FTI Publishers

Journal of Intelligent Systems and Applied Technology

REVIEW ARTICLE

OPEN ACCESS

Hypertext Transfer Protocol Status Codes in Application Programming Interfaces and Gateways: A Review of Standards and Practice

Luiz Henrique de Sousa Ribas*

Principal Cloud & Platform Architect, WP Engine, Dublin, Ireland

ABSTRACT

Every response on the web carries a three-digit status code, and in application programming interfaces that number is read far more often by software than by people. Monitoring systems, caches, retry policies, circuit breakers and gateways all act on it, so a wrongly chosen code causes failures well beyond the application that sent it. This article reviews the status code system from the point of view of the developers and architects who build and operate such interfaces. It covers the current standard and the official registry, which held 64 entries in October 2026; the meaning of each of the five classes; the accompanying headers and the standard error format; and the way status codes are produced by application frameworks and gateways, which the standards largely leave to vendors. The review draws on the standards documents, on the published documentation of widely used gateways and cloud platforms, and on a case study in which every status code discussed was produced and checked by automated tests in a reference implementation built with a current Java framework. The case study shows that the framework omits headers required by the standard in two situations, and that validation and rate-limiting errors need explicit configuration to produce the recommended codes. Widely used gateways also depart from the standard's recommendations, for example by answering a missing credential with 403 rather than 401. The article closes with a catalogue of recurring mistakes and a four-question procedure for choosing a status code.

Keywords: application programming interfaces; HTTP; status codes; API gateways; web services; software architecture

***Corresponding author:** Luiz Henrique de Sousa Ribas, Principal Cloud & Platform Architect, WP Engine, Dublin, Ireland.

Citation: de Sousa Ribas LH. Hypertext Transfer Protocol Status Codes in Application Programming Interfaces and Gateways: A Review of Standards and Practice. *J Intell Syst Appl Technol*. 2026. Available from: https://ftipublishers.com/jisat

Received: 30 October 2026 · **Published:** 30 October 2026

Copyright: © 2026 The Author. Published by FTI Publishers UAB. This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC BY 4.0), which permits unrestricted use, distribution and reproduction in any medium, provided the original author and source are credited.

1. Introduction

The Hypertext Transfer Protocol (HTTP) is the protocol through which most software systems now talk to each other. Public web services, internal microservices, mobile applications and the platforms that connect them all exchange HTTP requests and responses. Each response begins with a status code: a three-digit integer that states, before anything else, how the request was handled [1].

For a person reading a web page, the status code is almost invisible. For an application programming interface (API), it is the most important part of the response. A client program decides whether to parse the body, retry the request, ask the user to log in or give up, based on that number. The infrastructure around the API depends on it even more. A content delivery network decides what it may cache; a monitoring system decides whether to raise an alert; a gateway decides whether to retry, whether to open a circuit breaker and which team's dashboard turns red. In each case the decision is taken from the number alone, without reading the body.

This makes the choice of status code an architectural decision rather than a matter of style. An API that answers every request with 200 OK and puts the real outcome in the body hides its failures from every tool on the path. An API that answers invalid input with 500 Internal Server Error blames itself for its clients' mistakes, invites pointless retries and wakes the on-call engineer at night. Empirical studies of public APIs show that such problems are common: in an analysis of 500 public REST APIs, only 64.6% were found to use adequate status codes, 2.8% reported errors through custom codes in the body, and 32.6% gave no information about the codes they use [2]. Earlier work on the detection of REST antipatterns [3] and on the design quality of cloud APIs [4] reported similar inconsistencies, and a Delphi study with industry experts ranked correct status code use among the design rules with the greatest effect on API quality [5].

The standards themselves are precise but scattered. The core semantics are defined in RFC 9110 [1], but the official registry maintained by the Internet Assigned Numbers Authority (IANA) [6] collects codes from more than ten specifications. The error format most APIs should use is defined separately in RFC 9457 [7]. Guidance on how new protocols should use HTTP lives in yet another document [8]. The behaviour of frameworks and gateways, which in practice produce a large share of the status codes clients receive, is documented only by each vendor.

This article brings these sources together. Its contributions are:

1. A structured review of the status code system as standardised in 2026, including the official registry and the specifications behind each code.
2. A class-by-class account of the codes most relevant to APIs, with the headers the standard requires or recommends alongside them.
3. An analysis of how API gateways and cloud platforms produce status codes, including documented departures from the standard's recommendations.
4. A case study of the status codes produced automatically by a current Java framework, Quarkus, based on a reference implementation in which every behaviour was verified by automated tests.
5. A catalogue of recurring mistakes and a simple decision procedure for choosing a status code.

The article is written for developers and architects who design, build and operate APIs, and for educators who teach these topics. Section 2 describes the review method. Section 3 presents the structure of the status code system and its registry. Section 4 reviews each class. Section 5 covers error representation and accompanying headers. Section 6 examines status codes at the API gateway. Section 7 presents the framework case study. Section 8 discusses recurring mistakes, offers a decision procedure and states the limitations of the review. Section 9 concludes.

2. Review method

This is a narrative review of standards and practice. It does not report a systematic search of the research literature, so the PRISMA guideline does not apply. Its aim is to describe what the standards require, what widely used products actually do, and where the two differ.

Sources were selected in four groups, summarised in Table 1. The first group is normative: the Internet Engineering Task Force (IETF) specifications that define status codes and their accompanying headers, and the IANA registry that lists them. Where a specification has been replaced, only the current version is cited; RFC 9110, for example, replaces RFC 7231 and the earlier RFC 2616. The second group is the published documentation of widely deployed gateways, load balancers and content delivery networks, chosen because they sit in front of a large share of public APIs. The third group is industry API design guidelines and the peer-reviewed empirical literature on API design quality. The fourth is the documentation of the Java framework used in the case study.

Table 1. Source groups used in this review.

