PROGRAMMING

API Types & Design

HTTP Foundations · REST In Depth & Design Practices

SOAP · RPC-Style APIs & the Road to gRPC

Webhooks · Choosing the Right API Style

Capstone: Designing a Real API

Philip Osztromok

Generated with Claude

Table of Contents

API Types & Design — 9 Chapters

🔌 What Is an API? The Client-Server Request-Response Model

This course assumes no prior API knowledge, so this chapter starts from first principles: what "API" actually means, the client-server roles every API involves, and the basic request/response exchange nearly all of them share. Later chapters get concrete with real protocols and syntax — this one is just the mental model.

The Client-Server Model

Almost every API involves two sides: a **client**, which asks for something, and a **server**, which has the data or logic to answer. Think of a restaurant: you (the client) don't walk into the kitchen and cook your own meal — you tell a waiter what you want, and the kitchen (the server) prepares it and sends it back out. You never need to know how the kitchen works internally; you just need to know how to place an order and what to expect back.

Client

Whatever **initiates** the request — a browser, a mobile app, a script, or even another server. The client wants something: data, an action performed, or both.

Server

Whatever **holds the data or logic** the client wants, and knows how to respond to a properly formed request. One server can serve many different clients at once.

What "API" Actually Means

API stands for **Application Programming Interface** — a defined way for one piece of software to ask another for something, without needing to know how that other piece works internally. The key word is *interface*: it's a contract. As long as both sides honor the contract, the client doesn't care whether the server is written in Python, Go, or PHP, and the server doesn't care whether the client is a phone app or a script running on someone's laptop.

A Person Using a Website vs. a Program Using an API

> **Human via Browser (UI)**
>
> A person fills in a search box, clicks a button, and reads the results as rendered text and images on a page — built for *human* eyes.

> **Program via API**
>
> A program sends the same underlying search request in a structured format and gets back structured data it can parse — built for *another program* to consume, not for a human to read directly.

Both might hit the exact same underlying search feature — the difference is who's on the other end of the request, and what shape the response takes.

Request/Response: The Basic Exchange

Most APIs — though not all, as later chapters cover — follow the same basic pattern: the client sends a **request** describing what it wants, the server does whatever work is needed, and sends back a **response**. Stripped of any specific technology, a request/response exchange might look conceptually like this:

A Generic Request/Response Exchange

```
// Client sends a request:
"give me today's weather for London"
// Server sends back a response:
{ "city": "London", "tempC": 14, "conditions": "cloudy" }
```

That's the whole shape of it, conceptually. Chapter 2 makes this concrete with HTTP — the actual protocol most web APIs use to carry requests and responses across a network.

Not All APIs Talk Over a Network

"API" is actually a broader term than just "web API." When you call a function from a JavaScript library, or use the browser's built-in `fetch()` function, you're using a **library API** — an interface to code running in the *same* program, with no network involved at all. This course is specifically

about **web APIs**: interfaces for communication *between separate systems*, usually over a network, usually between a client and a server that are entirely different programs — possibly written by different teams, running on different machines, in different parts of the world.

Library API (not this course's focus)

Function signatures and classes exposed by a piece of code running inside your own program — e.g. the DOM API, or a Python library's public functions. No network call happens.

Web API (this course's focus)

A separate system — usually a server somewhere else — that your program talks to over a network, sending a request and getting back a response. This is what REST, SOAP, gRPC, and GraphQL all are.

🗺️ Where This Course Fits

If you've already come across **GraphQL** or **WebSockets** elsewhere on this site, both are web APIs too — just different styles, each with their own full course already. This course covers the broader family of request/response-style web APIs (REST, SOAP, RPC-style APIs, and Webhooks), and Chapter 8 closes with a direct comparison across all of them — including where GraphQL and WebSockets fit — so you can choose the right style for a given job instead of defaulting to whichever one you've heard of most.

API Style	One-Line Description	Where It's Covered
REST	Resource-based request/response, almost always over HTTP	This course, Chapters 3–4
SOAP	XML-based, strict formal contracts (WSDL)	This course, Chapter 5
RPC-style (JSON-RPC, gRPC)	"Call this specific function" rather than "fetch this resource"	This course, Chapter 6 — gRPC has its own full course too
GraphQL	Client asks for the exact shape of data it wants back	Already covered — its own course
WebSockets	A persistent, two-way connection — not really request/response at all	Already covered — its own course
Webhooks	The server calls *you* when something happens, instead of you asking it	This course, Chapter 7

💻 Coding Challenges

Challenge 1: Identify Client and Server

For each of the following, say which side is the client and which is the server: (a) a weather app on your phone showing today's forecast, (b) a browser loading a news website, (c) a smart thermostat reporting the current temperature to an app on your phone.

Goal: Practice spotting the client/server roles in everyday situations, not just abstract diagrams.

→ Solution

Challenge 2: Spot the API in Everyday Life

Pick one app or website you use regularly. Describe one moment where it's very likely making a web API call behind the scenes, and sketch — in plain words, no syntax needed — roughly what the request might ask for and what the response might contain.

Goal: Practice recognizing that a request/response exchange is happening even when it's invisible to the user.

→ Solution

Challenge 3: Library API or Web API?

Classify each of the following as a "library API" or a "web API," and justify each answer in one sentence: (a) calling `Array.map()` in JavaScript, (b) a weather app fetching a forecast from a remote server, (c) using Python's built-in `len()` function, (d) a mobile game submitting a high score to a leaderboard server.

Goal: Practice telling apart the two meanings of "API" this chapter distinguished.

→ Solution

⚠️ Gotcha: Assuming "API" Always Means "Web API"

In casual conversation, "API" gets used for almost anything — "the DOM API," "the Fetch API," "our internal API" — and not all of those involve a network at all. When someone says "call the API" in a professional context, they almost always mean a *web* API: a request going out to a separate server. But "the Array API" or "the DOM API" refers to functions built into a language or browser, running entirely inside your own program. Mixing these up is a common beginner confusion — when in doubt, ask whether a network request is actually involved.

🎯 What's Next

With client/server roles and the request/response idea established conceptually, the next chapter gets concrete: **HTTP as the Foundation of Most APIs** — the actual methods, status codes, and headers that carry a real request and response across a network.

🌐 HTTP as the Foundation of Most APIs

Chapter 1 described the request/response idea in the abstract — a client asks, a server answers. This chapter makes it concrete: HTTP is the actual protocol that carries almost every web API's requests and responses, and it's worth learning properly here, once, since REST, SOAP, GraphQL, and webhooks all ride on top of it (only WebSockets steps away from plain request/response after its initial handshake, and gRPC uses HTTP/2 more directly — both get their own treatment later).

