REST & GraphQL APIs

Table of Contents

Status	REVIEW
Version	1.0.0
Author(s)	@Daniel Steinhauf
Date of creation	Dec 18, 2025
Last updated	Dec 18, 2025
Approved on	
<name of the person(s) in charge to approve the document>	Date of approval: yyyy-mm-dd

Overview

This guideline establishes **consistent, scalable standards** for designing, implementing, and operating **REST and GraphQL APIs** across our engineering organization. It enforces an **API-First** approach, integrates **OpenAPI** for REST contracts, and provides framework-specific guidance for **Spring Boot (≥ 3.3)** and **Next.js**.

Scope

- **Design**: Resource modeling, schema design, versioning, error handling, pagination, caching, security, and retry policies.
- **Contracts**: OpenAPI (REST) and GraphQL schemas; mock servers; governance; CI validation.
- **Implementation**: Spring Boot services, Next.js consumers, and gateway concerns.
- **Operations**: Observability, reliability, performance, deprecation, and lifecycle management.

Audience

Backend and frontend engineers, architects, and platform teams working on services and clients that expose or consume REST/GraphQL APIs.

Outcomes

- Predictable APIs with **strong contracts**, **consistent error formats**, **robust security**, and **observable behavior**.
- Faster delivery via **contract-first development**, **mocking**, and **generated SDKs**.
- Reduced regressions through **linting**, **breaking-change checks**, and **contract tests**.

Core Principles

- **API-First**: Author and review API contracts **before** implementation (OpenAPI for REST, schema for GraphQL).
- **Consistency**: Shared conventions for naming, versioning, errors, pagination, authentication, and headers.
- **Backward Compatibility**: Prefer additive changes; deprecate before removing; publish sunset timelines.
- **Observability**: Tracing (OpenTelemetry), structured JSON logs with correlation IDs, per-endpoint/resolver metrics.
- **Security**: OIDC/JWT, scopes/roles, least privilege, strict validation, deny-by-default CORS, rate limiting.
- **Performance & Resilience**: Caching, idempotency keys, retries with backoff, circuit breakers, bulkheads, and efficient pagination.

API-First Process

Lifecycle

1. **Model** domain resources (REST) and entities/relations (GraphQL).
2. **Design** contracts: `openapi.yaml` and `schema.graphqls`.
3. **Mock** early; iterate with frontend consumers.
4. **Validate** in CI (lint, breaking-change checks, examples, security).
5. **Implement** behind contracts; enforce compatibility via contract tests.
6. **Publish** dev portal pages and SDKs; document changelogs.
7. **Operate** with SLOs, error budgets, and deprecation timelines.

Artifacts

- REST: `openapi.yaml` (+ error catalog, examples, deprecation log).
- GraphQL: `schema.graphqls` (+ directives for auth, deprecation notes, persisted queries list).

REST API Design Requirements

3.1 Resource Modeling & URLs

- Plural nouns for collections; singular for instances: `/v1/customers/{customerId}/orders/{orderId}`
- Avoid verbs in paths; use HTTP methods for actions.
- Use natural subresources; otherwise link via IDs.

3.2 Methods & Idempotency

- Use `GET`, `POST`, `PUT`, `PATCH`, `DELETE` with standard semantics.
- **Idempotency-Key**: Required for POST actions with side effects (recommend 24h scope per operation+actor).

3.3 Status Codes

Use standard HTTP status codes consistently across all REST endpoints:

- **2xx** for success (`200 OK`, `201 Created`, `202 Accepted`, `204 No Content`).
- **3xx** for redirects (`301 Moved Permanently`, `302 Found`, `304 Not Modified`).
- **4xx** for client errors (`400 Bad Request`, `401 Unauthorized`, `403 Forbidden`, `404 Not Found`, `409 Conflict`, `422 Unprocessable Entity`, `429 Too Many Requests`).
- **5xx** for server errors (`500 Internal Server Error`, `502 Bad Gateway`, `503 Service Unavailable`, `504 Gateway Timeout`).

Each status code should align with its semantic meaning (e.g., `201` for resource creation, `202` for async processing, `409` for state conflicts).
Error responses must use the standardized envelope described in REST & GraphQL APIs | 3.6 Error Envelope (uniform).