Group	Sources	Used for
Standards	RFC 9110 [1]; IANA registry [6]; RFC 9457 [7]; RFC 9205 [8]; RFCs 6585, 7538, 8297, 8470, 7725, 4918, 5842, 6455, 7617 [9]–[17]	Meaning of each code; required and recommended headers; error format
Platform documentation	Amazon API Gateway [18]; AWS Application Load Balancer [19]; Cloudflare [20]; GitHub REST API [21]	Status codes produced by gateways and platforms
Design guidelines and empirical studies	Neumann et al. [2]; Palma et al. [3]; Petrillo et al. [4]; Kotstein and Bogner [5]; Fielding [22]; industry guidelines [23]–[25]	Observed practice and recurring problems
Framework documentation and case study	Quarkus guides [26], [27]; Jakarta RESTful Web Services [28]; reference implementation [29]	Status codes produced automatically by a framework

Within each group, three criteria were applied. A source had to be current, which excluded specifications that have been replaced and documentation for discontinued products. It had to be authoritative for the behaviour it describes, which meant the standard itself for normative statements and the vendor's own documentation for product behaviour, rather than third-party summaries. And it had to concern status codes as seen by APIs, which excluded material that deals only with browser rendering of error pages. Where a vendor's documented behaviour differs from the standard, both are reported and the difference is stated explicitly rather than resolved in favour of either.

The state of the registry is reported as observed in October 2026, when it had last been updated on 15 September 2025 [6]. The framework case study is described in Section 7. All examples were run against Quarkus 3.37.4 on OpenJDK 21, and the observed behaviour was compared with the requirements of RFC 9110.

3. The status code system

3.1 Structure and the class rule

A status code is a three-digit integer. The first digit defines its class, and the remaining two digits carry no categorisation [1]. There are five classes, shown in Table 2. RFC 9110 requires a client to understand the class of any status code from its first digit, and to treat an unrecognised code as equivalent to the x00 code of its class [1]. A client that receives an unknown 524 therefore handles it as a 500, and a client that receives an unknown 451 handles it as a 400. This rule is what allows new codes to be added without breaking existing clients, and it is also what allows vendors to use unregistered codes without completely confusing clients (Section 3.3).

Table 2. The five classes of status code.

Class	Name	Meaning	Who should act
1xx	Informational	Interim response; the final answer follows	Usually no action
2xx	Successful	The request was received, understood and accepted	Client continues
3xx	Redirection	Further action is needed to complete the request	Client follows the redirect
4xx	Client error	The request contains an error or cannot be fulfilled as sent	Client must change the request
5xx	Server error	The server failed to fulfil an apparently valid request	Server side must fix; client may retry later

The status line of an HTTP/1.1 response also carries a reason phrase, such as "Not Found". The reason phrase is purely informative: RFC 9112 states that a client should ignore it because it is not a reliable channel for information [30], and HTTP/2 and HTTP/3 do not transmit it at all. Programs must therefore never parse it. This matters in practice because servers still emit outdated phrases: in the case study, the framework answered "413 Request Entity Too Large" and "422

Unprocessable Entity", the names from RFC 2616 and WebDAV, whereas RFC 9110 now calls these codes Content Too Large and Unprocessable Content (Section 7).

3.2 The registry and its evolution

The authoritative list of status codes is the IANA HTTP Status Code Registry [6]. In October 2026 it held 64 entries: 5 informational, 10 successful, 9 redirection, 29 client error and 11 server error codes. Not every entry is in active use. Two are marked as unused: 306, a former proxy code, and 418, reserved because of its widespread use as a joke after the April Fools' "Hyper Text Coffee Pot Control Protocol" [31]. One, 510 Not Extended, is obsolete, and one, 104 Upload Resumption Supported, is a temporary registration linked to an IETF draft on resumable uploads, with an expiry date of 13 November 2026.

RFC 9110 defines 46 of the 64 codes. The remaining 18 come from other specifications, summarised in Table 3. This distribution has practical consequences: several codes that APIs use every day, notably 429 Too Many Requests, are not defined in the core document, and developers who read only RFC 9110 will not find them.

Table 3. Registered status codes defined outside RFC 9110.

Specification	Codes	Purpose
RFC 6585 [9]	428, 429, 431, 511	Preconditions, rate limiting, oversized headers, captive portals
RFC 8297 [11]	103	Early Hints for preloading resources
RFC 8470 [12]	425	Replay protection for early data in TLS 1.3
RFC 7725 [13]	451	Resources withheld for legal reasons
RFC 4918 [14]	207, 423, 424, 507	WebDAV multi-status responses, locking and storage
RFC 5842 [15]	208, 508	WebDAV bindings and loop detection
RFC 2518 (WebDAV, original)	102	Processing; removed from WebDAV by RFC 4918
RFC 3229	226	Delta encoding
RFC 2295	506	Transparent content negotiation
RFC 2774	510	HTTP extensions (obsolete)
IETF draft (temporary)	104	Resumable uploads

The registry also records how meanings change over time. 422 started life in WebDAV as Unprocessable Entity and was adopted into the core standard as Unprocessable Content by RFC 9110. 413 was renamed from Payload Too Large to Content Too Large. 308 Permanent Redirect was first defined separately in RFC 7538 [10] and is now part of RFC 9110. The registry is therefore a living document, and any reference table of codes, including Table 3, should be checked against the live registry before it is relied upon.

3.3 Unregistered codes in practice

Several widely deployed products use codes that are not in the registry. The best known are the 52x codes of Cloudflare, which describe problems between its edge and the customer's origin server: 520 for an unknown error from the origin, 521 when the origin is down, 522 when the connection timed out, 523 when the origin cannot be reached, 524 when the origin took too long to answer, and 525 and 526 for TLS handshake and certificate failures [20]. The AWS Application Load Balancer uses 460 when the client closed the connection before the load balancer's idle timeout elapsed, 463 when the X-Forwarded-For header lists too many addresses, 464 for a protocol mismatch with the target group, and 561 and 562 for failures in its built-in authentication [19]. The nginx web server logs 499 when the client closes the connection before the response is sent.

These codes are useful for diagnosis, but they are not part of the standard, and clients cannot be expected to know them. The class rule in Section 3.1 is what keeps them from causing harm: a client that knows nothing about 524 still treats it as a server error, which is correct. The lesson for API designers is to use only registered codes in their own APIs and, where finer detail is needed, to put it in the response body (Section 5), not in a new number.