What HTTP Actually Is

HTTP (HyperText Transfer Protocol) was originally built for fetching web pages, but the same request/response mechanics — a method, a target, some headers, an optional body, and a status code back — turned out to be a perfectly good general-purpose way for any client and server to exchange structured data, not just HTML pages. That reuse is why it became the default transport for web APIs rather than something purpose-built.

HTTP Methods (Verbs)

Every HTTP request declares a **method** — a verb describing the kind of action being requested. The protocol itself doesn't enforce what each verb "means"; it's a widely followed convention that well-designed APIs stick to.

Method	Conventional Meaning	Has a Body?	Idempotent?
GET	Retrieve data, no side effects	No	Yes
POST	Create something new, or trigger an action	Yes	No
PUT	Replace a resource entirely with what's sent	Yes	Yes
PATCH	Partially update a resource	Yes	No (usually)
DELETE	Remove a resource	Usually not	Yes

Idempotent means: calling it once has the same end result as calling it five times in a row. `GET`ting the same resource repeatedly doesn't change anything, and `PUT`ting the exact same replacement data five times leaves the resource in the same final state as doing it once. `POST` is the odd one out — posting "create a new order" five times typically creates five separate orders, which is exactly why it's not idempotent. This distinction matters more than it might seem — Chapter 4 covers why idempotency is something real API design has to deliberately account for, especially when a client might retry a request after a dropped connection.

Anatomy of a Request

A Real GET Request

```
GET /users/42 HTTP/1.1
Host: api.example.com
Authorization: Bearer abc123
Accept: application/json
```

A Real POST Request (With a Body)

```
POST /users HTTP/1.1
Host: api.example.com
Content-Type: application/json
Content-Length: 47

{ "name": "Alice", "email": "alice@example.com" }
```

Every request has three parts: a **method + path** (what's being asked for, and how), a set of **headers** (metadata about the request), and — for methods like `POST`/`PUT`/`PATCH` — an optional **body** carrying the actual data being sent.

Status Codes: How the Server Reports Back

Every response starts with a three-digit **status code**, grouped into five classes by its first digit:

Range	Class	Common Examples
2xx	Success	200 OK, 201 Created, 204 No Content
3xx	Redirection	301 Moved Permanently, 304 Not Modified
4xx	Client Error — you did something wrong	400 Bad Request, 401 Unauthorized, 404 Not Found, 429 Too Many Requests
5xx	Server Error — it did something wrong	500 Internal Server Error, 503 Service Unavailable

The single most useful habit for reading an unfamiliar API's behavior is checking which *class* a status code falls into before worrying about the exact number — a 4xx means "fix your request," a 5xx means "the problem is on their end, maybe retry later."

Headers: Metadata Riding Alongside the Data

Headers are key-value pairs that travel with a request or response but sit outside the body — metadata *about* the exchange, rather than the actual content. `Content-Type` tells the receiving side how to parse the body (e.g. `application/json`), `Authorization` carries credentials, and `Accept` tells the server what response format the client can handle. If you want to see real headers in action, the Web Developer Tools course's Network panel chapter (`wdt_lesson_04`) shows exactly how to inspect every header on a live request in your browser — and the HTTPS/TLS course covers how headers like `Authorization` stay protected in transit via encryption.

Headers vs. Body — Different Jobs

Headers

Metadata: content type, auth token, caching rules, what formats are acceptable. Small, always present in some form.

Body

The actual payload: the JSON, XML, or file being sent or returned. Optional — many requests (like a plain `GET`) have none at all.

Putting It Together: A Full Exchange

Chapter 1's Weather Example, Made Concrete

```
// Request
GET /weather?city=London HTTP/1.1
Host: api.weatherexample.com
Accept: application/json

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

{ "city": "London", "tempC": 14, "conditions": "cloudy" }
```

💻 Coding Challenges

Challenge 1: Classify HTTP Methods

Match each action to the HTTP method that best fits it: (a) fetch a user's profile, (b) create a new order, (c) update just the email field of an existing user, (d) delete a comment, (e) replace an entire user record with new data.

Goal: Practice mapping real actions onto the conventional meaning of each verb, including the PUT/PATCH distinction.

→ Solution

Challenge 2: Read a Status Code

For each of these status codes — 201, 404, 429, 500 — explain in plain language what it tells the client, and what the client should probably do next in each case.

Goal: Practice treating status codes as actionable information, not just numbers to memorize.

→ Solution

Challenge 3: Spot the Missing Header

A client sends a `POST` request with a JSON body but no `Content-Type` header at all. Explain why this can cause problems on the server side, and what header should be added and set to what value.

Goal: Practice recognizing that headers aren't optional decoration — servers often depend on them to correctly parse the body.

→ Solution

⚠️ Gotcha: Using PUT When You Actually Mean PATCH

A very common real-world mistake: an API documents an endpoint as `PUT /users/42` for "updating a user," but the implementation only expects the fields being changed — not the full record. Genuine `PUT` semantics mean the entire resource is *replaced* with whatever's sent; if a client sends only `{ "email": "new@example.com" }` to a true `PUT` endpoint, a strictly correct server would wipe out every other field not included. Many APIs quietly use `PUT` to mean "partial update" anyway, which works until a client follows the spec literally and something else gets accidentally erased. If partial updates are the real intent, use `PATCH` — that's exactly what it's for.

🎯 What's Next

With HTTP's building blocks in place, the next chapter goes deeper into **REST APIs In Depth** — the architectural constraints (statelessness, uniform interface, HATEOAS) and resource-modeling philosophy that turn plain HTTP into a coherent, well-designed API style.

🏛️ REST APIs In Depth

Chapter 2 covered HTTP's raw building blocks — methods, status codes, headers. REST is what turns those building blocks into a coherent architectural style. It comes from Roy Fielding's year-2000 doctoral dissertation, and it's a set of *constraints*, not a format you can validate a request against — which is both REST's greatest flexibility and, as this chapter is honest about, its biggest practical weak spot.

What REST Actually Stands For

REST stands for **RE**presentational **S**tate **T**ransfer. Unpacked: a client interacts with a *representation* of some piece of server-side state (a JSON object representing a user, say), and every request/response transfers that representation back and forth. Unlike SOAP (Chapter 5), which has a formal contract language (WSDL) a request can be validated against, there is no "REST specification" — just a style. That means two APIs can both honestly call themselves "RESTful" while looking fairly different from each other.

