

API Design A Visual Guide

Sample Book

Chapters 1 to 2

Foundations, Practical Design, Security, Evolution, and Operations

Introduction

This book is designed for readers who are taking their first steps with API design, including systems engineers who want to build the right foundation from the start.

Rather than asking you to memorize more than you need, the book focuses on the essential concepts that will support your future study and day-to-day work. More advanced topics are included only when they are likely to become useful to many readers, and even then they are introduced only far enough to explain their purpose, context, and overall role.

Processes, relationships, and structures that can be difficult to picture from text alone are supported by diagrams. By moving between explanations, visuals, and examples, you can build a clearer and more practical understanding of the subject.

Product names, service names, logos, icons, and other identifying marks mentioned in this book are trademarks or registered trademarks of their respective owners. This book is not an official publication, certified training resource, or endorsed learning material of any company or organization.

How to Use This Book

Read the chapters in order when studying the subject for the first time. Each chapter builds on terms, mental models, and design decisions introduced earlier, so following the sequence will help you develop the required background step by step.

Before reading a chapter, review its **What You'll Be Able to Do** section. The learning outcomes identify the practical abilities the chapter is intended to develop and help you distinguish the central ideas from supporting detail.

Use the diagrams to clarify relationships, processes, structures, state changes, responsibility boundaries, and data movement that may be difficult to picture from prose alone. The diagrams support the written explanation rather than replace it, so read the surrounding text before drawing conclusions from a visual.

Try the code, HTTP messages, and hands-on examples yourself whenever the necessary environment is available. Reproducing a small example and changing one input at a time is often the fastest way to confirm how a contract behaves.

At the end of each chapter, use the **Chapter Summary** and **Review Questions** to confirm that you understand and can retain the fundamentals. Return to the relevant section when an answer is unclear rather than memorizing a phrase without its reasoning.

Advanced-topic overviews do not need to be memorized at this stage. Treat them as signposts: they show why a mechanism exists, how it relates to the foundation, and when you may need to study it in greater depth later.

Table of Contents

Introduction

How to Use This Book

Main Text

Part I — Understanding the API Contract

Chapter 1 — API Design Is Contract Design

- What You'll Be Able to Do
- The API as an agreement
- A boundary that hides change
- Design, implementation, and operation are different responsibilities
- Internal APIs still need discipline
- Quality is part of the contract
- The API lifecycle
- The support-ticket contract in context
- Two damaging misconceptions
 - "API design is mainly choosing URL paths"
 - "An internal API can be changed whenever the owning team wants"
- Before approving an API concept
- Chapter Summary
- Review Questions

Chapter 2 — How an HTTP API Exchange Works

- What You'll Be Able to Do
- The request-response model
- The four parts of a request
- Reading a response
- A first command-line request
- Method, URI, headers, and body work together
- Headers carry message metadata
- Stateless interactions
- Intermediaries in the path
- Following one ticket retrieval
- Two common misunderstandings
 - "The URL alone defines what an HTTP API call does"
 - "Stateless means that the server cannot store any data"
- A practical reading checklist
- Chapter Summary
- Review Questions

Main Text

An API should be designed as a dependable contract between systems, not as a thin wrapper around application code or database tables. The contract tells consumers what they may request, what the provider promises to return, which failures can occur, and which behaviors remain stable as the implementation changes.

This guide uses a JSON-based HTTP API over HTTPS as its primary teaching environment. That is one common environment for project work, not a universal rule for all APIs. The recurring example is a support-ticket service used by three consumers:

- A web application used by support agents.
- An automation service that creates and updates tickets.
- A reporting service that reads ticket data for analysis.

The chapters progress from the basic contract model to everyday HTTP API design, safe state changes, security, documentation, testing, compatibility, reliability, asynchronous communication, observability, API-style selection, and team governance. The most important habit throughout is to ask: **What may a consumer safely rely on?**

Part I — Understanding the API Contract

Chapter 1 — API Design Is Contract Design

The central idea is that an API is a long-lived agreement about observable behavior. URLs and payloads matter, but they are only parts of that agreement. A useful design also defines meanings, permissions, failures, timing expectations, compatibility rules, and ownership.

What You'll Be Able to Do

After completing this chapter, you will be able to:

- Explain why an API is a contract for observable behavior rather than merely a set of URLs.
- Distinguish API design, implementation, and operation responsibilities.
- Identify coupling, quality attributes, lifecycle stages, and ownership concerns that shape a durable boundary.

The API as an agreement

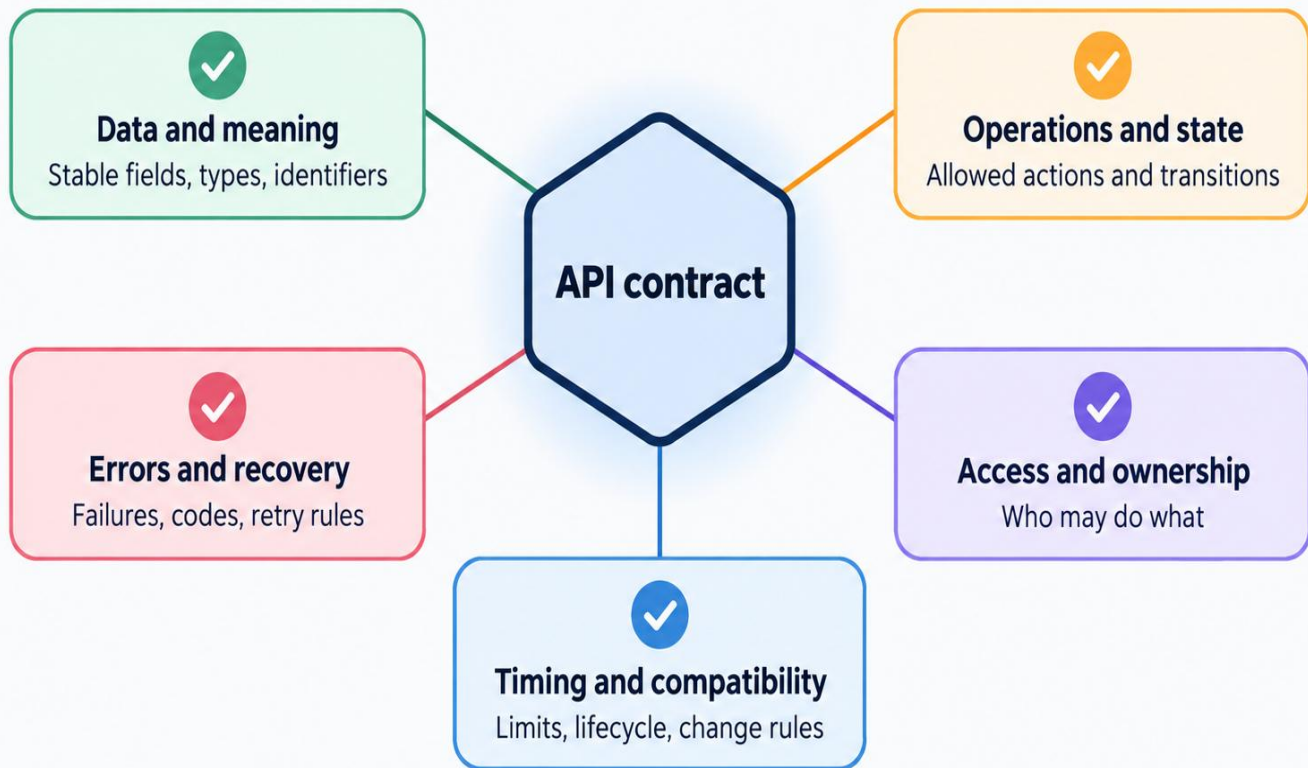
An **application programming interface (API)** is a defined way for one software component to request capabilities or exchange data with another. The component that offers the API is the **provider**. A component that calls it is a **client** or **caller**. The word **consumer** is broader: it may refer to a client application, another service, a team, or an organization that depends on the API. A **downstream dependency** is a system that the provider itself must call or rely on to complete some requests.

Suppose a support-ticket provider offers an operation for retrieving a ticket. The contract may promise that:

- A ticket has a stable identifier.
- A successful retrieval returns a specific data structure.
- A missing ticket produces a defined error.
- A caller must have permission to view the ticket.
- Existing fields will not silently change meaning.

Chapter 1: API Design Is Contract Design

1-1 The API contract as a bundle of promises



001

Figure 1-1: The API contract as a bundle of promises

Those promises allow the web application, automation service, and reporting service to be developed independently. The provider may rewrite its code or replace its database without forcing those consumers to change, provided that the observable contract remains compatible.

A useful everyday analogy is a restaurant menu. The kitchen is free to reorganize its equipment or change how it prepares a dish, but a customer relies on the menu's name, description, price, and ordering rules. If the kitchen silently changes "vegetable soup" into a meat dish, the problem is not an internal implementation detail; it is a broken contract.