3.4 Status codes across HTTP versions

The meaning of each status code is independent of the version of HTTP that carries it. RFC 9110 defines the semantics once, and the version-specific documents define only how they are transmitted [1], [30]. HTTP/1.1 sends the code in a text status line together with the reason phrase. HTTP/2 and HTTP/3 send it as a pseudo-header field that contains only the three digits, with no reason phrase. An API that behaves correctly over HTTP/1.1 therefore behaves the same way over the newer versions, provided that its clients never depended on the reason phrase.

Two practical differences remain. First, intermediaries often translate between versions: a content delivery network may accept HTTP/3 from the client and speak HTTP/1.1 to the origin. Any information carried only in the reason phrase is lost at that boundary, which is a further reason not to use it. Second, informational responses are handled differently in practice. 101 Switching Protocols belongs to HTTP/1.1, because HTTP/2 and HTTP/3 establish WebSocket connections through a different mechanism, and gateways that translate between versions must handle the upgrade explicitly. For API designers these differences are small, but for those who build or configure gateways they explain a class of failures that appear only when a new protocol version is enabled at the edge.

4. Class-by-class review

4.1 1xx Informational

Informational responses are interim: they are followed by a final response to the same request. Three are relevant to APIs. 100 Continue lets a client that is about to send a large body send only the headers first, with Expect: 100-continue, so that the server can refuse the request, for example with 401 or 413, before the body is transmitted [1]. 101 Switching Protocols is the answer to an Upgrade request and is the first response in every WebSocket connection [16]. 103 Early Hints lets a server

send Link headers for resources the client will need while the final response is still being prepared [11].

Most API developers never write code that produces a 1xx response; servers and frameworks handle them. Their main practical relevance is in gateways and proxies, which must forward or answer them correctly. A gateway that does not support the WebSocket upgrade, for example, will break every real-time feature behind it.

4.2 2xx Successful

The successful class is where precision is most often neglected, because any 2xx code "works" in a browser. For APIs, the distinctions carry information that clients and tools use.

200 OK is the general success code. Its meaning depends on the method: for GET, the body is the requested resource; for POST, the body describes the result of the action. 201 Created states that a new resource was created; the server should send a Location header that identifies it [1]. 202 Accepted states that the request was accepted for processing but that processing has not finished. It is the correct answer for long-running jobs, and the response should tell the client where to check progress. 204 No Content states that the request succeeded and there is deliberately nothing to return, which is typical of DELETE and of updates that return no representation. 206 Partial Content answers range requests, which allow large downloads to be resumed.

The choice between these codes is most often debated for updates. RFC 9110 states that a successful PUT that creates a new resource must be answered with 201, and that a successful PUT that modifies an existing resource should be answered with 200 or 204 [1]. The choice between 200 and 204 then depends on whether the API returns the updated representation. Returning it saves the client a second request but costs bandwidth; returning nothing is simpler and is common for idempotent updates. Either is correct, provided the API is consistent and documents its choice. The same reasoning applies to 202: an API that accepts a long-running job should return 202 immediately, with a Location header or a link in the body pointing to a status resource, and that status resource should later report the outcome using ordinary codes. Answering 200 after a long synchronous wait, by contrast, ties up connections, triggers gateway timeouts and turns a slow operation into a 504 (Section 6).

The remaining successful codes (203, 205, 207, 208 and 226) are rarely needed in general-purpose APIs. 207 Multi-Status is the exception in APIs that perform batch operations, where a single request may succeed for some items and fail for others; even there, many APIs now prefer 200 with a structured body.

4.3 3xx Redirection

Redirects matter to APIs for two reasons: resources move, and the HTTP method must be preserved or changed deliberately. RFC 9110 distinguishes two properties of a redirect: whether it is permanent or temporary, and whether the client may change the method [1]. 301 Moved Permanently and 302 Found are the historical codes. For historical reasons, user agents may change a POST into a GET when following them, so they are unreliable for any method other than GET. 307 Temporary Redirect and 308 Permanent Redirect [10] were introduced to remove this ambiguity: both require the client to repeat the same method with the same body. 303 See Other explicitly tells the client to fetch the result with GET, which is the correct answer after a form submission or a POST that created something viewable elsewhere.

Redirects deserve particular care in APIs because API clients, unlike browsers, do not always follow them. Many HTTP client libraries follow redirects for GET by default but not for other methods, and some do not follow them at all. A common example is the redirect from HTTP to HTTPS: a website can safely answer plain-HTTP requests with 301, but an API that does the same may silently turn a client's POST into a GET, or cause the client to send credentials over an unencrypted connection before being redirected. For APIs it is generally safer to refuse plain-HTTP requests, or to use 308, than to rely on a 301. Redirects are also sometimes used to retire old API versions. Here 308 preserves the method and body, but a client that does not follow redirects will see only the 3xx code, so retired versions are often better answered with 410 and a Problem Details body that names the replacement.

304 Not Modified is not a redirect in the everyday sense. It answers a conditional request, sent with If-None-Match or If-Modified-Since, and tells the client that its cached copy is still valid [1], [32]. In APIs it saves bandwidth on large or frequently polled resources, and together with 412 Precondition Failed (Section 4.4) it forms the basis of optimistic concurrency control.

4.4 4xx Client error

The client error class is the largest, with 29 registered codes, and the one in which API designers make most of their choices. A 4xx code tells the client that the request must change before it can succeed, and tells monitoring systems that the problem lies on the client's side. It also tells retry logic that repeating the same request will not help, with the notable exceptions of 408, 425 and 429.

Syntax and validation. 400 Bad Request is the general answer for a request the server cannot or will not process because of a client error, such as malformed JSON or an unparseable parameter. 422 Unprocessable Content is more specific: the request is syntactically correct, but its content is invalid, for example an order for minus three items [1]. Using 422 for validation failures gives the client a more precise signal, but 400 for both cases remains acceptable, and several industry guidelines prefer it [23], [25].

Authentication and authorisation. 401 Unauthorized means that the request lacks valid credentials. Despite its name, it is about authentication, not authorisation. RFC 9110 requires a 401 response to include a WWW-Authenticate header that tells the client how to authenticate [1]. 403 Forbidden means that the server understood the request and refuses to fulfil it, typically because the authenticated client lacks permission. Swapping the two is one of the most common mistakes in practice: a client that receives 403 for a missing token gives up when it should log in, and a client that receives 401 for a missing permission logs in again and again without success. A related design choice is to answer 404 instead of 403 for resources whose existence must not be revealed. GitHub's REST API, for example, returns 404 for private repositories to unauthenticated callers, to avoid confirming that they exist [21]. RFC 9110 explicitly allows this practice [1].