Resources: Nouns, Not Verbs

The core idea: REST organizes an API around **resources** — nouns, like `users`, `orders`, or `products` — rather than baking actions into the URL itself. The HTTP method (from Chapter 2) supplies the verb; the URL supplies the noun.

RPC-Style Naming vs. REST Resource Naming

Verb-in-the-URL (not REST)

```
POST /getUserData?id=42
POST /createOrder
POST /deleteComment?id=9
```

Resource + Method (REST)

```
GET /users/42
POST /orders
DELETE /comments/9
```

Notice the RPC-style column crams the verb into the path itself (`getUserData`, `createOrder`) and defaults everything to `POST`. The REST column lets the URL identify *what* and the HTTP method say *what to do with it* — which is exactly why Chapter 2's method table matters so much here.

The Architectural Constraints

Fielding's dissertation defines REST through a handful of constraints an API should follow. Client-server separation was already covered in Chapter 1 — here are the rest:

Constraint	What It Means	Practical Example
Statelessness	Each request carries everything needed to understand it; the server keeps no memory of prior requests	Every request includes its own auth token, not a server-side session lookup
Cacheability	Responses must say whether they can be cached	`Cache-Control` headers marking a response as cacheable or not
Uniform Interface	Resources identified by URL, manipulated via representations, self-descriptive messages, hypermedia-driven (HATEOAS)	A consistent `/resource/id` shape across the whole API
Layered System	The client can't necessarily tell if it's talking to the origin server directly or through intermediaries	A load balancer or caching proxy sitting transparently in front of the real server
Code-on-Demand (optional)	The server can optionally send executable logic to the client	Rarely used in modern REST APIs — mentioned for completeness

Statelessness in Practice

This is the constraint with the biggest everyday impact. An older, stateful web app might log a user in once and store their identity server-side in a session, referenced by a cookie on every later request. A stateless REST API instead expects *every single request* to prove who's asking, independently:

Two Stateless Requests — Each Self-Contained

```
// Request 1
GET /orders/101 HTTP/1.1
Authorization: Bearer abc123

// Request 2 — no memory of Request 1, so the token is sent again
GET /orders/102 HTTP/1.1
Authorization: Bearer abc123
```

The payoff: any server behind a load balancer can handle any request, since none of them depend on a specific server "remembering" a client from before — which is exactly why statelessness makes REST APIs so much easier to scale horizontally.

HATEOAS: The Constraint Everyone Skips

HATEOAS (Hypermedia As The Engine Of Application State) says a response shouldn't just return raw data — it should include links describing what the client can do *next*, so the client doesn't need to hard-code every related URL in advance.

Plain JSON vs. Hypermedia-Driven JSON

Without HATEOAS

```
{ "id": 101, "status": "pending" }
```

The client has to already know the URL for cancelling this order.

With HATEOAS

```
{ "id": 101, "status": "pending", "links": { "self": "/orders/101", "cancel": "/orders/101/cancel" } }
```

The response itself tells the client what's possible next.

Here's the honest part: most real-world "RESTful" APIs skip HATEOAS almost entirely — clients are usually hard-coded against documented URLs rather than following links dynamically. Strictly speaking, an API missing HATEOAS isn't fully REST by Fielding's own definition; in practice, the industry mostly calls "HTTP + JSON, organized around resources" REST anyway, HATEOAS or not.

Why REST Became the Default

Riding on plain HTTP means every browser, script, and tool already speaks it natively — no special client library required. Statelessness makes horizontal scaling straightforward. JSON is human-readable and easy to debug. And HTTP's own caching mechanisms can be reused rather than reinvented. Chapter 4 turns these architectural ideas into the concrete design patterns real REST APIs use day to day — versioning, pagination, and more.

💻 Coding Challenges

Challenge 1: Nouns Not Verbs

Rewrite each of these as a proper REST resource + HTTP method pair: `GET /getUser?id=5`, `POST /createOrder`, `POST /deleteComment?id=9`.

Goal: Practice separating the "noun" (the URL) from the "verb" (the HTTP method).

→ Solution

Challenge 2: Identify a Statelessness Violation

A server stores "the user's last search query" in a server-side session, and a later request just says `GET /search/repeat` expecting the server to remember it. Explain why this violates REST's statelessness constraint, and how you'd redesign the request to fix it.

Goal: Practice recognizing a stateful design masquerading as REST.

→ Solution

Challenge 3: Add HATEOAS Links

Given `{ "id": 55, "status": "shipped" }` for an order resource, add a plausible `links` object with at least two hypermedia links a client might need next.

Goal: Practice designing a response that describes its own next steps.

→ Solution

⚠️ Gotcha: Calling Anything With JSON Over HTTP "RESTful"

It's extremely common to hear "REST API" used to mean nothing more than "an HTTP API that returns JSON" — with no statelessness guarantee, no resource-based URL design, and no hypermedia at all. That's not necessarily wrong in casual usage, but it's worth knowing the difference: an API that stores session state server-side, uses verb-heavy URLs like `/processOrder`, and never includes hypermedia links is, strictly speaking, an HTTP API following some REST-ish conventions — not a fully Fielding-compliant REST API. Knowing which constraints an API actually honors (and which it's skipped) matters more than whether it wears the "REST" label.

🎯 What's Next

The next chapter, **REST API Design Practices**, gets practical: versioning strategies, pagination, filtering and sorting query parameters, idempotency in real designs, and rate limiting — the day-to-day patterns that turn this chapter's architectural ideas into a real, usable API.

🛠️ REST API Design Practices

Chapter 3 covered REST's architectural theory. This chapter is the practical companion — the concrete decisions every real REST API has to make. Several of these have already shown up as brief asides in the framework courses on this site (Express, FastAPI, Rails, Laravel, Django); this is the first time they get a proper, dedicated treatment on their own.

Versioning Strategies

APIs change over time, but existing clients can't be forced to update the instant something breaking ships. Versioning lets old and new behavior coexist.

Strategy	Example	Pros / Cons
URL versioning	`/v1/users`, `/v2/users`	Most visible and simplest to reason about; "impure" since a resource's URL technically shouldn't change just because its representation did
Header versioning	`Accept: application/vnd.api+json;version=2`	Keeps the URL stable and is arguably "purer" REST; much less visible/discoverable to developers browsing the API
Query parameter versioning	`/users?version=2`	Easy to add on top of an existing unversioned API; easy to forget or omit accidentally

URL versioning is by far the most common in practice — it's the most visible option, and visibility usually wins over architectural purity.

