feed
Authorization: Bearer {token}
Query Parameters:
- cursor: pagination cursor
- limit: items per page (default: 20)
```

```
- include: comments, likes

Response:
{
  "data": [
    {
      "id": "post1",
      "userId": "123",
      "content": "Hello world!",
      "createdAt": "2024-01-15T10:30:00Z",
      "likes": 42,
      "comments": 5,
      "isLiked": true,
      "user": {
        "id": "123",
        "username": "johndoe",
        "avatar": "https://cdn.example.com/avatars/123.jpg"
      }
    }
  ],
  "meta": {
    "nextCursor": "eyJpZCI6InBvc3QxMCJ9",
    "hasMore": true
  }
}
```

Key Design Considerations:

- Use cursor-based pagination for real-time feeds
- Include user information to avoid additional requests
- Use `isLiked` flag for personalized experience
- Cache aggressively with CDN
- Implement rate limiting per user

---

2. E-commerce Product Search API

```
GET /products/search
Query Parameters:
- q: search query
- category: filter by category
- price_min, price_max: price range
- brand: filter by brands (comma-separated)
- rating_min: minimum rating
- sort: sorting (price_asc, price_desc, rating, newest)
- page, limit: pagination

Response:
{
  "data": {
```

```
    "products": [
      {
        "id": "prod1",
        "name": "iPhone 15 Pro",
        "price": 999.99,
        "currency": "USD",
        "rating": 4.5,
        "reviewCount": 1234,
        "inStock": true,
        "imageUrl": "https://cdn.example.com/products/prod1.jpg",
        "brand": "Apple",
        "category": "Electronics"
      }
    ],
    "facets": {
      "brands": [
        {"name": "Apple", "count": 45},
        {"name": "Samsung", "count": 38}
      ],
      "priceRanges": [
        {"min": 0, "max": 500, "count": 120},
        {"min": 500, "max": 1000, "count": 80}
      ]
    }
  },
  "meta": {
    "page": 1,
    "limit": 20,
    "total": 156
  }
}
```

Key Design Considerations:

- Include facets for filtering
- Support complex filtering and sorting
- Use Elasticsearch for full-text search
- Cache popular searches
- Implement autocomplete endpoint

---

3. Payment Processing API

```
POST /payments
Authorization: Bearer {token}
Idempotency-Key: {unique-key}

Request:
{
  "orderId": "order123",
```

```
  "amount": 99.99,
  "currency": "USD",
  "paymentMethod": {
    "type": "credit_card",
    "cardToken": "tok_abc123"
  },
  "billingAddress": {...},
  "metadata": {
    "customerId": "cust123"
  }
}

Response (202 Accepted):
{
  "data": {
    "paymentId": "pay_xyz789",
    "status": "processing",
    "orderId": "order123",
    "amount": 99.99,
    "currency": "USD",
    "createdAt": "2024-01-15T10:30:00Z",
    "statusUrl": "/payments/pay_xyz789/status"
  }
}

// Check status
GET /payments/pay_xyz789/status

Response:
{
  "data": {
    "paymentId": "pay_xyz789",
    "status": "completed",
    "transactionId": "txn_abc123",
    "completedAt": "2024-01-15T10:30:05Z"
  }
}
```

Key Design Considerations:

- Use idempotency keys to prevent duplicate charges
- Async processing with status endpoint
- Never store raw card details
- Use payment tokens
- Implement webhooks for status updates
- PCI DSS compliance
- Strong authentication (3D Secure)

---

4. Notification System API

```
POST /notifications/send
Authorization: Bearer {token}

Request:
{
  "recipients": ["user123", "user456"],
  "channels": ["email", "push", "sms"],
  "template": "order_confirmation",
  "data": {
    "orderId": "order123",
    "amount": 99.99,
    "items": [...]
  },
  "priority": "high",
  "scheduledAt": "2024-01-15T15:00:00Z"
}

Response (202 Accepted):
{
  "data": {
    "notificationId": "notif_xyz789",
    "status": "scheduled",
    "recipients": 2,
    "estimatedDelivery": "2024-01-15T15:00:00Z"
  }
}