Resources and methods. 404 Not Found means that the server has no current representation for the target resource, or is not willing to disclose that one exists. 410 Gone is the stronger, permanent form, useful when an API version or resource has

been retired. 405 Method Not Allowed means that the resource exists but does not support the method; RFC 9110 requires the server to send an Allow header that lists the supported methods [1].

Representations and limits. 406 Not Acceptable and 415 Unsupported Media Type are the two sides of content negotiation: the server cannot produce any format the client accepts, or cannot consume the format the client sent. 413 Content Too Large and 414 URI Too Long reject requests that exceed the server's limits, and 431 Request Header Fields Too Large [9] does the same for headers.

State and concurrency. 409 Conflict reports a clash with the current state of the resource, such as a duplicate username. 412 Precondition Failed reports that a condition sent by the client, typically If-Match with an entity tag, no longer holds because the resource has changed. 428 Precondition Required [9] lets a server insist on such conditions, which protects against the "lost update" problem when several clients edit the same resource.

Retryable client errors. Three codes in this class describe conditions that may resolve on their own, so the general rule that a 4xx response should not be retried unchanged does not apply to them. 408 Request Timeout means that the server did not receive a complete request in time, for example because the client was slow to send the body, and the client may repeat the request on a new connection. 425 Too Early asks the client to repeat a request that was sent as TLS 1.3 early data, once the handshake is complete [12]. 429 Too Many Requests asks the client to wait and then try again. Retry logic in clients and gateways should therefore be written per code, not per class: retrying all 4xx responses wastes resources, and retrying none of them breaks these three cases.

Rate limiting. 429 Too Many Requests [9] tells the client that it has sent too many requests in a given period. It may include a Retry-After header stating how long to wait. In practice, gateways send 429 far more often than applications do (Section 6).

The remaining client error codes either serve specialised protocols (the WebDAV codes 423 and 424 [14]), address transport-level conditions (421 Misdirected Request, 425 Too Early [12]) or are reserved and unused (418 [31]). 402 Payment Required is registered but has no defined semantics, and APIs should not rely on it.

4.5 5xx Server error

A 5xx code states that the server failed to fulfil an apparently valid request. It moves responsibility to the server side, and it is the class that monitoring systems treat as an incident.

500 Internal Server Error is the generic code for unexpected conditions, such as an unhandled exception. Its body must not reveal internal details: stack traces, database errors and file paths help attackers and should be logged on the server, with only a short message and an error identifier returned to the client. 501 Not Implemented means that the server does not support the functionality required, for example a method it does not recognise. 503 Service Unavailable means that the server is temporarily unable to handle the request, because of overload or maintenance; the server may add a Retry-After header to say when the client should try again [1].

502 Bad Gateway and 504 Gateway Timeout are special: they are produced by an intermediary, not by the application. 502 means that the intermediary received an invalid response from the server behind it, or could not connect to it. 504 means that it did not receive a response in time. An application should almost never send them itself. When they appear, they say something about the path between components, which is why they are central to Section 6.

The question of whether a client should retry a failed request depends on both the code and the method. 502, 503 and 504 are usually temporary, and retrying with exponential backoff, respecting Retry-After where present, is reasonable. For non-idempotent methods such as POST, however, the first attempt may have succeeded before the failure was reported, and a retry may duplicate the action. Many payment APIs address this with idempotency keys, which let the server recognise a repeated request.

5. Error representation and accompanying headers

5.1 Problem Details

A status code tells machines what happened; people also need to know why. For many years every API invented its own error format, which forced clients to write different parsing code for each API. RFC 9457, Problem Details for HTTP APIs [7], which replaces RFC 7807, defines a common JSON (and XML) format, sent with the media type application/problem+json. It has five standard members: type, a URI that identifies the kind of problem (defaulting to about:blank, meaning "see the status code"); title, a short summary of that kind of problem; status, a copy of the status code; detail, an explanation specific to this occurrence; and instance, a URI that identifies this occurrence. APIs may add their own members, such as a list of invalid fields.

A typical response for a validation failure looks as follows:

```
HTTP/1.1 422 Unprocessable Content
Content-Type: application/problem+json

{
       "type":    "https://api.example/problems/invalid-
quantity",
  "title": "Invalid order",
  "status": 422,
  "detail": "quantity must be 1 or more, but was -3",
  "instance": "/orders/attempts/9f51bf0f"
}
```

Problem Details complements the status code; it does not replace it. The status member must match the code in the status line, and a Problem Details body sent with 200 OK would repeat the mistake it is meant to prevent. The format has been adopted by several major API guidelines and frameworks, which makes it the sensible default for new APIs [8], [23].

5.2 Headers that complete the status code

Several codes are incomplete without an accompanying header. Table 4 lists the most important cases. Two of these headers are requirements of RFC 9110, not recommendations, and the case study in Section 7 shows that a current framework omits one of them by default.

Table 4. Headers that accompany specific status codes.

Code	Header	Requirement in RFC 9110	Purpose
401	WWW-Authenticate	Required	Tells the client how to authenticate
405	Allow	Required	Lists the methods the resource supports
201	Location	Recommended	Identifies the created resource
3xx (except 304)	Location	Recommended	Gives the redirect target
429, 503	Retry-After	Optional	Tells the client how long to wait
206	Content-Range	Required	Identifies the part of the resource returned
304	ETag, Cache-Control and related	Required if they would have been sent with 200	Updates the cached copy

5.3 Describing and testing status codes

An API's status codes are part of its contract, and they should be described as explicitly as its request and response bodies. API description formats such as the OpenAPI Specification organise each operation's responses by status code, so that every code the operation can return is listed together with its body schema and headers [34]. A description that lists only 200 for an operation that can also return 401, 404 and 422 gives client developers, code generators and testing tools an incomplete picture, and the missing codes will then be discovered in production.