Pagination

Returning every row of a 2-million-row table in one response isn't practical for the server or the client. Two common patterns solve this differently:

Offset/Limit vs. Cursor-Based Pagination

Offset/Limit (Page-Based)

`GET /products?offset=20&limit=20`

Simple to implement and reason about, but if items are added or removed while a client is paging through results, rows can be skipped or duplicated between pages.

Cursor-Based

`GET /products?after=abc123&limit=20`

The cursor points to a specific position in the data itself (not just a numeric offset), staying stable even as the underlying data changes — the pattern used by production APIs like Stripe and GitHub.

Filtering & Sorting via Query Parameters

Query parameters are the conventional place for narrowing down or reordering a resource collection:

A Filtered, Sorted Request

```
GET /orders?status=shipped&sort=-createdAt&limit=10
```

Reads as: "give me `orders` where `status` is `shipped`, sorted by `createdAt` descending (the leading `-` conventionally means descending), limited to 10 results."

Idempotency in Practice

Chapter 2 introduced idempotency as a property of HTTP methods. Here's where it becomes a real, practical problem: a client sends `POST /payments`, the connection times out before a response arrives, and the client — reasonably — retries. Did the payment actually go through the first time? If it did, the retry risks creating a *second* charge, since `POST` isn't idempotent by design.

The common real-world fix is an **Idempotency-Key** header: the client generates a unique key (typically a UUID) once, and sends the *same* key on both the original request and any retry:

An Idempotency-Key in Action

```
POST /payments HTTP/1.1
Idempotency-Key: 6f9a1e2c-...
Content-Type: application/json

{ "amount": 4999, "currency": "usd" }
```

The server remembers which idempotency keys it has already processed; if the same key arrives again, it returns the original result instead of creating a second payment. This is exactly the pattern Stripe's API popularized.

Rate Limiting

Rate limiting caps how many requests a client can make in a given time window, protecting server infrastructure and keeping usage fair across clients. Rate-limited APIs conventionally return the `429 Too Many Requests` status introduced in Chapter 2, along with headers describing the current state:

Header	Meaning
`X-RateLimit-Limit`	The maximum requests allowed in the current window
`X-RateLimit-Remaining`	How many requests are left in the current window
`X-RateLimit-Reset`	When the window resets and the count starts over

Rate limiting also plays a security role, not just a capacity-management one — the Authentication & Session Security course's login-attacks chapter (`bc1-3`) covers rate limiting specifically as a brute-force defense, the same underlying mechanism applied to a different problem.

💻 Coding Challenges

Challenge 1: Choose a Versioning Strategy

Recommend a versioning strategy for (a) a public API with thousands of external consumers who don't read documentation closely, and (b) an internal API used only by one team who reviews every deploy together. Justify each choice.

Goal: Practice matching a design decision to its actual audience rather than picking one "correct" universal answer.

→ Solution

Challenge 2: Design a Paginated Response

A `/products` resource has 10,000 items and is updated constantly (items added/removed throughout the day). Design the query parameters and response shape for cursor-based pagination through it, and explain why cursor-based fits better than offset/limit here.

Goal: Practice applying the offset-vs-cursor tradeoff to a concrete, changing dataset.

→ Solution

Challenge 3: Fix a Non-Idempotent Retry Bug

A client's `POST /payments` request times out, the client retries automatically, and the customer ends up charged twice. Propose a fix using this chapter's Idempotency-Key pattern, describing what the client and server each need to do.

Goal: Practice applying the idempotency-key pattern to a real double-charge scenario.

→ Solution

⚠️ Gotcha: Offset Pagination Skipping or Duplicating Rows

Offset/limit pagination looks harmless until data changes mid-pagination. Say a client fetches `?offset=0&limit=20`, then a new row is inserted at the very front of the dataset, and the client fetches `?offset=20&limit=20` next — that new insertion just shifted everything over by one position, so the client's second page now starts one row too early, silently **duplicating** the last row of page one. The reverse happens with a deletion near the front: a row gets silently **skipped** entirely. Cursor-based pagination avoids this specific failure mode because the cursor tracks a stable position in the data itself, not a numeric offset that shifts every time the underlying rows change.

🎯 What's Next

The next chapter steps away from REST entirely to cover **SOAP APIs** — XML envelopes, WSDL contracts, and why plenty of enterprise, banking, and government systems still rely on this older, stricter style.

📜 SOAP APIs

Chapters 3–4 covered REST in depth. This chapter steps back to an older, stricter style that predates REST's popularity and still quietly runs large parts of enterprise, banking, and government infrastructure today: SOAP. It's worth understanding on its own terms rather than dismissing as "REST but worse" — it solves a genuinely different problem.

What SOAP Actually Is

SOAP stands for **S**imple **O**bject **A**ccess **P**rotocol — somewhat ironically, given it's rarely described as "simple" in practice. Unlike REST, which is an architectural *style* with no formal specification (Chapter 3), SOAP is an actual **protocol**: a strictly defined XML-based message format, standardized by the W3C, most commonly sent over HTTP but technically transport-agnostic (it can run over SMTP or other transports too). It originated at Microsoft and IBM in the late 1990s, aimed squarely at enterprise systems that needed strict contracts and strong typing more than they needed lightweight flexibility.

The SOAP Envelope

Every SOAP message — request or response — is wrapped in an **Envelope**, containing an optional **Header** (metadata like authentication or session tokens) and a required **Body** (the actual request or response payload), all expressed as XML:

A SOAP Request: GetWeather

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Header>
  <AuthToken>abc123</AuthToken>
 </soap:Header>
 <soap:Body>
  <GetWeather>
   <City>London</City>
  </GetWeather>
 </soap:Body>
</soap:Envelope>
```

Compare that to Chapter 2's plain JSON weather example — the same underlying idea (ask for London's weather), but SOAP wraps it in a much more formal, verbose structure, with the envelope/header/body pattern present on *every single message*, request or response.

WSDL: The Formal Contract

WSDL (Web Services Description Language) is where SOAP's real strength lies: an XML document describing *exactly* what operations a service offers, what parameters each one takes, what types they are, and what it returns — a strict, machine-readable contract. Tools can read a WSDL file and auto-generate fully-typed client code in languages like Java or C#, catching mismatches (wrong type, missing field) at code-generation time rather than discovering them as a runtime surprise.

No Formal Contract vs. a Formal Contract

REST

No enforced contract — documentation might describe the shape of a request/response, but nothing stops a server from silently changing a field's type or a client from sending malformed data. (OpenAPI/Swagger, covered in `express2-7`, adds documentation on top, but it's optional and not enforced by the protocol itself.)

SOAP + WSDL

The WSDL contract is the source of truth: exact operation names, exact parameter types, exact return shapes — generated client/server code is built directly from it, so mismatches tend to surface immediately rather than silently in production.

SOAP Faults: Error Handling the SOAP Way

SOAP doesn't lean on HTTP status codes the way REST does (Chapter 2) — errors are represented inside the XML body itself, as a **SOAP Fault**:

A SOAP Fault

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
  <soap:Fault>
   <faultcode>soap:Client</faultcode>
   <faultstring>City not found</faultstring>
  </soap:Fault>
 </soap:Body>
</soap:Envelope>
```