// Get notification status
GET /notifications/notif_xyz789

Response:
{
  "data": {
    "notificationId": "notif_xyz789",
    "status": "sent",
    "deliveryDetails": [
      {
        "recipient": "user123",
        "channel": "email",
        "status": "delivered",
        "deliveredAt": "2024-01-15T15:00:05Z"
      },
      {
        "recipient": "user123",
        "channel": "push",
        "status": "delivered",
        "deliveredAt": "2024-01-15T15:00:03Z"
      }
    ]
  }
}
```

Key Design Considerations:

- Support multiple channels
- Use message queues (SQS, Kafka)
- Template-based notifications
- Scheduling support
- User preferences for channels
- Delivery tracking
- Retry logic for failures

---

5. File Upload API

```
POST /uploads/initiate
Authorization: Bearer {token}

Request:
{
  "fileName": "document.pdf",
  "fileSize": 5242880,
  "contentType": "application/pdf",
  "metadata": {
    "description": "Contract document"
  }
}

Response:
{
  "data": {
    "uploadId": "upload_abc123",
    "uploadUrl": "https://s3.example.com/bucket/key?signature=...",
    "method": "PUT",
    "headers": {
      "Content-Type": "application/pdf"
    },
    "expiresAt": "2024-01-15T11:30:00Z"
  }
}

// Client uploads directly to S3
PUT https://s3.example.com/bucket/key?signature=...
Content-Type: application/pdf
(binary data)

// Confirm upload
POST /uploads/upload_abc123/complete
Authorization: Bearer {token}

Response:
{
```

```
  "data": {
    "fileId": "file_xyz789",
    "url": "https://cdn.example.com/files/file_xyz789",
    "size": 5242880,
    "uploadedAt": "2024-01-15T10:35:00Z"
  }
}
```

Key Design Considerations:

- Direct upload to S3 (signed URLs)
- Chunked upload for large files
- Resume capability
- Virus scanning
- Content validation
- CDN for delivery
- Access control

Example API Designs

Twitter-like API

```
# User endpoints
POST /v1/users/register
POST /v1/auth/login
GET /v1/users/{userId}
PUT /v1/users/{userId}
GET /v1/users/{userId}/followers
GET /v1/users/{userId}/following
POST /v1/users/{userId}/follow
DELETE /v1/users/{userId}/follow

# Tweet endpoints
POST /v1/tweets
GET /v1/tweets/{tweetId}
DELETE /v1/tweets/{tweetId}
POST /v1/tweets/{tweetId}/like
DELETE /v1/tweets/{tweetId}/like
POST /v1/tweets/{tweetId}/retweet
GET /v1/tweets/{tweetId}/replies

# Feed endpoints
GET /v1/feed/home
GET /v1/feed/user/{userId}
GET /v1/feed/trending

# Search endpoints
GET /v1/search/tweets
```

```
GET /v1/search/users
GET /v1/search/hashtags
```

E-commerce API

```
# Product catalog
GET /v1/products
GET /v1/products/{productId}
GET /v1/products/search
GET /v1/categories
GET /v1/categories/{categoryId}/products

# Shopping cart
POST /v1/cart/items
GET /v1/cart
PUT /v1/cart/items/{itemId}
DELETE /v1/cart/items/{itemId}
DELETE /v1/cart

# Orders
POST /v1/orders
GET /v1/orders
GET /v1/orders/{orderId}
PUT /v1/orders/{orderId}/cancel
GET /v1/orders/{orderId}/tracking

# Payments
POST /v1/payments
GET /v1/payments/{paymentId}
POST /v1/payments/{paymentId}/refund

# Reviews
POST /v1/products/{productId}/reviews
GET /v1/products/{productId}/reviews
PUT /v1/reviews/{reviewId}
DELETE /v1/reviews/{reviewId}
```

Ride-sharing API

```
# Riders
POST /v1/rides/request
GET /v1/rides/{rideId}
PUT /v1/rides/{rideId}/cancel
GET /v1/rides/active
GET /v1/rides/history
```

```
# Drivers
POST /v1/drivers/status
PUT /v1/drivers/location
POST /v1/rides/{rideId}/accept
POST /v1/rides/{rideId}/start
POST /v1/rides/{rideId}/complete

# Pricing
POST /v1/pricing/estimate
GET /v1/pricing/surge

# Payments
GET /v1/payment-methods
POST /v1/payment-methods
DELETE /v1/payment-methods/{methodId}
```

Video Streaming API

```
# Videos
GET /v1/videos
GET /v1/videos/{videoId}
POST /v1/videos
DELETE /v1/videos/{videoId}
GET /v1/videos/{videoId}/stream
GET /v1/videos/trending
GET /v1/videos/recommended