> **For detailed mappings, GraphQL-specific patterns, and recommended headers, see:** REST & GraphQL APIs | REST + GraphQL Status & Error Mapping

3.4 Pagination, Filtering, Sorting

- Offset/limit for simple lists; **cursor** for large/volatile datasets.
- Query params for filters (`?status=active`) and sorts (`?sort=-createdAt,name`).
- Multi-value filters allowed: `?tag=blue&tag=large`.

3.5 Caching

- `ETag` / `If-None-Match`, `Last-Modified` / `If-Modified-Since`.
- `Cache-Control` on GET when safe; leverage CDN for public cacheable endpoints.
- Consider `Prefer: return=minimal` for large payloads.

3.6 Error Envelope (uniform)

Use an RFC7807-inspired envelope across all REST services:

```
{
  "error": {
    "type": "https://errors.example.com/not-found",
    "title": "Resource Not Found",
    "status": 404,
    "code": "CUSTOMER_NOT_FOUND",
    "detail": "Customer 91e3... does not exist",
    "instance": "traceId-12345",
    "errors": [{ "field": "customerId", "message": "Invalid UUID" }]
  }
}
```

3.7 Security

- OAuth2/OIDC with JWT; scopes per resource/action.
- mTLS for service-to-service when applicable.
- Strict CORS (deny-by-default); allow exact origins/headers/methods.

GraphQL API Design Requirements

4.1 Schema Modeling

- Keep mutations focused; use input types for complex args.
- Use Relay-style connections for pagination (`edges` , `pageInfo`).
- Prefer nullability transitions (add as nullable → migrate → enforce non-null when safe).

4.2 Error Handling & Result Shapes

- Most application errors return `200 OK` with `errors[]` ; include `extensions.code` .
- Recommended mutation pattern with `userErrors` :

```
type Mutation {
  createCustomer(input: CreateCustomerInput!): CreateCustomerPayload!
}

type CreateCustomerPayload {
  customer: Customer
  userErrors: [UserError!]!
}

type UserError {
  field: String
  message: String!
  code: String
}
```

- Common `extensions.code` : `BAD_USER_INPUT` , `UNAUTHENTICATED` , `FORBIDDEN` , `NOT_FOUND` , `CONFLICT` , `PRECONDITION_FAILED` , `RATE_LIMITED` , `INTERNAL_SERVER_ERROR` .

> **Transport vs application errors**: Transport-level failures (missing/invalid token, gateway timeouts) should use HTTP error codes; resolver/business errors remain `200` . See §7.

4.3 Performance

- **DataLoader** per request for batching & caching.
- Enforce max query depth/complexity; consider persisted queries & allow GET for CDN caching (`304` friendly).
- Avoid N+1 and deep nesting in resolvers.

4.4 Authorization

- Field-level auth when needed; enforce at resolvers (server-side), never only in the client.
- Map auth failures to `UNAUTHENTICATED` / `FORBIDDEN` in `extensions.code`.

4.5 Schema Evolution

- Deprecate fields/types with reasons:

```
type Customer {
  legacyId: ID @deprecated(reason: "Use id")
}
```

- For breaking changes, route via the new schema/version behind the gateway if unavoidable.

OpenAPI Integration (REST)

- **Source of truth**: `openapi.yaml` at repo root with tags, reusable `components`, examples, and security schemes.
- **Docs**: Render with Swagger UI / Redoc; publish to dev portal.
- **Validation**: Lint with Spectral; detect breaking changes with `openapi-diff`.
- **SDKs**: Generate TypeScript and Java clients; publish to internal registries.

Minimal skeleton:

```
openapi: 3.0.3
info: { title: Customers API, version: 1.0.0 }
servers: [ { url: https://api.example.com/v1 } ]
paths:
  /customers:
    get:
      tags: [Customers]
      summary: List customers
      parameters:
        - $ref: '#/components/parameters/limit'
        - $ref: '#/components/parameters/offset'
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema: { $ref: '#/components/schemas/CustomerList' }
components:
  parameters:
    limit: { name: limit, in: query, schema: { type: integer, minimum: 1, maximum: 100 } }
    offset: { name: offset, in: query, schema: { type: integer, minimum: 0 } }
  schemas:
    CustomerList:
      type: object
      properties:
        items: { type: array, items: { $ref: '#/components/schemas/Customer' } }
        totalCount: { type: integer, minimum: 0 }
```

Framework-Specific Guidance

6.1 Spring Boot

- **OpenAPI**: Use `springdoc-openapi` to generate and serve docs; configure OAuth2 security schemes.
- **GraphQL**: Use `spring-graphql`. Place schema at `src/main/resources/graphql/schema.graphqls`.