Two practices follow from this. The first is to list, for every operation, the codes produced by the framework and the gateway as well as those produced by the application code, since all of them reach the client (Sections 6 and 7). The second is to test them. Contract tests and integration tests that assert the status code and the required headers of each response catch regressions that unit tests of business logic do not. In the case study described in Section 7, such tests were what revealed that the framework omitted a header required by the standard.

5.4 Security considerations

Status codes can leak information. The difference between 403 and 404 tells an unauthenticated caller that a resource exists, which is why RFC 9110 allows a server to answer 404 for a forbidden resource whose existence must remain hidden [1], and why some public APIs do so [21]. The same reasoning applies to authentication endpoints. If a login request with an unknown username returns a different code, body or response time from one with a known username and a wrong password, an attacker can enumerate valid accounts. Security guidance therefore recommends returning the same generic response in both cases [35].

The ordering of checks has similar effects. A server or gateway that checks whether a route exists before it checks credentials will answer 404 for unknown paths and 401 for known ones, which allows anyone to map the API without logging in. Checking credentials first gives every unauthenticated request the same 401, at the cost of less precise errors for legitimate clients that mistype a path. Section 6.5 shows this trade-off in a concrete gateway design. Finally, as noted in Section 4.5, 5xx responses must not reveal stack traces, database errors or internal addresses. The status code should tell the client that the server failed, and an error identifier in the body should allow the operator to find the details in the server's log.

6. Status codes at the API gateway

6.1 Three origins of a status code

An API gateway is a single entry point placed in front of a set of services. It typically handles routing, authentication, authorisation, rate limiting, TLS termination, request transformation, caching, retries and observability. Common examples include Kong, Amazon API Gateway, Apigee, Azure API Management, gateways based on the Envoy proxy, and the edge platforms of content delivery networks.

From the client's point of view, a response that leaves a gateway can have one of three origins. First, the gateway may create the code itself before any service is involved: 404 when no route matches, 401 or 403 when authentication or authorisation fails at the edge, 413 when the body exceeds a limit, and 429 when a rate limit or quota is exceeded. Second, the gateway may create a code about the service behind it, after trying and failing to obtain a valid response: 502 when the connection is refused or the response is invalid, 503 when no healthy instance is available, and 504 when the service does not answer in time. Third, and most often, the gateway passes the service's own response through unchanged.

These three origins have very different operational meanings. A 404 created by the gateway means that the route configuration is wrong or the client used an unknown path; a 404 passed through from a service means that a specific item does not exist. A 503 created by the gateway means that no instance is healthy; a 503 from the service means that the service itself has declared an overload. Telling them apart is essential for diagnosis (Section 6.4).

6.2 Documented behaviour of gateways and platforms

Table 5 summarises the status codes produced by two widely used AWS services and by Cloudflare's edge, as stated in their documentation. Three observations stand out.

Table 5. Status codes produced by selected gateways and platforms, from their documentation.

Situation	Amazon API Gateway [18]	AWS Application Load Balancer [19]	Cloudflare edge [20]
Missing credentials	403 (MISSING_AUTHENTICATION_TOKEN)	401 (authentication rule)	—
Credentials rejected by an authoriser	401 (UNAUTHORIZED)	561 (identity provider error)	—
Access denied, invalid API key or firewall block	403	403 (web application firewall)	—
Unknown resource or method	403 (MISSING_AUTHENTICATION_TOKEN) or 404 (RESOURCE_NOT_FOUND)	—	—
Request body too large	413	413	—
Rate limit or quota exceeded	429 (THROTTLED, QUOTA_EXCEEDED)	—	—
No healthy target	—	503	521, 523
Upstream connection or response failure	504 (INTEGRATION_FAILURE)	502	520, 525, 526
Upstream timeout	504 (INTEGRATION_TIMEOUT)	504	522, 524
Client closed the connection early	—	460	—

First, gateways do not always follow the standard's recommendations. Amazon API Gateway answers a request with no authentication token with 403, where RFC 9110 would suggest 401 with a WWW-Authenticate challenge, and uses the same 403 response for requests to methods or resources that do not exist [18]. It also maps an integration failure to 504, where 502 would describe the situation more precisely. These choices are documented and have historical reasons, but a client written strictly against RFC 9110 will misinterpret them.

Second, platforms use unregistered codes to add diagnostic precision, as discussed in Section 3.3. Cloudflare's 52x codes and the load balancer's 460 and 561 tell an operator exactly which segment of the path failed, but a generic client sees only "some server error" or "some client error".

Third, most gateways allow these defaults to be changed. Amazon API Gateway, for example, exposes each of its gateway responses as a configurable type whose status code and body can be customised [18]. This means that the codes a client receives depend not only on the product but on its configuration, which strengthens the case for clients relying on the class first and the specific code second.

6.3 Operational consequences

Behind a gateway, the status code drives automated behaviour at scale. Dashboards and alerts count error rates by class: a rise in 5xx responses usually indicates a failure on the provider's side and pages an engineer, while a rise in 4xx responses more often indicates a misbehaving client. Retry policies in gateways and service meshes commonly retry on 502, 503 and 504, especially for safe methods, and should never retry 400 or 422, since the same request will fail in the same way. Circuit breakers stop sending traffic to a service that keeps returning 5xx responses and answer 503 immediately, giving the service time to recover. Caches store certain responses, such as 200 and 301, and rely on 304 to revalidate them [32].

Platforms increasingly make the origin of a code visible in their metrics. The AWS Application Load Balancer, for example, counts the 4xx and 5xx responses that it generates itself separately from those returned by its targets [19]. An operator can therefore see at once whether a rise in 5xx responses comes from the load balancer, which suggests a connectivity or capacity problem, or from the application, which suggests a defect. Gateways that do not offer this separation can approximate it through the attribution techniques described in Section 6.4.

Each of these mechanisms fails silently when the application chooses the wrong code. A 200 response with an error in its body is counted as a success, cached, never retried and never alerted on. A 500 response for invalid input is counted as an incident, retried pointlessly and may trip a circuit breaker that then blocks valid traffic. The cost of a wrong status code therefore grows with the amount of infrastructure that trusts it, which is why the question becomes more important, not less, as a system moves behind a gateway.