# Interactions
POST /v1/videos/{videoId}/views
POST /v1/videos/{videoId}/like
POST /v1/videos/{videoId}/comments
GET /v1/videos/{videoId}/comments

# Channels
GET /v1/channels/{channelId}
POST /v1/channels/{channelId}/subscribe
GET /v1/channels/{channelId}/videos

# Playlists
POST /v1/playlists
GET /v1/playlists/{playlistId}
POST /v1/playlists/{playlistId}/videos
DELETE /v1/playlists/{playlistId}/videos/{videoId}
```

Interview Tips & Best Practices

During the Interview

1. **Clarify Requirements**
    - Ask about scale (users, requests/second)
    - Understand data model
    - Identify critical features
    - Discuss security requirements
2. **Start with High-Level Design**
    - Define main resources
    - Identify relationships
    - Plan URL structure
    - Choose HTTP methods
3. **Discuss Trade-offs**
    - REST vs GraphQL vs gRPC
    - Consistency vs Availability
    - Pagination approaches
    - Caching strategies
4. **Consider Non-Functional Requirements**
    - Performance (response time)
    - Scalability (concurrent users)
    - Reliability (uptime)
    - Security (authentication, authorization)
5. **Think About Edge Cases**
    - Rate limiting
    - Concurrent updates
    - Large payloads
    - Network failures

Common Mistakes to Avoid

1. ❌ Using verbs in URLs (`/getUsers`)
2. ❌ Not versioning APIs
3. ❌ Ignoring idempotency for critical operations
4. ❌ Not implementing proper error handling
5. ❌ Overlooking security considerations
6. ❌ Not considering pagination for large datasets
7. ❌ Forgetting about rate limiting
8. ❌ Not using appropriate HTTP status codes
9. ❌ Exposing internal implementation details
10. ❌ Not documenting breaking changes

Key Talking Points

1. **Scalability**
    - Horizontal scaling with load balancers
    - Stateless design
    - Caching strategies
    - Database sharding
2. **Performance**
    - CDN for static content
    - Response compression
    - Database optimization
    - Async processing
3. **Reliability**
    - Retry logic
    - Circuit breakers
    - Health checks
    - Graceful degradation
4. **Security**
    - HTTPS everywhere
    - Input validation
    - Rate limiting
    - Authentication/Authorization
5. **Monitoring**
    - Logging
    - Metrics
    - Alerting
    - Tracing

Quick Reference Checklist

API Design Checklist

- [ ] Use nouns for resources (not verbs)
- [ ] Use plural nouns for collections
- [ ] Implement proper HTTP methods (GET, POST, PUT, PATCH, DELETE)
- [ ] Return appropriate status codes
- [ ] Version your API (prefer URI versioning)
- [ ] Implement authentication and authorization
- [ ] Add rate limiting
- [ ] Support pagination for large datasets
- [ ] Provide filtering, sorting, and searching
- [ ] Implement proper error handling with clear messages

- [ ] Use HTTPS for all endpoints
- [ ] Add caching headers
- [ ] Document your API
- [ ] Consider idempotency for critical operations
- [ ] Implement request/response logging
- [ ] Add monitoring and alerting
- [ ] Validate all inputs
- [ ] Support CORS if needed
- [ ] Use compression for responses
- [ ] Implement health check endpoints

Additional Resources

Books

- "RESTful Web APIs" by Leonard Richardson
- "API Design Patterns" by JJ Geewax
- "Web API Design" by Brian Mulloy

Online Resources

- OpenAPI Specification: https://swagger.io/specification/
- REST API Tutorial: https://restfulapi.net/
- HTTP Status Codes: https://httpstatuses.com/

Tools

- Postman: API development and testing
- Swagger/OpenAPI: API documentation
- Insomnia: API client
- Stoplight: API design platform

Conclusion

Designing good APIs is both an art and a science. During system design interviews, focus on:

1. **Understanding requirements** - Ask clarifying questions
2. **Following best practices** - Use REST principles consistently
3. **Considering scale** - Think about performance and scalability
4. **Security first** - Always consider security implications
5. **Clear communication** - Explain your design decisions

Remember, there's often no single "correct" answer in system design. The key is to demonstrate your thought process, consider trade-offs, and justify your decisions based on the requirements.

Good luck with your interviews!