- **Security**: Spring Security 6; JWT resource server (`.oauth2ResourceServer(jwt())`), method security (`@PreAuthorize`).
- **Observability**: Micrometer + OpenTelemetry exporter; include `traceId` / `spanId` in MDC/logs.
- **Validation**: Jakarta Bean Validation and global `@ControllerAdvice` / `DataFetcherExceptionResolver` to normalize errors.

REST example (controller and security)

```java
@RestController
@RequestMapping("/v1/customers")
class CustomerController {
  private final CustomerService service;

  @GetMapping
  public ResponseEntity<CustomerList> list(
      @RequestParam(defaultValue = "50") int limit,
      @RequestParam(defaultValue = "0") int offset) {
    return ResponseEntity.ok(service.list(limit, offset));
  }

  @PostMapping
  @PreAuthorize("hasAuthority('SCOPE_customers:write')")
  public ResponseEntity<Customer> create(@Valid @RequestBody CreateCustomerInput input) {
    var created = service.create(input);
    return ResponseEntity.status(HttpStatus.CREATED).body(created);
  }
}

@Bean
SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
  http.authorizeHttpRequests(auth -> auth
        .requestMatchers(HttpMethod.GET, "/v1/customers/**").hasAuthority("SCOPE_customers:read")
        .anyRequest().authenticated())
      .oauth2ResourceServer(oauth -> oauth.jwt());
  return http.build();
}
```

GraphQL error mapping (centralized)

```java
@Component
class GraphQLExceptionResolver implements DataFetcherExceptionResolver {
  @Override
  public Mono<List<GraphQLError>> resolveException(Throwable ex, DataFetchingEnvironment env) {
    if (ex instanceof ConstraintViolationException cve) {
      var error = GraphqlErrorBuilder.newError(env)
        .message("Invalid input")
        .errorType(ErrorType.ValidationError)
        .extensions(Map.of(
          "code", "BAD_USER_INPUT",
          "fieldErrors", cve.getConstraintViolations().stream()
            .map(v -> Map.of("field", v.getPropertyPath().toString(), "message", v.getMessage()))
            .toList()
        ))
        .build();
      return Mono.just(List.of(error));
    }
    var error = GraphqlErrorBuilder.newError(env)
      .message("Unexpected error occurred")
      .extensions(Map.of("code", "INTERNAL_SERVER_ERROR"))
      .build();
    return Mono.just(List.of(error));
  }
}
```

DataLoader registry

```java
@Bean
```

```
public DataLoaderRegistry dataLoaderRegistry(OrdersBatchLoader ordersLoader) {
  var registry = new DataLoaderRegistry();
  registry.register("ordersByCustomer", DataLoader.newDataLoader(ordersLoader));
  return registry;
}
```

6.2 Next.js (App Router)

- **REST & GraphQL clients**:
  - Generate **TypeScript** clients from OpenAPI and GraphQL Codegen from schema/operations.
- **Route handlers** (`app/api/.../route.ts`):
  - Use for lightweight APIs/proxies; normalize transport errors (401/403/429/5xx).
- **Caching**:
  - `cache: 'no-store'` for personalized or auth'd fetches; use `revalidate` /ISR for public cacheable GET.
  - Persisted GraphQL queries via GET enable `ETag` / `304` at the edge/CDN.
- **Auth**:
  - Middleware to attach/validate tokens; never expose secrets in client components.

Proxy example (edge runtime)

```
export const runtime = 'edge';

export async function GET(req: Request) {
  const url = new URL(req.url);
  const limit = url.searchParams.get('limit') ?? '50';

  const res = await fetch(`${process.env.API_BASE}/v1/customers?limit=${limit}`, {
    headers: { Authorization: `Bearer ${await getAccessToken()}` },
    cache: 'no-store'
  });

  return new Response(await res.text(), { status: res.status, headers: res.headers });
}
```

REST + GraphQL Status & Error Mapping

> **Guiding rule**: In GraphQL, **transport-layer** failures (auth, gateway, rate limit, timeouts) should return appropriate **HTTP status codes**. **Application-layer** (resolver/business) errors typically return `200 OK` with entries in `errors[]` and a meaningful `extensions.code` .