6.4 Attributing a response

Because the same code can have different origins, operators need to know who produced a given response. Three techniques are common. The gateway can answer in its own body format, for example Problem Details with a gateway-specific type, while services use theirs. The gateway can generate a request identifier, send it to the service in a header such as X-Request-Id and return it to the client, so that one identifier connects the gateway log, the service log and the client's error report. Finally, many products add their own response headers that indicate whether the gateway or the service produced the answer. Together these make a 404 or a 503 diagnosable rather than ambiguous.

6.5 A minimal reference gateway

To make these behaviours concrete, the reference implementation described in Section 7 includes a minimal gateway in front of a single service. It applies five checks to every request, in a fixed order, and the first check that fails determines the response: (1) authentication, answered with 401 and a WWW-Authenticate challenge; (2) authorisation by role, answered with 403; (3) routing, answered with 404 and a Problem Details body when no service handles the path; (4) a per-client rate limit, answered with 429 and Retry-After; and (5) forwarding, answered with 504 when the service does not respond within one second and with 502 when the connection fails. A request that passes all five checks receives the service's own response unchanged, including the service's own 404 and 422 responses.

The design illustrates the three origins of Section 6.1 in a few hundred lines of code. The gateway's own errors use Problem Details, while the service uses a different body format, so a client or operator can tell from the body alone who answered. Every request receives a short random identifier before any check runs, which is returned to the client and forwarded to

the service in an X-Request-Id header, so that one identifier links the gateway log with the service log. The order of the checks is a deliberate security choice, as discussed in Section 5.4: because identity is checked before routing, an unauthenticated caller cannot discover which paths exist. Finally, the distinction between 502 and 504 is implemented by checking for a timeout before any other failure. This small detail determines whether an operator looks for a slow service or for one that is down.

7. Framework-generated status codes: a Quarkus case study

7.1 Setting

Application developers rarely write every status code themselves. Modern frameworks produce many codes automatically: when no route matches, when a method is not supported, when a body cannot be parsed, or when an exception escapes the application code. These automatic answers are part of the API's contract, but they are easy to overlook because no line of application code creates them.

To examine them, a small reference implementation was built with Quarkus, a Java framework that implements the Jakarta RESTful Web Services specification [26], [28]. The implementation consists of three applications: a "restaurant" service that demonstrates the status codes of Sections 4 and 5, a "menu" service, and a minimal API gateway in front of it. Each behaviour is covered by an automated test that checks the status code and the relevant headers. The applications were built with Maven and run on Quarkus 3.37.4 and OpenJDK 21; all 37 tests passed. The code is publicly available [29], and the same examples are explained step by step in a companion book [33].

7.2 Codes produced without application code

Table 6 lists the situations in which the framework produced a status code with no status-related code in the application. In most cases the framework's choice matches RFC 9110. The Jakarta RESTful Web Services specification defines this behaviour in its request-matching algorithm, which maps failures at each step to 404, 405, 406 or 415 [28].

Table 6. Status codes produced automatically by the framework in the case study.

Situation	Status code observed	Matches RFC 9110
No resource matches the path	404 Not Found	Yes
Resource exists, method not supported	405 Method Not Allowed	Code yes; Allow header missing
No acceptable representation for the client	406 Not Acceptable	Yes
Body is not valid JSON	400 Bad Request	Yes
No credentials for a protected resource	401 Unauthorized with WWW-Authenticate	Yes; Basic challenge without realm
Authenticated, but without the required role	403 Forbidden	Yes
Body larger than the configured limit	413 (reason phrase "Request Entity Too Large")	Yes; outdated reason phrase
Body in a media type the method does not consume	415 Unsupported Media Type	Yes
Unhandled exception in application code	500 Internal Server Error, with an error identifier and no stack trace in production mode	Yes
Method returns an object	200 OK	Yes
Method returns nothing	204 No Content	Yes

7.3 Deviations and gaps

The case study revealed four points where the default behaviour falls short of the standard's requirements or the recommended practice.

First, the framework answered an unsupported method with 405 but without the Allow header that RFC 9110 requires [1]. A small response filter was needed to add it. Clients and gateways that use Allow to discover supported methods would otherwise receive incomplete information.

Second, when authentication used the Basic scheme, the 401 response carried a WWW-Authenticate challenge without a realm parameter, which RFC 7617 requires for that scheme [17]. Browsers and most clients tolerate the omission, but strict clients may not.

Third, two common API behaviours do not produce the recommended codes without explicit configuration. Validation failures on request bodies, declared with Jakarta Bean Validation annotations, produced 400 by default; an exception mapper was needed to produce 422 for content that is well-formed but invalid. Rate limiting with the SmallRye Fault Tolerance annotation produced an exception rather than a response; a mapper was needed to turn it into 429 with a Retry-After header [27]. Both choices are defensible framework defaults, but developers who assume the recommended codes will appear automatically will be wrong.

Fourth, the reason phrases emitted over HTTP/1.1 used outdated names: "413 Request Entity Too Large" and "422 Unprocessable Entity" rather than RFC 9110's "Content Too Large" and "Unprocessable Content". Because reason phrases carry no meaning (Section 3.1), this has no functional effect, but it illustrates why clients must rely on the number alone.

On the positive side, the framework's handling of unexpected failures followed good security practice: in production mode, an unhandled exception produced a 500 response containing only an error identifier, while the full stack trace was written to the server log. A ServiceUnavailableException constructed with a delay produced 503 with the corresponding Retry-After header.

The general lesson is not specific to this framework. Automatic status codes are part of an API's contract and should be tested like any other behaviour. The tests in the reference implementation check both the code and the required headers, which is how the missing Allow header was found.

8. Discussion

8.1 Recurring mistakes

Combining the standards review, the platform documentation and the empirical studies [2]–[5], Table 7 lists the mistakes that recur most often, their consequences and the corrective practice.

Table 7. Recurring status code mistakes in APIs.