A boundary that hides change

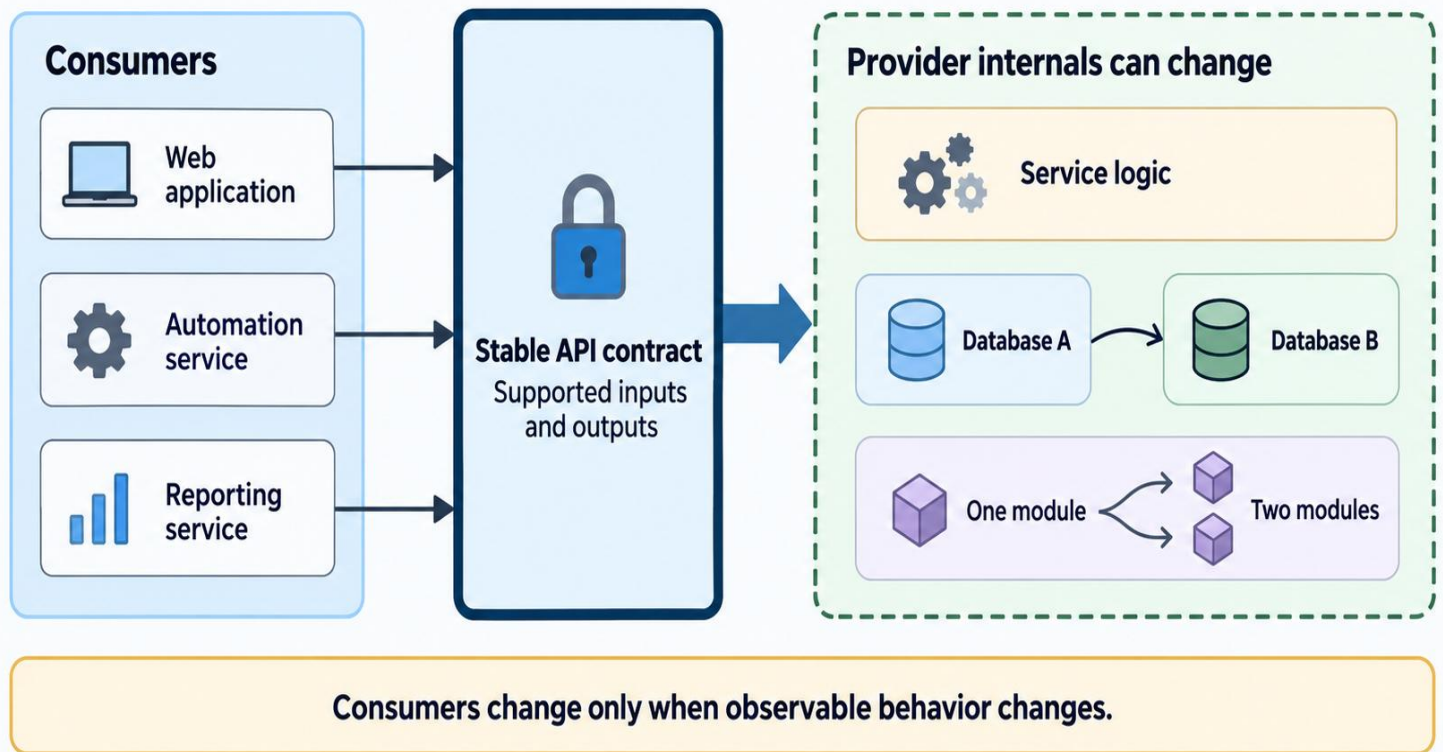
A **system boundary** is the point at which one part of a system interacts with another through defined inputs and outputs. A well-designed API exposes supported behavior at that boundary while hiding implementation details.

This hiding is valuable because implementations change frequently. A ticket service might move from one database product to another, split one internal module into several services, or calculate priority differently. Consumers should not need to know unless the public behavior actually changes.

The opposite condition is **coupling**, which means that one component depends on details of another component. Some coupling is unavoidable: a consumer must depend on the API contract. The design goal is to keep that coupling intentional and stable rather than accidental.

Chapter 1: API Design Is Contract Design

1-2 Stable boundary hiding changing internals



002

Figure 1-2: Stable boundary hiding changing internals

For example, consider an API response copied directly from a database row:

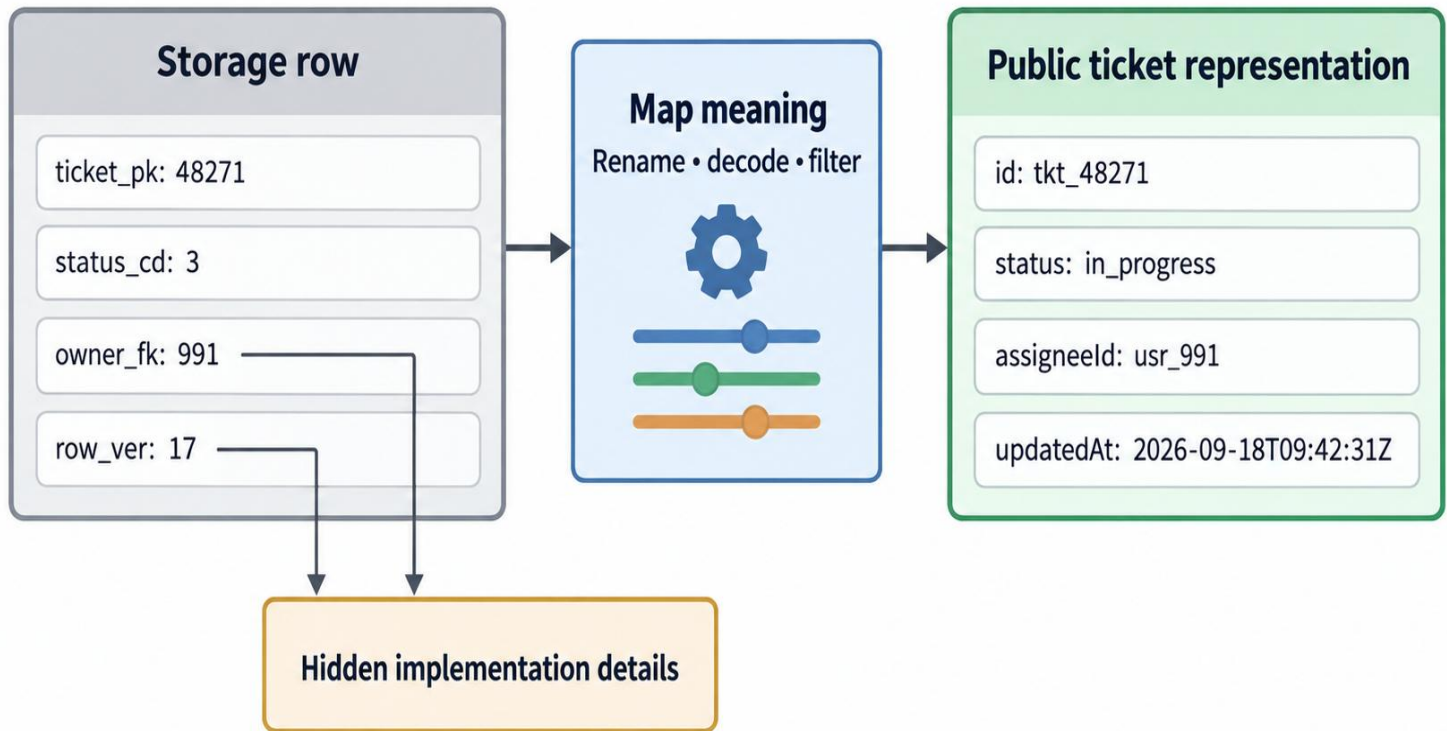
```
{
  "ticket_pk": 48271,
  "status_cd": 3,
  "owner_fk": 991,
  "row_ver": 17
}
```

The field names and numeric codes reveal storage choices but do not explain business meaning. A clearer contract might be:

```
{
  "id": "tk_48271",
  "status": "in_progress",
  "assigneeId": "usr_991",
  "updatedAt": "2026-09-18T09:42:31Z"
}
```

Chapter 1: API Design Is Contract Design

1-2 Database row versus domain representation



003

Figure 1-3: Database row versus domain representation

The second representation still has implementation behind it, but the contract uses domain concepts that consumers can understand. The provider can later change primary-key types, table names, or versioning columns without exposing those changes.

Design, implementation, and operation are different responsibilities

These three areas influence one another, but they answer different questions.

Area	Main question	Ticket-service example
API design	What behavior and data may consumers rely on?	Which fields identify a ticket, how assignment works, and which failures are defined.
API implementation	How does the provider produce that behavior?	Which framework, database, and internal classes are used.
API operation	How is the running API kept available and diagnosable?	Deployment, monitoring, capacity, incident response, and runtime configuration.

Confusing these areas leads to weak decisions. “Our framework automatically maps this class to JSON” is an implementation fact, not evidence that the resulting data contract is appropriate. “The database can update this column” does not prove that consumers should be allowed to perform that update.

Design begins with consumer-visible behavior. Implementation choices must support that behavior, not define it accidentally.

Internal APIs still need discipline

APIs are often described as public, partner, or internal.

- A **public API** is offered to a broad external audience.

- A **partner API** is shared with selected external organizations under an agreed relationship.
- An **internal API** is used within one organization, often between teams or services.

The labels affect documentation, onboarding, support, and risk, but none makes contract design optional. An internal API can be especially difficult to change because its consumers may be numerous, poorly recorded, or deployed on different schedules. The fact that two teams work for the same organization does not make a breaking change inexpensive.

A practical rule is: **Treat every API as independently consumed unless deployment and ownership arrangements prove otherwise.** Even when provider and client are currently released together, that condition may change as the system grows.

