API Design Best Practices

Building Clear, Maintainable APIs

Designing interfaces that developers love to use

Why API Design Matters

> "An API is a user interface for developers"

- **Developer Experience** - Easy to understand and use
- **Maintainability** - Evolve without breaking clients
- **Consistency** - Predictable patterns reduce errors
- **Documentation** - Self-describing interfaces

The Problem

Common API Issues:

- 🤔 Inconsistent naming and structure
- 💔 Breaking changes without warning
- 📚 Poor or missing documentation
- 🐛 Unclear error messages
- 🔒 Inadequate security considerations

Result: Frustrated developers and integration problems

API Design Principles

Core Guidelines:

1. **RESTful Design**
2. **Consistent Naming Conventions**
3. **Versioning Strategies**
4. **Error Handling**
5. **Authentication & Authorization**
6. **Documentation**

1. RESTful API Design

Resource-Oriented Architecture

Key Principles:

- **Resources** - Represent nouns, not verbs
- **HTTP Methods** - Use standard verbs (GET, POST, PUT, PATCH, DELETE)
- **Stateless** - Each request contains all needed information
- **Cacheable** - Responses indicate if they can be cached

RESTful Resource Naming

Good Examples:

```
GET    /users                # List all users
GET    /users/123            # Get specific user
POST   /users                # Create new user
PUT    /users/123            # Update entire user
PATCH  /users/123            # Partial update
DELETE /users/123            # Delete user

GET    /users/123/orders     # User's orders (nested resource)
```

Avoid:

```
GET    /getUsers             # Verb in URL
POST   /user/create          # Verb in URL
GET    /users/123/delete     # Wrong HTTP method
```

HTTP Methods & Status Codes

HTTP Methods:

- **GET** - Retrieve data (safe, idempotent)
- **POST** - Create new resource
- **PUT** - Replace entire resource (idempotent)
- **PATCH** - Partial update (idempotent)
- **DELETE** - Remove resource (idempotent)

Common Status Codes:

- **200 OK** - Successful GET, PUT, PATCH, or DELETE
- **201 Created** - Successful POST
- **204 No Content** - Successful request with no response body
- **400 Bad Request** - Invalid client request

2. Consistent Naming Conventions

Predictability Reduces Errors

Best Practices:

- **Use plural nouns** - `/users` not `/user`
- **Use lowercase** - `/user-profiles` not `/UserProfiles`
- **Use hyphens** - `/user-profiles` not `/user_profiles` or `/userProfiles`
- **Be consistent** - Choose a convention and stick to it

JSON Field Naming:

```
{
  "userId": 123,              // camelCase (JavaScript/JSON standard)
  "firstName": "Jane",
  "createdAt": "2024-01-15T10:30:00Z"
```

Filtering, Sorting, and Pagination

Query Parameters:

```
GET /users?status=active&role=admin          # Filtering
GET /users?sort=createdAt&order=desc         # Sorting
GET /users?page=2&limit=20                   # Pagination
GET /users?search=john                        # Search
```

Response with Pagination:

```
{
  "data": [...],
  "pagination": {
    "currentPage": 2,
    "totalPages": 10,
    "pageSize": 20,
    "totalItems": 195
  }
}
```

3. API Versioning

Managing Change Over Time

Versioning Strategies:

1. URL Path Versioning (Most Common):

```
/api/v1/users
/api/v2/users
```

2. Header Versioning:

```
Accept: application/vnd.myapi.v1+json
```

3. Query Parameter:

```
/api/users?version=1
```

When to Introduce a New Version

Breaking Changes:

- Removing or renaming fields
- Changing data types
- Changing endpoint URLs
- Modifying authentication methods

Non-Breaking Changes (Same Version):

- Adding new optional fields
- Adding new endpoints
- Making required fields optional
- Adding new query parameters

Versioning Best Practices

Guidelines:

- **Support multiple versions** - Give clients time to migrate
- **Deprecation notice** - Announce version end-of-life early
- **Semantic versioning** - v1, v2, v3 (not v1.2.3 for APIs)
- **Document changes** - Clear changelog between versions

Example Deprecation Header:

```
Sunset: Sat, 31 Dec 2024 23:59:59 GMT
Deprecation: true
Link: <https://api.example.com/v2/users>; rel="successor-version"
```

4. Error Handling

Clear, Actionable Error Messages

Good Error Response:

```
{
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Invalid user data provided",
    "details": [
      {
        "field": "email",
        "issue": "Invalid email format"
      },
      {
        "field": "age",
        "issue": "Must be at least 18"
      }
    ],
```

Error Response Structure

Essential Fields:

- **code** - Machine-readable error code
- **message** - Human-readable description
- **details** - Specific validation errors (if applicable)
- **timestamp** - When the error occurred
- **requestId** - For tracking and debugging

HTTP Status Code + Error Code:

```
HTTP/1.1 400 Bad Request

{
  "error": {
    "code": "INVALID_INPUT",
    "message": "The request contains invalid data"
```

Common Error Patterns

Validation Errors (400):

```
{
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Input validation failed",
    "details": [...]
  }
}
```

Authentication Errors (401):

```
{
  "error": {
    "code": "INVALID_TOKEN",
    "message": "Authentication token is invalid or expired"
  }
}
```

5. Authentication & Authorization

Securing Your API

Common Authentication Methods:

- **API Keys** - Simple, but less secure
- **OAuth 2.0** - Industry standard for delegated access
- **JWT (JSON Web Tokens)** - Stateless authentication
- **Basic Auth** - Username/password (HTTPS only)

Best Practices:

- **Always use HTTPS** - Encrypt data in transit
- **Token expiration** - Implement refresh tokens
- **Rate limiting** - Prevent abuse

JWT Authentication Example

Request with JWT:

```
GET /api/v1/users/profile
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

JWT Structure:

```
{
  "header": {
    "alg": "HS256",
    "typ": "JWT"
  },
  "payload": {
    "userId": 123,
    "role": "admin",
    "exp": 1700000000
  },
  "signature": "..."
}
```

Authorization Patterns

Role-Based Access Control (RBAC):

```
{
  "userId": 123,
  "roles": ["admin", "editor"]
}
```

Permission-Based:

```
{
  "userId": 123,
  "permissions": ["users:read", "users:write", "posts:delete"]
}
```

Resource-Based:

```
GET /users/123/orders  # Can only access own orders
```

6. API Documentation

Making Your API Discoverable

Documentation Standards:

- **OpenAPI/Swagger** - Industry standard specification
- **Interactive docs** - Try endpoints in browser
- **Code examples** - Multiple languages
- **Authentication guide** - Step-by-step setup

OpenAPI/Swagger Example

```yaml
openapi: 3.0.0
info:
  title: User API
  version: 1.0.0
  description: API for managing users

paths:
  /users:
    get:
      summary: List all users
      parameters:
        - name: page
          in: query
          schema:
            type: integer
            default: 1
      responses:
        '200':
          description: Successful response
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/User'

components:
  schemas:
    User:
      type: object
      properties:
        id:
          type: integer
        name:
          type: string
        email:
          type: string
          format: email
```

Documentation Best Practices

Include:

- **Getting started guide** - Authentication, base URLs
- **Endpoint reference** - All methods, parameters, responses
- **Code examples** - cURL, JavaScript, Python, etc.
- **Error codes** - What each error means and how to fix
- **Rate limits** - Request limits and quotas
- **Changelog** - Version history and breaking changes

Tools:

- **Swagger UI** - Interactive API documentation
- **Postman** - API testing and documentation
- **ReadMe.io** - Developer-friendly docs

GraphQL Considerations

An Alternative to REST

When to Use GraphQL:

- Complex data relationships
- Need to minimize over-fetching
- Multiple client types (web, mobile)
- Rapid frontend development

GraphQL Benefits:

- **Single endpoint** - `/graphql`
- **Flexible queries** - Request exactly what you need
- **Strong typing** - Schema-based

GraphQL Example

Query:

```
query GetUser {
  user(id: 123) {
    name
    email
    orders {
      id
      total
      items {
        product {
          name
          price
        }
      }
    }
  }
}
```

REST vs. GraphQL

Choosing the Right Approach

Aspect	REST	GraphQL
Learning Curve	Lower	Higher
Caching	HTTP caching	Requires custom logic
Over-fetching	Common issue	Eliminated
Versioning	URL-based	Schema evolution
Tooling	Mature ecosystem	Growing ecosystem
Best For	CRUD operations	Complex data needs

Rate Limiting & Throttling

Protecting Your API

Strategies:

- **Per-user limits** - 1000 requests/hour per API key
- **IP-based limits** - Prevent abuse
- **Endpoint-specific** - Different limits for different operations
- **Burst allowance** - Allow short spikes

Rate Limit Headers:

```
X-RateLimit-Limit: 1000
X-RateLimit-Remaining: 995
X-RateLimit-Reset: 1700000000
```

API Testing

Ensuring Quality

Types of API Tests:

- **Unit tests** - Individual endpoint logic
- **Integration tests** - API + database
- **Contract tests** - API matches specification
- **Load tests** - Performance under stress

Tools:

- **Postman** - Manual and automated testing
- **Jest/Mocha** - JavaScript testing
- **Pytest** - Python testing

API Test Example

```
describe('User API', () => {
  test('POST /users creates a new user', async () => {
    const response = await request(app)
      .post('/api/v1/users')
      .send({
        name: 'Jane Doe',
        email: 'jane@example.com'
      })
      .set('Authorization', `Bearer ${token}`);

    expect(response.status).toBe(201);
    expect(response.body).toHaveProperty('id');
    expect(response.body.name).toBe('Jane Doe');
  });

  test('GET /users returns 401 without auth', async () => {
    const response = await request(app)
      .get('/api/v1/users');

    expect(response.status).toBe(401);
  });
});
```

Key Takeaways

🎯 Remember:

1. **Be consistent** - Pick conventions and stick to them
2. **Think long-term** - Design for evolution with versioning
3. **Clear errors** - Help developers fix problems quickly
4. **Document everything** - Your API is a product
5. **Security first** - Authentication, HTTPS, rate limiting
6. **Test thoroughly** - Automated testing prevents regressions

Questions & Discussion

Discussion Points:

- What API design mistakes have you encountered?
- REST vs. GraphQL - when to use which?
- How do you handle API versioning in your projects?
- What makes an API "developer-friendly"?

Thank You

Next Steps:

- **Review your APIs** - Check against these best practices
- **Implement versioning** - Prepare for future changes
- **Write OpenAPI specs** - Document your existing APIs
- **Add comprehensive error handling** - Better developer experience

Resources:

- REST API Design Best Practices
- OpenAPI Specification
- API Design Patterns