Mistake	Consequence	Corrective practice
200 with an error in the body	Monitoring, caches, retries and gateways count the failure as success	Send the matching 4xx or 5xx code, with Problem Details
500 for invalid input	Blames the server, raises false alarms, invites pointless retries	400 or 422
401 and 403 swapped	Client logs in repeatedly, or gives up when a login would succeed	401 for missing or invalid credentials; 403 for insufficient permission
Stack traces in 5xx bodies	Exposes internal structure to attackers	Log the details; return a short message and an identifier
Missing required headers	Clients cannot recover automatically	WWW-Authenticate with 401, Allow with 405
302 to redirect a POST	Client may switch to GET and lose the body	307 or 308
Invented codes, or 418 in production	Clients and tools do not recognise them	Use registered codes; put detail in the body
Parsing the reason phrase	Breaks on outdated phrases and on HTTP/2 and HTTP/3	Use the numeric code only

8.2 A decision procedure

The variety of codes can make the choice seem harder than it is. In practice, four questions asked in order are enough for almost every API response.

1. Did the request succeed? If yes, choose a 2xx code: 201 with Location if a resource was created, 202 if work was only started, 204 if there is nothing to return, otherwise 200.
2. Must the client go somewhere else? If yes, choose a 3xx code: 308 or 307 if the method must be preserved, 303 to send the client to a result with GET, 304 for a valid cached copy.
3. Was the problem caused by the request? If yes, choose a 4xx code: 400 or 422 for invalid input, 401 or 403 for credentials and permissions, 404 or 410 for missing resources, 405 for unsupported methods, 409 or 412 for state conflicts, 429 for rate limits.
4. Otherwise, the failure is on the server side: 500 for unexpected errors, 503 with Retry-After for overload or maintenance, and 502 or 504 only when acting as an intermediary.

The first three questions select the class; the fourth question within each class selects the specific code. In every case where something went wrong, the body should then explain the problem using Problem Details.

8.3 Implications

For API designers, the main implication is that the status code is part of the interface contract and should be designed, documented and tested as such, including the codes produced by the framework rather than by application code. For architects, the review shows that the codes a client receives are shaped by every component on the path, and that gateways and platforms make their own choices, some of which differ from the standard. Clients should therefore handle each class sensibly before handling specific codes, and operators should make the origin of each response visible. For client developers, the review suggests a layered strategy. A robust client first acts on the class: it treats any 2xx response as success, follows 3xx redirects within a limit, treats 4xx as a problem with its own request and 5xx as a temporary server problem. It then refines that behaviour for the specific codes it understands, such as 401, 409, 412 and 429, and reads Problem Details where available for messages to show to users. Such a client continues to work when a gateway substitutes its own codes or when a platform introduces an unregistered one, because the class rule guarantees that the first layer still applies.

For educators, the class rule, the 401–403 distinction, the role of Problem Details and the three origins of a status code at a gateway form a compact core that can be taught early, before frameworks and gateways are introduced.

8.4 Limitations

This review has several limitations. It is a narrative review rather than a systematic one, and the selection of platforms in Section 6 is illustrative, not exhaustive; other gateways may behave differently, and the documented behaviour of the products discussed may change after publication. The case study uses a single framework and version, so its specific findings, such as the missing Allow header, may not apply to other frameworks or to later versions. The registry figures describe its state in October 2026 and will change as codes are added, as the temporary registration of 104 shows. Finally, the review does not measure how often each mistake occurs; for that, the empirical studies cited should be consulted.

9. Conclusion

The HTTP status code is a small field with large consequences. In APIs it is read mainly by software, and every tool on the path, from caches and monitoring systems to retry policies and gateways, acts on it without reading the body. This review has brought together the current standard, the official registry, the error format and the headers that complete each code, and has examined two sources of status codes that the standards largely leave to vendors: frameworks and gateways. Both produce a large share of the codes clients actually receive, and both were shown to depart from the standard in specific, documented ways.

The practical recommendations are simple. Use registered codes only, chosen by class first and then by meaning. Send the headers the standard requires. Describe errors in the body using Problem Details, never instead of the code. Test the automatic answers of the framework as well as the answers of the application. Behind a gateway, make the origin of each response visible and write clients that rely on the class before the specific code. Future work could extend the framework case study to other frameworks and languages, and measure

systematically how gateways' default behaviour differs from the standard.

Declarations

Author contributions (CRediT): L. H. de Sousa Ribas: Conceptualization, Methodology, Software, Validation, Investigation, Writing – original draft, Writing – review and editing.

Funding: This research received no external funding.

Conflicts of interest: The author's book on HTTP status codes [33] is published by FTI Publishers, which also publishes this journal. The author declares no other conflicts of interest.

Ethics approval and consent: Not applicable. This study did not involve human participants or personal data.

Data and code availability: The reference implementation and its tests are available at https://github.com/lhsribas/httpstatus-code-explained-with-quarkus [29].

Use of generative AI: Claude (Anthropic) was used to help structure and draft this manuscript from the author's book and reference implementation, and to check references. The author reviewed and edited all content, verified all technical statements and references, and takes full responsibility for the article.

Abbreviations

Abbreviation	Meaning
API	Application programming interface
AWS	Amazon Web Services
HTTP	Hypertext Transfer Protocol
IANA	Internet Assigned Numbers Authority
IETF	Internet Engineering Task Force
JSON	JavaScript Object Notation
REST	Representational State Transfer
RFC	Request for Comments
TLS	Transport Layer Security
URI	Uniform Resource Identifier
WebDAV	Web Distributed Authoring and Versioning

References

[1] R. Fielding, M. Nottingham, and J. Reschke, "HTTP Semantics," RFC 9110, IETF, Jun. 2022, doi: 10.17487/RFC9110.

[2] A. Neumann, N. Laranjeiro, and J. Bernardino, "An analysis of public REST web service APIs," *IEEE Trans. Services Comput.*, vol. 14, no. 4, pp. 957–970, 2021, doi: 10.1109/TSC.2018.2847344.

[3] F. Palma, J. Dubois, N. Moha, and Y.-G. Guéhéneuc, "Detection of REST patterns and antipatterns: A heuristics-based approach," in *Service-Oriented Computing – ICSOC 2014*, Lecture Notes in Computer Science, vol. 8831. Berlin, Germany: Springer, 2014, pp. 230–244, doi: 10.1007/978-3-662-45391-9_16.