Why Enterprise, Banking, and Government Systems Still Use SOAP

This isn't just legacy inertia (though that's certainly part of it — enormous amounts of core banking and government software were already built on SOAP long before REST was popular, and rewriting it is expensive and risky). SOAP also has genuine, deliberate advantages for certain domains:

WS-Security

A standard for message-level security — parts of the message itself can be encrypted or signed independently, rather than relying solely on transport-level encryption (contrast with the HTTPS/TLS course's transport-only approach).

Reliable Messaging & Transactions

Standards like WS-ReliableMessaging and WS-AtomicTransaction give formal guarantees about exactly-once delivery and multi-step transactional consistency — exactly what a bank-to-bank wire transfer needs, and something REST has no built-in equivalent for.

SOAP vs REST: The Real Tradeoffs

Aspect	SOAP	REST
Contract	Formal, enforced (WSDL)	Informal, optional (OpenAPI/Swagger)
Message format	XML only	Usually JSON, but flexible
Payload size	Heavier, more verbose	Lighter
Built-in security/transactions	WS-Security, WS-*ReliableMessaging/Transaction standards	None built in — handled at the application layer
Best fit	Enterprise/banking/government systems needing strict guarantees	Public web/mobile APIs prioritizing simplicity and flexibility

💻 Coding Challenges

Challenge 1: Read a SOAP Envelope

Given this chapter's `GetWeather` SOAP request example, identify what's in the Header versus the Body, and state exactly what operation is being invoked.

Goal: Practice reading a SOAP envelope's structure without getting lost in the XML verbosity.

→ Solution

Challenge 2: WSDL vs No Contract

A developer accidentally sends a string where an integer is expected. Explain what a WSDL-generated SOAP client would do differently from a plain, undocumented REST call in this situation, and at what point each would catch the mistake.

Goal: Practice explaining WHY a formal contract changes WHEN an error surfaces, not just THAT it does.

→ Solution

Challenge 3: SOAP or REST?

Recommend SOAP or REST for (a) a bank-to-bank wire transfer system requiring guaranteed-exactly-once delivery and formal contracts, and (b) a public weather API meant to be consumed easily by mobile app developers. Justify each choice.

Goal: Practice applying this chapter's tradeoff table to concrete, different scenarios.

→ Solution

⚠️ Gotcha: A SOAP Fault Can Arrive With HTTP 200 OK

Coming from REST, it's natural to assume "if the status code says success, the request succeeded." SOAP doesn't always work that way — because SOAP puts errors inside the XML body as a `soap:Fault` rather than relying on the HTTP status code, some SOAP implementations return a perfectly normal `HTTP 200 OK` even when the actual operation failed, with the failure only visible if you parse the body and check for a `Fault` element. Code that only checks the HTTP status code and assumes success can completely miss a genuine application-level error. Always parse the body of a SOAP response and check for a fault, regardless of what the HTTP status line says.

🎯 What's Next

The next chapter covers **RPC-Style APIs: JSON-RPC, XML-RPC, and the Road to gRPC** — simpler alternatives to SOAP that skip the heavy envelope/WSDL machinery while keeping the "call a specific function" mental model, and how that idea evolves into gRPC (which has its own full course on this site).

📞 RPC-Style APIs: JSON-RPC, XML-RPC, and the Road to gRPC

REST (Chapter 3) models an API around resources — nouns. SOAP (Chapter 5) is technically an RPC-style protocol too, just wrapped in a lot of formal ceremony. This chapter covers the lighter-weight RPC styles that skip SOAP's envelope-and-WSDL machinery while keeping its core idea — "call a specific function, with these arguments, and get a result back" — and traces how that same idea evolves into gRPC, which gets its own full course on this site.

The RPC Mental Model: Call a Function, Not a Resource

RPC stands for **R**emote **P**rocedure **C**all. Instead of asking "give me the `user` resource with id 42" (REST's noun-first thinking), an RPC-style API asks "call the `getUser` procedure, passing 42 as an argument" — a verb-first, function-call mental model.

Resource Call vs. Method Call — Same Operation

REST (resource-oriented)

`GET /users/42`

The URL names a resource; the HTTP method names the action.

RPC (method-oriented)

`getUser(42)`

The name of the call itself IS the action — there's no separate "resource" being addressed.

JSON-RPC: A Lightweight Spec

JSON-RPC is a small, precisely defined specification (currently version 2.0) for making RPC calls using plain JSON. Every request names a `method`, supplies `params`, and includes an `id` used to match the eventual response back to this specific request — important once multiple calls might be in flight or batched together:

A JSON-RPC 2.0 Request/Response Pair

```
// Request
{
 "jsonrpc": "2.0",
 "method": "subtract",
 "params": [42, 23],
 "id": 1
}

// Response
{
 "jsonrpc": "2.0",
 "result": 19,
 "id": 1
}
```

Notice there's just one endpoint conceptually — every call typically goes to the same URL, with the `method` field itself saying what to actually do. That's a real structural difference from REST, where the URL path is usually what distinguishes one operation from another.

XML-RPC: The Historical Predecessor

XML-RPC came first — dating to 1998, it's the same "call a named method with arguments" idea as JSON-RPC, just expressed in XML rather than JSON. It's notably simpler than SOAP (no WSDL, no elaborate envelope with headers), but SOAP itself was actually built as an extension of XML-RPC's original ideas. Today, JSON-RPC has largely displaced it for new development — JSON parses natively in JavaScript and most modern languages, while XML needs an extra parsing step. XML-RPC mostly survives now in older systems that predate JSON-RPC's rise.

Where JSON-RPC Actually Shows Up Today

JSON-RPC isn't just historical trivia — it's genuinely in active use. Ethereum and many other blockchain networks expose their node APIs as JSON-RPC. Closer to a web developer's daily tools: the **Language Server Protocol** (the protocol behind VS Code's and many editors' code intelligence — autocomplete, go-to-definition, error squiggles) is itself built on JSON-RPC 2.0, running between the editor and a separate "language server" process.