Quality is part of the contract

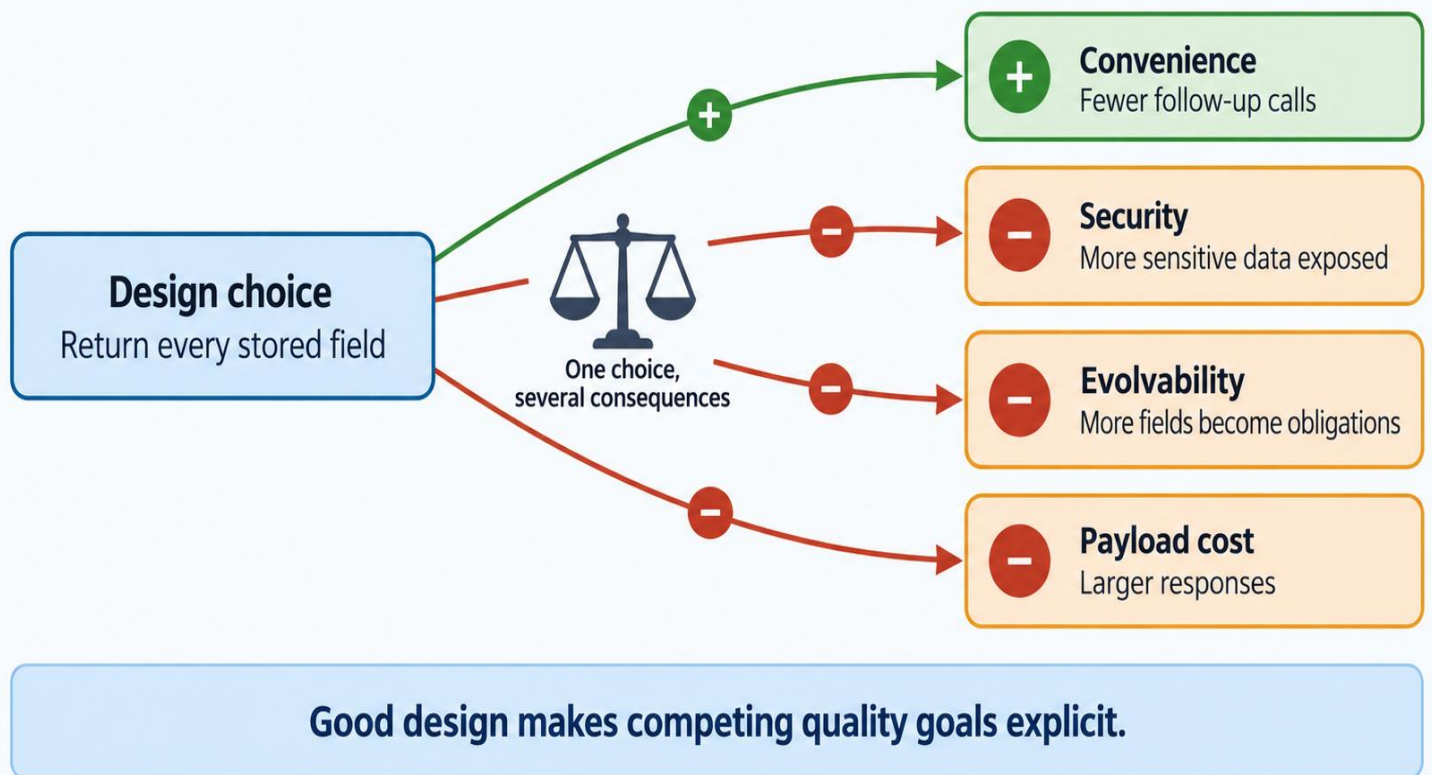
Functional requirements describe what an API must do. For the ticket service, examples include creating a ticket, assigning an agent, adding a comment, and retrieving open tickets.

A design also needs **quality attributes**, which are properties that describe how well the API must work. Common API quality goals include:

- **Clarity:** Consumers can understand the contract without reverse-engineering server behavior.
- **Consistency:** Similar operations use similar names, structures, and rules.
- **Security:** Only authorized callers can access or change protected data.
- **Reliability:** The API behaves predictably during normal failures and retries.
- **Performance:** Response time and payload size are suitable for the expected use.
- **Evolvability:** The provider can add capabilities without routinely breaking existing consumers.

Chapter 1: API Design Is Contract Design

1-5 Quality-attribute trade-offs around one contract



004

Figure 1-4: Quality-attribute trade-offs around one contract

These goals sometimes compete. Returning every ticket field in one response may appear convenient, but it can increase payload size, expose sensitive information, and create more compatibility obligations. Good design makes such trade-offs explicit.

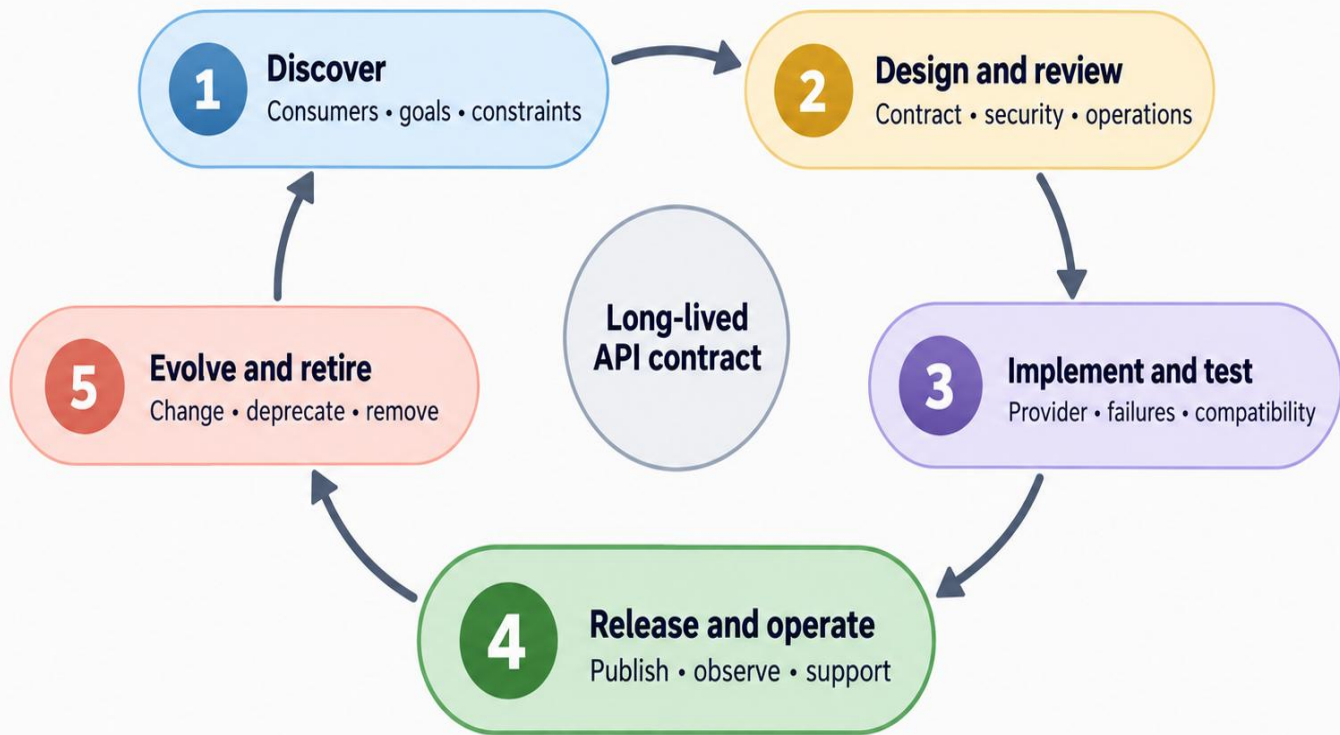
The API lifecycle

An API passes through more stages than implementation and release. A practical lifecycle is:

1. **Discovery:** Identify consumers, business goals, constraints, and existing systems.
2. **Design:** Define resources, operations, data, errors, permissions, and behavior.
3. **Review:** Check the contract with consumer, domain, security, and operations representatives.
4. **Implementation:** Build the provider according to the agreed contract.
5. **Testing:** Verify successful behavior, failures, permissions, and compatibility.
6. **Release:** Publish the contract, documentation, and migration information.
7. **Operation:** Observe usage, performance, errors, and abuse.
8. **Evolution:** Add or change capabilities while protecting consumers.
9. **Retirement:** Deprecate and remove an API through a managed process.

Chapter 1: API Design Is Contract Design

1-6 API lifecycle from discovery through retirement



005

Figure 1-5: API lifecycle from discovery through retirement

The important point is that **the contract continues to exist after the code is deployed**. Real consumers will discover assumptions that were not obvious during development. Operational evidence may show that a limit is too small or an error is too vague. Design therefore remains an active responsibility.

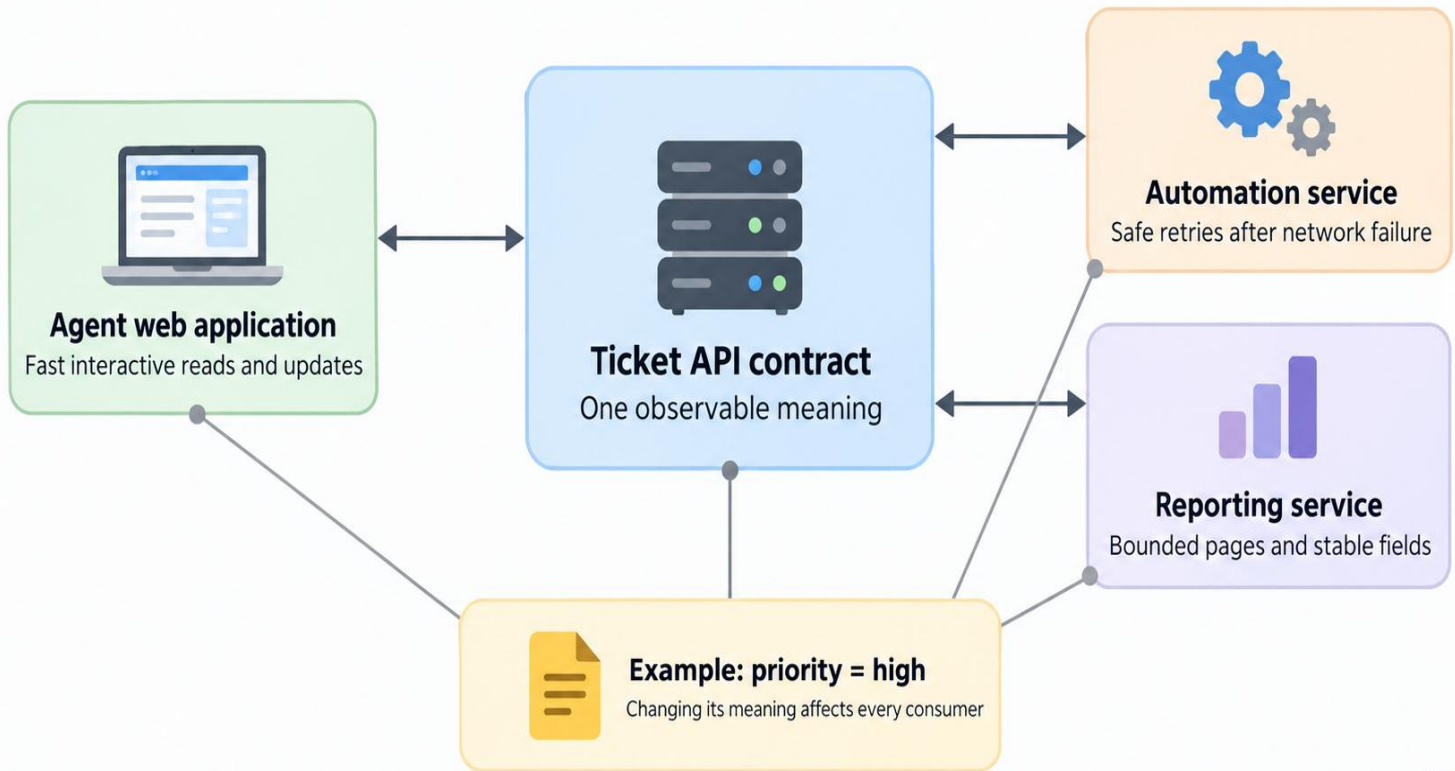
The support-ticket contract in context

Consider three consumers:

- The agent web application needs fast interactive reads and updates.
- The automation service needs safe retries because network failures can occur during batch processing.
- The reporting service needs bounded collection retrieval and stable meanings for fields over time.