[4] F. Petrillo, P. Merle, N. Moha, and Y.-G. Guéhéneuc, "Are REST APIs for cloud computing well-designed? An exploratory study," in *Service-Oriented Computing – ICSOC 2016*, Lecture Notes in Computer Science, vol. 9936. Cham, Switzerland: Springer, 2016, pp. 157–170, doi: 10.1007/978-3-319-46295-0_10.

[5] S. Kotstein and J. Bogner, "Which RESTful API design rules are important and how do they improve software quality? A Delphi study with industry experts," in *Service-Oriented Computing – SummerSOC 2021*, Communications in Computer and Information Science, vol. 1429. Cham, Switzerland: Springer, 2021, pp. 154–173, doi: 10.1007/978-3-030-87568-8_10.

[6] Internet Assigned Numbers Authority, "Hypertext Transfer Protocol (HTTP) Status Code Registry." [Online]. Available: https://www.iana.org/assignments/http-status-codes/ (accessed Oct. 30, 2026).

[7] M. Nottingham, E. Wilde, and S. Dalal, "Problem Details for HTTP APIs," RFC 9457, IETF, Jul. 2023, doi: 10.17487/RFC9457.

[8] M. Nottingham, "Building Protocols with HTTP," RFC 9205 (BCP 56), IETF, Jun. 2022, doi: 10.17487/RFC9205.

[9] M. Nottingham and R. Fielding, "Additional HTTP Status Codes," RFC 6585, IETF, Apr. 2012, doi: 10.17487/RFC6585.

[10] J. Reschke, "The Hypertext Transfer Protocol Status Code 308 (Permanent Redirect)," RFC 7538, IETF, Apr. 2015, doi: 10.17487/RFC7538.

[11] K. Oku, "An HTTP Status Code for Indicating Hints," RFC 8297, IETF, Dec. 2017, doi: 10.17487/RFC8297.

[12] M. Thomson, M. Nottingham, and W. Tarreau, "Using Early Data in HTTP," RFC 8470, IETF, Sep. 2018, doi: 10.17487/RFC8470.

[13] T. Bray, "An HTTP Status Code to Report Legal Obstacles," RFC 7725, IETF, Feb. 2016, doi: 10.17487/RFC7725.

[14] L. Dusseault, Ed., "HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)," RFC 4918, IETF, Jun. 2007, doi: 10.17487/RFC4918.

[15] G. Clemm, J. Crawford, J. Reschke, and J. Whitehead, "Binding Extensions to Web Distributed Authoring and Versioning (WebDAV)," RFC 5842, IETF, Apr. 2010, doi: 10.17487/RFC5842.

[16] I. Fette and A. Melnikov, "The WebSocket Protocol," RFC 6455, IETF, Dec. 2011, doi: 10.17487/RFC6455.

[17] J. Reschke, "The 'Basic' HTTP Authentication Scheme," RFC 7617, IETF, Sep. 2015, doi: 10.17487/RFC7617.

[18] Amazon Web Services, "Gateway response types for API Gateway," *Amazon API Gateway Developer Guide*. [Online]. Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/suppo gateway-response-types.html (accessed Oct. 30, 2026).

[19] Amazon Web Services, "Troubleshoot your Application Load Balancers," *Elastic Load Balancing User Guide*. [Online]. Available: https://docs.aws.amazon.com/elasticloadbalancing/latest/application/lo balancer-troubleshooting.html (accessed Oct. 30, 2026).

[20] Cloudflare, "Cloudflare 5xx errors," *Cloudflare Docs*. [Online]. Available: https://developers.cloudflare.com/support/troubleshooting/http-status-codes/cloudflare-5xx-errors/ (accessed Oct. 30, 2026).

[21] GitHub, "Troubleshooting the REST API," *GitHub Docs*. [Online]. Available: https://docs.github.com/en/rest/using-the-rest-api/troubleshooting-the-rest-api (accessed Oct. 30, 2026).

[22] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, Univ. of California, Irvine, CA, USA, 2000.

[23] Microsoft, "Microsoft REST API Guidelines." [Online]. Available: https://github.com/microsoft/api-guidelines (accessed Oct. 30, 2026).

[24] Google, "AIP-193: Errors," *API Improvement Proposals*. [Online]. Available: https://google.aip.dev/193 (accessed Oct. 30, 2026).

[25] Zalando SE, "Zalando RESTful API and Event Guidelines." [Online]. Available: https://opensource.zalando.com/restful-api-guidelines/ (accessed Oct. 30, 2026).

[26] Quarkus, "Writing REST services with Quarkus REST," *Quarkus Guides*. [Online]. Available: https://quarkus.io/guides/rest (accessed Oct. 30, 2026).

[27] Quarkus, "SmallRye Fault Tolerance," *Quarkus Guides*. [Online]. Available: https://quarkus.io/guides/smallrye-fault-tolerance (accessed Oct. 30, 2026).

[28] Eclipse Foundation, "Jakarta RESTful Web Services 3.1 Specification," 2022. [Online]. Available: https://jakarta.ee/specifications/restful-ws/3.1/

[29] L. H. de Sousa Ribas, "httpstatus-code-explained-with-quarkus," GitHub repository, 2026. [Online]. Available: https://github.com/lhsribas/httpstatus-code-explained-with-quarkus

[30] R. Fielding, M. Nottingham, and J. Reschke, "HTTP/1.1," RFC 9112, IETF, Jun. 2022, doi: 10.17487/RFC9112.

[31] L. Masinter, "Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0)," RFC 2324, IETF, Apr. 1998, doi: 10.17487/RFC2324.

[32] R. Fielding, M. Nottingham, and J. Reschke, "HTTP Caching," RFC 9111, IETF, Jun. 2022, doi: 10.17487/RFC9111.

[33] L. H. de Sousa Ribas, *Knock Knock, Who's There? HTTP Status Codes Explained with Quarkus: A Friendly First Step toward API Gateways, with Java and Quarkus*. Vilnius, Lithuania: FTI Publishers, 2026.

[34] OpenAPI Initiative, "OpenAPI Specification, Version 3.1.0," Feb. 2021. [Online]. Available: https://spec.openapis.org/oas/v3.1.0

[35] OWASP Foundation, "Authentication Cheat Sheet," *OWASP Cheat Sheet Series*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_ (accessed Oct. 30, 2026).