HTTP Code	Name	REST Use Case	GraphQL (Transport Layer)	GraphQL (Application Layer) – recommended `extensions.code`	Recommended Headers / Notes

200	OK	Successful GET/PUT/PATCH/POST returning data.	Successful GraphQL response (may contain `data` and/or `errors`).	`BAD_USER_INPUT`, `INTERNAL_SERVER_ERROR`, domain codes (e.g., `CUSTOMER_NOT_FOUND`).	Use caching headers for safe queries.
201	Created	Resource created; include `Location`.	Rare (mutations usually return 200 with payload).	Use payload + `userErrors` for validation issues.	`Location` with new resource URI (REST).
202	Accepted	Async job started; poll status.	Often still 200 with the job handle in the payload.	Payload includes job status (`PENDING`, `IN_PROGRESS`).	Provide job ID or status endpoint.
204	No Content	Success without body (DELETE).	Avoid—GraphQL expects a body.	Use payload flag (e.g., `deleted: true`).	Prefer explicit confirmations.
301	Moved Permanently	Resource moved.	Rarely used for GraphQL endpoints.	N/A	If used, set `Location`.
302	Found	Temporary redirect.	Generally avoid for GraphQL.	N/A	Gateway only.
304	Not Modified	Conditional GET with `ETag` / `Last-Modified`.	GET persisted queries can use `ETag` and `If-None-Match`.	N/A	Enable CDN/edge caching.
400	Bad Request	Malformed request/validation.	Syntax/validation errors of the GraphQL request can result in a `400`.	Domain validation → `200` + `BAD_USER_INPUT`.	Use consistent error envelopes (REST).

401	Unauthorized	Missing/invalid auth.	`401` from gateway/resource server.	Resolver-level auth → `200` + `UNAUTHENTICATED`.	`WWW-Authenticate` header.
403	Forbidden	Authenticated but no permission.	`403` at gateway/policy layer.	`FORBIDDEN` at resolver.	Include required scopes in detail.
404	Not Found	Resource/endpoint missing.	Unknown GraphQL endpoint → `404`.	Missing entity → `200`, `data.field = null` + `NOT_FOUND` (optional).	Keep the REST envelope consistent.
405	Method Not Allowed	HTTP method not supported.	Rare for GraphQL (POST/GET).	N/A	Include `Allow` header.
409	Conflict	State conflict or version mismatch.	Use transport `409` if the operation cannot start.	`CONFLICT` or domain-specific (e.g., `VERSION_CONFLICT`).	Provide remediation hints.
410	Gone	Resource permanently removed.	Rare; prefer app-level handling.	Explain deprecation in the payload.	Pair with `Sunset` headers in REST.
412	Precondition Failed	`If-Match` / `If-Unmodified-Since` failed.	Typically handled in REST.	`PRECONDITION_FAILED` in extensions.	Favor ETag concurrency in REST.
415	Unsupported Media Type	Wrong `Content-Type`.	`415` if non-JSON or invalid GraphQL type.	N/A	Enforce `application/json`.
422	Unprocessable Entity	Semantic validation failure.	Schema-valid but semantically invalid → often `200` + `BAD_USER_INPUT`.	`BAD_USER_INPUT` with field errors.	Include per-field details.

429	Too Many Requests	Rate limit exceeded.	`429` at gateway; GraphQL not executed.	Resolver throttling → `200` + `RATE_LIMITED`.	`Retry-After` (seconds or HTTP date).
500	Internal Server Error	Generic server error.	Server/runtime failure → `500`.	Resolver exception → `200` + `INTERNAL_SERVER_ERROR`.	Don't leak internals; log traceId.
502	Bad Gateway	Upstream invalid response.	Gateway/proxy to GraphQL returns `502`.	N/A	Trace upstream.
503	Service Unavailable	Maintenance/overload.	Gateway returns `503`.	Resolver-level overload → optionally `200` with error; prefer `503` if not processed.	`Retry-After`; circuit breakers.
504	Gateway Timeout	Upstream timeout.	Gateway returns `504`.	N/A	Tune timeouts; propagate correlation IDs.