Chapter 1: API Design Is Contract Design

1-7 One contract serving three consumers



006

Figure 1-6: One contract serving three consumers

A single careless choice can affect them differently. For example, changing `priority` from a named value such as `high` to a number such as `3` may require user-interface changes, automation changes, and report corrections. The database migration might be simple, while the contract migration is expensive.

This is why design review must ask not only “Can the server implement this?” but also:

- Can each consumer understand it?
- Can each consumer recover from failure?
- Can the provider change its internals later?
- Can the contract evolve without coordinated replacement of every client?

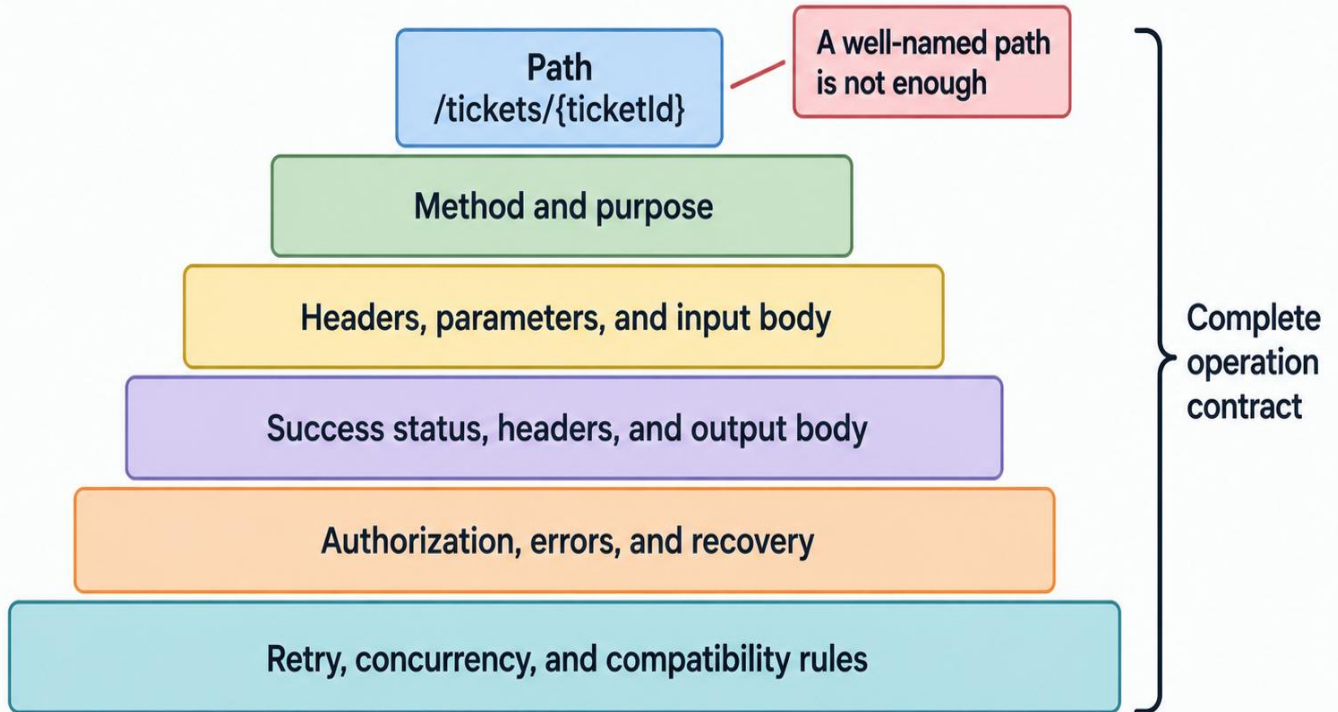
Two damaging misconceptions

“API design is mainly choosing URL paths”

Paths are visible and easy to debate, so teams may spend disproportionate time on them. A path such as `/tickets/123` says little by itself. The complete contract also includes the method, input fields, status codes, response body, authorization rules, retry behavior, concurrency behavior, and compatibility expectations.

Chapter 1: API Design Is Contract Design

1-8 A URL path versus the complete operation contract



007

Figure 1-7: A URL path versus the complete operation contract

A well-named path cannot compensate for ambiguous behavior.

“An internal API can be changed whenever the owning team wants”

Ownership gives a team responsibility for the API; it does not erase consumer dependencies. Before changing an internal API, the provider needs to know who uses it, whether the change is compatible, and how consumers will migrate.

Before approving an API concept

Confirm that the proposed API has:

- Named consumers and concrete use cases.
- An explicit provider and owner.
- Consumer-oriented domain language rather than raw implementation structures.
- Defined successful behavior and important failure behavior.
- Quality goals appropriate to expected use.
- A plan for review, release, operation, and future change.

The foundation for every later design decision is the same: expose a clear, stable boundary and keep internal choices behind it.

Chapter Summary

- An API gives consumers a stable boundary while allowing the provider's internals to change.
- The contract includes meanings, permissions, failures, quality expectations, compatibility, and ownership—not only paths and payloads.
- Internal APIs require the same design discipline because consumers and release schedules can diverge over time.

Review Questions

1. What may an API consumer rely on besides a URL and a JSON structure?
2. Why is exposing a database row directly likely to create harmful coupling?
3. Which API lifecycle activities continue after the first production release?

Chapter 2 — How an HTTP API Exchange Works

Every HTTP API call can be understood as one request message followed by one response message. Each message has a small number of parts. Once those parts are clear, endpoint definitions and troubleshooting become much easier.

What You'll Be Able to Do

After completing this chapter, you will be able to:

- Identify the method, target URI, headers, and optional body in an HTTP request.
- Read a response by interpreting its status code, headers, and optional representation.
- Explain how HTTPS, statelessness, gateways, proxies, and load balancers affect an API exchange.

The request-response model

Hypertext Transfer Protocol (HTTP) is an application protocol that defines how a client sends a request and how a server returns a response. An HTTP API uses that exchange to expose system capabilities.

HTTPS is HTTP carried over a protected transport connection that encrypts data in transit and authenticates the server to the client. In ordinary API design, HTTPS should be treated as the baseline rather than an optional enhancement.

A simplified exchange looks like this:

1. The client connects to the API host over HTTPS.
2. The client sends an HTTP request.
3. The server identifies the requested operation and processes it.
4. The server sends an HTTP response.
5. The client interprets the status, metadata, and body according to the contract.

Chapter 2: How an HTTP API Exchange Works

2-1 Logical HTTP request-response exchange

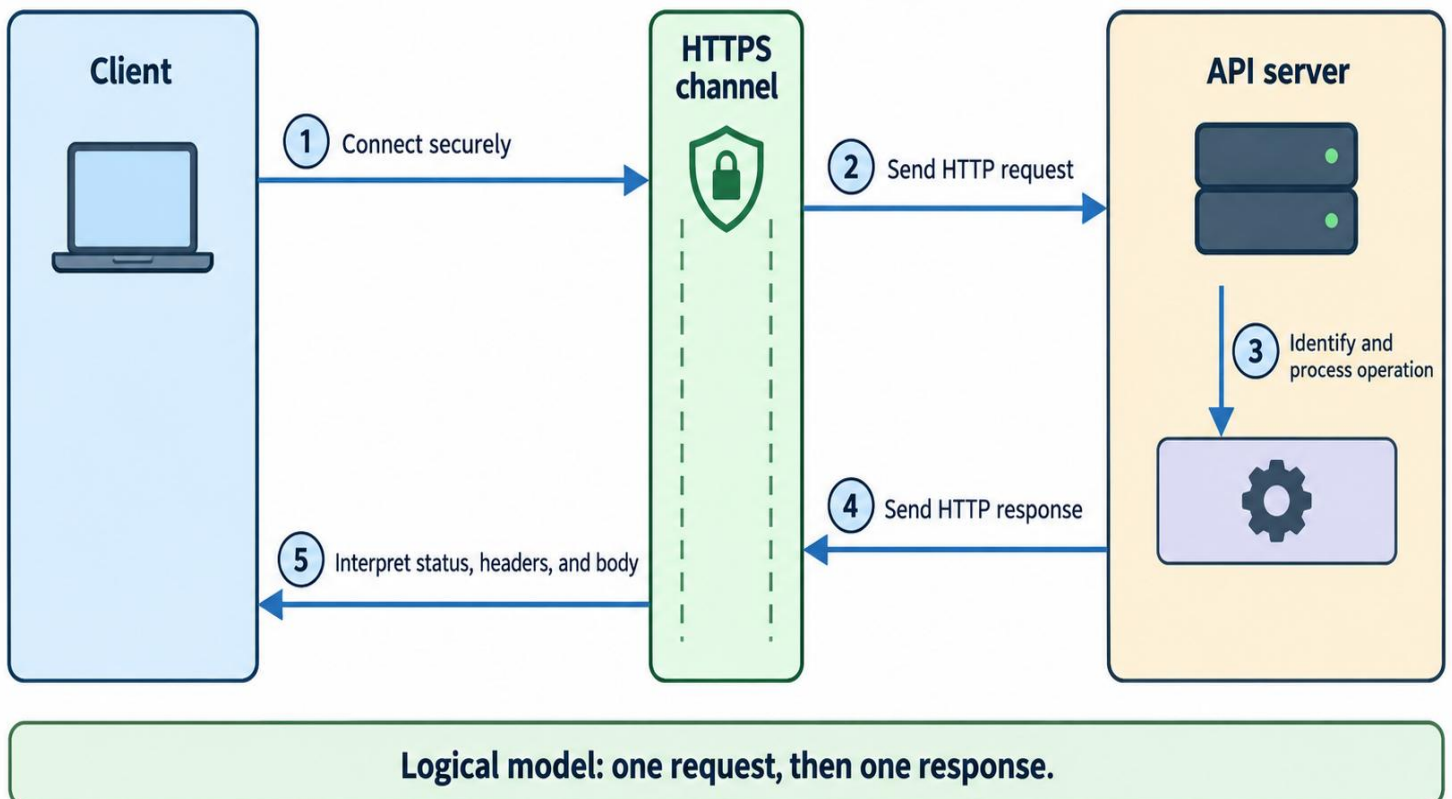


Figure 2-1: Logical HTTP request-response exchange

The network may contain intermediaries, but the logical exchange is still request followed by response.