The Road to gRPC

gRPC takes this exact same "call a function" mental model and modernizes it: instead of JSON or XML, it uses **Protocol Buffers** — a compact, strongly-typed binary schema format that plays a similar role to SOAP's WSDL (a formal, enforceable contract) but is far lighter weight and easier to

work with. Instead of HTTP/1.1, it runs on **HTTP/2**, which supports genuine bidirectional streaming rather than one request producing exactly one response. This chapter won't go further into gRPC's actual mechanics — Protocol Buffer schema design, the four streaming modes, code generation — that's exactly what the site's dedicated **gRPC course** covers in full.

	XML-RPC	JSON-RPC	gRPC
Format	XML	JSON	Protocol Buffers (binary)
Transport	HTTP/1.1	HTTP/1.1 (usually)	HTTP/2
Formal contract	None	None (informal convention)	Yes — the `.proto` file itself
Status today	Mostly legacy	Actively used (blockchain nodes, LSP)	Actively growing, especially in microservices

💻 Coding Challenges

Challenge 1: Convert REST to RPC

Given the REST call `GET /users/42`, write the equivalent JSON-RPC 2.0 request calling a `getUser` method with the same effective meaning.

Goal: Practice translating between REST's resource-oriented thinking and RPC's method-oriented thinking.

→ Solution

Challenge 2: Write a JSON-RPC Request/Response Pair

Write a correct JSON-RPC 2.0 request calling an `add` method with the numbers 7 and 5, along with the matching response — make sure the `id` values line up correctly.

Goal: Practice the exact JSON-RPC 2.0 shape, including the field that ties a response back to its request.

→ Solution

Challenge 3: Choosing Within the RPC Family

Match each scenario to XML-RPC, JSON-RPC, or gRPC, and justify each choice: (a) integrating with a 15-year-old legacy system that only speaks an older RPC format, (b) a new internal microservice needing very high-throughput, low-latency binary communication with strict typing, (c) a small hobby project's simple JSON-based API between two of your own scripts.

Goal: Practice applying this chapter's comparison table to realistic, distinct scenarios.

→ Solution

⚠️ Gotcha: RPC-Style APIs Don't Get REST's Free HTTP Caching

Chapter 3 covered cacheability as one of REST's architectural constraints — a `GET` request to a stable resource URL can be cached by browsers, proxies, and CDNs using ordinary HTTP caching mechanisms, for free. RPC-style APIs usually break this: since every call typically goes to the *same endpoint* (often via `POST`, with the actual operation named inside the body rather than the URL), there's no unique, stable URL per operation for standard HTTP caching to key off. This isn't a flaw exactly — it's a genuine tradeoff that comes with the method-call model — but it means caching has to be built deliberately at the application layer if it's needed at all, the same caveat GraphQL's course raises about its own single-endpoint design.

🎯 What's Next

The next chapter covers **Webhooks: APIs That Call You** — a genuinely different direction, where instead of your code asking a server for something, the server reaches out and calls *you* the moment something happens.

Webhooks: APIs That Call You

Every API style covered so far — REST, SOAP, RPC/gRPC — has your code as the one initiating a request. Webhooks flip that entirely: your server becomes the one *receiving* a request, sent by someone else's system the instant something happens on their end.

The Reverse Direction: Why Webhooks Exist

Imagine wanting to know the moment a payment succeeds. Without webhooks, the only option is **polling** — repeatedly asking "has it succeeded yet?" (`GET /payments/123/status`) every few seconds, wasting requests and adding delay between the real event and when you find out about it. A webhook lets the payment provider *push* a notification to you the instant it happens, instead.

Polling vs. Webhooks

Polling

Your code repeatedly asks "did anything change?" — wasteful when nothing has, and laggy since you only find out on your next poll, not the instant it happens.

Webhooks

The other system tells YOU, immediately, the moment something happens — no wasted requests, no polling delay.

How a Webhook Actually Works

1. You register a URL ("endpoint") with the provider — e.g. "call this URL when a payment succeeds."
2. Something happens on their end (a payment succeeds).
3. Their server sends an HTTP `POST` request to *your* registered URL, with a body describing the event.
4. Your server processes it and responds quickly with a `2xx` status to acknowledge receipt.

A Realistic Payment-Succeeded Webhook Payload

```
POST /webhooks/payments HTTP/1.1
Content-Type: application/json
X-Signature: t=1717000000,v1=5257a869e7...

{
  "id": "evt_1Nx8k",
  "type": "payment.succeeded",
  "data": { "amount": 4999, "currency": "usd" }
}
```

Signature Verification: Proving the Request Is Genuine

Your webhook endpoint is just a **public URL** — anyone on the internet could send a fake `POST` request to it pretending to be the real provider. To guard against that, providers **sign** each payload using a shared secret (agreed on when you registered the webhook) and include the signature in a header, like `X-Signature` above. Your server recomputes the same signature — typically an **HMAC** (Hash-based Message Authentication Code) over the raw request body, using that same shared secret — and only trusts the request if the computed signature matches the one in the header.

What HMAC Verification Confirms

That the payload genuinely came from someone holding the shared secret (presumably the real provider), and that it wasn't altered in transit — both of which a plain, unsigned `POST` can't guarantee.

What It Doesn't Do

It doesn't encrypt the payload (HTTPS/TLS already handles confidentiality in transit) — it only proves authenticity and integrity of the message itself.

Common Real-World Providers

Stripe sends webhooks for payment and subscription lifecycle events (succeeded, failed, refunded). **GitHub** sends webhooks for repository events (a push, a pull request opened, a comment posted) — the exact mechanism that powers a lot of CI/CD automation. **Slack** sends webhooks for message and interaction events, letting external apps react to things happening inside a workspace. All three follow the same shape this chapter describes: register a URL, get a signed `POST` the moment something happens.

Reliability: Retries and Idempotency

Providers generally can't guarantee your endpoint is always up, so most implement **at-least-once delivery**: if your endpoint doesn't respond with a `2xx` quickly (or times out), they'll retry — which

means your endpoint might genuinely receive the *same* event more than once. This is the same underlying problem Chapter 4's Idempotency-Key pattern solved for outgoing requests, just from the receiving side: each webhook payload includes a unique event `id` (like `evt_1Nx8k` above), and your handler should track which ids it's already processed, skipping any repeat instead of, say, charging a customer or sending a confirmation email a second time.