GraphQL error payload examples:

- **Validation**

```
{
  "data": { "createCustomer": null },
  "errors": [{
    "message": "Email is invalid",
    "path": ["createCustomer"],
    "extensions": {
      "code": "BAD_USER_INPUT",
      "fieldErrors": [{ "field": "email", "message": "Must be a valid email address" }],
      "traceId": "abc123"
    }
  }]
}
```

- **Auth**

```
{
  "data": null,
  "errors": [{
    "message": "Authentication token missing",
    "extensions": { "code": "UNAUTHENTICATED", "traceId": "abc123" }
  }]
}
```

- **Not Found (nullable field)**

```
{
  "data": { "customer": null },
```

```
  "errors": [{
    "message": "Customer not found",
    "path": ["customer"],
    "extensions": { "code": "NOT_FOUND", "traceId": "abc123" }
  }]
}
```

- **Internal Resolver Error**

```
{
  "data": null,
  "errors": [{
    "message": "Unexpected error occurred",
    "extensions": { "code": "INTERNAL_SERVER_ERROR", "traceId":
"abc123" }
  }]
}
```

When to Use REST vs GraphQL

- **REST**: Stable resource boundaries; cacheable GETs; external/public APIs; strong OpenAPI ecosystem.
- **GraphQL**: Complex UIs; avoid over-/under-fetching; rich relationships; field-level auth; internal/front-end heavy apps.

Hybrid: Provide REST externally and GraphQL internally; unify via a gateway with common auth, observability, and rate limits.

Gateway & Edge

- Central API gateway for **auth, rate limiting, request/response size caps, header normalization, compression, error translation**, and routing (REST & GraphQL).
- **Caching**: CDN for REST GET; **persisted GraphQL queries** (GET) for edge caching and `304` handling.

Testing, Governance & CI/CD

- **Contract tests**: REST (Pact), GraphQL (breaking-change checks, schema linting, query whitelists).
- **Integration**: Testcontainers in Spring.
- **Performance**: k6/Gatling; track P95/P99 and error budgets.
- **Security**: Scope enforcement tests; ZAP fuzzing; JWT/CSRF/CORS checks.
- **Governance**: Spectral (OpenAPI), eslint-plugin-graphql; block merges on breaking changes or missing examples/security.
- **Automation**: Generate & publish SDKs on release; dev portal updates; deprecation trackers.

Retry Policy

Purpose

Ensure resilience and graceful degradation when backend services or upstream dependencies are temporarily unavailable.

Scope

Applies to:

- **Client-side SDKs** (generated from OpenAPI or GraphQL schema).
- **Service-to-service calls** within Spring Boot.
- **Next.js API routes** acting as proxies.

Policy

- **Trigger conditions**:
  - Any **5xx HTTP status code** (`500`, `502`, `503`, `504`).
  - **Connection timeouts** or transient network failures.
- **Excluded**:
  - **4xx client errors** (do not retry).
  - **Validation/business logic errors** (GraphQL `BAD_USER_INPUT`, REST `422`).

Strategy

- **Max retries**: 3 attempts (initial + 2 retries).
- **Backoff**: Exponential with jitter:
  - Initial delay: **500 ms**.
  - Multiply by a factor of **2** for each retry.
  - Add random jitter ±20% to avoid thundering herd.
- **Cap**: Maximum delay per attempt = **5 seconds**.
- **Timeout**: Overall request timeout = **15 seconds** (including retries).
- **Circuit breaker**:
  - Open after **5 consecutive failures**.
  - Half-open after **30 seconds**.
- **Idempotency**:
  - Only retry **idempotent operations** (GET, HEAD, PUT, DELETE).
  - For POST, require `Idempotency-Key` header.

Example (Spring Boot using Resilience4j)

```java
@Bean
public RetryConfig retryConfig() {
  return RetryConfig.custom()
    .maxAttempts(3)
    .waitDuration(Duration.ofMillis(500))
    .intervalFunction(IntervalFunction.ofExponentialBackoff(500, 2.0))
    .retryExceptions(IOException.class, TimeoutException.class)
    .ignoreExceptions(HttpClientErrorException.class)
    .build();
}

@Bean
public RetryRegistry retryRegistry() {
  return RetryRegistry.of(retryConfig());
}
```

Example (Next.js fetch wrapper)

```ts
async function fetchWithRetry(url: string, options: RequestInit,
retries = 2, delay = 500): Promise<Response> {
  try {
    const res = await fetch(url, options);
    if (res.ok || (res.status >= 400 && res.status < 500)) return res;
    throw new Error(`Retryable status: ${res.status}`);
  } catch (err) {
    if (retries <= 0) throw err;
    await new Promise(r => setTimeout(r, delay + Math.random() * delay
* 0.2));
    return fetchWithRetry(url, options, retries - 1, delay * 2);
  }
}
```

Observability

- Log each retry attempt with `traceId`, attempt count, and delay.
- Emit metrics: `retry_attempts_total`, `retry_failures_total`.