The four parts of a request

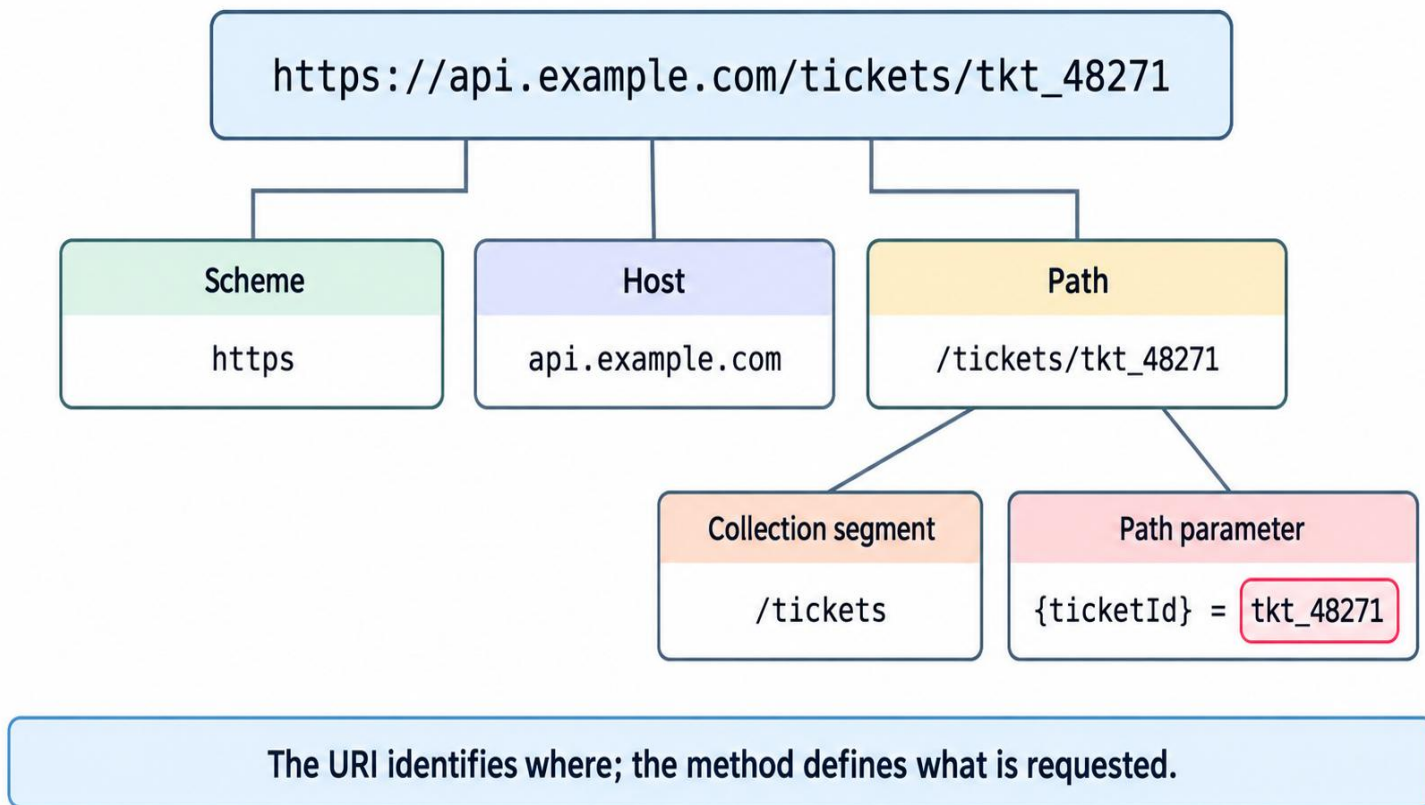
An HTTP request contains:

1. A **method**, which indicates the intended semantics of the request, such as retrieving or creating data.
2. A **target URI**, which identifies the addressed resource or operation.
3. **Headers**, which carry metadata about the message and its handling.
4. An optional **body**, which carries input data when the operation needs it.

A **Uniform Resource Identifier (URI)** is a string used to identify a resource or operation. In common HTTPS APIs, it appears in a form such as `https://api.example.com/tickets/tkt_48271`.

Chapter 2: How an HTTP API Exchange Works

2-2 URI, host, path, and path parameter relationship



010

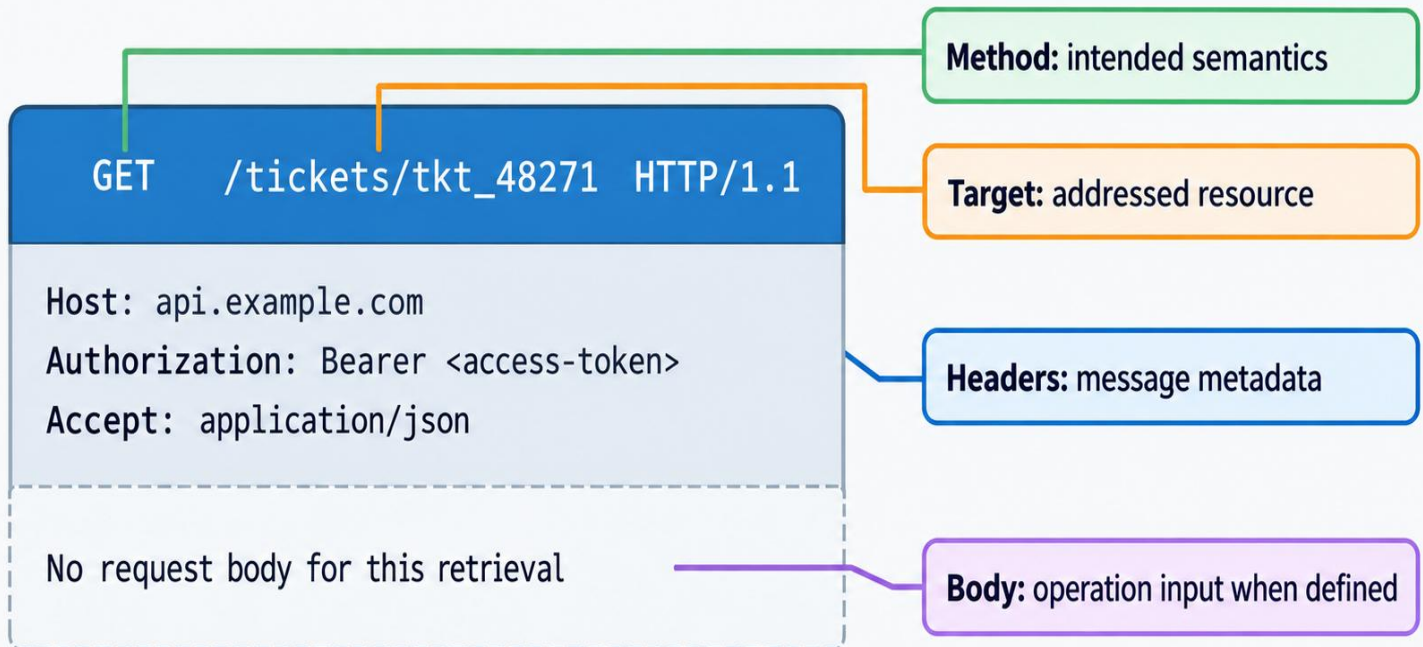
Figure 2-3: URI, host, path, and path parameter relationship

Here is a minimal request in raw HTTP form:

```
GET /tickets/tkt_48271 HTTP/1.1
Host: api.example.com
Authorization: Bearer <access-token>
Accept: application/json
```

Chapter 2: How an HTTP API Exchange Works

2-2 Anatomy of an HTTP request



009

Figure 2-2: Anatomy of an HTTP request

The important lines are:

- `GET` is the method.
- `/tickets/tkt_48271` is the target path on the host.
- `Authorization` carries an access token in this example; an **access token** is a credential that represents a caller's granted access, and `Bearer` names the credential scheme used here.
- `Accept: application/json` says that the client can receive JSON.

JavaScript Object Notation (JSON) is a text format for structured data using objects, arrays, strings, numbers, booleans, and `null`. It is common in HTTP APIs because it is widely supported and human-readable.

A **media type** identifies the format and intended interpretation of message content. `application/json` is the media type for ordinary JSON data. A **schema** is a formal definition of the fields, types, and constraints allowed in a message.

Reading a response

An HTTP response contains:

1. A **status code**, which gives a standardized summary of the outcome.
2. Headers containing response metadata.
3. An optional body containing a representation, result, or error.