💻 Coding Challenges

Challenge 1: Polling vs Webhook

A food delivery app needs to know the instant a driver marks an order as delivered, to notify the customer immediately. Recommend polling or a webhook, and justify using this chapter's comparison.

Goal: Practice recognizing when the instant-notification property of webhooks genuinely matters.

→ Solution

Challenge 2: Verify a Webhook Signature

Describe, step by step, what a server should do upon receiving a webhook with an `X-Signature` header, to determine whether to trust the payload or reject it.

Goal: Practice explaining the HMAC verification process concretely, not just naming it.

→ Solution

Challenge 3: Handle a Duplicate Webhook Delivery

A `payment.succeeded` webhook with event id `evt_1Nx8k` arrives twice, a few seconds apart, because your server's first acknowledgement was slow. Describe how your handler should avoid sending the customer two confirmation emails.

Goal: Practice applying event-id-based deduplication to a concrete double-delivery scenario.

→ Solution

⚠️ Gotcha: Slow Webhook Handlers Cause Their Own Duplicate Deliveries

It's tempting to do the real work — sending a confirmation email, updating several database records, calling another service — directly inside the webhook handler, before responding. But if that work takes too long, the provider's request can time out waiting for your `2xx` response, conclude the delivery failed, and retry — even though your server actually received and is still processing the original one. The fix mirrors the background-job pattern already covered in the Django, Rails, and Laravel courses: acknowledge receipt immediately with a fast `2xx`, then hand the real work off to a background job or queue, processed asynchronously after the response has already gone out.

🎯 What's Next

The next chapter, **Choosing the Right API Style**, brings everything together — a decision framework comparing REST, SOAP, RPC/gRPC, GraphQL, WebSockets, and Webhooks by use case, so you can pick the right one deliberately instead of by habit.

⚖️ Choosing the Right API Style

Six styles are now on the table: REST, SOAP, RPC/gRPC, GraphQL, WebSockets, and Webhooks. This chapter doesn't crown one winner — it gives a small decision framework: four questions to ask about your actual situation, and how the answers point toward one style over another.

The Full Lineup, One Line Each

Style	One-Liner	Best For
REST	Resource-based request/response over HTTP	General-purpose public and web APIs
SOAP	Formal XML contracts, WS-* standards	Enterprise/banking/government systems needing strict guarantees
RPC / gRPC	Call a specific function directly	High-performance internal microservices
GraphQL	Client asks for the exact shape of data it wants	Complex UIs assembling data from many sources
WebSockets	A persistent, two-way connection	Real-time bidirectional communication
Webhooks	The server calls you when something happens	Event notification, avoiding polling

Question 1: Who's the Client — the Public, or You?

A broad, external, uncoordinated audience with variable needs generally points toward **REST** or **GraphQL** — both are widely understood and tooling-friendly. Tightly coupled internal services you also control point toward **gRPC** — its performance and strict typing pay off precisely because you control both ends and can regenerate client code whenever the contract changes. Formal enterprise partners requiring guaranteed contracts and transactional integrity point toward **SOAP**.

Question 2: Is It Actually Request/Response, or Something Else?

If the communication is genuinely two-way and continuous — a chat app, a live dashboard, a multiplayer game — that's **WebSockets** territory, already covered in its own full course. If the real need is "notify me later, don't make me keep asking," that's **Webhooks**. If it's a client asking a question and getting one answer back, the request/response styles (REST, SOAP, RPC/gRPC, GraphQL) are the right family to be choosing among at all.

Question 3: How Complex Are the Client's Data Needs?

If clients mostly need simple, fixed-shape access to a resource, plain **REST** is enough — and simpler to build and reason about than the alternative. If a client needs to assemble deeply nested, *varying* combinations of data across many different screens (recall the GraphQL course's over-fetching/under-fetching problem), that's exactly the complexity **GraphQL** earns its own extra weight by solving.

Question 4: What Guarantees Do You Actually Need?

If you need enforced, machine-validated contracts plus built-in transactional and message-level security guarantees, that's **SOAP**'s specific strength (Chapter 5). If you need raw performance and binary efficiency for service-to-service calls, that's **gRPC**. Otherwise, REST's informal contract — optionally documented with OpenAPI, as covered in `express2-7` — is enough for the vast majority of APIs.

A Worked Example

Consider building the backend for a public social media platform:

The Main API

Broad external audience (Q1 → REST or GraphQL), and mobile screens assembling posts + comments + likes + author info in different combinations per screen (Q3 → the complexity GraphQL is built for). **Conclusion: GraphQL** for the main data API.

Live Notifications

Genuinely continuous, two-way communication the instant a new like or comment arrives (Q2 → not request/response at all). **Conclusion: WebSockets**, already covered in its own course.

Third-Party Integrations

A partner app wants to know whenever a user posts something, without polling constantly (Q2 → event notification, not a live connection). **Conclusion: Webhooks** for outbound event notifications to partners.

Notice the answer wasn't "one style for the whole platform" — real systems very often combine several API styles, each handling the specific part of the problem it's actually good at.

💻 Coding Challenges

Challenge 1: A Public Weather API

You're building a public developer-facing weather API expecting millions of simple, mostly-uniform requests (get the forecast for a location). Which style fits, and why?

Goal: Practice recognizing when the SIMPLEST fitting style is the right choice, not the most sophisticated one.

→ Solution

Challenge 2: Two Internal, Latency-Sensitive Microservices

An order service and an inventory service, both owned by your own team, need extremely low-latency communication with each other. Which style fits, and why?

Goal: Practice applying Question 1 and Question 4 together — control over both ends, plus a genuine performance requirement.

→ Solution

Challenge 3: Design a Multi-Style System

An e-commerce platform needs: (a) a public product-browsing API, (b) live order-tracking updates for a customer watching their delivery, and (c) letting partner logistics companies know the instant stock runs low. Identify which API style fits each of the three needs.

Goal: Practice the worked example's lesson — that one real system often needs more than one API style.

→ Solution

⚠️ Gotcha: Forcing One API Style to Do Everything

It's tempting, once you've committed to building a REST API, to make it handle EVERYTHING — including needs it's genuinely bad at, like simulating "real-time" updates by having the client poll a REST endpoint every second. That approach technically works, but it reproduces exactly the wasteful, laggy pattern Chapter 7 described polling as a problem in the first place — when a WebSocket connection or a webhook would solve the same need properly. The worked example above is the real lesson: don't default to whichever style you already have running: ask, per feature, which style that specific piece of functionality actually needs.

🎯 What's Next

The final chapter, **Capstone: Designing an API for a Real Scenario**, puts everything from this course into practice — given a real scenario, choosing and justifying an API style and sketching out its actual design.

🎓 Capstone: Designing an API for a Real Scenario

Every chapter so far has worked in isolated pieces. This capstone puts them all together — one real scenario, walked through end to end, from picking a style with Chapter 8's decision framework all the way to a concrete endpoint and webhook design.

The Scenario

A task-management SaaS company wants to let outside developers build integrations against their platform — a Slack bot that creates tasks from messages, a calendar sync tool, a reporting dashboard. Those integrators need to read and create tasks, and be notified the moment a task's status changes, without hammering the platform with constant polling.

Applying the Decision Framework

Question (from Chapter 8)	Answer for This Scenario
Who's the client?	A broad, external, uncoordinated audience of third-party developers → points toward REST
Request/response, or something else?	Reading/creating tasks is request/response; "notify me when status changes" is not → that piece needs Webhooks
How complex are the client's data needs?	Mostly uniform CRUD access to tasks, not wildly varying nested shapes per integrator → plain REST is enough, GraphQL's extra complexity isn't earned here
What guarantees are needed?	No enterprise/transactional guarantees needed (ruling out SOAP); no internal extreme-performance requirement (ruling out gRPC) for this external-facing part

Conclusion: REST for the main API, Webhooks for change notifications — exactly the kind of two-styles-together answer Chapter 8's worked example arrived at.

Designing the REST Endpoints

Method	Path	Purpose
GET	/v1/tasks	List tasks (paginated, filterable)
GET	/v1/tasks/{id}	Retrieve one task
POST	/v1/tasks	Create a new task
PATCH	/v1/tasks/{id}	Partially update a task (e.g. just its status)
DELETE	/v1/tasks/{id}	Delete a task

The `/v1/` prefix applies Chapter 4's URL versioning strategy — the right call here since the audience is broad and external, exactly the case Chapter 4 recommended it for. `PATCH` rather than `PUT` for updates reflects Chapter 2's distinction: most real updates here (like changing just a status field) are partial, not full replacements.

Handling Creation Safely: Idempotency

Creating a Task, Safely Retryable

```
POST /v1/tasks HTTP/1.1
Authorization: Bearer <token>
Idempotency-Key: 3a1f9c2e-...
Content-Type: application/json

{ "title": "Follow up with client", "dueDate": "2026-08-01" }
```

If an integrator's request times out and their code retries automatically, the same `Idempotency-Key` (Chapter 4) ensures the platform creates exactly one task, not two.

Pagination for Task Lists

Since tasks are added and completed constantly by many users at once, cursor-based pagination (Chapter 4) is the right fit — offset-based pagination would risk exactly the skipped/duplicated-row problem Chapter 4's gotcha described:

Paginating /v1/tasks

```
GET /v1/tasks?after=eyJpZCI6NTIwfQ==&limit=20
```

Authentication & Rate Limiting

Each request authenticates with a bearer token in the `Authorization` header — the token-based approach covered in depth in the Authentication & Session Security course (`bc1-7`), not re-taught here. Requests are capped per integrator using the `X-RateLimit-*` headers and `429` status introduced in Chapter 4, protecting the platform from any single integrator's runaway retry loop.

Adding Webhooks for Real-Time Notification

Rather than making integrators poll `GET /v1/tasks/{id}` repeatedly to detect a status change, each integrator registers a webhook URL once. When a task's status changes, the platform sends a signed event:

A task.updated Webhook

```
POST https://integrator-app.com/webhooks HTTP/1.1
Content-Type: application/json
X-Signature: t=1717000000,v1=8a41c...

{
  "id": "evt_88f2",
  "type": "task.updated",
  "data": { "taskId": 501, "status": "completed" }
}
```

The integrator verifies `X-Signature` (Chapter 7's HMAC pattern) before trusting the payload, and deduplicates on `evt_88f2` in case the same event is delivered more than once.

What About Internal Services?

If the platform's own task service needed to talk to its own internal search-indexing service at very low latency, gRPC (Chapter 6, with its own full course) would be the right internal choice — but that's a separate concern from this external-facing design, and folding it in here would repeat exactly the mistake Chapter 8's gotcha warned against: reaching for whichever style is already in front of you rather than asking what each specific piece of the system actually needs.

Course Recap

Across nine chapters: the client-server model and what "API" really means (Ch.1), HTTP's methods/status codes/headers (Ch.2), REST's architectural constraints (Ch.3), REST's practical design patterns (Ch.4), SOAP's formal contracts (Ch.5), the RPC lineage into gRPC (Ch.6), Webhooks (Ch.7), a decision framework (Ch.8), and this capstone applying all of it to one real design. From here, the site's **API Testing & Tooling** course covers how to actually test a design like this one with Postman, cURL, and more, and the dedicated **gRPC** course goes deep on the internal-services piece only briefly touched on above.

💻 Coding Challenges

Challenge 1: Add a Projects Resource

Tasks belong to projects. Design the REST endpoints for a new `/v1/projects` resource, following the same conventions (versioning, method choices, pagination) established in this capstone.

Goal: Practice extending a design consistently rather than inventing new conventions per resource.

→ Solution

Challenge 2: Design a task.completed Webhook

Design the webhook payload for a new `task.completed` event, including a plausible event id and signature header, following this chapter's `task.updated` example.

Goal: Practice applying the webhook payload shape and signature pattern to a new event type.

→ Solution

Challenge 3: Justify Cursor Pagination for /v1/tasks

Explain specifically why cursor-based pagination was chosen for `/v1/tasks` instead of offset/limit, referencing what would concretely go wrong with offset-based pagination given how this platform's tasks change over time.

Goal: Practice justifying a design decision with a concrete failure mode, not just a general preference.

→ Solution

⚠️ Gotcha: Designing Only the Happy Path

It's easy to design an API around what happens when everything goes right — a valid request, a successful create, a clean 200. A real design also has to account for what happens when a task id doesn't exist (`404`), the bearer token is missing or expired (`401`), a required field is missing from the request body (`400`), or the integrator is sending requests too fast (`429`) — all status codes and concepts this course already covered, but easy to forget to actually design for once the "main" functionality is working. A design that only handles success is an incomplete design, however good the happy path looks.

🎉 Course Complete

That's the full **API Types & Design** course — from "what is an API?" through REST, SOAP, RPC/gRPC, Webhooks, a decision framework, and this final end-to-end design. Next steps from here: the **API Testing & Tooling** course to learn how to actually exercise an API like the one designed above, or the dedicated **gRPC** course to go deep on the internal-services piece this capstone only touched on briefly.