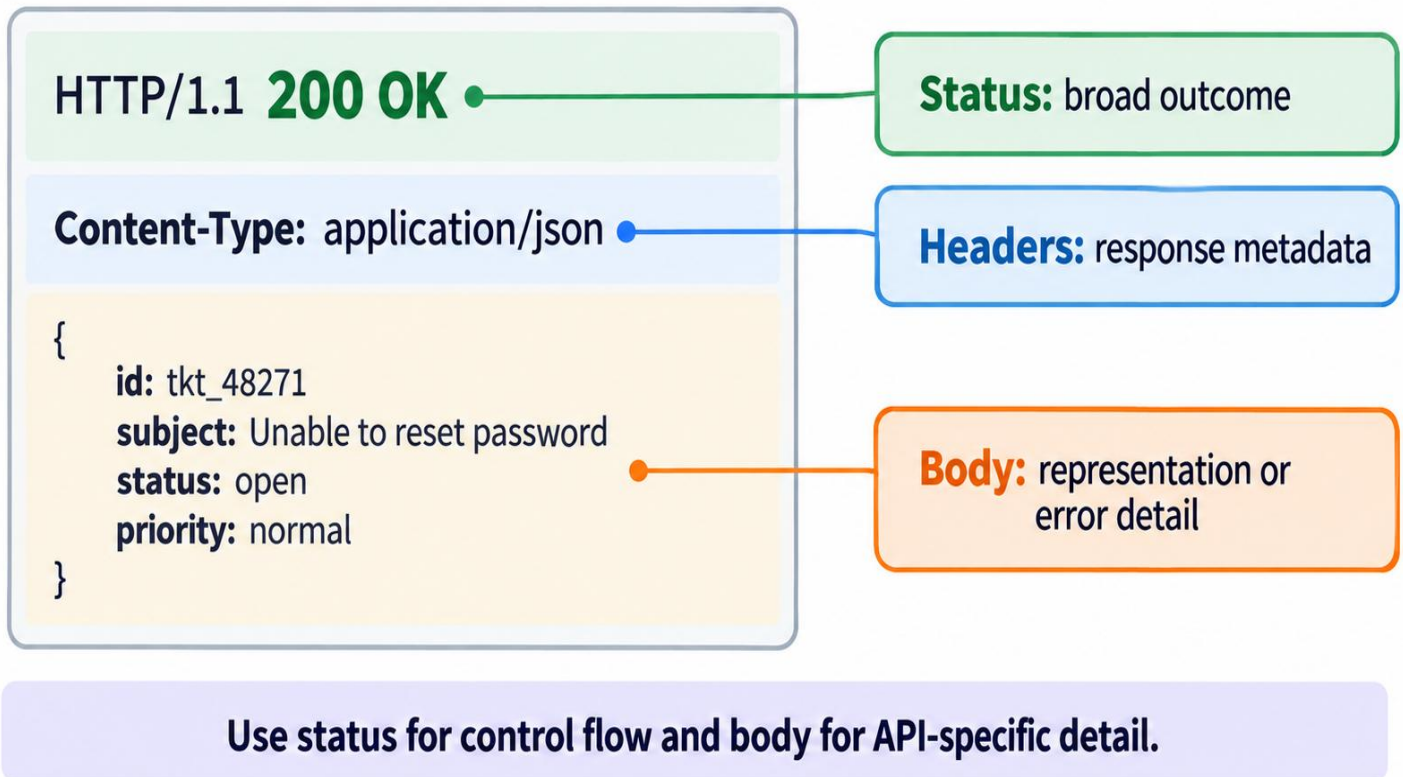
A possible successful response is:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": "tkkt_48271",
  "subject": "Unable to reset password",
  "status": "open",
  "priority": "normal",
  "createdAt": "2026-09-18T09:42:31Z"
}
```

Chapter 2: How an HTTP API Exchange Works

2-3 Anatomy of an HTTP response



011

Figure 2-4: Anatomy of an HTTP response

`200 OK` indicates successful handling. `Content-Type` tells the client how to interpret the body. The JSON object is a **representation**, meaning the data form used in the message to describe the ticket.

The status code and body serve different purposes. The status code gives a general result that HTTP-aware software can understand. The body supplies API-specific details. A client should not have to parse a sentence in the body to discover whether the request succeeded.

A first command-line request

The following command is one concrete example using `curl`, a command-line HTTP client, in a POSIX-style shell.

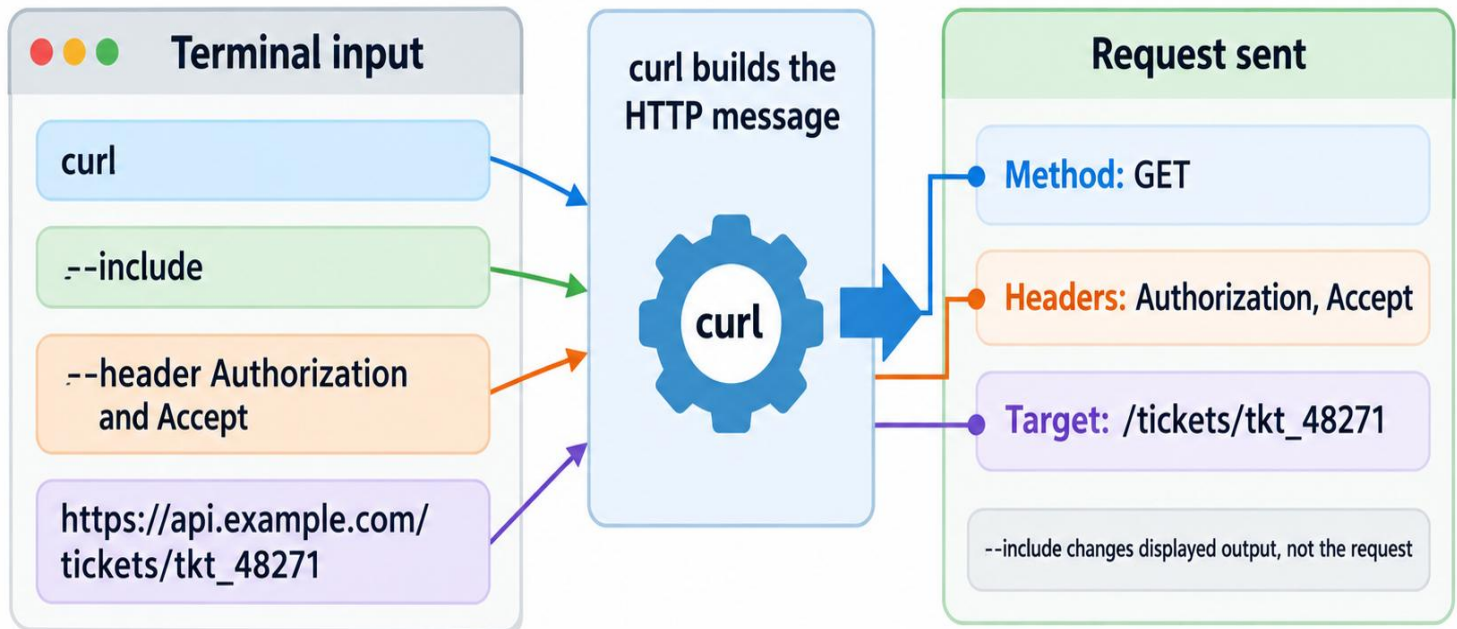
Prerequisites: `curl` is installed, the example API is reachable, and the caller has a valid access token stored in the `ACCESS_TOKEN` environment variable.

Purpose: Retrieve ticket `tkt_48271` and include response headers in the output.

```
curl --include \
  --header "Authorization: Bearer $ACCESS_TOKEN" \
  --header "Accept: application/json" \
  "https://api.example.com/tickets/tkt_48271"
```

Chapter 2: How an HTTP API Exchange Works

2-4 How a curl command becomes an HTTP request



012

Figure 2-5: How a `curl` command becomes an HTTP request

The significant options are:

- `--include` prints response headers as well as the body.
- Each `--header` option adds request metadata.
- The final argument is the target URI.

Expected result: A successful call starts with a success status such as `200 OK`, followed by headers and a JSON ticket representation. A real service may return a defined error instead when the token is invalid, the caller lacks permission, or the ticket does not exist.

This is an environment-specific example. The exact host, credential method, and output format depend on the API being called.

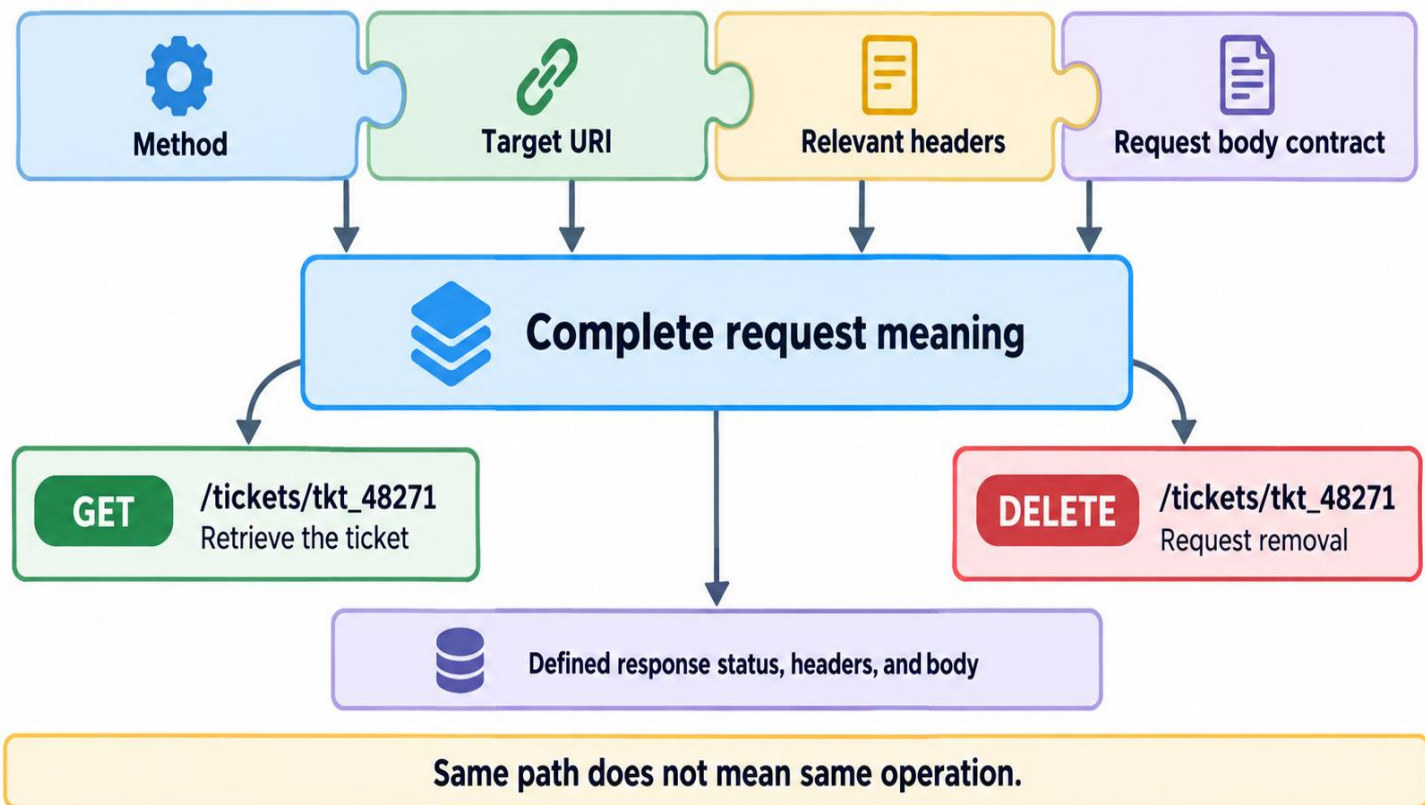
Method, URI, headers, and body work together

A frequent beginner error is to treat the path as the whole operation. These requests have the same path but different meanings:

```
GET /tickets/tkt_48271
DELETE /tickets/tkt_48271
```

Chapter 2: How an HTTP API Exchange Works

2-5 The operation is the combination of method, URI, headers, and body



013

Figure 2-6: The operation is the combination of method, URI, headers, and body

The method changes the intended operation. Headers can also change handling. For example, a **conditional request**, which applies an operation only when a stated condition is true, may tell the server to update a ticket only if it has not changed since the client last read it. The body supplies operation-specific data, such as the new priority.

Therefore, an HTTP operation is identified and understood through the combination of:

- Method.
- Target URI.
- Relevant request headers.
- Request-body contract.
- Defined response status, headers, and body.

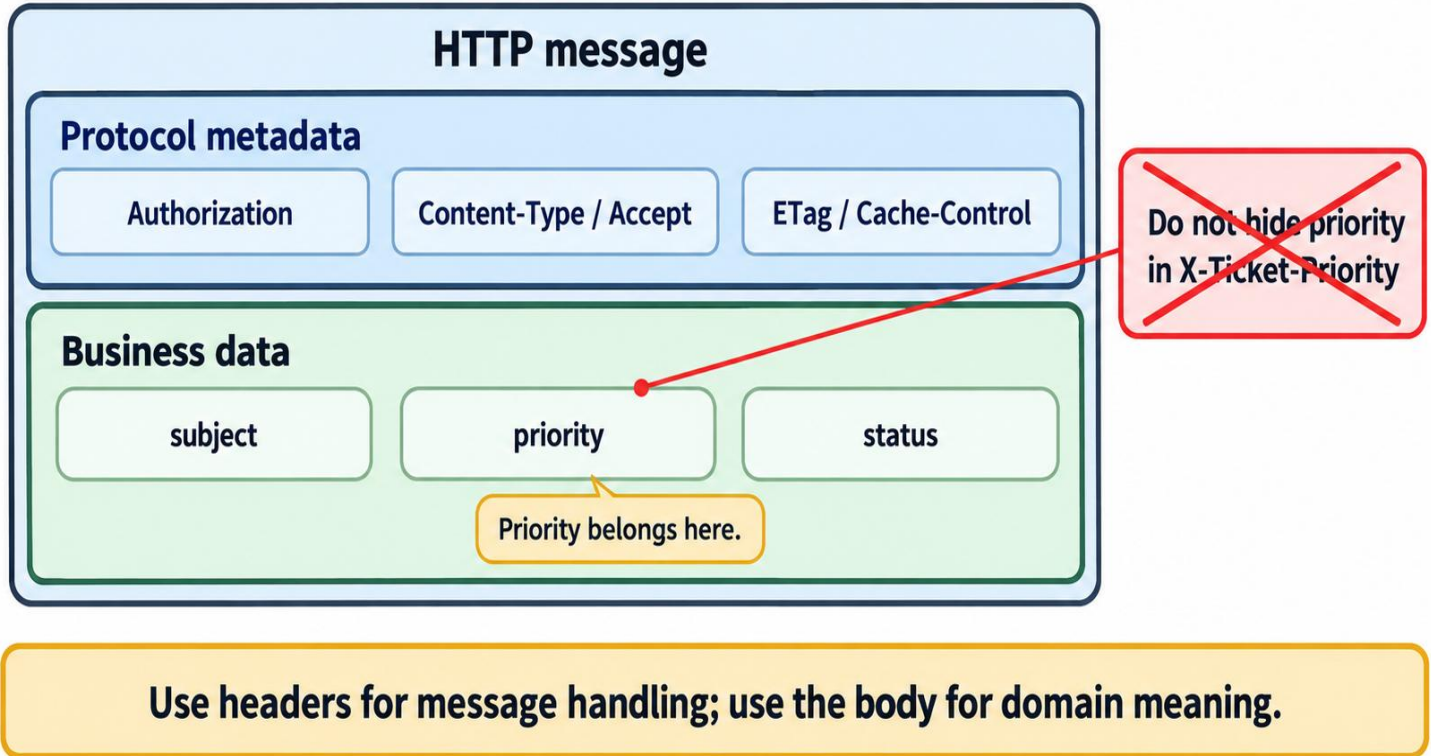
Headers carry message metadata

A **header** is a named metadata field attached to an HTTP request or response. Common examples include:

- **Authorization**, which carries credentials in many API designs.
- **Accept**, which states which response media types the client can process.
- **Content-Type**, which identifies the format of a body that is present.
- **Location**, which may identify a newly created resource.
- **ETag**, which can identify a particular version of a representation.
- **Cache-Control**, which provides instructions for **caching**, meaning reuse of a stored response under defined conditions.

Chapter 2: How an HTTP API Exchange Works

2-6 Protocol metadata versus business data



014

Figure 2-7: Protocol metadata versus business data

Headers should be used when the information describes message handling or protocol behavior. Business data usually belongs in the body or resource model. For example, ticket priority is business data; it should not be hidden in a custom protocol header.

Stateless interactions

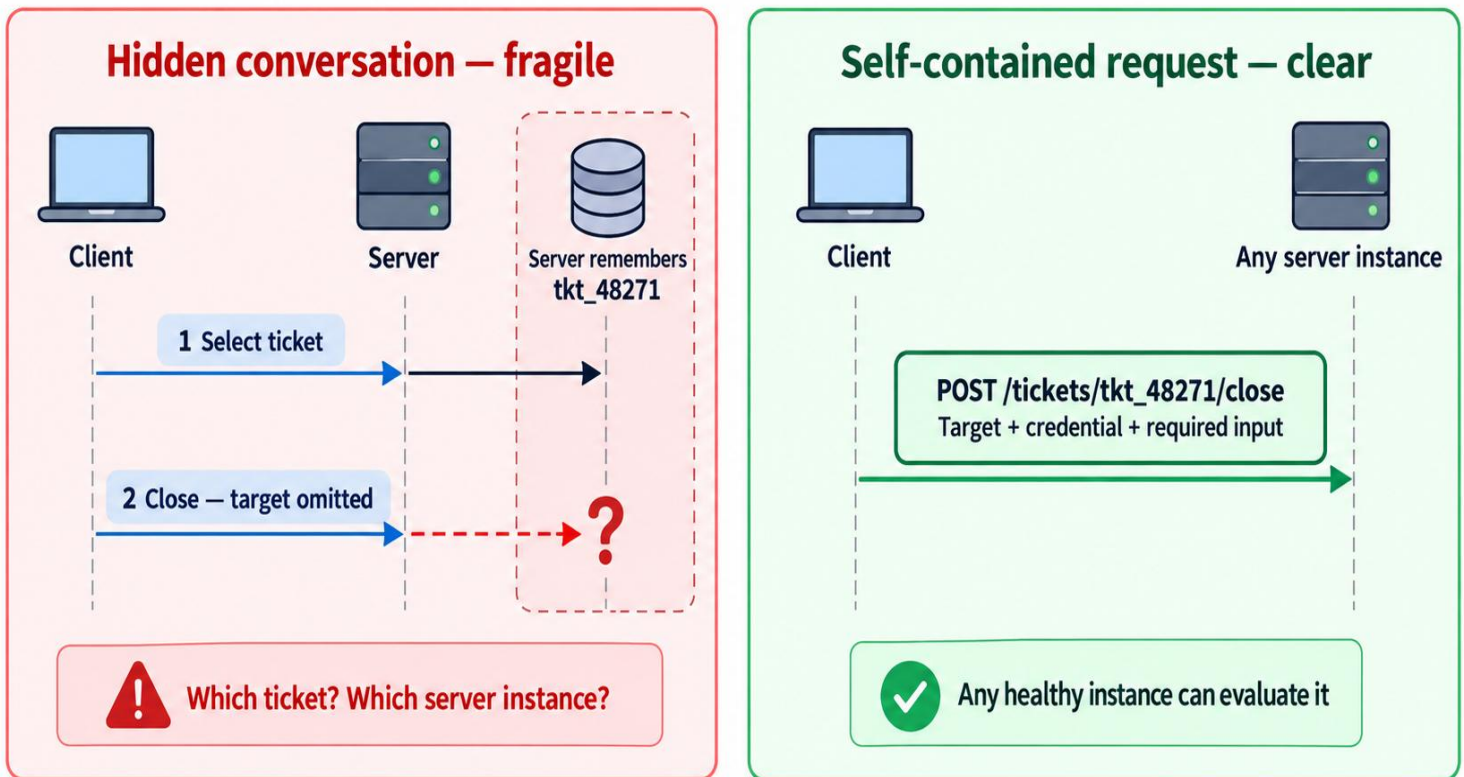
A **stateless interaction** is one in which each request carries the information needed to understand and authorize that request, rather than relying on undocumented conversational context from earlier calls.

Stateless does **not** mean that the server stores no data. The ticket service certainly stores tickets, users, and assignments. It may also maintain authorized sessions. Statelessness means the meaning of a call should not depend on a fragile hidden sequence such as:

1. Call `/select-ticket`.
2. The server silently remembers the selected ticket for this connection.
3. Call `/close` without identifying which ticket to close.

Chapter 2: How an HTTP API Exchange Works

2-7 Stateless request versus hidden conversational state



015

Figure 2-8: Stateless request versus hidden conversational state

A clearer request identifies its target directly:

```
POST /tickets/tkt_48271/close
```

An everyday analogy is submitting a complete form to a service desk. The clerk may have records in a system, but the form still identifies the requester and the case being changed. A note saying only “close the one we discussed before” depends on hidden context and is easy to misinterpret.

Stateless interactions support independent retries, operation across multiple interchangeable server instances, logging, and troubleshooting because each request can be evaluated on its own. They do not eliminate all server-side state.

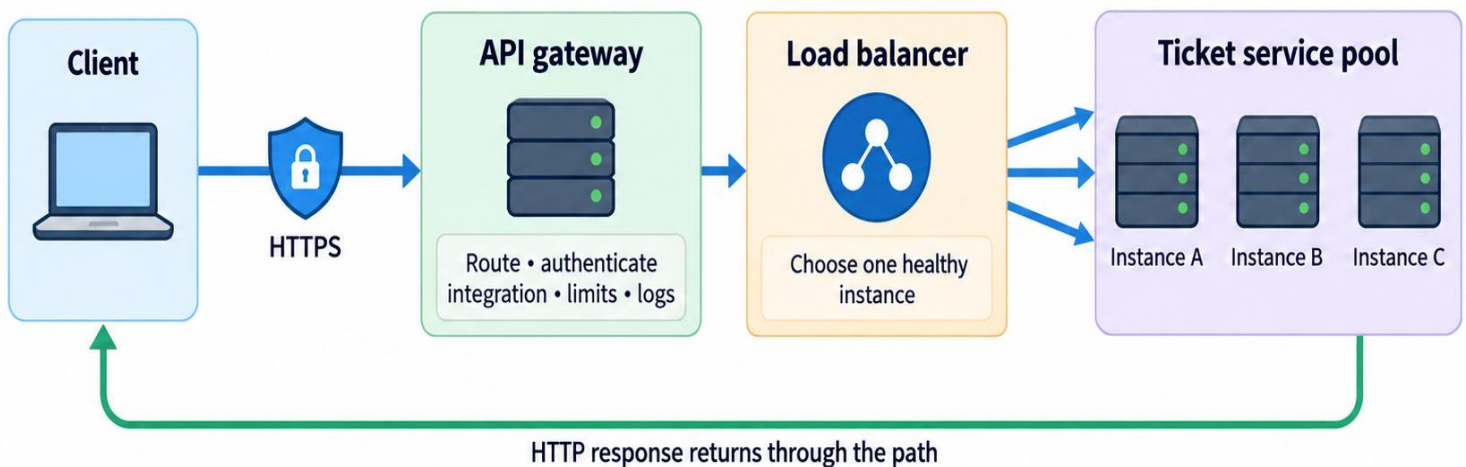
Intermediaries in the path

The client and application server are not always directly connected. Common intermediaries include:

- A **proxy**, which sends requests on behalf of another component.
- An **API gateway**, which provides shared functions such as routing, authentication integration, limits, and logging; **authentication** is the process of establishing the caller's identity.
- A **load balancer**, which distributes traffic across multiple server instances.

Chapter 2: How an HTTP API Exchange Works

2-8 Client-to-service path through gateway and load balancer



Intermediaries also rely on correct HTTP semantics.

016

Figure 2-9: Client-to-service path through gateway and load balancer

These components matter because HTTP semantics are not used only by application code. An intermediary may cache a safe retrieval, retry a connection, reject an oversized request, or route by host and path. Using methods, status codes, and headers consistently gives these components enough information to behave correctly.

A design should not assume that every request travels over one permanent connection to one server instance. That assumption creates hidden state and unreliable behavior in a **distributed system**, where multiple components communicate over a network.

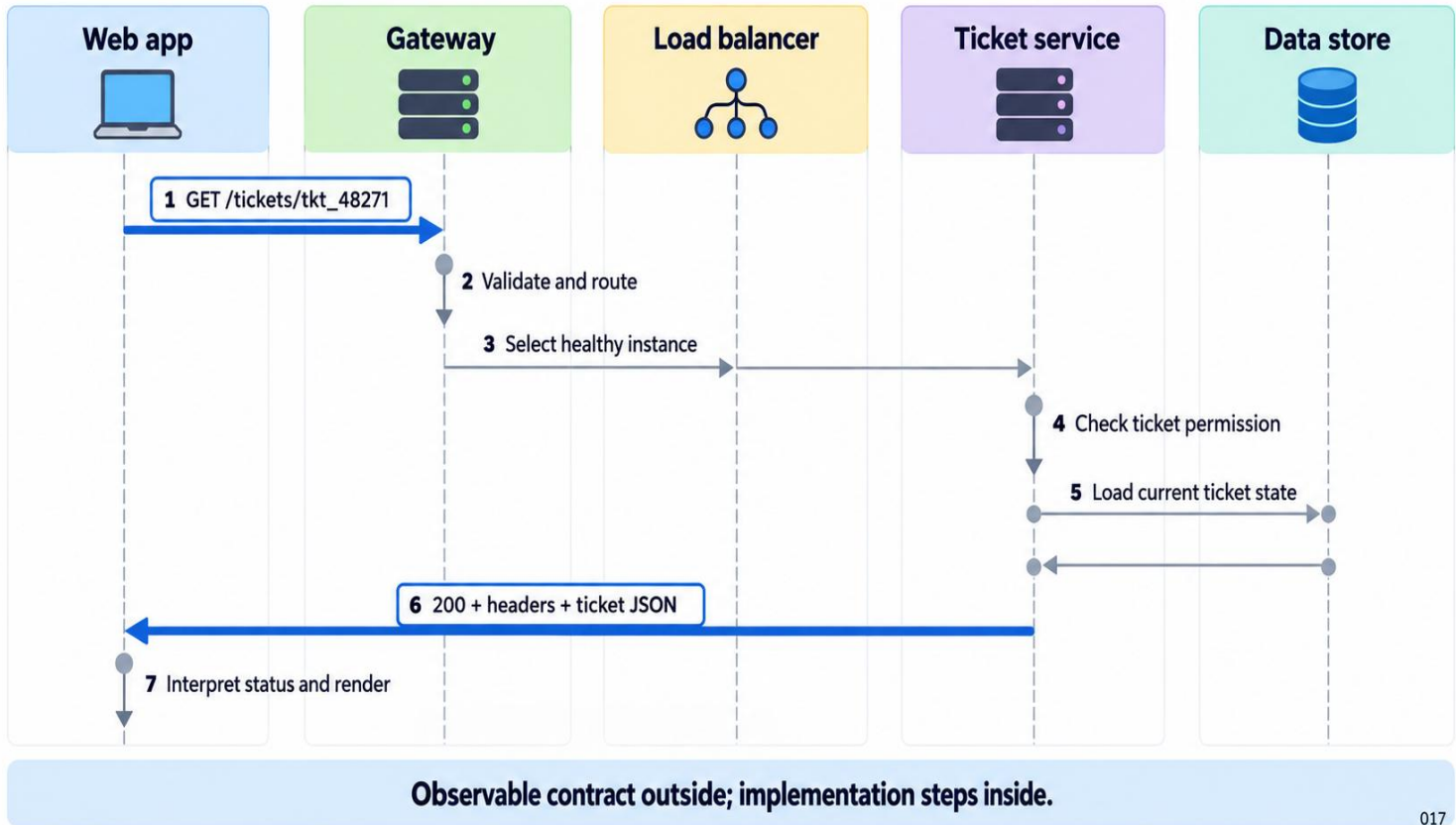
Following one ticket retrieval

Consider `GET /tickets/tkt_48271` from end to end:

1. The web application constructs an HTTPS request with `GET`, the ticket URI, credentials, and an accepted response type.
2. A gateway receives the request and may validate basic credentials and route it to the ticket service.
3. A load balancer selects one healthy service instance.
4. The service checks whether the caller may view the ticket.
5. The service loads the current ticket state and builds the public representation.
6. The service returns a status, headers, and JSON body.
7. The client uses the status to select its control path and the body to render ticket data.

Chapter 2: How an HTTP API Exchange Works

2-9 End-to-end ticket retrieval



017

Figure 2-10: End-to-end ticket retrieval

The API contract concerns the observable request and response. The specific gateway, server instance, database query, and object mapping are implementation and operational details unless they change that observable behavior.

Two common misunderstandings

“The URL alone defines what an HTTP API call does”

The path identifies a target, but the method, headers, and body complete the request semantics. Documentation and tests must describe the whole operation, not only a list of paths.

“Stateless means that the server cannot store any data”

A stateless API can store extensive persistent data. The requirement is that a request be understandable without an undocumented dependency on a previous conversational step or a particular server instance.

A practical reading checklist

When examining any HTTP API call, identify:

- The method and its intended meaning.
- The target URI.
- Headers that affect interpretation or authorization.
- The request body’s media type and schema, if a body exists.
- Every defined response status relevant to the operation.
- Response headers that clients must process.
- The success or error body contract.

An HTTP exchange becomes predictable when every part of the request and response has an explicit role.

Chapter Summary

- An HTTP interaction is a request message followed by a response message.
- The operation is defined by the combination of method, URI, headers, and body rather than by the URL alone.
- Statelessness requires each request to carry the context needed to understand it; it does not prevent the server from storing durable data.

Review Questions

1. Which four parts of a request determine what the client is asking the server to do?
2. What does HTTPS protect, and what does it not decide?
3. Why can an intermediary care about correct HTTP semantics even when it does not understand the business domain?